Mix Your Contexts Well: Opportunities Unleashed by Recent Advances in Scaling Context-Sensitivity

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

V. Krishna Nandivada
IIT Madras, India
nvk@iitm.ac.in

Abstract

Existing precise context-sensitive heap analyses do not scale well for large OO programs. Further, identifying the right context abstraction becomes quite intriguing as two of the most popular categories of context abstractions (call-site- and object-sensitive) lead to theoretically incomparable precision. In this paper, we address this problem by first doing a detailed comparative study (in terms of precision and efficiency) of the existing approaches, both with and without heap cloning. In addition, we propose novel context abstractions that lead to a new sweet-spot in the arena.

We first enhance the precision of level-summarized relevant value (LSRV) contexts (a highly scalable abstraction with precision matching that of call-site-sensitivity) using heap cloning. Then, motivated by the resultant scalability, we propose the idea of mixing various context abstractions, and add the advantages of k-object-sensitive analyses to LSRV contexts, in an efficient manner. The resultant context abstraction, which we call *lsrvkobjH*, also leads to a novel connection between the two broad variants of otherwise incomparable context-sensitive analyses. Our evaluation shows that the newer proposals not only enhance the precision of both LSRV contexts and object-sensitive analyses (to perform control-flow analysis of Java programs), but also scale well to large programs.

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

Keywords Static analysis, Context-sensitivity, Java

ACM Reference Format:
Manas Thakur and V. Krishna Nandivada. 2020. Mix Your Contexts Well: Opportunities Unleashed by Recent Advances in Scaling Context-Sensitivity. In *Proceedings of the 29th International Conference on Compiler Construction (CC '20), February 22–23, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377555.3377902

CC '20, February 22–23, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-7120-9/20/02…$15.00
https://doi.org/10.1145/3377555.3377902

1 Introduction

Context-sensitive analyses, specially for object-oriented languages such as Java, are known to be notorious for their precision-scalability trade-offs. Compared to their context-insensitive counterparts, context-sensitive analyses have been shown to improve the precision significantly; however, the scalability of precise context-sensitive analyses continues to be a cause of concern. The degree by which a given context-sensitive analysis improves precision depends on how well the context abstraction partitions the dataflow facts across the various created contexts. Similarly, the degree by which a context-sensitive analysis may increase the analysis time depends on whether the partitioned dataflow facts succeed or fail in leading to an enhanced precision for the analysis under consideration. Thus, as also noted by prior works [7, 26], the choice of context abstraction plays a very important role in deciding whether a given context-sensitive analysis gives good enough precision for its associated cost.

The classical call-strings approach [23, 24], which uses the string formed by the callers of a method as the context, statically models the run-time stack, and hence is arguably the most intuitive context abstraction. The value-contexts approach [10, 20] uses the dataflow values reaching the call-sites of a method to restrict the unbounded growth of call-strings, and thus provides a way to realize the precision of infinite-length call-strings. Thakur and Nandivada [30] identify the relevant portions of value contexts and summarize them to form the analysis-specific abstraction of LSRV contexts; their approach scales individual value-contexts based analyses, and hence call-strings based analyses, without compromising on the precision of the analysis.

Object-sensitivity [16], another popular context abstraction, distinguishes contexts based on the chain formed by the allocation sites of successive receiver objects. Over the last few years, limited-length versions of object-sensitivity (abbreviated as $k$obj) have been shown to offer reasonably well-balanced precision-scalability trade-offs for object-oriented programs [28, 29]; their variants have also been introduced with a great interest, specially in order to make the original approaches scale to large programs [13, 14, 22].

Nystrom et al. [18] proposed heap cloning as yet another technique to improve the precision of existing context abstractions. Heap cloning specializes allocation sites with the context of creation, which usually improves the partitioning of the per-context information computed by a given analysis.

Though this improves the precision of context-sensitive analyses significantly, typical heap analyses with even one-level of heap-cloning do not scale to large benchmarks. Recent approaches [14, 27] solve this problem by coming up with heuristics to perform context-sensitivity selectively, thus scaling such analyses by sacrificing some precision.

We note two salient points in the above discussion. First, the popular choices of context abstractions can be divided broadly into two variants: call-string and object-sensitivity based. As shown in Figure 1, the highly scalable LSRV contexts (and its basis - value contexts) are based on call-site-sensitivity, and have the same precision as full-length classical call strings. However, the precision of the other popular variant (object-sensitivity) is incomparable with the call-site-sensitive variants. Second, though heap cloning is known to improve the precision of individual variants, the effects of heap cloning on the newer context abstractions, such as value contexts and LSRV contexts, are not known. In this paper, we address the resulting challenges from these points, while giving a clear picture of the theoretical and practically achievable precision of the popular existing context-sensitivity abstractions, with and without heap-cloning. In the pursuit to do so, we also present a novel way of mixing the existing, otherwise incomparable, context abstractions to realize a novel sweet-spot between precision and scalability.

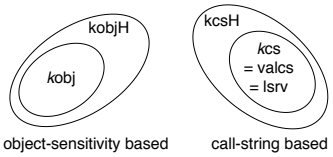


Figure 1. Relative precision of existing context abstractions.

We show that heap-cloning leads to interesting conclusions in terms of the relative precision of different existing context abstractions. We enhance the precision of LSRV contexts using heap cloning, we call it *lsrvH*. We show that unlike their counterparts without heap-cloning, the heap-cloned versions of value-contexts and *lsrvH* do not have the same precision. But their precision remains incomparable with the object-sensitive variants with heap-cloning.

Motivated by the scalability of the LSRV approach, we next extend its definition to also consider the k-level object context as the context abstraction. This not only leads to a very novel experiment of performing a very precise context-sensitive analysis (we term the resultant parameterized abstraction as *lsrvkobjH*), but also uniquely results in scalable analyses that are more precise than both call-string based and object-sensitivity based analyses.

We implement the proposed approaches to perform Java control-flow analysis, in the Soot framework [32]. We evaluate the implementations by comparing them with standard k-object-sensitive analyses with heap cloning. The results show that not only do the newer proposals scale well to large benchmarks, they also enhance the precision of both LSRV-contexts based and object-sensitive analyses.

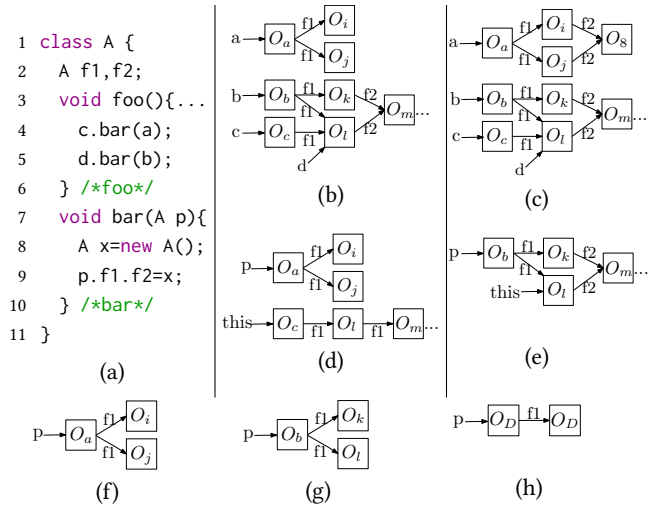


Figure 2. (a) A Java code snippet. (b) and (c) The assumed points-to graphs at lines 4 and 5. (d) and (e) The value contexts for `bar` at lines 4 and 5. (f) and (g) The relevant value-contexts for `bar` at lines 4 and 5. (h) The LSRV context for `bar` at lines 4 and 5 (for escape analysis), assuming O_a, O_b, O_i, O_j, O_k and O_l do not escape; O_D represents a universal non-escaping object. Figure derived from a similar code used by Thakur and Nandivada [30].

2 Background

In this section, we give a brief background of some popular context abstractions (for heap analyses) and heap cloning.

2.1 Existing Context Abstractions

Call-site sensitivity. The call-string based approach [23, 24] identifies contexts based on the call-string formed by a method's callers. For example, the call-strings approach would analyze the method `bar` in Figure 2a in two contexts – one each for the calls at lines 4 and 5. A major drawback of the call-strings approach is that in the presence of recursion and deep nesting of multiple calls, the length of the call-strings, and hence the number of contexts, may grow combinatorially. This makes the analysis unscalable to large real-world programs. Consequently, to improve scalability, typical call-site sensitive analyses usually impose a limit on the call-string length, and treat the greater length contexts conservatively; however, it compromises on the precision.

Value contexts. Using value-contexts [10], in a top-down context-, flow-, and field-sensitive points-to analysis [20], on reaching a call-statement for a method m, the method is (re-)analyzed, if the current *value context* is different from the prior value contexts (if any) in which m was analyzed. Here, the value context at a call to m is the points-to (sub) graph passed to m, which is the *parameter-reachable graph* of m. For the code shown in Figure 2a, the points-to relationships (represented as parameter-reachable graphs in Figures 2d and 2e) at both the calls to `bar` are different, and hence `bar` would be analyzed in two different contexts. The disadvantage of value contexts for heap analyses is that the "values", that is, the points-to graphs tend to grow very large resulting

into significant memory overheads and costly comparison at each call-site, thus making the analysis unscalable.

LSRV contexts. Recently, Thakur and Nandivada [30] proposed *level-summarized relevant value-contexts* (LSRV contexts) as a way to scale value-contexts based heap analyses. For a given points-to graph based analysis, instead of comparing the complete points-to graphs at each call-site, the LSRV approach compares only the *relevant* portion of the *level-summarized* points-to graphs. The approach is divided into three stages: pre-, main-, and post-analysis.

The notion of relevance determines the portion of the callers' heap that is actually accessed by a called method (approximated in the pre-analysis). For example, the method `bar` in Figure 2a accesses only the parameter `p` passed from its caller(s) and that too only up to two levels (for `p.f1.f2`), and hence the corresponding *relevant points-to graphs* at lines 4 and 5 are as shown in Figures 2f and 2g, respectively.

On top of the notion of relevance, level-summarization computes a per-level summary of the relevant points-to graphs by taking a meet of the represented analysis-specific dataflow values (for example, escape-statuses D and E representing *DoesNotEscape* and *Escapes*, respectively, for escape analysis, and types of the represented objects for control-flow analysis), instead of the actual nodes of the points-to graphs. For example, assuming none of the objects in Figures 2f and 2g escape, the LSRV context while performing escape analysis for both the calls to `bar` in Figure 2a is as shown in Figure 2h. As a result, the method `bar` would be analyzed only once. Note that if the lattice of dataflow values of the analysis being performed is smaller than that of the points-to graphs (which is quite often the case), then LSRV contexts significantly scale value-contexts based analyses, without compromising on their precision (because of the sound but conservative definition of relevance; see [30]).

Object sensitivity. An object-sensitive analysis [16] distinguishes the contexts of a method based on the allocation site of the receiver object. For scalability, similar to call-string based analyses, object-sensitive analyses also use a limit k on the length of the chain formed by the receivers. For example, a one-level object-sensitive analysis (denoted as 1obj) would analyze the method `bar` (see Figure 2a) in two contexts – at lines 4 and 5 – as the receiver objects at both the sites are different (O_c and O_l, respectively). Over the last few years, object sensitivity has become a much used context abstraction; further, there are several recent works that focus on scaling existing object-sensitive analyses, usually by compromising on their precision [13, 14, 22, 29].

2.2 Heap Cloning

Heap cloning [18] is a technique to specialize the different instances of the objects allocated on the heap, based on the context in which they are created. This may lead to the removal of some spurious points-to facts, and hence may improve the precision of the analysis being performed. Say we are performing a call-string based context-sensitive points-to analysis. In Figure 2a, without heap cloning, though `bar` is analyzed it two contexts (created at lines 4 and 5), after the call, both `a.f1.f2` and `b.f1.f2` in `foo` point to the same object O_8, and thus are identified as aliases (though they will point to different objects at runtime). With heap cloning, the object created at line 8 is qualified by the context in which it is created (say O_{4_8} and O_{5_8}). As a result, in method `foo`, `a.f1.f2` and `b.f1.f2` would point to different objects (O_{4_8} and O_{5_8}, respectively), and would (correctly) not be identified as aliases, hence possibly triggering other optimizations.

```
1 class D {
2   void fb(A y) {/*y not null*/
3     A t1 = t2 = y;
4     while (t1 != null) {
5       t2 = t1;
6       t1 = t2.f; } /*while*/
7     t2.f = new A();
8     if (*) { fb(y); }
9   } /*fb*/ } /*class D*/
```

(a)

```
1 class D {
2   void m1() {
3     P p = new P();
4     Q q1 = p.m2();
5     Q q2 = p.m2(); }
6   void m2() {
7     return new Q();
8   } /*m2*/
9 } /*class D*/
```

(b)

Figure 3. Code snippets to show that (a) *valcsH* may not terminate; (b) *valcsH* is less precise than *kcsH*.

3 Value/LSRV Contexts with Heap Cloning

In this section, we enhance the precision of value contexts and LSRV contexts using heap cloning.

3.1 Adding Heap Cloning to Value Contexts

We now briefly discuss an as-yet unexplored context abstraction – value contexts with heap cloning; we call it *valcsH*. In *valcsH*, each abstract object can be represented using two values: the allocation site and the value context in which it is created. As expected by specializing the heap, *valcsH* might partition the dataflow facts better than *valcs* and thus lead to an enhanced precision. An interesting fact about each cloned object in *valcsH* based analysis is that there is a corresponding cloned object in the corresponding call-site sensitive analysis with heap cloning.

Termination. Note that without heap cloning, the number of abstract objects that could be created while performing a heap analysis is bound by the number of allocation statements in the program, which is finite. However, with heap cloning, the number of objects (and hence the termination of the analysis) also depends on the number of contexts that could be created. We now present an interesting observation concerning *valcsH*, which shows that unlike traditional value contexts, *valcsH* might not terminate in the presence of recursion. Consider the code snippet shown in Figure 3a. Here, for each subsequent call to the method `fb` at line 8, with heap cloning, the value context keeps changing (as the object allocated at line 7 in each context is different). As a

result, an analysis using *valcsH* contexts might not terminate for such recursive calls that keep adding a heap-cloned object as part of the "value" (that is, the points-to graph at the entry of a method), which leads to a new value context at each call-site to the method being called recursively.

3.2 Adding Heap Cloning to LSRV Contexts

Similar to value contexts, we now extend LSRV contexts with heap cloning; we call the resultant version *lsrvH*. As heap cloning may only improve the partitioning efficacy, the precision of *lsrvH* can be more but never less than LSRV contexts, with a possible trade-off in the analysis time. In Section 6, we evaluate *lsrvH* by comparing it with plain LSRV contexts, for performing control-flow analysis of Java programs; as we will see, the *lsrvH* approach scales over all the benchmarks under consideration, and noticeably improves precision over the corresponding analysis using plain LSRV contexts.

Termination. Similar to *lsrv* contexts, *lsrvH* contexts restrict the growth of the values, that is, points-to graphs, using level-summarization. Thus, for LSRV contexts with heap cloning, if the lattice of the analysis under consideration is finite (which bounds the number of possible contexts), the number of objects that could get created with heap cloning is also finite, and the analysis would still be guaranteed to terminate. For example, in Figure 3a, contrary to *valcsH*, the *lsrvH* approach for analyses with a finite lattice (such as escape statuses in escape analysis and types in control-flow analysis) is guaranteed to terminate.

4 Relative Precision

Given the variety of choices we have discussed in the previous sections (existing context abstractions in Section 2 and two new abstractions with heap cloning in Section 3), an analysis writer needs to carefully choose a particular abstraction to suit the requirements of the context-sensitive analysis under consideration. The major factors affecting the decision are the precision and the scalability of the analysis with a given context abstraction. The focus of this section is the relative precision of these abstractions, with and without heap cloning; later, Section 6 evaluates the different approaches to compare the same empirically.

The context abstractions discussed in Sections 2 and 3 can be broadly categorized into two classes: call-string based, which includes value contexts and LSRV contexts, and object-sensitivity based. Figure 4 shows the abbreviations used to represent the context abstractions discussed in this paper; as higher levels of heap cloning have not been used in literature in a scalable manner, we limit the discussion to a single level of heap cloning (denoted as XH for an abstraction X), for all the variants. We now describe our comparison scheme, followed by discussions to establish the precision relations.

Terminology. In order to measure the precision of an analysis, we use the number of *optimization opportunities* generated by that analysis; for example, the number of calls that could be resolved as monomorphic can be a measure of the precision of control-flow analysis. While comparing two context abstractions X_1 and X_2, we say that X_1 has a higher *theoretical precision* than X_2, if an analysis using X_1 is guaranteed to cover all the optimization opportunities generated by an analysis using X_2. On the other hand, if an analysis using a context abstraction X_1 is known to terminate faster than another using X_2, then we say that X_1 has a higher *scalability* than X_2. Further, if X_1 and X_2 have the same theoretical precision, but X_1 has a higher scalability than X_2, then we say that the *practical precision* of X_1 is higher than that of X_2. Note: to establish a hypothesis that X_1 can be less precise than X_2, it is enough to show an example where X_1 misses an optimization opportunity captured by X_2.

Abbreviation	Abstraction
kcs	*k*-length call-site-sensitive analysis.
kobj	*k*-length object-sensitive analysis.
valcs	Value-contexts based analysis.
lsrv	LSRV-contexts based analysis.
XH	Variant *X* with heap cloning.

Figure 4. Notations used to represent context abstractions.

4.1 *kcs* versus *valcs*

The value-contexts approach (*valcs*) simply scales call-string based analyses (*kcs*), and as shown by Padhye and Khedker [20], each value context for a method can be mapped back to a call-string based context. Thus, the theoretical precision of *kcs* and *valcs* is the same. However, note that this relation holds for a full-length call-string based analysis (that is, $k = \infty$). In practice, call-string based analyses for higher values of k (> 2 or 3) are known not to scale, and hence, *valcs* offers a better practical precision than *kcs*.

4.2 *valcs* versus *lsrv*

For a given analysis, Thakur and Nandivada [30] show that LSRV contexts (*lsrv*) only scale the corresponding value-contexts based analysis, without affecting its precision. Further, *lsrv* can be trivially extended to maintain a map that records the value contexts for which *lsrv* skips analyzing a method. Thus, for a given analysis, the theoretical precision of both the approaches is the same. However, as shown by Thakur and Nandivada [30], *valcs* does not scale for popular whole-program heap analyses. Hence, *lsrv* offers a much better practical precision than *valcs*.

4.3 *kcs*/*valcs*/*lsrv* versus *kobj*

The contexts created in the call-string based and the object-sensitive approaches are quite different: in the former, the contexts created for a method can be directly mapped to the runtime call-stack; whereas in the latter, the contexts created depend on the possible receiver objects. Consequently, the per-context theoretical precision of object-sensitive analyses

```
1  class D {
2    ...
3    void foo() {
4      B x = new B();
5      Y y = new Y();
6      Z z = new Z();
7      x.m3(y);
8      x.m3(z); } /*foo*/
9    void bar() {
10     B o1 = new B();
11     B o2 = new B();
12     B o3 = * ? o1 : o2;
13     o3.m4(); } /*bar*/
14   B m3(B p) { p.f = p;}/*m3*/
15   B m4() {
16     this.f = this; } /*m4*/ }
```

Figure 5. Example to show the incomparability of *kcs*/*valcs*/*lsrv* and *kobj*.

cannot be compared with those of call-string based analyses. As value contexts, and hence LSRV contexts, are also based on call-site sensitivity, they are also theoretically incomparable with object-sensitive analyses.

Theoretical precision aside, there are cases where one of the above two approaches may enable an optimization opportunity, and not the other. Figure 5 shows one case where *kcs*/*valcs*/*lsrv* are more precise than *kobj*, and another case where *kobj* is more precise than *kobj*/*valcs*/*lsrv*. The method `foo` calls the method `m3` at lines 7 and 8. In *kobj*, as the receiver object in both the calls is O_4, the analysis happens in only one context, both `y.f` and `z.f` conservatively point to both O_5 and O_6, and hence are aliases (imprecise).

On the other hand, in the method `bar`, where using *kobj* leads to distinct contexts for the method `m4` (one each for the receivers O_{10} and O_{11}), using *kcs*/*valcs*/*lsrv* results in only one context for `m4`. Consequently, though `o1.f` and `o2.f` point to different objects (O_{10} and O_{11}, respectively) in *kobj*, they point to the same set of objects $\{O_{10}, O_{11}\}$ in *kcs*/*valcs*/*lsrv* and become aliases (imprecise). The above examples further illustrate the incomparability of the precision of the call-string based and the object-sensitive approaches.

Sections 4.1-4.3 establish the precision relations among various context abstractions in the absence of heap cloning. The next three sections illustrate how heap cloning changes the obtained relations in a somewhat surprising manner.

4.4 *kcsH* versus *valcsH*

As discussed in Section 4.1, without heap cloning, the theoretical precision of analyses that use value contexts (*valcs*) is the same as the ones that use call-strings (*kcs*). However, specializing the heap gives a different result, as illustrated in Figure 3b. With heap cloning, the object O_7 in the method `m2` is qualified with the context in which it is created. However, in *valcsH* (and also in *lsrvH*), `m2` is analyzed in only one context (as the respective values at lines 4 and 5 are the same). As a result, in *valcsH*, the variables `q1` and `q2` are marked as aliases (imprecise); whereas in *kcsH*, `m2` is analyzed twice (for the calls at lines 4 and 5 in `m1`), `q1` and `q2` point to different objects (O_{4_7} and O_{5_7}, respectively), and do not alias (precise). As observed in Section 3.1, for each context where *valcsH* clones an object, the object would have been cloned even by *kcsH*. However, the above example shows that the reverse is not true, and hence unlike the versions without heap cloning, *valcsH* is less precise than *kcsH*. Note that this relation holds only for $k = \infty$, and thus for finite values of k the precisions of *valcsH* and *kcsH* are incomparable.

```
1  class W {
2    X f;
3    W() { f = new X(); }
4    void setG(Y y) {
5      f.g = y; }
6    Y getG() {
7      return f.g; } }
8  class X { Y g; }
9  class Y {
10   void m() {...} }
11 class Z extends Y {
12   void m() {...} }
13 class D {
14   void bar() {
15     W w1 = new W();
16     Y y1 = new Y();
17     w1.setG(y1);
18     W w2 = new W();
19     Z z1 = new Z();
20     w2.setG(z1);
21     Y p = w1.getG();
22     p.m();
23     Y q = w2.getG();
24     q.m(); } }
```

Figure 6. An example where an *lsrvh* control-flow analysis is less precise than one based on *kobjh*.

4.5 *lsrvH* versus *kobjH*

Another interesting precision relationship can be observed when heap cloning is added to both LSRV contexts (to obtain *lsrvH*) and to object-sensitivity (to obtain *kobjH*). Figure 6 shows a case where a control-flow analysis using *lsrvH* may be less precise than one using *kobjH*. Here, the class `W` is like a container with a field `f` of class `X` (initialized in the constructor of `W`). The field `g` of `X` objects may in turn store an object either of class `Y` or of class `Z` (as `Z` extends `Y`).

The method `bar` of class `D` creates two `W` instances at lines 15 and 18, pointed-to by `w1` and `w2`, respectively. However, as the *lsrvH* contexts for the `W` constructor at both the allocation sites are the same (the level-summarized receiver), an *lsrvH* control-flow analysis would not re-analyze the constructor at line 18. As a result, the object pointed-to by `w1.f` and `w2.f` would be the same, that is, O_3. Consequently, though the method `setG` is analyzed in two *lsrvH* contexts (due to the types of the parameter being different) at lines 17 and 20, both `w1.f.g` and `w2.f.g` would point to both O_{16} and O_{19}. Thus, the variables `p` and `q` would point to both `Y` and `Z` objects (O_{16} and O_{19}, respectively), leading to both the calls to the method `m` (that is, at lines 22 and 24) being deemed as polymorphic (having multiple targets – imprecise).

Contrary to *lsrvH*, a *kobjH* control-flow analysis would forcefully re-analyze the constructor of class `W` at line 18 (as the receiver object is different). As a result, `w1.f` and `w2.f` would point to two different objects (O_{15_3} and O_{18_3}, respectively), `w1.f.g` and `w2.f.g` would respectively point to O_{16} and O_{19}, and hence both the calls to the method `m` would be monomorphic (having a single target – precise).

The above example shows that *lsrvH* can be less precise than *kobjH*. We now argue the other way and show that *kobjH* can also be less precise than *lsrvH*. In Figure 5, owing to different types of parameters, the *lsrvH* approach will

create two different contexts at lines 7 and 8, and precisely identify that `o1.f` and `o2.f` do not alias after the calls. On the other hand, a *kobjH* analysis would imprecisely identify `o1.f` and `o2.f` as aliases (see Section 4.3). Thus, we conclude that the precisions of *lsrvH* and *kobjH* are incomparable.

4.6 *lsrvH* versus *valcsH*

The addition of heap cloning modifies yet another precision relation with respect to the one without heap cloning. As the value context changes if the receiver has changed, in Figure 6, *valcs* also re-analyzes the constructor of class `W` at line 18. Also, it is trivial to note that whenever *lsrvH* clones an object, the object would have been cloned even by *valcsH*. Thus, we conclude that *lsrvH* is less precise than *valcsH*. However, as we show in Section 6, contrary to *lsrvH*, the *valcsH* approach does not scale well (reasons being the possible non-termination as discussed in Section 3.1 and the non-scalability of *valcs* itself [30]).

Discussion. Overall, for the existing context abstractions and the newer variants introduced in Section 3, we can make three important observations: (i) As the LSRV approaches are not *k*-limited, they offer the best practical precision among all context abstractions. (ii) Compared to object-sensitivity with heap cloning, there are cases where the LSRV approach with heap cloning is not able to capture some optimization opportunities. This is because the LSRV (and value context) approaches might not create new contexts for some methods (such as the constructor of class `W` in Figure 6) as against the object-sensitive approach. (iii) The relative precision of the call-site-sensitive (*kcs*, *kcsH*, *valcs*, *valcsH*, *lsrv*, *lsrvH*) and the object-sensitive (*kobj* and *kobjH*) context abstractions can still not be compared. However, the theoretical precisions of various variants within each set can be summarized as follows: $kcsH > valcsH > lsrvH > (kcs = valcs = lsrv)$ and $kobjH > kobj$. Further, the practical precisions of the variants with the same theoretical precision can be summarized as: $lsrv > valcs > kcs$ [30].

Using the first two insights discussed above, we next decribe a novel way of mixing the LSRV and the object-sensitive approaches, which finally provides a context abstraction that is more precise than both *kobjH* and *lsrvH*.

5 Mixing Contexts for Enhanced Precision

As seen in Section 4, there is no "most precise" existing context abstraction, and each abstraction might miss out on some optimization opportunities covered by another. Further, many of the context abstractions, though theoretically very precise, do not scale to large programs and hence are not practical (for example, an infinite-length call-strings based analysis with heap cloning). However, among the existing context abstractions, the LSRV approaches (with *lsrvH* being more precise than plain *lsrv*) offer the best practical precision in terms of scalability to large programs. On the other hand, the object-sensitive approaches, though not as efficient as the LSRV approaches, cover some optimization opportunities that might get missed by the latter. Based on the above observations, given a context abstraction c_1, in order to cover the optimization opportunities missed by c_1 but covered by another context abstraction c_2, we propose a simple, yet novel idea: *mix* c_1 *and* c_2 to derive a new context abstraction $c_{1\bullet 2}$, such that at each call-site, $c_{1\bullet 2}$ creates a new context if either of the component contexts of c_1 and c_2 has changed. An advantage of such a mixing scheme is that the resultant context abstraction covers the optimization opportunities of both the component abstractions. That is, by construction, $c_{1\bullet 2}$ is more precise than both c_1 and c_2.

5.1 Mixing LSRV Contexts and Object-sensitivity

While mixing two context abstractions to come up with a new one, one can visualize two intuitive requirements. First, the component abstractions should include non-overlapping optimization opportunities. In order to satisfy this criterion, we propose to mix one abstraction from the call-site-sensitive approaches and another from the object-sensitive approaches. Second, as mixing two context abstractions is likely to increase the cost of performing the resultant context-sensitive analysis, it is crucial that the chosen abstractions be as scalable as possible, that is, with a good practical precision.

Candidate 1. As discussed in Section 4, among the call-site-sensitive variants, the LSRV approaches offer the best practical precision. This is because of the identification of relevance and the notion of level-summarization, and the splitting of the overall approach into three stages: pre-analysis, main-analysis, and post-analysis [30]. The pre-analysis identifies which portions of the callers' heaps may be affected by each method (expressed as the *access-depth* of each parameter). This information is then used to create and compare smaller contexts during the main-analysis, and to defer the analysis of methods that do not affect their callers (zero access-depth for all the parameters). The deferred methods are analyzed later in a post-analysis pass. Motivated by the high scalability (along with precision) of LSRV contexts, and the further increase in precision with the addition of heap cloning (as discussed in Section 3.2), we pick *lsrvH* as our mixing candidate from the call-site-sensitive abstractions.

Candidate 2. Among the object-sensitive approaches, as shown by our examples in Section 4 and by prior works [12, 16, 26], *k*-level object-sensitivity with one level of heap cloning (that is, *kobjH*) offers the best precision-scalability trade-off for typical pointer analyses. Hence we choose *kobjH* as our second pick for the mix.

We now discuss how we mix *lsrvH* and *kobjH* to derive a new context abstraction called *lsrvkobjH*.

5.1.1 Computation

When we reach a call-site for a method *m* that has been previously analyzed in a context *c*, we need to decide whether *m*

needs to be re-analyzed. This is done by computing the current context (based on the context abstraction being used), and comparing it with the existing context c. More the number of existing contexts for m, more number of comparisons may need to be performed – a costly operation, specially for heap analyses where the contexts are typically larger than non-heap analyses. Similarly, more the number of contexts for m, more is the number of times m needs to be analyzed – again a costly operation. The computation of the *lsrv* part of our mixed context is already optimized based on per-parameter access-depths. For *lsrvkobjH*, we further optimize the context computation, using the following insight:

Insight 1. *If a method m and its callees do not store any new object to a non-primitive field of the receiver, then a change in the receiver object will not contribute towards changing the summary of m or its callees.*

In order to find whether a method or its callees satisfy Insight 1, we modify the multi-stage analysis approach (consisting of a pre-, a main- and a post-analysis) already in place for LSRV contexts [30], as discussed next.

Pre-analysis. The pre-analysis pass of LSRV processes each statement of each function (in the bottom up order of the call-graph). We modify the handling of three statements: (i) store, (ii) call , and (iii) endProcedure statements in the existing pre-analysis, by maintaining a special field `receiverStore` (initialized to `unknown`) for each method. At the end of the pre-analysis, if the field `receiverStore` for a method m is set to true, it indicates that m satisfies Insight 1.

procStorePre. At a store statement $a.f = b$, where f is a non-primitive field, if an object O_b pointed to by b does not flow from the caller (that is, it is allocated either in the current method m or one of its callees), then we set the field m.`receiverStore` to `true`.

procCallPre. At a call statement $a.n()$ in the current method m, if we find that n.`receiverStore` was `true` (indicating that the callee n satisfies Insight 1), we set the field `receiverStore` of each object O_a pointed-to by a to `true`. We handle recursion conservatively: if the information about a method n is not known, then we conservatively set the `receiverStore` field of all the objects in the points-to set of a to `true`.

procEndProcedurePre. At the end of processing a method m, if m.`receiverStore` still has the `unknown` value, then m.`receiverStore` is set to `false`.

Main-analysis. In order to keep the main analysis (which uses the mixed *lsrvkobjH* context abstraction) efficient, we conditionally check the *kobj* context for a method m, only if the `receiverStore` field for m is set. Thus, at each call-site for a method m, we either construct and compare the full *lsrvkobjH* context or only the *lsrvH* context, based on the satisfiability of Insight 1 for m.

Post-analysis. Similar to LSRV contexts [30], in the main-analysis, we defer the analysis of methods that do not access their callers' heap, and analyze them in a post-analysis pass.

5.1.2 Precision

As *lsrvkobjH* contexts are the result of mixing *lsrvH* and the *kobjH*, and by the definition of mixing the precision of *lsrvkobjH* is higher than each of its constituents. For example, using *lsrvkobjH* for the code in Figure 6, we would re-analyze the constructor of class `W` at line 18, be able to distinguish the objects pointed-to by `w1.f` and `w2.f`, and hence resolve both the calls to the method `m` as monomorphic.

5.1.3 Termination

As discussed in Section 3.2, analyses using *lsrvH* contexts are guaranteed to terminate. Further, if k is a finite positive integer, k-object-sensitive analyses are guaranteed to terminate. The number of *lsrvkobjH* contexts that could be created depends not only on the number of contexts that could be created individually by the *lsrvH* and the *kobj* approaches, but also on the newer combinations of contexts that could be created due to the new dataflow facts generated. However, the newer contexts that could be created are still bound by the number of combinations possible due to the cross product of the number of allocation sites, the number of contexts in *kobj* and the ones in *lsrvH*, which is still finite. As a result, analyses that use *lsrvkobjH* contexts are also guaranteed to terminate. Note that in practice, due to the additional precision achievable using *lsrvkobjH* contexts, the contexts created do not reach the worst case proportions.

5.2 The Updated Precision View

Recall the precision view shown in Figure 1 that compared the theoretical precision of the various context abstractions prior to this manuscript. We had two non-overlapping blocks for context abstractions: one each for call-site- and object-sensitivity. Further, for both the non-overlapping blocks, heap cloning was known to improve the precision of each of the base approaches.

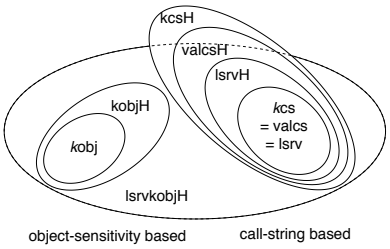


Figure 7. Updated relative precision of context abstractions.

Based on the proposals and insights in this manuscript, Figure 7 shows the updated view of the precision of the various context abstractions that are now in picture. We can observe that with heap cloning, the precisions of the value-contexts (*valcsH*) and the LSRV (*lsrvH*) approaches do not match the precision of call-site-sensitivity (*kcsH*). Importantly, we now have another context abstraction *lsrvkobjH*, which is more precise than the heap-cloning enabled versions of value contexts (*valcsH*), LSRV contexts (*lsrvH*), as well as object-sensitivity (*kobjH*). To the best of our knowledge, *lsrvkobjH* is the first context abstraction

that combines the precision of the call-site- and the object-sensitive approaches, in a scalable manner (see Section 6). In the world view of Figure 7, *kcsH* and *valcsH* may still lead to improved precision in some cases (compared to *lsrvkobjH*). Similarly, a mix of *valcsH* and *kobjH* will have more precision than *lsrvkobjH*. However, none of these abstractions may terminate in general. Thus, in practice (as validated in Section 6), our newly proposed abstraction (*lsrvkobjH*) currently offers the *best practical precision*.

Note that Figure 7 is not complete, and could show a few more relationships. For example, we may have a mix *lsrvkobj* (without heap cloning), which will be more precise than both *lsrv* and *kobj*. We focus only on the heap-cloning enabled variants, as they have been used more popularly in recent literature [13, 14, 22] due to their higher precision.

6 Implementation and Evaluation

We have implemented the various context abstractions to perform control-flow analysis [21, 24] of Java programs, in Soot [32] version 2.5.0. The experiments were performed on a 64-core 512GB AMD Abu Dhabi system with OpenJDK 8 as the installed JVM. We have evaluated our techniques on seven benchmarks from the DaCapo-9.12 suite [2], and three benchmarks from Section C (with large applications) of the JGF suite [5]; these benchmarks are listed in Figure 8, along with some static characteristics. We used the extremely helpful tool TamiFlex [3] to resolve reflective calls in the original DaCapo benchmarks, so that they could be analyzed by Soot. The benchmarks excluded from the DaCapo suite are the ones which either could not be translated by TamiFlex, or could not be analyzed by Soot (using OpenJDK8) after the TamiFlex pass. The number of classes in the benchmarks (excluding the JDK library) varied from 13 (small programs) to 1.6K (large applications). Figure 8 also shows the number of JDK classes referred by each benchmark (gives the total number of analyzed classes), computed using the call-graph generated by our default call-graph tool Spark [11].

We compare four different context abstractions: (i) *lsrv*: the base implementation of LSRV contexts [30]; (ii) *lsrvH*: LSRV contexts with heap cloning; (iii) *kobjH*: one-level object-sensitivity with heap cloning, based on *full object-sensitivity* as defined by Smaragdakis et al. [26]; and (iv) *lsrvkobjH*: a mix of (ii) and (iii), as discussed in Section 5.1. The *lsrvkobjH* and *kobjH* implementations are parameterized with the value of k. For our evaluation, we set $k = 1$, as it is well-known [13, 14, 22] that the next more precise object-sensitive analysis 2*objH*, does not directly scale for many large benchmarks. We have also run the evaluation *valcsH*, but it did not terminate for any of the benchmarks under consideration and hence not reported. This behavior of *valcsH* is in line with the observations of Thakur and Nandivada [30] that the base *valcs* itself does not terminate for any of these benchmarks under consideration, within the cutoff time (of 3 hours).

We now present an evaluation to compare the various context abstractions, with respect to their relative empirical precision (Section 6.1) and their scalability (Section 6.2).

6.1 Precision of Various Context Abstractions

As many of the context abstractions under consideration are in principle incomparable, we compare their precision, similar to existing works [12–14, 26, 30], by normalizing the numbers for two precision-indicating clients over all the contexts: (i) *#polyCall*: the number of call-sites that could not be resolved to a single method (Figure 9a); and (ii) *#callEdge*: the number of edges in the on-the-fly call-graph (Figure 9b). For both the clients, a lower value indicates higher precision.

Figure 9a compares the numbers for #polyCall. We can see that the number of polymorphic calls reduces, albeit marginally, from *lsrv* to *lsrvH*. The number of polymorphic calls in case of 1*objH* is the highest among the four reported versions, which indicates that overall, the number of optimization opportunities enabled (over all contexts) by the LSRV approaches is better than that by the object-sensitive approach. Importantly, it can be seen that even though 1*objH* leads to more number of polymorphic calls (compared to *lsrvH*), *lsrv*1*objH* leads to improved precision (better than both *lsrvH* and 1*objH*) – this attests to the non-overlapping cases reported by 1*objH* and *lsrvH* and the theoretically superior precision of *lsrv*1*objH* over both. Across all the benchmarks, we can see that *lsrv*1*objH* leads to the least number of polymorphic calls (5.73% less than *lsrv*, 5.65% less than *lsrvH*, and 12.23% less than 1*objH* for cases where 1*objH* terminates, on average), which establishes it as a superior alternative over the other context abstractions.

Figure 9b compares the numbers for #callEdge for each context abstraction. We see that similar to #polyCall, the normalized #callEdge reduces, though by a small amount, from *lsrv* to *lsrvH*. The number of call-graph edges is the highest for 1*objH*, and the reduction is the highest for *lsrv*1*objH*: 0.41%, 0.35% and 4.97% fewer call-edges compared to *lsrv*, *lsrvh* and 1*objH* (where 1*objH* terminates), respectively.

Overall, among the context abstractions considered, we see that *lsrv*1*objH*, as also expected from its theoretical model (Section 5), offers the best precision in terms of generated optimization opportunities. We next compare the scalability of the various approaches to find out how does *lsrv*1*objH* fare compared to the rest of the abstractions.

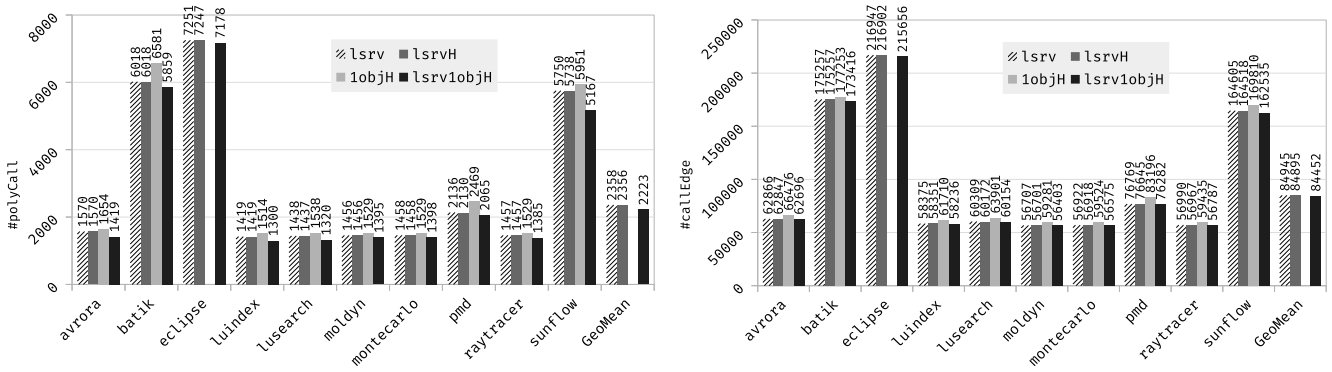
6.2 Scalability of Various Context Abstractions

We now evaluate the scalability of the various context abstractions, by comparing them with the base *lsrv* approach, in terms of two parameters: the analysis time and the peak memory requirement. Columns 4 and 5 of Figure 8 show the corresponding numbers for *lsrv* [30].

Analysis time. Columns 6-8 of Figure 8 show the additional analysis time (in percentage) taken by the various context abstractions over *lsrv*. We can see that though the

1	2	3	4	5	6	7	8	9	10	11
Benchmark	**#classes**		**time**	**memory**	**% increase in time**			**% increase in memory**		
	application	**jdk***	*lsrv* (s)	*lsrv* (GB)	*lsrvh*	1*objh*	*lsrv*1*objh*	*lsrvh*	1*objh*	*lsrv*1*objh*
avrora	527	1588	55	11.2	24.5	646.2	26.5	6.3	19.7	7.9
batik	1038	3700	946	64.4	167.3	709.8	129.2	31.1	-21.2	-6.1
eclipse	1608	2589	988	49.1	224.0	-	225.9	121.8	-	-
luindex	199	1485	46	11.1	38.5	653.9	23.3	-2.3	14.3	-9.0
lusearch	198	1481	57	11.2	44.6	672.1	61.4	2.3	30.6	5.8
moldyn	697	1607	53	10.5	85.3	448.5	83.6	18.7	13.4	20.1
montecarlo	225	3509	53	9.4	58.3	474.3	67.2	0.7	67.2	0.7
pmd	13	1555	108	13.3	44.8	587.3	2263.2	25.0	21.8	309.6
raytracer	19	1555	53	9.6	62.1	452.6	68.7	0.0	11.9	0.0
sunflow	19	1555	684	52.9	40.8	1097.1	53.2	8.7	34.1	42.9
GeoMean	174	1928	130	17.7	62.3	-	93.6	17.5	-	-

Figure 8. Scalability results for various context abstractions as % increase over *lsrv* numbers (obtained using the techniques of Thakur and Nandivada [30]). *JDK classes computed using Spark's [11] call graph.



(a) **(b)**

Figure 9. Normalized number of (a) polymorphic calls; (b) call-graph edges. Lower the better for both.

time taken by *lsrvH* is in most cases higher than *lsrv* (62.3% higher, on average), the analysis nevertheless terminates for all the benchmarks in a reasonable time (on average under 4 minutes, and maximum 53 minutes for ECLIPSE).

Figure 8 also shows that the 1*objH* approach takes a much higher time than the LSRV approaches for most of the benchmarks, and does not terminate (in 3 hours – our cutoff) for the largest benchmark ECLIPSE. The *lsrv*1*objH* approach, on the other hand, terminates for all the benchmarks, albeit taking 93.6% higher time than *lsrv*; this overhead is the cost we pay for the improved precision (see Section 6.1). Interestingly, for most benchmarks (except PMD, where object-sensitivity adds a significant overhead), *lsrv*1*objH* takes lesser time than even 1*objH*. We attribute this pattern to two facts: (i) a higher precision (as shown in Section 6.1) leading to the analysis of a fewer number of methods; and (ii) skipping of the creation of redundant contexts resulting from Insight 1.

Memory consumption. Columns 9-11 of Figure 8 show the peak memory consumption for the four approaches, as percentage increase over the *lsrv* approach. For small benchmarks, we observe that the memory requirements of all the approaches, though higher than *lsrv*, remain in the range of about 16-32 GB. For larger benchmarks (ECLIPSE and SUNFLOW), the memory requirements are significantly higher than *lsrv* (absolute numbers were between 100-200 GB). For ECLIPSE, though *lsrv*1*objH* terminated normally (see column 4), it timed out when we enabled memory measurement. For BATIK and LUINDEX, we notice a slight drop in the peak memory requirement compared to *lsrv*; we attribute it to the flattening of the overall memory requirement by the GC passes. We also observe that for many benchmarks, the 1*objH* approach takes lesser memory than the LSRV approaches; however, as seen in Section 6.1, the number of optimization opportunities generated by the LSRV approaches is higher.

We highlight an interesting point here: the benchmarks for which *lsrv*1*objH* leads to improvement/deterioration in the analysis time (compared to the other variants), there is a corresponding comparative increase/decrease in the memory consumption. This is in line with the above discussion of improvement in analysis time due to precision.

Effect of Insight 1. In order to study the impact of Insight 1 on *lsrv*1*objH*, Figure 10 shows the analysis time and

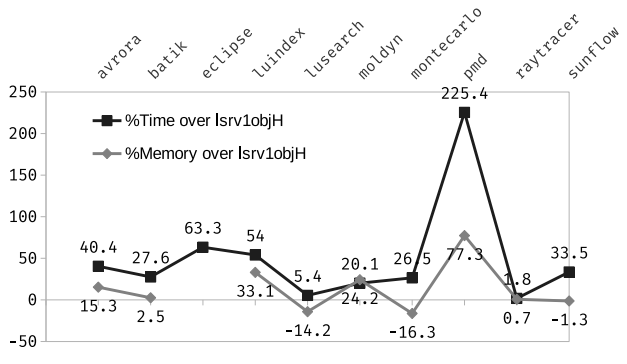


Figure 10. Percentage overhead without Insight 1.

the peak memory consumption when it was not used. Here, a method was re-analyzed if any of the *lsrvH* or the 1*objH* contexts changed, without considering whether that method (or its callees) store(s) an object to the receiver or not. As we can see, the analysis time without the optimization resulting from Insight 1 increases significantly; on average, 41.6% over *lsrv*1*objH* (though the precision of both the approaches is the same). A similar trend can be seen for the memory requirement as well: the average peak memory consumption without Insight 1 is mostly higher (and sometimes slightly less or similar) than *lsrv*1*objH*. These statistics show two things. First, the impact of Insight 1 for applying object-sensitivity selectively is significant. Second, the remaining difference between 1*objH* and *lsrv*1*objH* (which adds *lsrvH* to, and uses Insight 1 over, 1*objH*) is the result of the additional precision gained by the latter (see Section 6.1).

Overall, among the considered variants, the novel mix of *lsrvkobjH* offers the best practical precision: scales quite well and generates more number of optimization opportunities than the existing approaches that scale. Further, the approaches proposed in this paper to scale the various context abstractions can be used as it is, and augmented with more pre-analysis based techniques (for example, the ones proposed by Li et al. [13]), to scale more existing context abstractions as well as to come up with even newer ones.

7 Related Work

There have been several recent works that scale context abstractions for analyzing Java programs. Tan et al. [29] scale context-sensitive analyses for call-graph construction by merging type-consistent objects identified using equivalent automata. Li et al. [14] use a pre-analysis to gain meta-information about methods, and then select among the different variants of object-sensitivity for each method (with a small dip in the precision). Some prior works [13, 15] use a pre-analysis to approximate the methods/objects that follow some insightful patterns, and apply context-sensitivity partially to the identified methods/objects. Rama et al. [22] use slicing to incrementally enhance the precision of k-limited object-sensitive analysis, and increase the value of k for objects that are identified as precision-critical. On the other hand, in this paper, we propose the idea of mixing different context abstractions, with non-overlapping optimization opportunities, in an efficient manner.

There have been more works that scale the "main" context-sensitive analysis using a pre-analysis. Oh et al. [19] estimate the impact of context-sensitivity on different methods for a given set of queries in a pre-analysis, and use the computed information during the main analysis to reduce the precision on methods that might not benefit from the enhanced precision. Tan et al. [28] use a pre-analysis to identify and eliminate redundant objects from object-sensitive analyses and reduce the number of effective contexts. Recently, Karkare [8] uses a pre-analysis to mark variables whose shape cannot be refined, and skips them in a following precise (slow) pass. Prior works [25, 27] use a pre-analysis to identify code portions that do not affect the analysis results or may degrade scalability, and analyze them conservatively. Thakur and Nandivada [30] use a pre-analysis to identify the portions of the callers' heap that are accessed by each method, and use them to compute a new scalable context abstraction called LSRV contexts. A variation of LSRV contexts is also used by the PYE framework [31] to write efficient static partial analyses. In this paper, we enhance the precision of LSRV contexts using heap cloning [18] (to obtain *lsrvH*), mix *lsrvH* with object-sensitivity to obtain a newer, more precise context abstraction (called *lsrvkobjH*), and use a pre-analysis to scale the proposed context abstraction.

In this paper, we have focussed our discussion on the context abstractions that are representative of the recent and the classical advancements in the area. There are some more context abstractions in the literature that could also be of academic interest, such as the cartesian product algorithm [1] (may generate a large number of spurious contexts and be highly inefficient, especially in the context of heap analysis) and type-sensitivity [26] (scales well, but trades-off precision). We leave the corresponding empirical study, comparing type-sensitivity with the LSRV approaches (with and without heap cloning), as a future work.

The idea of combining abstractions has been explored earlier in various domains. Codish et al. [4] combine multiple analyses by performing them together over a combined domain. Kastrinis et al. [9] observe that combining call-site- and object-sensitivity is infeasible due to its cost, and apply them selectively based on program features (such as static versus virtual method calls). On the other hand, we show that by choosing suitable representatives of the two broad variants, we are able to scale the combined abstraction for the whole program.

It has been observed [12, 16] that in general, for object-oriented programs, object-sensitivity is expected to give a

better trade-off between precision an scalability than call-site-sensitivity. We note that with our approach, it is possible to get the benefits of both object-sensitive and call-site-sensitive words in a scalable manner. Further, it should be straightforward to augment our approaches with recent approaches [13, 14, 27] that adapt existing context abstractions based on various program features, where our pre-analysis can be used to compute the heuristics involved therein. It would also be interesting to compare our approaches to scale context-sensitivity with approaches used in other domains such as logic programming [6, 17].

There have been prior works that explain existing context abstractions and present new insights about when to use a particular abstraction. Smaragdakis et al. [26] clarify the original definition of object-sensitivity proposed by [16], and propose type-sensitivity as a closer sibling that scales better than object-sensitivity. Kanvar and Khedker [7] present a survey of existing heap abstractions, including for context-sensitivity, and assert the importance of the abstraction used towards the precision and the scalability of a given analysis. Lhoták and Hendren [12] evaluate call-site- and object-sensitive abstractions using a binary decision diagrams based framework. On the other hand, in this paper, we present a detailed comparison of various context abstractions, including recent ones whose placement in the area was not well understood, and also come up with a more precise abstraction that for the first time leads to a scalable connection between the call-site- and the object-sensitive approaches. Overall, we believe that our work significantly expands the understanding of the relative advantages and disadvantages of various classical as well as recent context abstractions, and thus advances the state of the art in this space.

8 Conclusion

In this paper, we first presented a detailed study of the various abstractions used for performing context-sensitive analyses. We then expanded the space of available choices by adding heap cloning to the recently introduced abstraction of LSRV contexts. Based on the differences between the theoretical and realistically achievable precisions of the various abstractions, we then proposed mixing of contexts as a way forward. To demonstrate the idea, we utilized the scalability of the LSRV approaches and augmented them with k-level object-sensitivity to propose a new context abstraction called *lsrvkobjH* that includes the benefits of the two approaches. We evaluated the abstractions under discussion by using them to perform control-flow analysis of Java programs. The evaluation showed that among the approaches under consideration, *lsrvkobjH* generated more optimization opportunities, while also scaling to large benchmarks.

Acknowledgments

V. Krishna Nandivada is partially supported by an IBM CAS grant (1101) and SERB CRG grant (CRG/2018/002488).

References

[1] Ole Agesen. 1995. The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP '95)*. Springer-Verlag, Berlin, Heidelberg, 2–26. http://dl.acm.org/citation.cfm?id=646153.679533

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

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

[4] Michael Codish, Anne Mulkers, Maurice Bruynooghe, Maria García de la Banda, and Manuel Hermenegildo. 1995. Improving Abstract Interpretations by Combining Domains. *ACM Trans. Program. Lang. Syst.* 17, 1 (Jan. 1995), 28–44. https://doi.org/10.1145/200994.200998

[5] Charles Daly, Jane Horgan, James Power, and John Waldron. 2001. Platform Independent Dynamic Java Virtual Machine Analysis: The Java Grande Forum Benchmark Suite. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande (JGI '01)*. ACM, New York, NY, USA, 106–115. https://doi.org/10.1145/376656.376826

[6] Isabel Garcia-Contreras, José F. Morales, and Manuel V. Hermenegildo. 2018. Towards Incremental and Modular Context-Sensitive Analysis. In *Technical Communications of the 34th International Conference on Logic Programming, ICLP 2018, July 14-17, 2018, Oxford, United Kingdom*. 7:1–7:2. https://doi.org/10.4230/OASIcs.ICLP.2018.7

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

[8] Amey Karkare. 2018. TwAS: Two-stage Shape Analysis for Speed and Precision. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC '18)*. ACM, New York, NY, USA, 1857–1864. https://doi.org/10.1145/3167132.3167330

[9] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid Context-Sensitivity for Points-to Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. Association for Computing Machinery, New York, NY, USA, 423–434. https://doi.org/10.1145/2491956.2462191

[10] Uday P. Khedker and Bageshri Karkare. 2008. Efficiency, Precision, Simplicity, and Generality in Interprocedural Data Flow Analysis: Resurrecting the Classical Call Strings Method. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction (CC'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 213–228. http://dl.acm.org/citation.cfm?id=1788374.1788394

[11] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction (CC'03)*. Springer-Verlag, Berlin, Heidelberg, 153–169. http://dl.acm.org/citation.cfm?id=1765931.1765948

[12] Ondřej Lhoták and Laurie Hendren. 2008. Evaluating the Benefits of Context-sensitive Points-to Analysis Using a BDD-based Implementation. *ACM Trans. Softw. Eng. Methodol.* 18, 1, Article 3 (Oct. 2008), 53 pages. https://doi.org/10.1145/1391984.1391987

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

[14] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-first Pointer Analysis with Self-tuning Context-sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 129–140. https://doi.org/10.1145/3236024.3236041

[15] Jingbo Lu and Jingling Xue. 2019. Precision-preserving Yet Fast Object-sensitive Pointer Analysis with Partial Context Sensitivity. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 148 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360574

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

[17] Kalyan Muthukumar and Manuel V. Hermenegildo. 1990. *Deriving a Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs.* Technical Report. Informatica, Madrid, Spain. http://oa.upm.es/15292/

[18] Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. 2004. Importance of Heap Specialization in Pointer Analysis. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '04)*. ACM, New York, NY, USA, 43–48. https://doi.org/10.1145/996821.996836

[19] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective Context-sensitivity Guided by Impact Pre-analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 475–484. https://doi.org/10.1145/2594291.2594318

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

[21] Jens Palsberg and Michael I. Schwartzbach. 1991. Object-oriented Type Inference. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '91)*. ACM, New York, NY, USA, 146–161. https://doi.org/10.1145/117954.117965

[22] 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 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276512

[23] M Sharir and A Pnueli. 1978. *Two approaches to interprocedural data flow analysis.* New York Univ. Comput. Sci. Dept., New York, NY. https://cds.cern.ch/record/120118

[24] Olin Grigsby Shivers. 1991. *Control-flow Analysis of Higher-order Languages or Taming Lambda.* Ph.D. Dissertation. Pittsburgh, PA, USA. UMI Order No. GAX91-26964.

[25] Yannis Smaragdakis, George Balatsouras, and George Kastrinis. 2013. Set-based Pre-processing for Points-to Analysis. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 253–270. https://doi.org/10.1145/2509136.2509524

[26] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 17–30. https://doi.org/10.1145/1926385.1926390

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

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

[29] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and Precise Points-to Analysis: Modeling the Heap by Merging Equivalent Automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 278–291. https://doi.org/10.1145/3062341.3062360

[30] Manas Thakur and V. Krishna Nandivada. 2019. Compare Less, Defer More: Scaling Value-contexts Based Whole-program Heap Analyses. In *Proceedings of the 28th International Conference on Compiler Construction (CC 2019)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/3302516.3307359

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

[32] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 13–23. http://dl.acm.org/citation.cfm?id=781995.782008