Making *k*-Object-Sensitive Pointer Analysis More Precise with Still *k*-Limiting

Tian Tan[1], Yue Li[1], and Jingling Xue[1,2]

[1] School of Computer Science and Engineering, UNSW Australia
[2] Advanced Innovation Center for Imaging Technology, CNU, China

Abstract. Object-sensitivity is regarded as arguably the best context abstraction for pointer analysis in object-oriented languages. However, a k-object-sensitive pointer analysis, which uses a sequence of k allocation sites (as k context elements) to represent a calling context of a method call, may end up using some context elements redundantly without inducing a finer partition of the space of (concrete) calling contexts for the method call. In this paper, we introduce Bean, a general approach for improving the precision of any k-object-sensitive analysis, denoted k-*obj*, by still using a k-limiting context abstraction. The novelty is to identify allocation sites that are redundant context elements in k-*obj* from an Object Allocation Graph (OAG), which is built based on a pre-analysis (e.g., a context-insensitive Andersen's analysis) performed initially on a program and then avoid them in the subsequent k-object-sensitive analysis for the program. Bean is generally more precise than k-*obj*, with a precision that is guaranteed to be as good as k-*obj* in the worst case. We have implemented Bean as an open-source tool and applied it to refine two state-of-the-art whole-program pointer analyses in Doop. For two representative clients (*may-alias* and *may-fail-cast*) evaluated on a set of nine large Java programs from the DaCapo benchmark suite, Bean has succeeded in making both analyses more precise for all these benchmarks under each client at only small increases in analysis cost.

1 Introduction

Pointer analysis, as an enabling technology, plays a key role in a wide range of client applications, including bug detection [3, 25, 35, 34], security analysis [1, 13], compiler optimisation [6, 33], and program understanding [12]. Two major dimensions of pointer analysis precision are flow-sensitivity and context-sensitivity. For C/C++ programs, flow-sensitivity is needed by many clients [11, 16, 37, 32]. For object-oriented programs, e.g., Java programs, however, context-sensitivity is known to deliver trackable and useful precision [17, 19–21, 28–30], in general.

There are two general approaches to achieving context-sensitivity for object-oriented programs, call-site-sensitivity (k-CFA) [27] and object-sensitivity [23, 24, 29] (among others). A k-CFA analysis represents a calling context of a method call by using a sequence of k call sites (i.e., k labels with each denoting a call site). In contrast, a k-object-sensitive analysis uses k object allocation sites (i.e., k labels with each denoting a `new` statement) as context elements.

Among all the context abstractions (including k-CFA) proposed, object-sensitivity is regarded as arguably the best for pointer analysis in object-oriented languages [14, 17, 29]. This can be seen from its widespread adoption in a number of pointer analysis frameworks for Java, such as DOOP [7, 4], CHORD [5] and WALA [36]. In addition, object-sensitivity has also been embraced by many other program analysis tasks, including typestate verification [9, 38], data race detection [25], information flow analysis [1, 10, 22], and program slicing [21].

Despite its success, a k-object-sensitive pointer analysis, which uses a sequence of k allocation sites (as k context elements) to represent a calling context of a method call, may end up using some context elements redundantly in the sense that these redundant context elements fail to induce a finer partition of the space of (concrete) calling contexts for the method call. As a result, many opportunities for making further precision improvements are missed.

In this paper, we introduce BEAN, a general approach for improving the precision of a k-object-sensitive pointer analysis, denoted k-*obj*, for Java, by avoiding redundant context elements in k-*obj* while still maintaining a k-limiting context abstraction. The novelty lies in identifying redundant context elements by solving a graph problem on an OAG (Object Allocation Graph), which is built based on a pre-analysis (e.g., a context-insensitive Andersen's analysis) performed initially on a program, and then avoid them in the subsequent k-object-sensitive analysis. By construction, BEAN is generally more precise than k-*obj*, with a precision that is guaranteed to be as good as k-*obj* in the worst case.

We have implemented BEAN and applied it to refine two state-of-the-art (whole-program) pointer analyses, *2obj+h* and *S-2obj+h* [14], provided in DOOP [7], resulting in two BEAN-directed pointer analyses, *B-2obj+h* and *B-S-2obj+h*, respectively. We have considered *may-alias* and *may-fail-cast*, two representative clients used elsewhere [8, 29, 30] for measuring the precision of a pointer analysis on a set of nine large Java programs from the DaCapo benchmark suite. Our results show that *B-2obj+h* (*B-S-2obj+h*) is more precise than *2obj+h* (*S-2obj+h*) for every evaluated benchmark under each client, at some small increases in analysis cost.

This paper presents and validates a new idea on improving the precision of object-sensitive pointer analysis by exploiting an object allocation graph. Considering the broad applications of object-sensitivity in analysing Java programs, we expect more clients to benefit from the BEAN approach, in practice. Specifically, this paper makes the following contributions:

- We introduce a new approach, BEAN, for improving the precision of any k-object-sensitive pointer analysis, k-*obj*, for Java, by avoiding its redundant context elements while maintaining still a k-limiting context abstraction.
- We introduce a new kind of graph, called an OAG (object allocation graph), constructed from a pre-analysis for the program, as a general mechanism to identify redundant context elements used in k-*obj*.
- We have implemented BEAN as a soon-to-be released open-source tool, which is expected to work well with various object-sensitive analyses for Java.

– We have applied Bean to refine two state-of-the-art object-sensitive pointer analyses for Java. Bean improves their precision for two representative clients on a set of nine Java programs in DaCapo at small time increases.

2 Motivation

When analysing Java programs, there are two types of context, a *method context* for local variables and a *heap context* for object fields. In *k-obj*, a *k*-object-sensitive analysis [24, 29], a method context is a sequence of k allocation sites and a heap context is typically a sequence of $k-1$ allocation sites. Given an allocation site at label ℓ, ℓ is also referred to as an abstract object for the site.

Currently, *k-obj*, where $k = 2$, represents a 2-object-sensitive analysis with a 1-context-sensitive heap (with respect to allocation sites), denoted *2obj+h* [14], which usually achieves the best tradeoff between precision and scalability and has thus been widely adopted in pointer analysis for Java [8, 21, 29]. In *2obj+h*, a heap context for an abstract object ℓ is a receiver object of the method that made the allocation of ℓ (known as an *allocator object*). A method context for a method call is a receiver object of the method plus its allocator object.

Below we examine the presence of redundant context elements in *2obj+h*, with two examples, one for method contexts and one for heap contexts. This serves to motivate the Bean approach proposed for avoiding such redundancy.

2.1 Redundant Elements in Method Contexts

We use an example in Fig. 1 to illustrate how *2obj+h* analyses it imprecisely due to its use of a redundant context element in method contexts and how Bean avoids the imprecision by avoiding this redundancy. We consider a *may-alias* client that queries for the alias relation between variables `v1` and `v2`.

In Fig. 1(a), we identify the six allocation sites by their labels given in their end-of-line comments (in green), i.e., `A/1`, `A/2`, `O/1`, `O/2`, `B/1`, and `C/1`.

In Fig. 1(b), we give the context-sensitive call graph computed by *2obj+h*, where each method is analysed separately for each different calling context, denoted by [...] (in red). `C.identify()` has two concrete calling contexts but analysed only once under [`B/1,C/1`]. We can see that `B/1` is redundant (relative to `C/1`) since adding `B/1` to [`C/1`] fails to separate the two concrete calling contexts. As a result, variables `v1` and `v2` are made to point to both `O/1` and `O/2` at the same time, causing *may-alias* to report a spurious alias. During any program execution, `v1` and `v2` can only point to `O/1` and `O/2`, respectively.

In Fig. 1(c), we give the context-sensitive call graph computed by Bean, where `C.identify()` is now analysed separately under two different contexts, [`A/1, C/1`] and [`A/2, C/1`]. Due to the improved precision, `v1` (`v2`) now points to `O/1` (`O/2`) only, causing *may-alias* to conclude that both are no longer aliases.

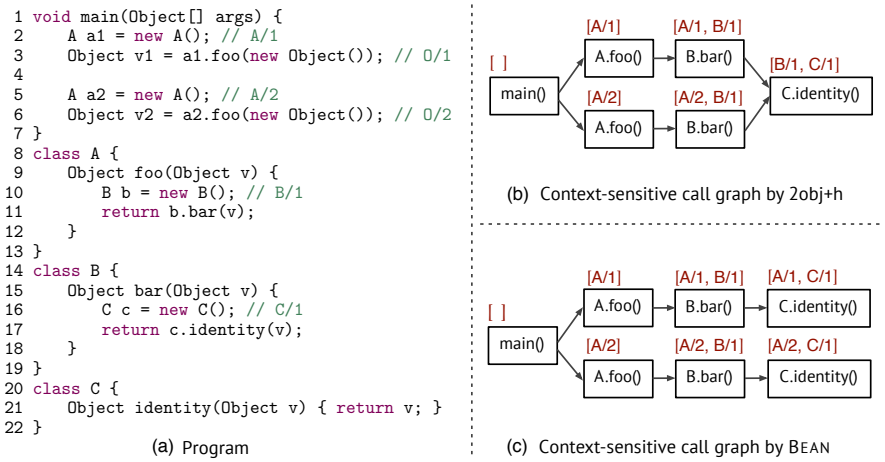


Fig. 1: Method contexts for *2obj+h* and BEAN.

2.2 Redundant Elements in Heap Contexts

We now use an example in Fig. 2 to illustrate how *2obj+h* analyses it imprecisely due to its use of a redundant element in heap contexts and how BEAN avoids the imprecision by avoiding this redundancy. Our *may-alias* client now issues an alias query for variables `emp1` and `emp2`. In Fig. 2(a), we identify again its six allocation sites by their labels given at their end-of-line comments (in green).

Fig. 2(b) shows the context-sensitive field points-to graph computed by *2obj+h*, where each node represents an abstract heap object created under the corresponding context, denoted [...], (in red), and each edge represents a field points-to relation with the corresponding field name being labeled on the edge. An array object is analysed with its elements collapsed to one pseudo-field, denoted `arr`. Hence, `x[i] = y` (`y = x[i]`) is handled as `x.arr = y` (`y = x.arr`).

In this example, two companies, `Co/1` and `Co/2`, maintain their employee information by using two different `ArrayLists`, with each implemented internally by a distinct array of type `Object[]` at line 22. However, *2obj+h* has modelled the two array objects imprecisely by using one abstract object `Obj[]/1` under [`AL/1`]. Note that `AL/1` is redundant since adding it to [ ] makes no difference to the handling of `Obj[]/1`. As a result, `emp1` and `emp2` will both point to `Emp/1` and `Emp/2`, causing *may-alias* to regard both as aliases conservatively.

Fig. 2(c) shows the context-sensitive field points-to graph computed by BEAN. This time, the `Object[]` arrays used by two companies `Co/1` and `Co/2` are distinguished under two distinct heap contexts [`Co/1`] and [`Co/2`]. As a result, our *may-alias* client will no longer report `emp1` and `emp2` to be aliases.

2.3 Discussion

As illustrated above, *k-obj* selects blindly a sequence of k-most-recent allocation sites as a context. To analyse large-scale software scalably, k is small, which is 2

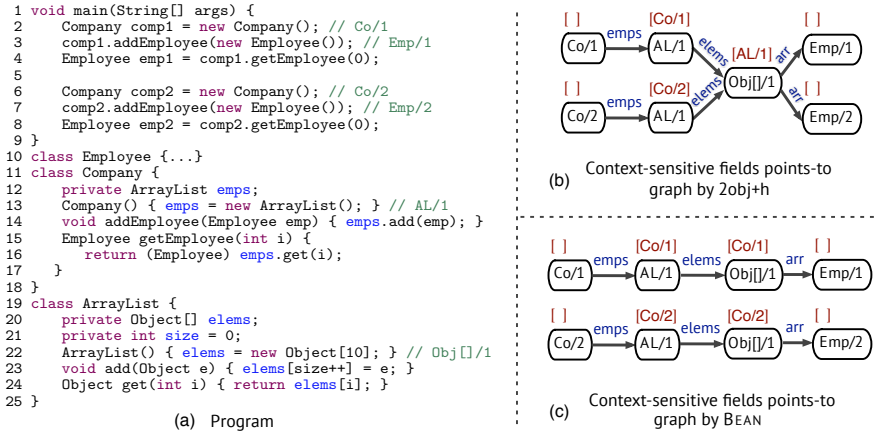


Fig. 2: Heap contexts for *2obj+h* and BEAN.

for a method context and 1 for a heap context in *2obj+h*. Therefore, redundant context elements, such as `B/1` in [`B/1,C/1`] in Fig. 1(b) and `AL/1` in [`AL/1`] in Fig. 2(b), should be avoided since they waste precious space in a context yet contribute nothing in separating the concrete calling contexts for a call site.

This paper aims to address this problem in *k-obj* by excluding redundant elements from its contexts so that their limited context positions can be more profitably exploited to achieve better precision, as shown in Figs. 1(c) and 2(c).

3 Methodology

We introduce a new approach, BEAN, as illustrated in Fig. 3, to improving the precision of a *k*-object-sensitive pointer analysis, *k-obj*. The basic idea is to refine *k-obj* by avoiding its redundant context elements while maintaining still a *k*-limiting context abstraction. An element e in a context c for a call or allocation site is *redundant* if c with e removed does not change the context represented by c. For example, `B/1` in [`B/1,C/1`] in Fig. 1(b) and `AL/1` in [`AL/1`] in Fig. 2(b) are redundant.

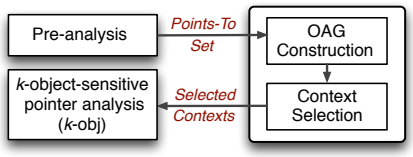


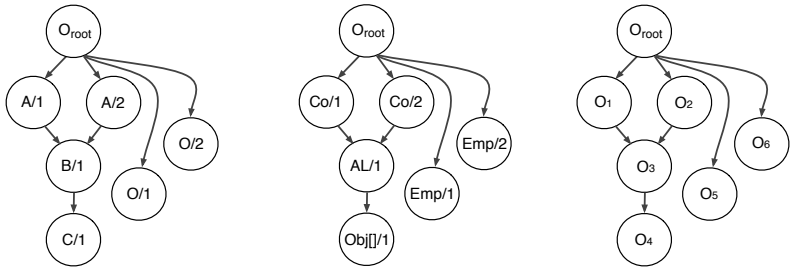
Fig. 3: Overview of BEAN.

BEAN proceeds in two stages. In Stage 1, we aim to identify redundant context elements used in *k-obj*. To do so, we first perform usually a fast but imprecise pre-analysis, e.g., a context-insensitive Andersen's pointer analysis on a program to obtain its points-to information. Based on the points-to information

discovered, we construct an object allocation graph (OAG) to capture the object allocation relations in k-*obj*. Subsequently, we traverse the OAG to select method and heap contexts by avoiding redundant context elements that would otherwise be used by k-*obj*. In Stage 2, we refine k-*obj* by avoiding its redundant context elements. Essentially, we perform a k-object-sensitive analysis in the normal way, by using the contexts selected in the first stage, instead.

3.1 Object Allocation Graph

The OAG of a program is a directed graph, $G = (N, E)$. A node $\ell \in N$ represents a label of an (object) allocation site in the program. An edge $\ell_1 \rightarrow \ell_2 \in E$ represents an object allocation relation. As G is context-insensitive, a label $\ell \in G$ is also interchangeably referred to (in the literature) as the (unique) abstract heap object that models all the concrete objects created at the allocation site ℓ. Given this, $\ell_1 \rightarrow \ell_2$ signifies that ℓ_1 is the the receiver object of the method that made the allocation of ℓ_2. Therefore, ℓ_1 is called an *allocator object* of ℓ_2 [29].



(a) Fig. 1 (b) Fig. 2 (c) Example 2

Fig. 4: The OAGs for the two motivating programs in Figs. 1 and 2.

Figure 4 gives the OAGs for the two programs in Figs. 1 and 2, which are deliberately designed to be isomorphic. In Fig. 4(a), `A/1` and `A/2` are two allocators of `B/1`. In Fig. 4(b), `AL/1` is an allocator of `Obj[]/1`. Some objects, e.g., those created in `main()` or static initialisers, have no allocators. For convenience, we assume the existence of a dummy node, $\texttt{O}_{root}$, so that every object has at least one allocator. The isomorphic OAG in Fig. 4(c) will be referred to in Example 2.

The concept of allocator object captures the essence of object sensitivity. By definition [24, 29], a context for an allocation site ℓ, i.e., an abstract object ℓ consists of its allocator object (ℓ'), the allocator object of ℓ', and so on. The OAG provides a new perspective for object sensitivity, since a context for an object ℓ is simply a path from $\texttt{O}_{root}$ to ℓ. As a result, the problem of selecting contexts for ℓ can be recast as one of solving a problem of distinguishing different paths from $\texttt{O}_{root}$ to ℓ. Traditionally, a k-object-sensitive analysis selects blindly a suffix of a path from $\texttt{O}_{root}$ to ℓ with length k.

3.2 Context Selection

Given an object ℓ in G, BEAN selects its contexts in G as sequences of its direct or indirect allocators that are useful to distinguish different paths from $\texttt{O}_{root}$ to ℓ while avoiding redundant ones that would otherwise be used in *k-obj*. The key insight is that in many cases, it is unnecessary to use all nodes of a path to distinguish the path from the other paths leading to the same node. In contrast, *k-obj* is not equipped with G and thus has to select blindly a suffix of each such path as a context, resulting in many redundant context elements being used.

Method Contexts Figure 1 compares the method contexts used by *2obj+h* and BEAN for the first example given. As shown in Figure 1(b), *2obj+h* analyses `C.identity()` under one context [`B/1,C/1`], where `B/1` is redundant, without being able to separate its two concrete calling contexts. In contrast, BEAN avoids using `B/1` by examining the OAG of this example in Fig. 4(a). There are two paths from $\texttt{O}_{root}$ to `C/1`: $\texttt{O}_{root} \rightarrow$ `A/1` $\rightarrow$ `B/1` $\rightarrow$ `C/1` and $\texttt{O}_{root} \rightarrow$ `A/2` $\rightarrow$ `B/1` $\rightarrow$ `C/1`. Note that *2obj+h* has selected a suffix of the two paths, `B/1` $\rightarrow$ `C/1`, which happens to represent the same context [`B/1,C/1`] for `C.identity()`. BEAN distinguishes these two paths by ignoring the redundant node `B/1`, thereby settling with the method contexts shown in Figure 1(c). As a result, `C.identity()` is now analysed under two different contexts [`A/1,C/1`] and [`A/2,C/1`] more precisely.

Heap Contexts Figure 2 compares the heap contexts used by *2obj+h* and BEAN for the second example given. As shown in Figure 2(b), *2obj+h* fails to separate the two array objects created at the allocation site `Obj[]/1` for two companies `Co/1` and `Co/2` by using one context [`AL/1`], where `AL/1` is redundant. In contrast, BEAN avoids using `AL/1` by examining the OAG of this example in Fig. 4(b). There are two paths from $\texttt{O}_{root}$ to `Obj[]/1`: $\texttt{O}_{root} \rightarrow$ `Co/1` $\rightarrow$ `AL/1` $\rightarrow$ `Obj[]/1` and $\texttt{O}_{root} \rightarrow$ `Co/2` $\rightarrow$ `AL/1` $\rightarrow$ `Obj[]/1`. Note that *2obj+h* has selected a suffix of the two paths, `AL/1` $\rightarrow$ `Obj[]/1`, which happens to represent the same heap context [`AL/1`] for `Obj[]/1`. BEAN distinguishes these two paths by ignoring the redundant node `AL/1`, thereby settling with the heap contexts shown in Figure 2(c). As a result, the two array objects created at `Obj[]/1` are distinguished under two different contexts [`Co/1`] and [`Co/2`] more precisely.

3.3 Discussion

BEAN, as shown in Fig. 3, is designed to be a general-purpose technique for refining *k-obj* with three design goals, under the condition that its pre-analysis is sound. First, as the pre-analysis is usually less precise than *k-obj*, the OAG constructed for the program may contain some object allocation relations that are not visible in *k-obj*. Therefore, BEAN is not expected to be optimal in the sense that it can avoid all redundant context elements in *k-obj*. Second, if the pre-analysis is more precise than *k-obj* (e.g., in some parts of the program), then the OAG may miss some object allocation relations that are visible in *k-obj*. This allows BEAN to avoid using context elements that are redundant with respect to the pre-analysis but not *k-obj*, making the resulting analysis even more precise.

Finally, Bean is expected to be more precise than k-*obj* in general, with a precision that is guaranteed to be as good as k-*obj* in the worst case.

4 Formalism

Without loss of generality, we formalise Bean as a k-object-sensitive pointer analysis with a $(k-1)$-context-sensitive heap (with respect to allocation sites), as in [14]. Thus, the depth of its method (heap) contexts is k $(k-1)$.

4.1 Notations

We will use the notations in Fig. 5. The top section lists the domains used. As we focus on object-sensitive analysis, a context is a sequence of objects. The middle section gives the five key relations used. *pt* and *fpt* store the analysis results: $pt(c, x)$ represents the points-to set of variable x under context c and $fpt(c, o_i, f)$ represents the points-to set of the field f of an abstract object o_i under context c. *mtdCtxSelector* and *heapCtxSelector* choose the method contexts for method calls and heap contexts for allocation sites, respectively. *contextsOf* maps a method to its contexts. The last section defines the OAG used for a program: $o_i \rightarrow o_j$ means that o_i is an allocator object of o_j, i.e., the receiver object of the method that made the allocation of o_j.

variable	$x, y \in \mathbb{V}$
heap object	$o_i, o_j \in \mathbb{H}$
method	$m \in \mathbb{M}$
field	$f \in \mathbb{F}$
context	$c \in \mathbb{C} = \mathbb{H}^0 \cup \mathbb{H}^1 \cup \mathbb{H}^2 ...$
pt	$\mathbb{C} \times \mathbb{V} \rightarrow \mathcal{P}(\mathbb{C} \times \mathbb{H})$
fpt	$\mathbb{C} \times \mathbb{H} \times \mathbb{F} \rightarrow \mathcal{P}(\mathbb{C} \times \mathbb{H})$
mtdCtxSelector	$\mathbb{C} \times \mathbb{H} \rightarrow \mathbb{C}$
heapCtxSelector	$\mathbb{C} \times \mathbb{H} \rightarrow \mathbb{C}$
contextsOf	$\mathbb{M} \rightarrow \mathcal{P}(\mathbb{C})$
OAG	$G = (N, E)$
node	$o_i, o_j \in N \subseteq \mathbb{H}$
edge	$o_i \rightarrow o_j \in E \subseteq N \times N$

Fig. 5: Notations.

4.2 Object Allocation Graph

Figure 6 gives the rules for building the OAG, $G = (N, E)$, for a program, based on the points-to sets computed by a pre-analysis, which may or may not be context-sensitive. As G is (currently) context-insensitive, the context information that appears in a points-to set (if any) is simply ignored. [OAG-Node] and [OAG-DummyNode] build N. [OAG-Edge] and [OAG-DummyEdge] build E.

By [OAG-Node], we add to N all the pointed-to target objects found during the pre-analysis. By [OAG-DummyNode], we add a dummy node o_{root} to N.

By [OAG-Edge], we add to E an edge $o_i \rightarrow o_j$ if o_i is an allocator object of o_j. Here, m_{this}, where m is the name of a method, represents the `this` variable of method m, which points to the receiver object of method m. By [OAG-DummyEdge], we add an edge from o_{root} to every object o_i without any incoming edge yet, to indicate that o_{root} is now a pseudo allocator object of o_i. Note that an object allocated in `main()` or a static initialiser does not have an allocator object. Due to o_{root}, every object has at least one allocator object.

$$\frac{\langle \_, o_i \rangle \in pt(\_, \_)}{o_i \in N}\ \text{[OAG-Node]} \qquad \frac{}{o_{root} \in N}\ \text{[OAG-DummyNode]}$$

$$\frac{\langle \_, o_i \rangle \in pt(\_, m_{this}) \quad m \in \mathbb{M} \quad o_j \textbf{ is allocated in } m}{o_i \to o_j \in E}\ \text{[OAG-Edge]}$$

$$\frac{o_i \in N \quad o_i \neq o_{root} \quad o_i \textbf{ does not have any incoming edge}}{o_{root} \to o_i \in E}\ \text{[OAG-DummyEdge]}$$

Fig. 6: Rules for building the OAG, $G = (N, E)$, for a program based on a pre-analysis.

Example 1. Figure 4 gives the OAGs for the two programs in Figs. 1 and 2. For reasons of symmetry, let us apply the rules in Fig. 6 to build the OAG in Fig. 4(a) only. Suppose we perform a context-insensitive Andersen's pointer analysis as the pre-analysis on the program in Fig. 1. The points-to sets are: $pt(\texttt{v1}) = pt(\texttt{v2}) = \{\texttt{O/1}, \texttt{O/2}\}$, $pt(\texttt{a1}) = \{\texttt{A/1}\}$, $pt(\texttt{a2}) = \{\texttt{A/2}\}$, $pt(\texttt{b}) = \{\texttt{B/1}\}$, and $pt(\texttt{c}) = \{\texttt{C/1}\}$. By [OAG-Node] and [OAG-DummyNode], $N = \{o_{root}, \texttt{A/1}, \texttt{A/2}, \texttt{B/1}, \texttt{C/1}, \texttt{O/1}, \texttt{O/2}\}$. By [OAG-Edge], we add $\texttt{A/1} \to \texttt{B/1}$, $\texttt{A/2} \to \texttt{B/1}$ and $\texttt{B/1} \to \texttt{C/1}$, since `B/1` is allocated in `foo()` with the receiver objects being `A/1` and `A/2` and `C/1` is allocated in `bar()` on the receiver object `B/1`. By [OAG-DummyEdge], we add $o_{root} \to \texttt{A/1}$, $o_{root} \to \texttt{A/2}$, $o_{root} \to \texttt{O/1}$ and $o_{root} \to \texttt{O/2}$. □

Due to recursion, an OAG may have cycles including self-loops. This means that an abstract heap object may be a direct or indirect allocator object of another heap object, and conversely (with both being possibly the same).

4.3 Context Selection

Figure 7 establishes some basic relations in an OAG, $G = (N, E)$, with possibly cycles. By [Reach-Reflexive] and [Reach-Transitive], we speak of graph reachability in the standard manner. In [Confluence], $\curlyvee o_i$ identifies a conventional confluence point. In [Divergence], $o_i \prec o_t$ states that o_i is a divergence point, with at least two outgoing paths reaching o_t, implying that either o_t is a confluence point or at least one confluence point exists earlier on the two paths.

$$\frac{o_i \in N}{o_i \rightsquigarrow o_i}\ \text{[Reach-Reflexive]} \qquad \frac{o_i \to o_j \in E \quad o_j \rightsquigarrow o_k}{o_i \rightsquigarrow o_k}\ \text{[Reach-Transitive]}$$

$$\frac{o_j \to o_i \in E \quad o_k \to o_i \in E \quad o_j \neq o_k}{\curlyvee o_i}\ \text{[Confluence]}$$

$$\frac{o_i \to o_j \in E \quad o_i \to o_k \in E \quad o_j \neq o_k \quad o_j \rightsquigarrow o_t \quad o_k \rightsquigarrow o_t}{o_i \prec o_t}\ \text{[Divergence]}$$

Fig. 7: Rules for basic relations in an OAG, $G = (N, E)$.

$$\frac{o_t \in N \quad o_{root} \to o_i \in E \quad o_i \rightsquigarrow o_t}{o_i^t : \langle o_{root} \prec o_t,\ [\,]\rangle \quad \textbf{if } o_i = o_t \textbf{ then } heapCtxSelector([\,], o_i) = [\,]} \text{ [HCTX-INIT]}$$

$$\frac{o_j^t : \langle rep, c\rangle \quad o_j \to o_i \in E \quad o_i \rightsquigarrow o_t \quad o_j \neq o_t}{o_i^t : \langle rep', c'\rangle \quad \begin{cases} rep' = \textbf{true}, \quad c' = c & \textbf{if } \neg rep \wedge o_j \prec o_t \ \textcircled{1} \\ rep' = o_j \prec o_t,\ c' = c +\!\!+ o_j & \textbf{if } rep \wedge \curlyvee o_i \quad \textcircled{2} \\ rep' = rep, \quad c' = c & \textbf{otherwise} \end{cases} \textbf{if } o_i = o_t \textbf{ then } heapCtxSelector(c +\!\!+ o_j, o_i) = c'} \text{ [HCTX-DIV]}$$

$$\frac{o_j^t : \langle rep, c\rangle \quad o_j \to o_i \in E \quad o_i \rightsquigarrow o_t \quad o_j = o_t}{o_i^t : \langle rep', c'\rangle \quad \begin{cases} rep' = \textbf{true},\ c' = c +\!\!+ o_j, & \textbf{if } rep \wedge \curlyvee o_i \ \textcircled{3} \\ rep' = \textbf{true},\ c' = c, & \textbf{otherwise} \end{cases} \textbf{if } o_i = o_t \textbf{ then } heapCtxSelector(c +\!\!+ o_j, o_i) = c'} \text{ [HCTX-CYC]}$$

$$\frac{heapCtxSelector(\_, o_i) = c \quad c' = c +\!\!+ o_i}{mtdCtxSelector(c, o_i) = c'} \text{ [MCTX]}$$

Fig. 8: Rules for context selection in an OAG, G=(N, E). $+\!\!+$ is a concatenation operator.

Figure 8 gives the rules for computing two context selectors, *heapCtxSelector* and *mtdCtxSelector*, used in refining an object-sensitive pointer analysis in Fig. 11. In $heapCtxSelector(c, o_i) = c'$, c denotes an (abstract calling) context of the method that made the allocation of object o_i and c' is the heap context selected for o_i when o_i is allocated in the method with context c. In $mtdCtxSelector(c, o_i) = c'$, c denotes a heap context of object o_i, and c' is the method context selected for the method whose receiver object is o_i under its heap context c.

For *k-obj* [24, 29], both context selectors are simple. In the case of full-object-sensitivity, we have $heapCtxSelector([o_1, ..., o_{n-1}], o_n) = [o_1, ..., o_{n-1}]$ and $mtdCtxSelector([o_1, ..., o_{n-1}], o_n) = [o_1, ..., o_n]$ for every path from o_{root} to a node o_n in the OAG, $o_{root} \to o_1 \to ... \to o_{n-1} \to o_n$. For a k-object-sensitive analysis with a $(k-1)$-context-sensitive heap, $heapCtxSelector([o_{n-k}, ..., o_{n-1}], o_n) = [o_{n-k+1}, ..., o_{n-1}]$ and $mtdCtxSelector([o_{n-k+1}, ..., o_{n-1}], o_n) = [o_{n-k+1}, ..., o_n]$. Essentially, a suffix of length of k is selected from $o_{root} \to o_1 \to ... \to o_{n-1} \to o_n$, resulting in potentially many redundant context elements to be used blindly.

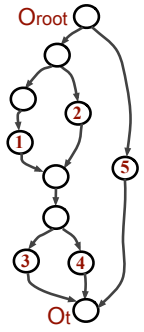


Fig. 9: An OAG.

Let us first use an OAG in Fig. 9 to explain how we avoid redundant context elements selected by *k-obj*. The set of contexts for a given node, denoted o_t, can be seen as the set of paths reaching o_t from o_{root}. Instead of using all the nodes on a path to distinguish it from the other four, we use only the five *representative nodes*, labeled by 1 – 5, and identify the five paths uniquely as ① → ③, ① → ④, ② → ③, ② → ④, and ⑤. The other six nodes are redundant with respect to o_t. The rules in Fig. 8 are used to identify such representative nodes

(on the paths from a divergence node to a confluence node) and compute the set of contexts for o_t.

In Fig. 8, the first three rules select heap contexts and the last rule selects method contexts based on the heap contexts selected. The first three rules traverse the OAG from o_{root} and select heap contexts for a node o_t. Meanwhile, each rule also records at o_i, which reaches o_t, a set of pairs of the form $o_i^t : \langle rep, c\rangle$. For a pair $o_i^t : \langle rep, c\rangle$, c is a heap context of o_i that uniquely represents a particular path from o_{root} to o_i. In addition, rep is a boolean flag considered for determining the suitability of o_i as a representative node, i.e., context element for o_t *under* c (i.e., for the path c leading to o_i). There are two cases. If $rep = \textbf{false}$, then o_i is redundant for o_t. If $rep = \textbf{true}$, then o_i is potentially a representative node (i.e., context element) for o_t. $c +\!\!+ o$ returns the concatenation of c and o.

Specifically, for the first three rules on heap contexts, [Hctx-Init] bootstraps heap context selection, [Hctx-Cyc] handles the special case when o_t is in a cycle such that $o_j = o_t$, and [Hctx-Div] handles the remaining cases. In [Mctx], the contexts for a method are selected based on its receiver objects and the heap contexts of these receiver objects computed by the first three rules. Thus, removing redundant elements from heap contexts benefits method contexts directly.

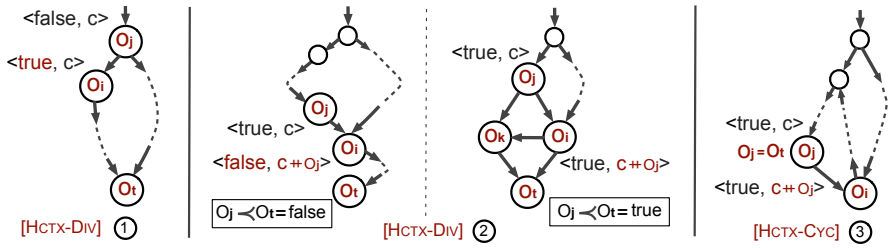


Fig. 10: Three Cases marked for [Hctx-Div] and [Hctx-Cyc] in Fig. 8.

Figure 10 illustrates the four non-trivial cases marked in Fig. 8, i.e., ①, ② (split into two sub-cases), and ③. In ①, o_i appears on a divergent path from o_j leading to o_t, o_i^t's rep' is set to **true** to mark o_i as a potential context element for o_t. In ②, there are two sub-cases: $\neg o_j \prec o_t$ and $o_j \prec o_t$. In both cases, o_j is in a branch (since o_j^t's rep is **true**) and o_i is a confluence node (since $\curlyvee o_i$ holds). Thus, o_j is included as a context element for o_t. In the case of $\neg o_j \prec o_t$, o_i is redundant for o_t under c. In the case of $o_j \prec o_t$, the paths to o_t diverge at o_j. Thus, o_i can be potentially a context element to distinguish the paths from o_j to o_t via o_i. If o_i is ignored, the two paths $o_j \rightarrow o_k \rightarrow o_t$ and $o_j \rightarrow o_i \rightarrow o_k \rightarrow o_t$ as shown cannot be distinguished. In [Hctx-Cyc], its two cases are identically handled as the last two cases in [Hctx-Cyc], except that [Hctx-Cyc] always sets o_i^t's rep' to **true**. If [Hctx-Cyc] is applicable, o_t must appear in a cycle such that $o_j = o_t$. Then, any successor of o_t may be a representative node to be used to distinguish the paths leading to o_t via the cycle. Thus, o_i^t's rep' is set to **true**. The first case in [Hctx-Cyc], marked as ③ in Fig. 8, is illustrated in Fig. 10.

To enforce k-limiting in the rules given in Fig 8, we simply make every method context $c +\!\!+ o_i$ k-bounded and every heap context $c +\!\!+ o_j$ $(k-1)$-bounded.

Example 2. For the two programs illustrated in Figs. 1 and 2, BEAN is more precise than *2obj+h* (with $k = 2$) in handling the method and heap contexts of o_4, shown in their isomorphic OAG in Fig. 4(c). We give some relevant derivations for o_i^t, with $t = 4$, only. By [HCTX-INIT], we obtain $o_1^4 : (\textbf{true}, [\,])$ and $o_2^4 : (\textbf{true}, [\,])$. By [HCTX-DIV], we obtain $o_3^4 : (\textbf{false}, [\texttt{o1}])$, $o_3^4 : (\textbf{false}, [\texttt{o2}])$, $o_4^4 : (\textbf{false}, [\texttt{o1}])$ and $o_4^4 : (\textbf{false}, [\texttt{o2}])$. Thus, $heapCtxSelector([\texttt{o1,o3}], o4) = [\texttt{o1}]$ and $heapCtxSelector([\texttt{o2,o3}], o4) = [\texttt{o2}]$. By [MCTX], $mtdCtxSelector([\texttt{o1}], o4) = [\texttt{o1,o4}]$, and $mtdCtxSelector([\texttt{o2}], o4) = [\texttt{o2,o4}]$. For *2obj+h*, the contexts selected for o_4 are $heapCtxSelector([\texttt{o1,o3}], o4) = [\texttt{o3}]$, $heapCtxSelector([\texttt{o2,o3}], o4) = [\texttt{o3}]$ and $mtdCtxSelector([\texttt{o3}], o4) = [\texttt{o3,o4}]$. As result, BEAN can successfully separate the two concrete calling contexts for o_4 and the two o_4 objects created in the two contexts but *2obj+h* fails to do this. □

4.4 Object-Sensitive Pointer Analysis

Figure 11 gives a formulation of a k-object-sensitive pointer analysis that selects its contexts in terms of *mtdCtxSelector* and *heapCtxSelector* to avoid redundant context elements that would otherwise be used in *k-obj*. In addition to this fundamental difference, all the rules are standard, as expected.

In [NEW], o_i identifies uniquely the abstract object created as an instance of T at allocation site i. In [ASSIGN], a copy assignment between two local variables is dealt with. In [LOAD] and [STORE], object field accesses are handled.

In [CALL], the function $dispatch(o_i, g)$ is used to resolve the virtual dispatch of method g on the receiver object o_i to be method m'. As in Fig. 6, we continue to use m'_{this} to represent the `this` variable of method m'. Following [31], we assume that m' has the k formal parameters $m'_{p1}, ..., m'_{pk}$ other than m'_{this} and that a pseudo-variable m'_{ret} is used to hold the return value of m'.

Compared to *k-obj*, BEAN avoids its redundant context elements in [NEW] and [CALL]. In [NEW], *heapCtxSelector* (by [HCTX-INIT], [HCTX-DIV] and [HCTX-CYC]) is used to select the contexts for object allocation. In [CALL], *mtdCtxSelector* (by [MCTX]) is used to select the contexts for method invocation.

4.5 Properties

Theorem 1. *Under full-context-sensitivity (i.e., when $k = \infty$),* BEAN *is as precise as the traditional k-object-sensitive pointer analysis (k-obj).*

Proof Sketch. The set of contexts for any given abstract object, say o_t, is the set P_t of its paths reaching o_t from o_{root} in the OAG of the program. Let R_t be the set of representative nodes, i.e., context elements identified by BEAN for o_t. We argue that R_t is sufficient to distinguish all the paths in P_t (as shown in Fig. 9).

For the four rules given in Fig. 8, we only need to consider the first three for selecting heap contexts as the last one for method contexts depends on the first three. [HCTX-INIT] performs the initialisation for the successor nodes of o_{root}.

m: the containing method for each statement being analysed

$$\frac{i: x = new\ T() \quad c \in contextsOf(m) \quad c' = heapCtxSelector(c, o_i)}{\langle c', o_i \rangle \in pt(c, x)} \text{ [New]}$$

$$\frac{x = y \quad c \in contextsOf(m)}{pt(c, y) \subseteq pt(c, x)} \text{ [Assign]}$$

$$\frac{x = y.f \quad c \in contextsOf(m) \quad \langle c', o_i \rangle \in pt(c, y)}{fpt(c', o_i, f) \subseteq pt(c, x)} \text{ [Load]}$$

$$\frac{x.f = y \quad c \in contextsOf(m) \quad \langle c', o_i \rangle \in pt(c, x)}{pt(c, y) \subseteq fpt(c', o_i, f)} \text{ [Store]}$$

$$\frac{\begin{array}{c} x = y.g(arg_1, ..., arg_n) \quad c \in contextsOf(m) \quad \langle c', o_i \rangle \in pt(c, y) \\ m' = dispatch(o_i, g) \quad c'' = mtdCtxSelector(c', o_i) \end{array}}{\begin{array}{c} c'' \in contextsOf(m') \quad \langle c', o_i \rangle \in pt(c'', m'_{this}) \\ \forall\ 1 \leq k \leq n : pt(c, arg_k) \subseteq pt(c'', m'_{pk}) \quad pt(c'', m'_{ret}) \subseteq pt(c, x) \end{array}} \text{ [Call]}$$

Fig. 11: Rules for pointer analysis.

[Hctx-Div] handles all the situations except the special one when o_t is in a cycle such that $o_t = o_j$. [Hctx-Div] has three cases. In the first case, marked ① (Fig. 10), our graph reachability analysis concludes conservatively whether it has processed a divergence node or not during the graph traversal. In the second case, marked ② (Fig. 10), o_i is a confluence node. By adding o_j to c in $c +\!\!+ o_j$, we ensure that for each path p from o_i's corresponding divergence node to o_i traversed earlier, at least one representative node that is able to represent p, i.e., o_j, is always selected, i.e., to R_t. In cases ① and ②, as all the paths from o_{root} to o_t are traversed, all divergence and confluence nodes are handled. The third case simply propagates the recorded information across the edge $o_j \rightarrow o_i$.

[Hctx-Cyc] applies only when o_t is in a cycle such that $o_t = o_j$. Its two cases are identical to the last two cases in [Hctx-Div] except o_i^t's rep' is always set to **true**. This ensures all the paths via the cycle can be distinguished correctly. In the case, marked ③ and illustrated in Fig. 10, o_j is selected, i.e., added to R_t.

Thus, R_t is sufficient to distinguish the paths in P_t. Hence, the theorem. □

Theorem 2. *For any fixed context depth k,* Bean *is as precise as the traditional k-object-sensitive pointer analysis (k-obj) in the worst case.*

Proof Sketch. This follows from the fact that, for a fixed k, based on Theorem 1, Bean will eliminate some redundant context elements in a sequence of k-most-recent allocation sites in general or nothing at all in the worst case. Thus, Bean may be more precise than (by distinguishing more contexts for a call or allocation site) or has the same precision as *k-obj* (by using the same contexts). □

5 Evaluation

We have implemented BEAN as a standalone tool for performing OAG construction (Fig. 6) and context selection (Fig. 8), as shown in Fig. 3, in Java. To demonstrate the relevance of BEAN to pointer analysis, we have integrated BEAN with DOOP [7], a state-of-the-art context-sensitive pointer analysis framework for Java. In our experiments, the pre-analysis for a program is performed by using a context-insensitive Andersen's pointer analysis provided in DOOP. To apply BEAN to refine an existing object-sensitive analysis written in Datalog from DOOP, it is only necessary to modify some Datalog rules in DOOP to adopt the contexts selected by *heapCtxSelector* and *mtdCtxSelector* in BEAN (Fig. 8).

Our entire BEAN framework will be released as open-source software at http://www.cse.unsw.edu.au/∼corg/bean.

In our evaluation, we attempt to answer the following two research questions:

RQ1. Can BEAN improve the precision of an object-sensitive pointer analysis at slightly increased cost to enable a client to answer its queries more precisely?
RQ2. Does BEAN make any difference for a real-world application?

To address RQ1, we apply BEAN to refine two state-of-the-art whole-program object-sensitive pointer analyses, *2obj+h* and *S-2obj+h*, the top two most precise yet scalable solutions provided in DOOP [7, 14], resulting in two BEAN-directed analyses, *B-2obj+h* and *B-S-2obj+h*, respectively. Altogether, we will compare the following five context-sensitive pointer analyses:

- *2cs+h*: 2-call-site-sensitive analysis [7]
- *2obj+h*: 2-object-sensitive analysis with 1-context-sensitive heap [7]
- *B-2obj+h*: the BEAN-directed version of *2obj+h*
- *S-2obj+h*: selective hybrids of 2 object-sensitive analysis proposed in [7, 14]
- *B-S-2obj+h*: the BEAN-directed version of *S-2obj+h*

Note that *2obj+h* is discussed in Section 2. *S-2obj+h* is a selective 2-object-sensitive with 1-context-sensitive heap hybrid analysis [14], which applies call-site-sensitivity to static call sites and *2obj+h* to virtual call sites. For *S-2obj+h*, BEAN proceeds by refining its object-sensitive part of the analysis, demonstrating its generality in improving the precision of both pure and hybrid object-sensitive analyses. For comparison purposes, we have included *2cs+h* to demonstrate the superiority of object-sensitivity over call-site-sensitivity.

We have considered *may-alias* and *may-fail-cast*, two representative clients used elsewhere [8, 29, 30] for measuring the precision of pointer analysis. The *may-alias* client queries whether two variables may point to the same object or not. The *may-fail-cast* client identifies the type casts that may fail at run time.

To address RQ2, we show how BEAN can enable *may-alias* and *may-fail-cast* to answer alias queries more precisely for `java.util.HashSet`. This container from the Java library is extensively used in real-world Java applications.

5.1 Experimental Setting

All the five pointer analyses evaluated are written in terms of Datalog rules in the DOOP framework [4]. Our evaluation setting uses the LogicBlox Datalog engine (v3.9.0), on an Xeon E5-2650 2GHz machine with 64GB of RAM.

We use all the Java programs in the DaCapo benchmark suite (2006-10-MR2) [2] except `hsqldb` and `jython`, because all the four object-sensitive analyses, cannot finish analysing each of the two in a time budget of 5 hours. All these benchmarks are analysed together with a large Java library, JDK 1.6.0_45.

DOOP handles native code (in terms of summaries) and (explicit and implicit) exceptions [4]. As for reflection, we leverage SOLAR [20] by adopting its string inference to resolve reflective calls but turning off its other inference mechanisms that may require manual annotations. We have also enabled DOOP to merge some objects, e.g., reflection-irrelevant string constants, in order to speed up each analysis without affecting its precision noticeably, as in [7, 14].

When analysing a program, by either a pre-analysis or any of the five pointer analyses evaluated, its native code, exceptions and reflective code are all handled in exactly the same way. Even if some parts of the program are unanalysed, we can still speak of the soundness of all these analyses with respect to the part of the program visible to the pre-analysis. Thus, Theorems 1 and 2 still hold.

5.2 RQ1: Precision and Performance Measurements

Table 1 compare the precision and performance results for the five analyses.

Precision We measure the precision of a pointer analysis in term of the number of may-alias variable pairs reported by *may-alias* and the number of may-fail-casts reported by *may-fail-cast*. For the *may-alias* client, the obvious aliases (e.g., due to a direct assignment) have been filtered out, following [8]. The more precise a pointer analysis is, the smaller these two numbers will be.

Let us consider *may-alias* first. *B-2obj+h* improves the precision of *2obj+h* for all the nine benchmarks, ranging from 6.2% for `antlr` to 16.9% for `xalan`, with an average of 10.0%. In addition, *B-S-2obj+h* is also more precise than *S-2obj+h* for all the nine benchmarks, ranging from 3.7% for `antlr` to 30.0% for `xalan`, with an average of 8.8%. Note that the set of non-aliased variable pairs reported under *2obj+h* (*S-2obj+h*) is a strict subset of the set of non-aliased variable pairs reported under *B-2obj+h* (*B-S-2obj+h*), validating practically the validity of Theorem 2, i.e., the fact that BEAN is always no less precise than the object-sensitive analysis improved upon. Finally, *2obj+h*, *S-2obj+h*, *B-2obj+h* and *B-S-2obj+h* are all substantially more precise than *2cs+h*, indicating the superiority of object-sensitivity over call-site-sensitivity.

Let us now move to *may-fail-cast*. Again, *B-2obj+h* improves the precision of *2obj+h* for all the nine benchmarks, ranging from 5.4% for `fop` to 11.2% for `luindex`, with an average of 8.4%. In addition, *B-S-2obj+h* is also more precise than *S-2obj+h* for all the nine benchmarks, ranging from 6.7% for `fop` to 15.6% for `luindex`, with an average of 10.8%. Note that the casts that are shown to

Table 1: Precision and performance results for all the five analyses. The two precision metrics shown are the number of variable pairs that may be aliases generated by *may-alias* ("may-alias pairs") and the number of casts that cannot be statically proved to be safe by *may-fail-cast* ("may-fail casts"). In both cases, *smaller is better*. One performance metric used is the analysis time for a program.

		2cs+h	*2obj+h*	*B-2obj+h*	*S-2obj+h*	*B-S-2obj+h*
xalan	may-alias pairs	25,245,307	6,196,945	**5,146,694**	5,652,610	**3,958,998**
	may-fail casts	1154	711	**653**	608	**550**
	analysis time (secs)	1400	8653	11450	1150	1376
chart	may-alias pairs	43,124,320	4,189,805	**3,593,584**	3,485,082	**3,117,825**
	may-fail casts	2026	1064	**979**	923	**844**
	analysis time (secs)	3682	630	1322	1145	1814
eclipse	may-alias pairs	20,979,544	5,029,492	**4,617,883**	4,636,675	**4,346,306**
	may-fail casts	1096	722	**655**	615	**551**
	analysis time (secs)	1076	119	175	119	188
fop	may-alias pairs	38,496,078	10,548,491	**9,870,507**	9,613,363	**9,173,539**
	may-fail casts	1618	1198	**1133**	1038	**973**
	analysis time (secs)	3054	796	1478	961	1566
luindex	may-alias pairs	10,486,363	2,190,854	**1,949,134**	1,820,992	**1,705,415**
	may-fail casts	794	493	**438**	408	**353**
	analysis time (secs)	650	90	140	88	145
pmd	may-alias pairs	13,134,083	2,868,130	**2,598,100**	2,457,457	**2,328,304**
	may-fail casts	1216	845	**787**	756	**698**
	analysis time (secs)	816	131	191	132	193
antlr	may-alias pairs	16,445,862	5,082,371	**4,768,233**	4,586,707	**4,419,166**
	may-fail casts	995	610	**551**	525	**466**
	analysis time (secs)	808	109	162	105	163
lusearch	may-alias pairs	11,788,332	2,251,064	**2,010,780**	1,886,967	**1,771,280**
	may-fail casts	874	504	**450**	412	**358**
	analysis time (secs)	668	94	153	91	155
bloat	may-alias pairs	43,408,294	12,532,334	**11,608,822**	12,155,175	**11,374,583**
	may-fail casts	1944	1401	**1311**	1316	**1226**
	analysis time (secs)	10679	4508	4770	4460	4724

be safe under *2obj+h* (*S-2obj+h*) are also shown to be safe by *B-2obj+h* (*B-S-2obj+h*), verifying Theorem 2 again. For this second client, *2obj+h*, *S-2obj+h*, *B-2obj+h* and *B-S-2obj+h* are also substantially more precise than *2cs+h*.

Performance Bean improves the precision of an object-sensitive analysis at some small increase in cost, as shown in Table 1. As can be seen in Figs. 1 and 2, Bean may spend more time on processing more contexts introduced. *B-2obj+h* increases the analysis cost of *2obj+h* for all the nine benchmarks, ranging from 5.8% for `bloat` to 109.8% for `chart`, with an average of 54.8%. In addition, *B-S-2obj+h* also increases the analysis cost of *S-2obj+h* for all the nine benchmarks, ranging from 5.9% for `bloat` to 70.3% for `lusearch`, with an average of 49.1%.

Table 2: Pre-analysis times of BEAN (secs). For a program, its pre-analysis time comes from three components: (1) a context-insensitive points-to analysis ("CI"), (2) OAG construction per Fig. 6 (OAG), and (3) object-sensitive context computation per Fig. 8 ("CTX-COMP").

Benchmark	xalan	chart	eclipse	fop	luindex	pmd	antlr	lusearch	bloat
CI	82.6	112.2	49.6	105.5	39.0	65.3	56.9	39.1	52.5
OAG	0.2	0.2	0.1	0.2	0.2	0.1	0.2	0.1	0.1
CTX-COMP	83.0	168.0	32.1	236.5	11.7	13.9	13.9	18.3	13.3
Total	165.8	280.4	81.8	342.2	50.9	79.3	71.0	57.5	65.9

Table 2 shows the pre-analysis times of BEAN for the nine benchmarks. The pre-analysis is fast, finishing within 2 minutes for the most of the benchmarks and in under 6 minutes in the worst case. In Table 1, the analysis times for *B-2obj+h* and *B-S-2obj+h* do not include their corresponding pre-analysis times. There are three reasons: (1) the points-to information produced by "CI" in Table 2 (for some other purposes) can be reused, (2) and the combined overhead for "OAG" and "CTX-COMP" is small, and (3) the same pre-analysis is often used to guide BEAN to refine many object-sensitive analyses (e.g., *2obj+h* and *S-2obj+h*).

2obj+h and *S-2obj+h* are the top two most precise yet scalable object-sensitive analyses ever designed for Java programs [14]. BEAN is significant as it improves their precision further at only small increases in analysis cost.

5.3 RQ2: A Real-World Case Study

Let us use `java.util.HashSet`, a commonly used container from the Java library to illustrate how *B-2obj+h* improves the precision of *2obj+h* by enabling *may-alias* and *may-fail-cast* to answer their queries more precisely. In Fig. 12, the code in `main()` provides an abstraction of a real-world usage scenario for `HashSet`, with some code in `HashSet` and its related classes being extracted directly from JDK 1.6.0_45. In `main()`, `X` and `Y` do not have any subtype relation.

We consider two queries: (Q1) are `v1` and `v2` at lines 5 and 11 aliases (from *may-alias*)? and (Q2) may the casts at lines 6 and 12 fail (from *may-fail-cast*)?

Let us examine `main()`. In lines 2 – 6, we create a `HashSet` object, `HS/1`, insert an `X` object into it, retrieve the object from `HS/1` through its iterator into `v1`, and finally, copy `v1` to `x` via a type cast operation (`X`). In lines 7 – 12, we proceed as in lines 1 – 6 except that another `HashSet` object, `HS/2`, is created, and the object inserted into `HS/2` is a `Y` object and thus cast back to `Y`.

Let us examine `HashSet`, which is implemented in terms of `HashMap`. Each `HashSet` object holds a backing `HashMap` object, with the elements in a `HashSet` being used as the keys in its backing `HashMap` object. In `HashMap`, each key and its value are stored in an `Entry` object pointed to its field `table`.

In `main()`, the elements in a `HashSet` object are accessed via its iterator, which is an instance of `KeyIterator`, an inner class of `HashMap`.

As before, we have labeled all the allocation sites in their end-of-line comments. Figure 13 gives the part of the OAG related to the two `HashSet` objects,

```
1  void main(String[] args) {
2      HashSet xSet = new HashSet(); // HS/1
3      xSet.add(new X()); // X/1
4      Iterator xIter = xSet.iterator();
5      Object v1 = xIter.next();
6      X x = (X) v1;
7
8      HashSet ySet = new HashSet(); // HS/2
9      ySet.add(new Y()); // Y/1
10     Iterator yIter = ySet.iterator();
11     Object v2 = yIter.next();
12     Y y = (Y) v2;
13 }
14 class HashSet ... {
15     HashMap map = new HashMap(); // HM/1
16     public boolean add(Object e) {
17         return map.put(e, ...) == null;
18     }
19     public Iterator iterator() {
20         return map.newKeyIterator();
21     }
22     ...
23 }
```

```
23 class HashMap ... {
24     Entry[] table = new Entry[16]; // Entry[]/1
25     public Object put(Object key, ...) { ...
26         table[bucketIndex] =
27             new Entry(key, ...); // Entry/1
28     ... }
29     static class Entry {
30         final Object key;
31         Entry(Object k, ...) {
32             key = k;
33         }
34     }
35     private final class KeyIterator ... {
36         public Object next() { ...
37             Entry e = table[index];
38             return e.key;
39         }
40     }
41     Iterator newKeyIterator() {
42         return new KeyIterator(); // KeyIter/1
43     }
44     ...
45 }
```

Fig. 12: A real-world application for using `java.util.HaseSet`.

`HS/1` and `HS/2`, which are known to own their distinct `HM/1`, `Entry/1`, `Entry[]/1` and `KeyIter/1` objects during program execution.

2obj+h. To answer queries Q1 and Q2, we need to know the points-to sets of `v1` and `v2` found at lines 5 and 11, respectively. As revealed in Fig. 13, *2obj+h* is able to distinguish the `HashMap` objects in `HS/1` and `HS/2` by using two different heap contexts, [`HS/1`] and [`HS/2`], respectively. However, the two iterator objects associated with `HS/1` and `HS/2` are still modeled under one context [`HM/1`] as one abstract object `KeyIter/1`, which is pointed to by `xIter` at line 5 and `yIter` at line 11. By pointing to `X/1` and `Y/1` at the same time, `v1` and `v2` are reported as aliases and the casts at lines 6 and 12 are also warned to be unsafe.

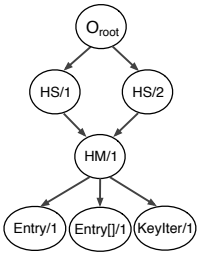


Fig. 13: Part of OAG related to `HS/1` and `HS/2`.

B-2obj+h. By examining the part of the OAG given in Fig. 13, *B-2obj+h* recognises that `HM/1` is redundant in the single heap context [`HM/1`] used by *2obj+h* for representing `Entry/1`, `Entry[]/1` and `KeyIter/1`. Thus, it will create two distinct sets of these three objects, one under [`HS/1`] and one under [`HS/2`], causing `v1` (`v2`) to point to `X/1` (`Y/1`) only. For query Q1, `v1` and v2 are no longer aliases. For query Q2, the casts at lines 6 and 12 are declared to be safe.

6 Related Work

Object-sensitivity, introduced by Milanova et al. [23, 24], has now been widely used as an excellent context abstraction for pointer analysis in object-oriented languages [14, 18, 29]. By distinguishing the calling contexts of a method call in terms of its receiver object's k-most-recent allocation sites (rather than k-most-recent call sites) leading to the method call, object-sensitivity enables object-oriented features and idioms to be better exploited. This design philosophy enables a k-object-sensitive analysis to yield usually significantly higher precision

at usually much less cost than a k-CFA analysis [17, 8, 14]. The results from our evaluation have also validated this argument further. In Table 1, *2obj+h* is significantly more precise than *2cs+h* in all the configurations considered and also significantly faster than *2cs+h* for all the benchmarks except `xalan`.

There once existed some confusion in the literature regarding which allocation sites should be used for context elements in a k-object-sensitive analysis [9, 15, 17, 24, 30]. This has recently been clarified by Smaragdakis et al. [29], in which the authors demonstrate that the original statement of object-sensitivity given by Milanova et al. [24], i.e., full-object-sensitivity in [29], represents a right approach in designing a k-object-sensitive analysis while the other approaches (e.g., [15]) may result in substantial loss of precision. In this paper, we have formalised and evaluated Bean based on this original design [24, 29].

For Java programs, hybrid object-sensitivity [14] enables k-CFA (call-site-sensitivity) to be applied to static call sites and object-sensitivity to virtual call sites. The resulting hybrid analysis is often more precise than their corresponding non-hybrid analyses at sometimes less and sometimes more analysis cost (depending on the program). As a general approach, Bean can also improve the precision of such a hybrid pointer analysis, as demonstrated in our evaluation.

Type-sensitivity [29], which is directly analogous to object-sensitivity, provides a new sweet spot in the precision-efficiency tradeoff for analysing Java programs. This context abstraction approximates the allocation sites in a context by the dynamic types (or their upper bounds) of their allocated objects, making itself more scalable but less precise than object-sensitivity [14, 29]. In practice, type-sensitivity usually yields an acceptable precision efficiently [20, 21]. How to generalise Bean to refine type-sensitive analysis is a future work.

Oh et al. [26] introduce a selective context-sensitive program analysis for C. The basic idea is to leverage a pre-impact analysis to guide a subsequent main analysis in applying context-sensitivity to where the precision improvement is likely with respect to a given query. In contrast, Bean is designed to improve the precision of a whole-program pointer analysis for Java, so that many clients may benefit directly from the improved points-to information obtained.

7 Conclusion

In the past decade, object-sensitivity has been recognised as an excellent context abstraction for designing precise context-sensitive pointer analysis for Java and thus adopted widely in practice. However, how to make a k-object-sensitive analysis even more precise while still using a k-limiting context abstraction becomes rather challenging. In this paper, we provide a general approach, Bean, to addressing this problem. By reasoning about an object allocation graph (OAG) built based on a pre-analysis on the program, we can identify and thus avoid redundant context elements that are otherwise used in a traditional k-object-sensitive analysis, thereby improving its precision at a small increase in cost.

In our future work, we plan to generalise Bean to improve the precision of other forms of context-sensitive pointer analysis for Java that are formulated in

terms of k-CFA and type-sensitivity (among others). Their redundant context elements can be identified and avoided in an OAG-like graph in a similar way.

8 Acknowledgement

The authors wish to thank the anonymous reviewers for their valuable comments. This work is supported by Australian Research Grants, DP130101970 and DP150102109.

References

1. Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. PLDI '14.
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. The DaCapo benchmarks: Java benchmarking development and analysis. OOPSLA '06.
3. Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Selective control-flow abstraction via jumping. OOPSLA '15.
4. Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. OOPSLA '09.
5. Chord. A program analysis platform for Java. http://www.cc.gatech.edu/∼naik/chord.html.
6. Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. SAS '01.
7. DOOP. A sophisticated framework for Java pointer analysis. http://doop.program-analysis.org.
8. Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. Bottom-up context-sensitive pointer analysis for Java. APLAS '15.
9. Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.
10. Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. Information flow analysis of android applications in droidsafe. NDSS '15.
11. Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. CGO '11.
12. Michael Hind. Pointer analysis: Haven't we solved this problem yet? PASTE '01.
13. Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. Scalable and precise taint analysis for android. ISSTA '15.
14. George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. PLDI '13.
15. Ondrej Lhoták. *Program Analysis using Binary Decision Diagrams.* PhD thesis, 2006.

16. Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. POPL '11.
17. Ondrej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: Is it worth it? CC '06.
18. Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1), 2008.
19. Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. Self-inferencing reflection resolution for Java. ECOOP '14.
20. Yue Li, Tian Tan, and Jingling Xue. Effective soundness-guided reflection analysis. SAS '15.
21. Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. Program tailoring: Slicing by sequential criteria. ECOOP '16.
22. Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. A user-guided approach to program analysis. FSE '15.
23. Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. ISSTA '02.
24. Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1), 2005.
25. Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. PLDI '06.
26. Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. PLDI '14.
27. Olin Grigsby Shivers. *Control-flow Analysis of Higher-order Languages of Taming Lambda*. PhD thesis, 1991.
28. Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2, 2015.
29. Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. POPL '11.
30. Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. PLDI '06.
31. Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J Fink, and Eran Yahav. *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, chapter Alias Analysis for Object-Oriented Programs.
32. Yulei Sui, Peng Di, and Jingling Xue. Sparse flow-sensitive pointer analysis for multithreaded programs. CGO '16.
33. Yulei Sui, Yue Li, and Jingling Xue. Query-directed adaptive heap cloning for optimizing compilers. CGO '13.
34. Yulei Sui, Ding Ye, and Jingling Xue. Static memory leak detection using full-sparse value-flow analysis. ISSTA '12.
35. Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Trans. Softw. Eng.*, 40(2), 2014.
36. WALA. T.J. Watson libraries for analysis. http://wala.sf.net.
37. Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. CGO '10.
38. Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in datalog. PLDI '14.