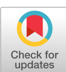


Generic Sensitivity: Customizing Context-Sensitive Pointer Analysis for Generics

Haofeng Li
lihaofeng19b@ict.ac.cn
SKLP, Institute of Computing Technology, CAS, China
University of Chinese Academy of Sciences, China

Jie Lu
lujie@ict.ac.cn
SKLP, Institute of Computing Technology, CAS, China

Haining Meng
menghaining@ict.ac.cn
SKLP, Institute of Computing Technology, CAS, China
University of Chinese Academy of Sciences, China

Liqing Cao
caoliqing19s@ict.ac.cn
SKLP, Institute of Computing Technology, CAS, China
University of Chinese Academy of Sciences, China

Yongheng Huang
huangyongheng20s@ict.ac.cn
SKLP, Institute of Computing Technology, CAS, China
University of Chinese Academy of Sciences, China

Lian Li*†
lianli@ict.ac.cn
SKLP, Institute of Computing Technology, CAS, China
University of Chinese Academy of Sciences, China

Lin Gao
gaolin@tianqisoft.cn
TianqiSoft Inc, China

ABSTRACT

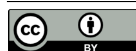
Generic programming has been extensively used in object-oriented programs such as Java. However, existing context-sensitive pointer analyses perform poorly in analyzing generics. This paper introduces *generic sensitivity*, a new context customization scheme targeting generics. We design our context customization scheme in such a way that generic instantiation sites, i.e., locations instantiating generic classes/methods with concrete types, are always preserved as key context elements. This is realized by augmenting contexts with a type variable lookup map, which is efficiently updated during the analysis in a context-sensitive manner.

We have implemented different variants of generic-sensitive analysis in Wala and experimental results show that the generic customization scheme can significantly improve performance and precision of context-sensitive pointer analyses. For instance, generic context customization significantly improves precision of 1-object-sensitive analysis, with an average speedup of 1.8×. In addition, generic context customization enables a 1-object-sensitive analysis to achieve overall better precision than a 2-object-sensitive analysis, with an averagely speed up of 12.6 × (62 × for `chart`).

CCS CONCEPTS

• **Theory of computation → Program analysis**.

*corresponding author: lianli@ict.ac.cn
†Also with TianqiSoft Inc, China.



This work is licensed under a Creative Commons Attribution 4.0 International License.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9413-0/22/11.
https://doi.org/10.1145/3540250.3549122

KEYWORDS

pointer analysis, generic programming, context sensitivity

ACM Reference Format:
Haofeng Li, Jie Lu, Haining Meng, Liqing Cao, Yongheng Huang, Lian Li, and Lin Gao. 2022. Generic Sensitivity: Customizing Context-Sensitive Pointer Analysis for Generics. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3540250.3549122

1 INTRODUCTION

Pointer analysis statically computes the possible run time values (abstract memory locations) of pointer variables in a program, and it provides a foundation for a variety of applications, such as bug detection [10, 29, 34], compiler optimization [46], security analysis [3, 12, 13, 15], etc. The effectiveness and precision of those client applications directly depend on the precision of the underlying pointer analysis results.

There is a rich literature optimizing the efficiency and precision of pointer analysis [17, 30, 42, 45, 55], and one of the key mechanism to improve precision is *context-sensitivity* [31, 37, 38, 40, 41, 43]. Context-sensitive pointer analyses differ values of a pointer variable under different calling contexts, effectively reducing spurious results introduced by infeasible inter-procedural control flow paths and drastically improving precision. In general, a context is represented by a sequence of k context elements, where context elements can be call-sites (k-call-site-sensitivity), allocation sites of receiver objects (k-object-sensitivity), or types of receiver objects (k-type-sensitivity). For object-oriented programs, object-sensitivity is believed to be better than call-site-sensitivity in achieving precision and efficiency [37, 38], and type-sensitivity is regarded as a more efficient, but less precise alternative to object-sensitivity [43].

Under k-limiting, the most recent k context-elements are picked to represent a context. For instance, k-object-sensitive pointer analysis analyzes a method *m* with its context $[O_k, ..., O_1]$, where O_1 is a receiver object of *m* and O_{i+1} is an *allocator* of O_i, i.e., a receiver object of a method allocating O_i. In practice, k is often limited to 1 or 2 in analyzing large real-world applications [23, 33].

This paper, for the first time, proposes a new context customization scheme for *generics*. Generic programming allows to write generic algorithms for different data representations using type variables, and has been widely adopted and used in modern programming languages including `C++`, `Java`, `C#`, etc. For instance, the previous study [8] over a large corpus of open-source projects demonstrated that generics, since its introduction to Java in 2004, is one of the most frequently used features in Java. With generics, we can define classes or methods with type variables as parameters, and later instantiate those classes or methods by giving them specific actual types.

Our context customization scheme for generics is based on the observation that *generic instantiation sites*, i.e., locations instantiating generic with concrete types, are key context elements. However, those key context elements are not preserved in existing context-sensitive analyses, often leading to poor performance and precision. Hence, we propose *generic sensitivity*: instead of always picking the most recent context elements, we keep generic instantiation sites as part of context and propagate such information within generic classes and generic methods. This may sound trivial but can be challenging, since type variables are propagated across generic classes (e,g., generic classes/objects defined within generic classes) or generic methods (by calling other generic methods). In our approach, this challenge is addressed by augmenting contexts with generic instantiation information, which is efficiently updated during the analysis in a context-sensitive manner.

We have implemented our approach in WALA [18] and evaluated it against a set of 18 real-world applications, including the Dacapo benchmark suite [5] and another 7 popular open-source applications. Experimental results show that our context customization scheme can significantly improve performance and precision of context-sensitive pointer analyses. For instance, generic sensitivity significantly improves the precision of 1-object-sensitive analysis, with noticeable performance improvements. In addition, generic context customization enables a 1-object-sensitive analysis to achieve overall better precision than a 2-object-sensitive analysis, with an average speedup of 12.6 × (up to 62 × for `chart`).

To summarize, the paper makes the following contributions:

- We present generics sensitivity, a new context customization scheme targeting generics. To the best of our knowledge, this is the first attempt to optimize context-sensitive pointer analysis for generics.
- We demonstrate how to apply our context customization scheme to two mainstream context-sensitive variants: k-object-sensitivity and k-type-sensitivity.
- We have implemented different variants of generics sensitive pointer analysis in Wala [18] and evaluated our implementations against a large set of 18 popular real-world applications, including the Dacapo benchmark suite and 7 popular open-source applications. Experimental results show that our generic customization scheme can significantly improve performance and precision of context-sensitive pointer analyses. For instance, generic context customization enables a 1-object-sensitive to achieve overall better precision than a 2-object-sensitive analysis, with a significant speedup: 12.6 × on average and up to 62 × for `chart`.

The rest of the paper is organized as follows. Section 2 motivates our approach with an example and highlights its key challenges. Section 3 formally describes generic sensitivity and demonstrates how it can be adapted to object-sensitivity and type-sensitivity. We evaluate the effectiveness and efficiency of generic sensitivity in Section 4. Section 5 reviews related work and Section 6 concludes this paper.

2 MOTIVATION

We first give a brief introduction on context-sensitive pointer analysis (Section 2.1). Then we illustrate the limitations of existing context-sensitive pointer analysis in analyzing generics with an example (Section 2.2). Finally, we motivate our context customization scheme and discuss its main challenges (Section 2.3).

2.1 Context Sensitivity

Pointer analysis computes the points-to sets of program variables, i.e., set of *abstract locations* that can be pointed to by a variable v (denoted as $pts(v)$). Typically, abstract locations are represented as allocation sites (instructions allocating objects, e.g., `new` in Java), denoting all dynamic object instancess allocated by the instruction at run time. In context-sensitive analysis, both variable v and abstract location o are qualified with a context, effectively distinguishing their different dynamic instances. Hence, instead of computing whether $o \in pts(v)$ as in context-insensitive analysis, context-sensitive analysis computes the relation $(c_o, o) \in pts(c_v, v)$, where c_o and c_v are the context for abstract location o and variable v, respectively.

Call-site sensitivity, object sensitivity, and type sensitivity are three main variants of context sensitivity, where call-sites, allocation sites of receiver objects, and types of receiver objects are considered as context elements, respectively. To ensure termination, *k-limiting* is applied to bound the number of context elements to k. In practice, k is often set to less than 2 for scalability.

Among the above three variants, object sensitivity and type sensitivity (as a cheaper alternative) are considered to be more suitable in analyzing object-oriented programs. In particular, object sensitivity is more precise and efficient than call-site sensitivity and is considered as the most precise context-sensitivity variant for Java [6, 27]. In k-object-sensitivity, an object o_0 is cloned multiple times, each with a different context of length $k-1$, referred to as the *heap context*. A heap context is in the form of $[o_{k-1}, ..., o_1]$, where o_i $(1 < i <= k-1)$ is an allocator of o_{i-1}, i.e., o_{i-1} is allocated in a method with o_i being a receiver object. Thus, method $o_0.m$ (with o_0 be a receiver object) will be analyzed context-sensitively multiple times: for each distinct heap context c_o, the method is analyzed once under the *method context* $[c_o, o_0]$. In type sensitivity, contexts are constructed in the same fashion, except that the context element o_i (in object-sensitivity) is replaced with its type. As a result, multiple object-sensitive contexts will be merged and analyzed together in type-sensitive analysis, yielding imprecise results.

```
1  public static void main(String[] args) {
2    HashMap<String, A> map1 = new HashMap<>();//O1
3    map1.put("A", new A());//OA
4    Object v1 = map1.get("A");
5    A a = (A) v1;//cast may fail?
6
7    HashMap<String, B> map2 = new HashMap<>();//O2
8    map2.put("B", new B());//OB
9    Object v2 = map2.get("B");
10   B b = (B) v2;//cast may fail?
11 }
12
13 class HashMap<K,V> ... {
14
15   Node<K,V>[] table = new Node[16];//O3
16   public void put(K k, V v, ...) {
17     Node<K,V> n = new Node<>(k, v, ...);//O4
18     table[hash(k)] = n;
19   }
20   public final V get(K k) {
21     Node<K,V> n = table[hash(k)];
22     return n.getValue();
23   }
24   class Node<M,N> ... {
25     M key;
26     N value;
27     Node(M k, N v, ...) {
28       key = k;
29       value = v;
30     }
31     public final M getKey() {
32       return key;
33     }
34     public final N getValue() {
35       return value;
36     }
37   }
38 }
```

Figure 1: Simplified code example from `java.util.HashMap`.

2.2 A Motivating Example

Let us study context-sensitive pointer analysis with an example in Figure 1. The example uses generic class `java.util.HashMap`. Hereafter, we only discuss the two main stream context-sensitive variants for object-oriented programs: object-sensitivity and type-sensitivity.

In the `main` method, there are two `HashMap` objects: O_1 (line 2) and O_2 (line 7). Object O_A is created and put into O_1 at line 3, then retrieved back via the `get` method at line 4. Similarly, object O_B is created and put into O_2 at line 8, then retrieved back at line 9. As a result, the two cast operations (line 5 and 10) will never fail.

The simplified code snippet of `HashMap` is given in lines 13 - 38. `HashMap` stores data in `table`, an array of `Node` objects (line 15). The `put` method creates a `Node` object and stores it in `table` (lines 16-19). The `get` method retrieves the corresponding `Node` object from `table`, then returns its value via the `getValue` interface (lines 20-23). Note that the `Node` class (lines 24-38) is implemented as an inner generic class, and it is instantiated with the type variables (i.e., `K` and `V`) of its outer class `HashMap` when creating a `Node` object.

k-object sensitivity. In 1-object sensitive analysis (abbreviated as 1-obj), the receiver object of the call to `put`/`get` method at line 3/4 and line 8/9 are O_1 and O_2, respectively. Hence, the call to `put`/`get` methods at different call-sites can be distinguished using contexts $[O_1]$ and $[O_2]$. In `put` (line 17), with 1-obj analysis, we get $pts(O_1,\texttt{n}) = \{O_4\}$ and $pts(O_2,\texttt{n}) = \{O_4\}$. Then in the constructor of `Node` (lines 27-30), since O_4 is the only one receiver object, we get $pts(O_4,\texttt{key}) = \{\text{"A","B"}\}$ and $pts(O_4,\texttt{value}) = \{O_A, O_B\}$. As a result, call to O_1.`get` and O_2.`get` will return a value pointing to both O_A and O_B, leading to cast-may-fail false alarms at line 5 and line 10.

The example can only be precisely analyzed when the context depth is set to more than 1. In `put` (line 17), with 2-obj analysis, we get $pts(O_1,\texttt{n}) = \{(O_1, O_4)\}$ and $pts(O_2,\texttt{n}) = \{(O_2, O_4)\}$, where object O_4 is qualified with a heap context. Hence, the constructor of class `Node` (lines 27-30) is analyzed twice with 2 distinct contexts: $[O_1, O_4]$ and $[O_2, O_4]$. Thus, we can precisely compute the pointer values of `key` and `value` as $pts([O_1, O_4],\texttt{key}) = \{\text{"A"}\}$, $pts([O_2, O_4],\texttt{key}) = \{\text{"B"}\}$, $pts([O_1, O_4],\texttt{value}) = \{O_A\}$, and $pts([O_1, O_4],\texttt{value}) = \{O_B\}$. Finally, we can correctly analyze that $pts(\texttt{v1}) = \{O_A\}$ and $pts(\texttt{v2}) = \{O_B\}$, avoiding false cast-may-fail alarms.

k-type sensitivity. Type-sensitive analysis is less precise than object-sensitive analysis. Hence, 1-type analysis cannot distinguish the pointer values of `v1` and `v2`, yielding same false alarms. Moreover, the standard 2-type analysis cannot distinguish the context in analyzing the constructor (and other methods) of `Node` either, since both O_1 and O_2 have type `HashMap` (for efficiency, the actual type parameters of generic-typed local variables are often omitted in the byte code). The default type-sensitive analysis can be extended with a simple analysis as illustrated in Section 3.3, to infer the actual parameters for variables with generic types. Thus, 2-type analysis can then distinguish the context using the distinct generic types `HashMap<String,A>` and `HashMap<String,B>`.

Discussion. For clarity, we simplify the example in Figure 1 so that it can be precisely analyzed by a 2-obj analysis. The real implementation of `HashMap` is much more complicated and may require a deeper context. Many algorithms and design patterns wrap generic classes insider other generic classes, which can be precisely analyzed only with a very deep context. For instance, `HashSet` is implemented by encapsulating `HashMap` and it can only be precisely analyzed with at least 3-object-sensitivity. Existing work [32, 35] also summarized numerous scenarios where a deeper context (>=3) is required. However, since the number of contexts grows exponentially with the depth, it is often infeasible to scale 3-obj analysis to real-world applications.

2.3 Generic Sensitivity

For generics, the key to ensure precision is to keep the *instantiation location*, i.e., location instantiating generic type parameters

```
1  public static void main(String [] args) {
2      G<A> g = new G<A>(); // O1
3      B b = new B(); // O2
4      g.foo(b);
5  }
6  public class G<T> {
7      public void <E> foo(E e) {
8          M<T> m = new M<T>(); // O3
9          m.bar();
10         M<E> n = new M<E>(); // O4
11         n.bar();
12     }
13 }
14 public class M<K> {
15     public void bar(){ }
16 }
```

Figure 2: Example of generics.

with concrete types, as part of context. As such, distinct pointer values flow into/from generic methods and generic objects can be effectively identified. In our example in Figure 1, line 2 and line 7 instantiate the generic class `HashMap` with actual types. Those actual types are passed as type variables of `HashMap` to instantiate `Node` at line 17. Hence, the corresponding location (O_1 and O_2) should be regarded as the context in analyzing methods of class `Node`. As a result, with 1-object-sensitivity, we can compute the same precise result as 2-obj analysis: $pts(O_1,\text{key}) = \{\text{"A"}\}$, $pts(O_2,\text{key}) = \{\text{"B"}\}$, $pts(O_1,\text{value}) = \{O_A\}$, and $pts(O_1,\text{value}) = \{O_B\}$.

For the example in Figure 1, an *omitting-generics* approach may work by simply omitting all invocation contexts within generic classes. However, this approach does not work for generic methods. As shown in Figure 2, the generic method `foo` (with type parameter `E`) is defined in generic class `G` (lines 6 - 13) with type parameter `T`. At line 8, we create a new generic object with type variable `T`, whose actual type is instantiated at line 2. Hence, we should pick O_1 as the context in analyzing the method call `m.bar` at line 9. On the other hand, line 10 instantiates the generic class `M` with type variable `E`. The instantiate location of `E`, i.e., O_2 at line 3, will be picked as the context in analyzing the method call `n.bar` at line 11. However, the straight forward omit-generics approach will use O_1 as the context instead.

To effectively analyze the above example, we need to precisely identify the actual instantiation location of type variables under different contexts. This may require a context-sensitive pointer analysis to compute.

3 GENERIC SENSITIVE POINTER ANALYSIS

In our approach, we precisely track propagation of type variables by augmenting context with generic instantiation locations, which are efficiently updated context-sensitively during the analysis.

3.1 Context Customization

The traditional context c is extended to a tuple $\langle c, G\rangle$, where G records all instantiation sites for available type variables. For non generic-related methods, G is $\emptyset$. The size of G is bounded to the number of available type variables.

For object-sensitive analysis, G maps a type variable to its instantiate location (more precisely, to the object created at the instantiate location). For type-sensitive analysis, G maps a type variable to its instantiated concrete type. In Java, developers can instantiate a generic class with explicit types (Figure 3(a)), or without giving any actual type arguments. In the later case, the generic class is by default instantiated with type `Object`. For instance, in Figure 3 (b), `s` is created at line 3 with type `HashSet<Object>`. At line 5, an object with type `A` is firstly created and implicitly cast to `Object`, before it is put in `s`.

With Actual Type Arguments	Without Actual Type Arguments

`1 class C {`
`2   void foo() {`
`3     Set<A> s =`
`4       new HashSet<>();`
`5   }`
`6 }`
	`1 class C {`
	`2   Set foo() {`
	`3     Set s =`
	`4       new HashSet();`
	`5     s.add(new A();)`
	`6     return s;`
	`7   }`
	`8 }`

(a)	(b)

Figure 3: Generic instatiation in Java.

Since type-sensitive analysis relies on concrete type information, it will fail to distinguish different contexts when generic classes are instantiated without giving actual type parameters, leading to imprecise results. We can employ a precise inter-procedural pre-analysis to infer actual type arguments of generics as [7, 11, 50]. However, the cost of such a pre-analysis may offset the benefits brought by more precise type information. Hence, we apply a simple analysis to infer actual instantiated types of a generic object by examining its local usages, as follows.

Definition 3.1. Type parameter inference: if generic object O instantiating generic class with formal type parameter T does not escape its declared scope and all its usages of T can be resolved to type C, we can safely regard C as the actual type parameter instantiating T.

We perform a simple conservative escape analysis where a variable escapes a scope if 1) it is accessible outside the scope, 2) it returns from the scope, or 3) it is stored to another escaping variable. As in Figure 3(b), if `s` is not returned (i.e., does not escape its declared scope `foo`), we can infer that `s` instantiates `HashSet` with type `A`, i.e., `s` has type `HashSet<A>`.

Finally, if we fail to resolve the actual type parameters instantiating a generic class, we use the instantiation location as a pseudo type. In the example Figure 3 (b), a pseudo type T_3 is introduced to instantiate `s`, i.e., the statement at line 3 is regarded as `Set<`T_3`> s = new HashSet()` in our analysis. As such, we are effectively applying object-sensitivity in analyzing generics since each instantiation location is regarded as a distinct type.

Kind	Statements
New	$l : x = new\, C\, \langle \mathcal{T} : A \rangle$
Assign	$l : x = y;$
Load	$l : x = y.f;$
Store	$l : x.f = y;$
Call	$l : x = v_0.m' \langle \mathcal{T} : A \rangle\, (v_1)$

Figure 4: Five types of statements analyzed by context-sensitive pointer analyses.

3.2 Formalization

Without loss of generality, we consider a simplified subset of Java, with five types of labeled statements in Figure 4. We write "$x = new\ C\, \langle \mathcal{T} : A \rangle$" for object allocation. If C is a generic class, $\mathcal{T}$ is its formal type parameter, and A is the actual type parameter instantiating $\mathcal{T}$. Otherwise, both $\mathcal{T}$ and A is Nil. Similarly, a generic method call "$x = v_0.m' \langle \mathcal{T} : A \rangle\, (v_1)$" instantiates its formal type parameter $\mathcal{T}$ with actual type parameter A. Both $\mathcal{T}$ and A are Nil for non-generic method invocations. For clarity, our formalization considers NEW and Call statements with one type parameter only. The general forms of NEW and Call statements with multiple parameters can be analyzed in the same fashion.

The statement "x = new C(...)" in Java is modeled as "x = new C; x.<init>(...)", where <init>() is the corresponding constructor invoked. Control flow statements are irrelevant for context-sensitive flow-insensitive analysis hence skipped. Accesses to array elements are modeled by collapsing all the elements into a special field of the array. In addition, every method is assumed to return via the variable ret. Since we formalize a method call with only one actual parameter, each method also has only one formal parameter p.

Given a program, let $\mathbb{M}, \mathbb{F}, \mathbb{H}, \mathbb{V}, \mathbb{L}, \mathbb{T}$ be its sets of methods, fields, allocation sites, local variables, statement labels and types, respectively. We use the symbol $\mathbb{C}$ for the universe of contexts. The following auxiliary functions are used in our rules:

- **methodOf**: $\mathbb{L} \mapsto \mathbb{M}$
- **methodCtx** : $\mathbb{M} \mapsto \wp(\mathbb{C})$
- **dispatch** : $\mathbb{M} \times \mathbb{H} \mapsto \mathbb{M}$
- **pts** : $(\mathbb{V} \bigcup \mathbb{H} \times \mathbb{F}) \times \mathbb{C} \mapsto \wp(\mathbb{H} \times \mathbb{C})$
- **typeOf** : $\mathbb{V} \mapsto \mathbb{T}$

where **methodOf** gives the containing method of a statement, **methodCtx** maintains the contexts used for analyzing a method, **dispatch** resolves a call to a target method, **pts** records the context-sensitive points-to information for a variable or field, and **typeOf** returns the declared type of a variable.

Given a list of context element $c = [e_n, ..., e_1]$ and a context element e, we use the notation $e +\!\!+ c$ for $[e, e_n, ..., e_1]$ and c_k for $[e_k, ..., e_1]$ where $k < n$.

3.2.1 Customizing Object Sensitivity. Let $\mathbb{G} := \overline{\mathbb{T} \mapsto \mathbb{H}}$ maps a type variable $\mathcal{T} \in \mathbb{T}$ to an allocation site $O_l \in \mathbb{H}$ (identified by label l). The universe context is $\mathbb{C} = \mathbb{H}^* \times \mathbb{G}$. We define the following two functions:

$$\textbf{Update}(G, \mathcal{T}, A, O_l) = \begin{cases} \emptyset & \mathcal{T} \equiv Nil \\ [\mathcal{T} \rightarrow O_l] & \mathcal{T} \neq Nil \wedge A \notin G \\ [\mathcal{T} \rightarrow G(A)] & \mathcal{T} \neq Nil \wedge A \in G \end{cases}$$

$$\frac{l : x = new\, C\, \langle \mathcal{T} : A \rangle \quad m = \textbf{methodOf}(l) \quad ctx = \langle c, G \rangle \in \textbf{methodCtx}(m) \quad G' = \textbf{Update}(G, \mathcal{T}, A, O_l) \quad hctx = \langle c_{k-1}, G' \rangle}{(O_l, hctx) \in \textbf{pts}(x, ctx)} \text{ [New]}$$

$$\frac{l : x = y \quad m = \textbf{methodOf}(l) \quad ctx \in \textbf{methodCtx}(m)}{\textbf{pts}(y, ctx) \subseteq \textbf{pts}(x, ctx)} \text{ [Assign]}$$

$$\frac{l : x = y.f \quad m = \textbf{methodOf}(l) \quad ctx \in \textbf{methodCtx}(m) \quad (O, hctx) \in \textbf{pts}(y, ctx)}{\textbf{pts}(O.f, hctx) \subseteq \textbf{pts}(x, ctx)} \text{ [Load]}$$

$$\frac{l : x.f = y \quad m = \textbf{methodOf}(l) \quad ctx \in \textbf{methodCtx}(m) \quad (O, hctx) \in \textbf{pts}(x, ctx)}{\textbf{pts}(y, ctx) \subseteq \textbf{pts}(O.f, hctx)} \text{ [Store]}$$

$$\frac{\begin{array}{c} l : x = a_0.f\, \langle \mathcal{T} : A \rangle\, (a_1) \quad m = \textbf{methodOf}(l) \\ ctx =< c_m, G_m > \in \textbf{methodCtx}(m) \\ (O_0, hctx) \in \textbf{pts}(a_0, ctx) \quad (O_1, -) \in \textbf{pts}(a_1, ctx) \\ hctx =< c, G > \quad G' = \textbf{Append}(G, G_m, \mathcal{T}, A, O_1) \\ ctx' = \langle O_0 +\!\!+ c, G' \rangle \quad m' = \textbf{dispatch}(f, O_0) \end{array}}{\begin{array}{c} ctx' \in \textbf{methodCtx}(m') \quad (O_0, hctx) \in \textbf{pts}(this^{m'}, ctx') \\ \textbf{pts}(a_1, ctx) \subseteq \textbf{pts}(p^{m'}, ctx') \\ \textbf{pts}(ret^{m'}, ctx') \subseteq \textbf{pts}(x, ctx) \end{array}} \text{ [Call]}$$

Figure 5: Rules for k-Obj analysis with generic customization.

$$\textbf{Append}(G, G_1, \mathcal{T}, A, O_l) = \begin{cases} G & \mathcal{T} \equiv Nil \\ G \uplus [\mathcal{T} \rightarrow O_l] & \mathcal{T} \neq Nil \wedge \\ & A \notin G \uplus G_1 \\ G \uplus [\mathcal{T} \rightarrow G \uplus G_1(A)] & \mathcal{T} \neq Nil \wedge \\ & A \in G \uplus G_1 \end{cases}$$

where the function $G(A)$ looks up the mapped allocation site of type variable A.

The two functions are used to update G for NEW and CALL statements. Figure 5 gives the five rules for analyzing the five kind of statements in Figure 4. Except for [New] and [Call], the other 3 rules are same as standard k-obj analysis.

In [New], $O_l \in \mathbb{H}$ is an abstract heap object created from the allocation site at l, identified by its heap context $hctx$. Given the method context $ctx = \langle c, G \rangle$, O_l's heap context $hctx$ is constructed as $\langle c_{k-1}, G' \rangle$, where c_{k-1} selects the first $k-1$ context elements from c as in standard k-obj analysis and G' is updated by the **Update** function, as follows.

- If O_l is a non-generic object, i.e., formal type parameter $\mathcal{T}$ is Nil, G' is set to $\emptyset$. Thus, method calls with non-generic objects as their receiver objects are analyzed same as in standard object-sensitive analysis.
- If O_l is intantiated with a concrete type, i.e., $\mathcal{T} \neq Nil \wedge A \notin G$, G' is set to $[\mathcal{T} \mapsto O_l]$. As a result, the instantiate location l is regarded as part of context in analyzing a method call with O_l being a receiver object.
- At last, if O_l is instantiated with a type variable, i.e., $\mathcal{T} \neq Nil \wedge A \in G$, we identify the actual instantiate location of A by looking up the context of l's containing method. G' is updated to $[\mathcal{T} \mapsto G(A)]$, enforcing that the actual generic instantiation location is always part of the context.

Note that the **Update** function does not preserve existing mapping of type variables in G. Since those type variables are invisible

in analyzing method calls where O_l is the receiver object, there is little benefit to preserve them in the heap context of O_l.

In [CALL], a call to an instance method $x = a_0.f\,\langle\mathcal{T} : A\rangle$ is analyzed. Let m' be a resolved target method, we write $this^{m'}, p^{m'}, ret^{m'}$ for the "this" variable, parameter, and return variable of m', respectively. Let O_0 be a receiver object of the method call with heap context $hctx =< c, G >$ and let $ctx = \langle c_m, G_m\rangle$ be a context of m. Similar to [NEW], a context $ctx' = \langle O_0 +\!\!+ c, G'\rangle$ is constructed in analyzing m', where $O_0 +\!\!+ c$ appends the receiver object O_0 with O_0's heap context in a standard manner and G' is updated by the **Append** function, as follows.

- If f is a non-generic call, i.e., $\mathcal{T} \equiv Nil$, G remains unchanged.
- If f is a generic call instantiated with a concrete type, i.e., $\mathcal{T} \neq Nil \wedge A \notin G \uplus G_m$, G' is updated by adding the new mapping $\mathcal{T} \mapsto O_l$ to G.
- If f is a generic call instantiated with a type variable, i.e., $\mathcal{T} \neq Nil \wedge A \in G \uplus G_m$, G' is updated by introducing to G a mapping: from $\mathcal{T}$ to its actual instantiate site $G \uplus G_m(A)$. Note that available type variables can be propagated from the receiver object (in which case $A \in G$), or from the caller method (in which case $A \in G_m$).

One may wonder whether the same type variable A may exists in both G and G_m. In that case, by construction, A must be introduced at the allocation site of generic object O_0, by the **Update** function. Such information may be further propagated to contexts of method m. In that case, both $G(A)$ and $G_m(A)$ are resolved to the same location instantiating A.

In the conclusion of the rule, $ctx' \in$ **methodCtx**(m') shows how the context of a method are introduced. Initially, we have **methodCtx**(main)=$\{\langle[], \emptyset\rangle\}$.

Let us revisit the example in Figure 2. A generic object O_1 is created at line 2. Hence, we have $(O_1, \langle[], T \mapsto O_1\rangle) \in pts(g, \langle[], \emptyset\rangle)$ ([NEW]). Line 4 invokes the generic method `foo<E>` where O_1 is the receiver object and O_2 is the actual parameter, i.e., `g.foo<E:B>(b)`. Hence, we analyze the target method `foo` with an updated context $\langle[O_1], [T \mapsto O_1, E \mapsto O_2]\rangle$ ([CALL]). In `foo`, the object created at line 8 (O_3) is instantiated with type variable `T`. Hence, it has the updated heap context $\langle[O_1], \texttt{K} \mapsto O_1\rangle$. Similarly, O_4 at line 10 has the heap context $\langle[O_1], \texttt{K} \mapsto O_2\rangle$. The two method call at line 9 and 10 are then analyzed with distinct contexts. To summarize, G always maps an available type variable to its actual instantiation location, to encode actual instantiation location of generics as part of context.

3.2.2 Customizing Type Sensitivity. . For type-sensitive analysis, $\mathbb{G} := \overline{\mathbb{T} \mapsto \mathbb{T}}$ maps a type variable $\mathcal{T} \in \mathbb{T}$ to an actual type $T \in \mathbb{T}$. The **Update** and **Append** functions are defined as follows.

$$\textbf{Update}(G, \mathcal{T}, A) = \begin{cases} \emptyset & \mathcal{T} \equiv Nil \\ [\mathcal{T} \to A] & \mathcal{T} \neq Nil \wedge A \notin G \\ [\mathcal{T} \to G(A)] & \mathcal{T} \neq Nil \wedge A \in G \end{cases}$$

$$\textbf{Append}(G, G_1, \mathcal{T}, A) = \begin{cases} G & \mathcal{T} \equiv Nil \\ G \uplus [\mathcal{T} \to A] & \mathcal{T} \neq Nil \wedge \\ & A \notin G \uplus G_1 \\ G \uplus [\mathcal{T} \to G \uplus G_1(A)] & \mathcal{T} \neq Nil \wedge \\ & A \in G \uplus G_1 \end{cases}$$

$$\frac{l : x = new\, C\,\langle\mathcal{T} : A\rangle \quad m = \textbf{methodOf}(l) \qquad ctx = \langle c, G\rangle \in \textbf{methodCtx}(m) \qquad G' = \textbf{Update}(G, \mathcal{T}, A) \quad hctx = \langle[c]_{k-1}, G'\rangle}{(O_l, hctx) \in \textbf{pts}(x, ctx)} \quad \text{[NEW]}$$

$$\frac{\begin{array}{c} l : x = a_0.f\,\langle\mathcal{T} : A\rangle\,(a_1) \quad m = \textbf{methodOf}(l) \\ ctx =< c_m, G_m >\in \textbf{methodCtx}(m) \\ (O_0, hctx) \in \textbf{pts}(a_0, ctx) \\ hctx =< c, G > \quad G' = \textbf{Append}(G, G_m, \mathcal{T}, A) \\ ctx' = \langle\textbf{typeOf}(O_0) +\!\!+ c, G'\rangle \quad m' = \textbf{dispatch}(f, O_0) \end{array}}{\begin{array}{c} ctx' \in \textbf{methodCtx}(m') \;\; (O_0, hctx) \in \textbf{pts}(this^{m'}, ctx') \\ \textbf{pts}(a_1, ctx) \subseteq \textbf{pts}(p^{m'}, ctx') \\ \textbf{pts}(ret^{m'}, ctx') \subseteq \textbf{pts}(x, ctx) \end{array}} \quad \text{[CALL]}$$

Figure 6: K-type analysis with generic customization.

Compared to object-sensitivity, the object allocation site is not used by the two functions.

Figure 6 gives the k-type analysis rules with generic customization. We only list [NEW] and [CALL], since the other 3 rules are same as in Figure 5. Compared to object-sensitivity, types of receiver objects, instead of allocation sites, are picked as context elements.

Let us study the example in Figure 2 again. A generic object O_1 is instantiated with actual type `A` at line 2. Hence, we have $(O_1, \langle[], T \mapsto \texttt{A}\rangle) \in pts(g, \langle[], \emptyset\rangle)$ ([NEW]). At line 4, we have the generic method call `g.foo<E:B>(b)` where O_1 is the receiver object. Since O_1 has the declared type `G`, we analyze the target method `foo` with an updated context $\langle[\texttt{G}], [T \mapsto \texttt{A}, E \mapsto B]\rangle$ ([CALL]). In `foo`, the object created at line 8 (O_3) is instantiated with type variable `T`. Hence, it has the updated heap context $\langle[\texttt{G}], \texttt{K} \mapsto \texttt{A}\rangle$. Similarly, O_4 at line 10 has the heap context $\langle[\texttt{G}], \texttt{K} \mapsto \texttt{B}\rangle$. Finally, the method `bar` is analyzed under two contexts $\langle[\texttt{M}], \texttt{K} \mapsto \texttt{A}\rangle$ and $\langle[\texttt{M}], \texttt{K} \mapsto \texttt{B}\rangle$. It is worth noting that in our extended type analysis, the generic type and the recorded actual instantiated types form the complete instantiated type signatures for generics.

3.3 Implementation

We have implemented our generic customization scheme in WALA and applied it to several pointer analysis variants: object-sensitive analysis, type-sensitive analysis, and insensitive analysis. There is a default implementation of k-obj analysis in WALA. However, instead of setting the heap context depth to *k-1*, it sets both method context and heap context to the same depth *k*. Hence, we revised the default implementation to be consistent with the standard k-obj definition [37, 38, 43]. We also implemented in WALA a new k-type analysis according to its original definition [43].

Sometimes, generic instantiation information may be optimized out in Java byte-code. For instance, Java tends to optimize out the actual instantiation type of a local variable if it is assigned from another generic typed variable. Hence, we apply simple type inference based on the rule that the lhs and rhs of an assignment must have identical generic types.

Following previous works [32, 33, 35], we use JDK1.6 (1.6.0_30) to analyze our benchmarks. We disable the exclusion option in WALA which can exclude some packages of JDK because those packages in the exclusion are wildly used in all benchmarks. For native code, we use the summaries provided by WALA. We disable the reflection

Table 1: Number of generic object allocations (abbreviated as GC) and generic method invocations (GM). #S is the number of instantiate locations, and #A is the number of actual type arguments.

Programs	Application		JDK	
	GC	GM	GC	GM
	#S(#A)	#S(#A)	#S(#A)	#S(#A)
antlr	21(37)	1(1)	503(701)	262(397)
bloat	304(376)	8(8)	342(472)	217(348)
chart	198(277)	39(39)	532(741)	280(420)
eclipse	74(97)	32(32)	349(479)	222(353)
fop	128(195)	4(4)	586(810)	293(431)
hsqldb	102(126)	4(4)	626(851)	342(483)
jython	101(138)	5(5)	346(476)	221(352)
luindex	54(70)	8(8)	452(620)	254(399)
lusearch	54(70)	8(8)	452(620)	254(399)
pmd	187(249)	9(9)	590(810)	308(445)
xalan	35(56)	1(1)	348(482)	217(348)
antlr4	345(450)	1,213(1,225)	610(846)	324(466)
byte-buddy	342(430)	257(261)	380(523)	244(376)
findbugs	455(617)	93(102)	570(794)	295(432)
javassist	60(74)	13(13)	341(470)	218(349)
jflex	21(28)	0(0)	509(704)	265(400)
junit	46(52)	55(56)	393(537)	251(383)
modelmapper	335(416)	206(210)	379(522)	244(376)

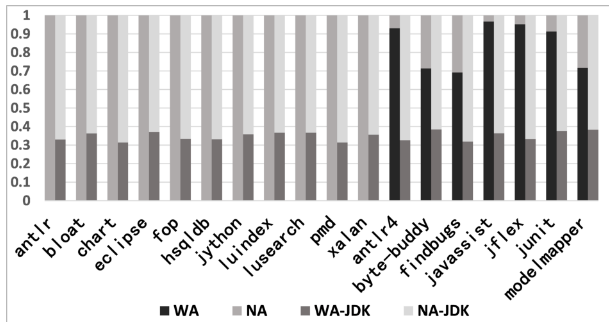


Figure 7: Percentages of generic object allocations with actual types. WA is with actual types, NA is without actual types, WA-JDK and NA-JDK are generic object allocations in JDK with or without actual types.

option in WALA since it fails to analyze most benchmarks even with insensitive pointer analysis.

4 EVALUATION

We evaluate generic sensitive pointer analysis by applying and comparing our context customization schemes to an array of pointer analyses at different precision. In total, there are 11 variants of pointer analyses. For illustration, we mainly compare 2 groups of them in this section: object-sensitive group (GenO, 1-Obj, Gen+1-Obj, 2-Obj) and type-sensitive group (GenT, 1-type, Gen+1-type, 2-type).

Hereafter, we use GenT as the customization scheme for type-sensitivity and GenO as the customization scheme for object-sensitivity. By default, GenT and GenO are our context customization schemes applied to the insensitive Andersen's analysis [1]. In this case, only generic objects/methods are analyzed context-sensitively. The notation Gen+k-obj represents the GenO scheme applied to k-obj analysis, and Gen+k-type represents the GenT scheme applied to k-type analysis.

We evaluate the 18 Benchmarks in Table 1, including the popular DaCapo suite (top half of the table) and 7 popular open source programs (bottom half of the table). All experiments are conducted on an Intel Core(TM) i5-10210U laptop (1.6GHz) with 40 GB of RAM, running Unbuntu 20.04.01. As in previous work [23, 32, 33], the JDK version is JDK6 (1.6.0_30) and we set a time budget of 90 minutes in analyzing each benchmark. We run each benchmark 5 times and report the average analysis time of the 5 runs.

Our evaluation answers the following research questions:

- RQ1. How extensive is generics used in real-world applications?
- RQ2. Can generic-sensitivity improve precision over existing context-sensitive approaches?
- RQ3. Can generic-sensitivity improve efficiency over existing context-sensitive approaches?
- RQ4. Does generic-sensitivity offer a better trade-off than standard context-sensitive analyses?

4.1 RQ1. Generic Usages

Table 1 summarizes the generic usages in each benchmark. We separate the usages in application code with those in JDK libraries which are transitively invoked by applications. As shown in Table 1, there are extensive usages of generics: `findbugs` has the largest number of generic object allocations (455) and `antlr4` has the largest number of generic method invocations (1,213). Although some application, e.g., `antlr`, uses generics infrequently. Its underlying JDK library makes extensive usages of generics, suggesting the necessity of an optimized context-sensitive pointer analysis targeting generics.

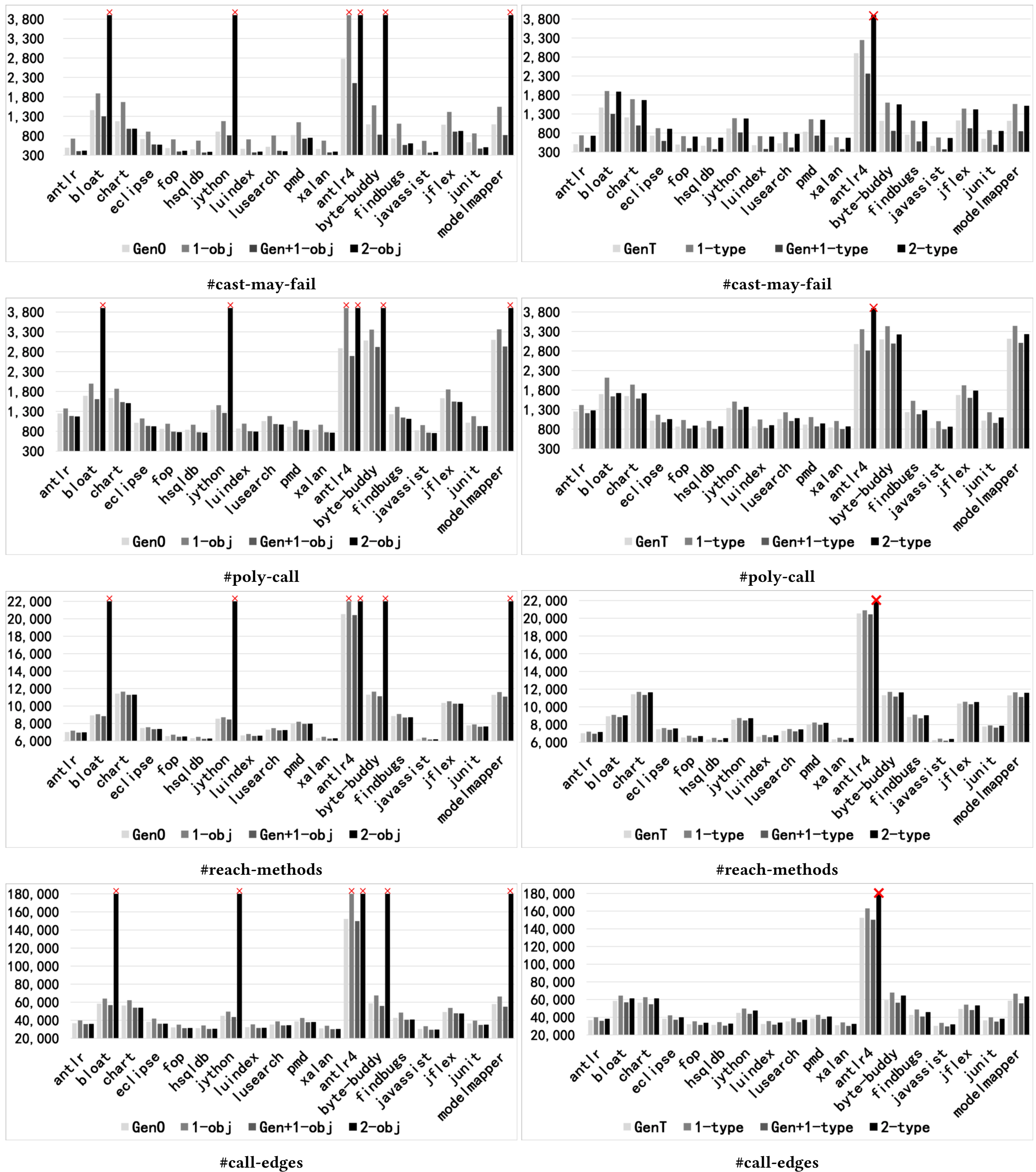
Figure 7 depicts the percentages of generic usages with actual type arguments, including those usages where our simple conservative type inference analysis (Section 3) can infer actual type arguments. The number of actual types inferred is small. As shown in Figure 7, the percentages are quite low for DaCapo Benchmarks. The reason is that DaCapo is released only a few years after generics being introduced into Java, and many Java applications at that time did not use the new generic feature (i.e., instantiating generics with actual type parameters). The percentage is much higher (>%70) for the 7 open-source applications, showing that new applications commonly use modern generic features supported by the language.

Comparing `antlr4` to its earlier version `antlr`, there are much more generic usages in `antlr4`, confirming that Java generics is widely adopted in modern Java applications.

Discussion. Generics is extensively used in modern Java applications and the underlying JDK library, suggesting the necessity to develop customized context-sensitive pointer analysis for generics.

4.2 RQ2. Precision

Following [35, 47, 49], we measure the precision of context-sensitive analyses using the four metrics: #call-edges (number of call graph



#cast-may-fail #cast-may-fail

#poly-call #poly-call

#reach-methods #reach-methods

#call-edges #call-edges

Figure 8: Precision metrics, the lower the better. Those timeout cases are outlined with cross.

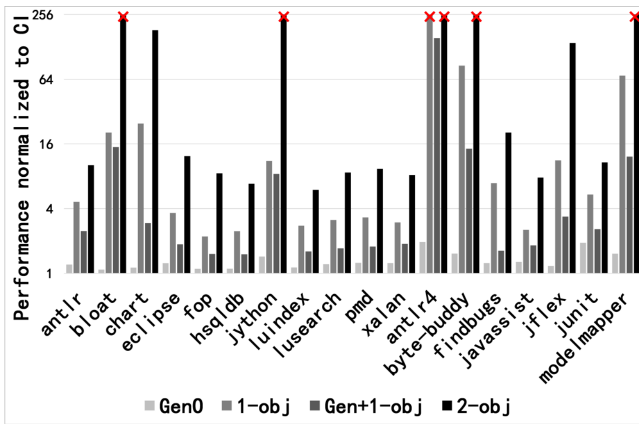


Figure 9: Performance of object sensitive group.

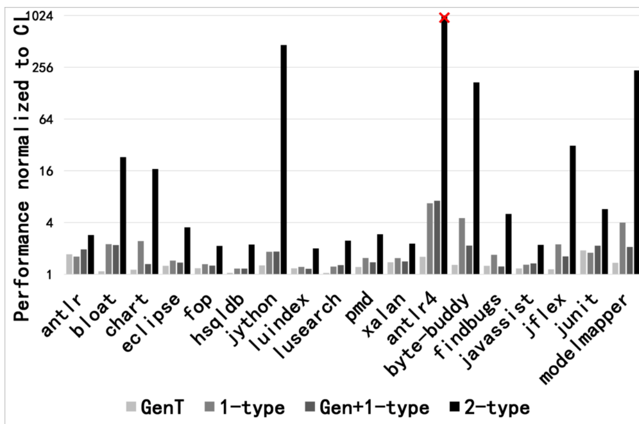


Figure 10: Performance of type sensitive group.

edges), #reach-methods (number of reachable methods), #poly-call (number of polymorhpic calls discovered), and #cast-may-fail (number of cast operations that may fail). As their names suggest, these metrics are obtained by different client applications of context-sensitive pointer analyses. All client applications are sound. Hence, for all the metrics, lower is better.

Since our approach introduces extra context-elements on top of standard k-obj or k-type analyses, Gen+k-obj (Gen+k-type) is always more precise than k-obj (k-type) analysis. Figure 8 compares precision metrics in 2 groups: object-sensitive group (Gen0, 1-Obj, Gen+1-Obj, 2-Obj) and type-sensitive group (GenT, 1-type, Gen+1-type, 2-type). The left half of Figure 8 shows results for object-sensitive group, and results of type-sensitive group are given in the right half.

Under the given 90 minutes budget, 2-Obj analysis fails to process the 5 benchmarks: `bloat`, `jython`, `antlr4`, `byte-buddy`, and `modelmaper`. Both 1-obj and 2-type analyses timeout on `antlr4`. Those timeout cases are outlined with crosses in Figure 8.

Object-sensitive group. Gen0, where only generic objects and methods are analyzed context-sensitively, is noticeably more precise than 1-obj for all metrics across all benchmarks. 2-Obj is more precise than Gen0 in those benchmarks that it runs to finish: for #cast-may-fail, #poly-call, #reach-methods, and #call-edge, the ratio of the number reported by Gen0 against that reported by 2-obj is 120.34%, 109.65%, 101.14%, and 103.53%, respectively. Gen+1-obj successfully analyzes all benchmarks without timeouts, and it achieves slightly better precision than 2-Obj, reporting 97.87%, 101.85%, 99.92%, and 100.10% of the number reported by 2-Obj for the 4 metrics, respectively.

Type-sensitive group. Gen+1-type is by far the most precise variant in the group, reporting 59.1%, 92.7%, 97.3%, and 92.1% of the number reported by 2-type for the above 4 metrics, respectively. Surprisingly, GenT also achieves better precision than than 2-type, reporting 72.98%, 96.81%, 98.27%, and 94.90% of the number reported by 2-type, respectively.

Discussion. Generic sensitivity can significantly improve precision for both object-sensitivity and type-sensitivity: a Gen+k-Obj analysis can achieve similar or better precision as a k+1-obj analysis. For type-sensitivity, the precision gains are even more significant with GenT outperforming 2-type analysis in precision.

4.3 RQ3. Performance

Figure 9 and Figure 10 compare analysis times for the object-sensitive group and type-sensitive group, where analysis times are normalized to the context-insensitive analysis (CI). As shown in Figure 9, 2-Obj timeouts for 5 benchmarks: `bloat`, `jython`, `antlr4`, `byte-buddy` and `modelmapper`. Comparing Gen+1-obj to 1-obj, Gen+1-obj achieves an average speed up of 1.8 ×, despite the fact that it is much more precise. Compared to 2-obj with similar precision, Gen+1-obj achieves a speed up of 62 × for `chart`, with an average speed up of 12.6 × for the 13 applications that 2-obj run to completion. Similarly, in Figure 10, Gen+1-type also achieves noticeably better performance than 1-type, with an average performance improvement of 20%.

Discussion. Although Gen+k-obj (Gen+k-type) introduces extra context elements to k-obj (k-type) analysis, the performance gain brought by more precise results can often compensate for the cost of introduced extra context elements. As an evidence, the 2 generic sensitive approaches Gen+1-obj and Gen+1-type outperforms 1-obj and 1-type, respectively.

4.4 RQ4. Precision and Performance Trade-off

Figure 11 depicts the performance and precision spectrum for an array of 11 pointer analysis variants. The figure plots precision in #reach-methods metric (with other precision metrics showing similar results) against analysis time over a set of 9 benchmarks in DaCapo. The other 2 benchmarks, `bloat` and `jython`, are not included in the graph since both 2-obj analysis and Gen+2-obj analysis fail to analyze them.

In Figure 11, lower numbers are better on both axes. Hence, analyses in the bottom left corner are superior in both precision and performance. As shown in the graph, the 3 variants Gen+1-obj, Gen+1-type, and Gen+2-type achieve overall best trade-offs between precision and performance. The most precise analysis is Gen+2-obj. However, its performance is similar to 2-Obj and

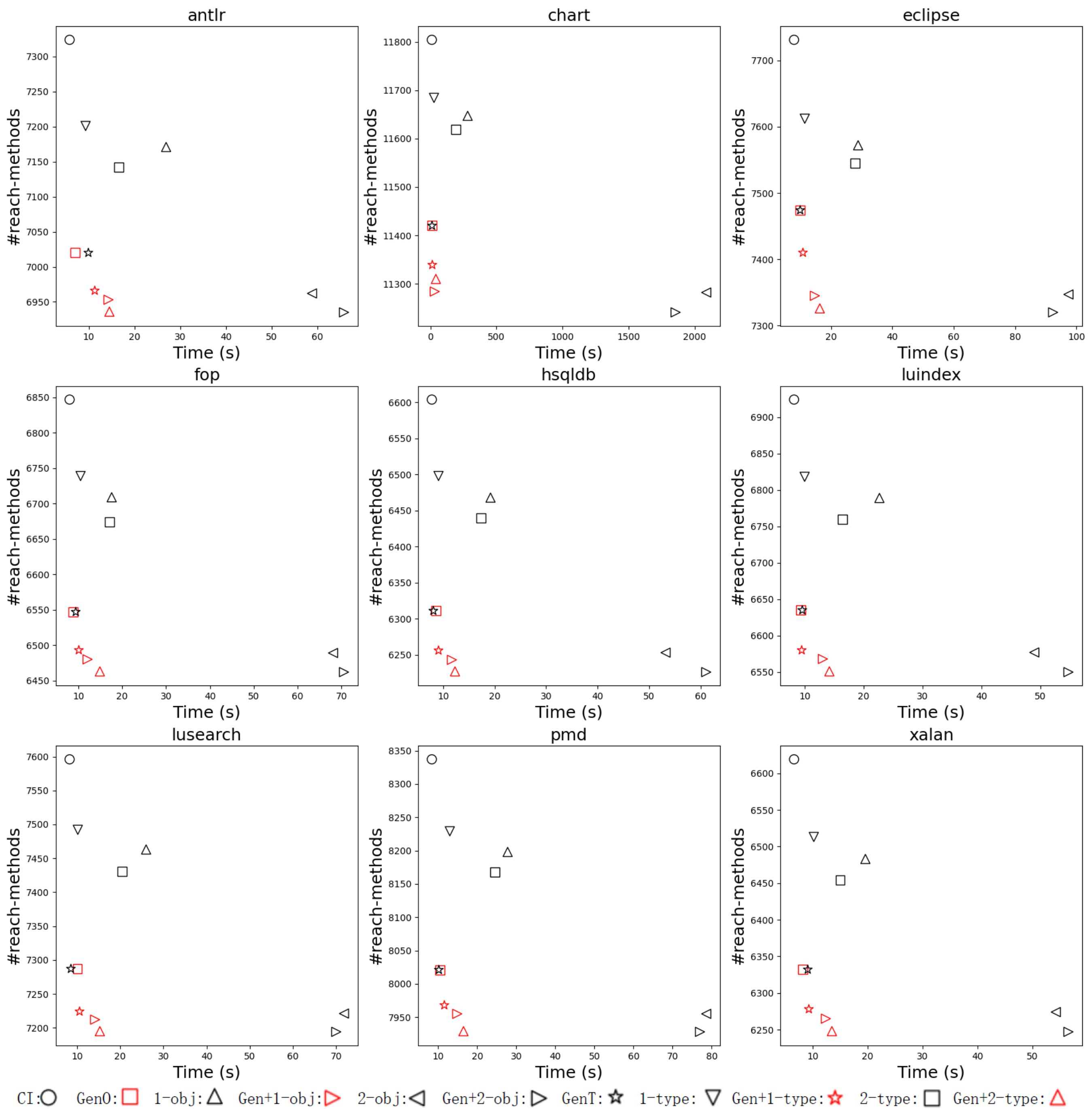


Figure 11: Precision(#reach-methods)/Performance spectrum for DaCapo benchmarks. Lower is better along both axes.

both are significantly slower than the other variants. For #reach-methods, Gen+2-type achieves similar precision to Gen+2-obj, with significant performance improvements. Let us compare Gen+1-obj with 2-Obj, Gen+1-obj is much faster and it is also more precise than 2-obj for all benchmarks, except for `hsqldb`. Between Gen+1-type and Gen+1-obj, Gen+1-type is slightly faster for all benchmarks but incur significant precision loss for `luindex`.

Discussion. Generic sensitivity offers a good solution in balancing precision and performance. The three variants Gen+1-obj, Gen+1-type, and Gen+2-type achieve overall best precision and performance trade-offs.

5 RELATED WORK

Context-sensitive pointer analysis for Java has been extensively studied in the literature. There are three mainstream variants of context sensitivity:k-object sensitivity, k-type sensitivity and k-call-site sensitivity. In addition to the above three variants, the work [23] proposes a hybrid approach which applies object sensitivity to instance method invocations and call-site sensitivity to static method invocations. The hybrid approach is superior to pure object-sensitivity since static methods don't have receiver objects. Jonas and Welf [36] use the points-to set of receiver object to approximate a context [36]. In [39], the cartesian product of the points-to sets of all arguments (including this) are used to symbolically represent a context. Our generic customization scheme can be adapted to the above context-sensitive variants as well.

Selective context-sensitivity has gained much attention recently since if may offer a better trade-off between precision and efficiency, where methods can be analyzed with different context elements and depths. Researchers have applied manually-selected metrics and heuristics [14, 44, 52], or learning-based approaches [19–22] to selectively analyze a subset of methods context-sensitivity. Scaler [33] determines whether to analyze a method context-sensitively or not based on an estimation of its potential memory consumption. Zipper [32] introduces 3 kinds of value-flow patterns to identify precision critical methods, and those patterns can be computed by solving a graph reachablility problem on a precision flow graph. Zipper achieves significant performance improvements, but does not guarantee precision. Eagle [35] performs a CFL-reachability-based pre-analysis to enable selective context-sensitivity in k-obj, while guaranteeing precision. Turner [16] finds a sweet spot between Zipper and Eagle, which enables k-obj analysis to run significantly faster than Eagle while achieve notably better precision than Zipper. CONCH finds context-dependent objects, avoiding contexts bloating. Baton [47] proposes a Unit-Relay framework by collectively integrating different context selectors. Instead of selecting which methods to be context-sensitive analyzed, Bean [48] makes k-obj sensitive analysis more precise by skipping those unhelpful context elements. In [49], Tan et al. apply a pre-analysis to selectively apply type-based abstractions to heap objects, provided that such approximation does not affect the precision of type-based clients, e.g., call graph construction. Compared to the above selective sensitive approaches, we propose a context customization scheme targeting generics and our approach can be applied together with the above optimization techniques, to further improve performance and efficiency.

There have been numerous approaches leveraging efficient data structure implementation to scale context-sensitive pointer analysis, e.g., using bit vectors or bit sets [4, 26], using binary decision diagrams (BDDs) [24, 25, 53], using geometric encoding techniques [54], or graph systems [51]. The work [2, 9] investigated on how to manually model semantics of data structures, to effectively speed up an analysis by omitting their complicated implementation details. Compared to the above approaches, we target a different problem on how to effectively analyze generics in a context-sensitive manner and our approach can also benefit from the above optimization techniques.

6 CONCLUSION

We introduce generic-sensitive pointer analysis, a new context customization scheme designed for generics. To the best of our knowledge, this is the first context-sensitive pointer analysis targeting generics. We design our context customization scheme based on the observation that generic instantiate location is key context elements, and apply generic customization to two mainstream context-sensitive variants: object sensitivity and type sensitive. Experimental results demonstrate that generic context customization can significantly improve performance and precision: a `Gen+1-obj` analysis can achieve overall better precision than 2-obj analysis, with a better performance than 1-obj analysis. The average speedup over 1-obj analysis is 1.8×, and is 12.6× (up to 62 × for `chart`) over 2-obj analysis.

7 DATA-AVAILABILITY STATEMENT

The tool is available at https://doi.org/10.6084/m9.figshare.20486556.v2 [28].

ACKNOWLEDGMENTS

We thank all anonymous reviewers for their valuable inputs. This work is supported by the National Science Foundation of China (NSFC) under grant number 62132020, the Alibaba Group through the Alibaba Innovative Research Program, and the CCF-Ant group research Foundation.

REFERENCES

[1] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph.D. Dissertation. Citeseer.

[2] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. 2020. Static analysis of Java enterprise applications: frameworks and caches, the elephants in the room. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* 794–807.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[4] Mohamad Barbar and Yulei Sui. 2021. Compacting points-to sets through object clustering. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27.

[5] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 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.* 169–190.

[6] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications.* 243–262.

[7] Wei-Ngan Chin, Florin Craciun, Siau-Cheng Khoo, and Corneliu Popeea. 2006. A flow-based approach for variant parametric types. *ACM SIGPLAN Notices* 41, 10 (2006), 273–290.

[8] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. 2014. Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 779–790. https://doi.org/10.1145/2568225.2568295

[9] Pratik Fegade and Christian Wimmer. 2020. Scalable Pointer Analysis of Data Structures Using Semantic Models. In *Proceedings of the 29th International Conference on Compiler Construction* (San Diego, CA, USA) *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 39–50. https://doi.org/10.1145/3377555.3377885

[10] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2016. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 377–396.

[11] Robert Fuhrer, Frank Tip, Adam Kieżun, Julian Dolby, and Markus Keller. 2005. Efficiently refactoring Java applications to use generic libraries. In *European Conference on Object-Oriented Programming*. Springer, 71–96.

[12] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information flow analysis of android applications in droidsafe.. In *NDSS*, Vol. 15. 110.

[13] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: unified points-to and taint analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28.

[14] Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. 2017. An efficient tunable selective points-to analysis for large codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*. 13–18.

[15] Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. 2019. Performance-Boosting Sparsification of the IFDS Algorithm with Applications to Taint Analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 267–279. https://doi.org/10.1109/ASE.2019.00034

[16] Dongjie He, Jingbo Lu, and Jingling Xue. 2021. Context Debloating for Object-Sensitive Pointer Analysis. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 79–91.

[17] Michael Hind. 2001. Pointer analysis: Haven't we solved this problem yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 54–61.

[18] IBM. [n.d.]. *WALA: T.J. Watson Libraries for Analysis*. Retrieved September 7, 2021 from http://wala.sourceforge.net

[19] Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and scalable points-to analysis via data-driven context tunneling. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29.

[20] Minseok Jeon, Myungho Lee, and Hakjoo Oh. 2020. Learning graph-based heuristics for pointer analysis without handcrafting application-specific features. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[21] Minseok Jeon and Hakjoo Oh. 2022. Return of CFA: call-site sensitivity can be superior to object sensitivity even for object-oriented programs. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–29.

[22] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28.

[23] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. *ACM SIGPLAN Notices* 48, 6 (2013), 423–434.

[24] Ondřej Lhoták, Stephen Curial, and José Nelson Amaral. 2007. Using ZBDDs in points-to analysis. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 338–352.

[25] Ondřej Lhoták, Stephen Curial, and Jose Nelson Amaral. 2009. Using XBDDs and ZBDDs in points-to analysis. *Software: Practice and Experience* 39, 2 (2009), 163–188.

[26] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*. Springer, 153–169.

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

[28] Haofeng Li, Jie Lu, Haining Meng, Liqing Cao, Yongheng Huang, Lian Li, and Lin Gao. 2022. Generic Sensitivity: Customizing Context-Sensitive Pointer Analysis for Generics. (8 2022). https://doi.org/10.6084/m9.figshare.20486556.v1

[29] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2010. Practical and Effective Symbolic Analysis for Buffer Overflow Detection. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Santa Fe, New Mexico, USA) *(FSE '10)*. Association for Computing Machinery, New York, NY, USA, 317–326. https://doi.org/10.1145/1882291.1882338

[30] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the Performance of Flow-Sensitive Points-to Analysis Using Value Flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 343–353. https://doi.org/10.1145/2025113.2025160

[31] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2013. Precise and scalable context-sensitive pointer analysis via value flow graph. *ACM SIGPLAN Notices* 48, 11 (2013), 85–96.

[32] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided context sensitivity for pointer analysis. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29.

[33] 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*. 129–140.

[34] Chen Liu, Jie Lu, Guangwei Li, Ting Yuan, Lian Li, Feng Tan, Jun Yang, Liang You, and Jingling Xue. 2021. Detecting TensorFlow Program Bugs in Real-World Industrial Environment. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 55–66. https://doi.org/10.1109/ASE51524.2021.9678891

[35] Jingbo Lu and Jingling Xue. 2019. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[36] Jonas Lundberg and Welf Löwe. 2012. Points-to Analysis: A Fine-Grained Evaluation. *J. Univers. Comput. Sci.* 18, 20 (2012), 2851–2878.

[37] Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. 1–11.

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

[39] 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*. 31–36.

[40] Micha Sharir, Amir Pnueli, et al. 1978. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences ....

[41] Olin Grigsby Shivers. 1991. *Control-flow analysis of higher-order languages or taming lambda*. Carnegie Mellon University.

[42] Yannis Smaragdakis and George Balatsouras. 2015. Pointer analysis. *Foundations and Trends in Programming Languages* 2, 1 (2015), 1–69.

[43] 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*. 17–30.

[44] 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*. 485–495.

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

[46] Yulei Sui, Sen Ye, Jingling Xue, and Jie Zhang. 2014. Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation. *Software: Practice and Experience* 44, 12 (2014), 1485–1510.

[47] Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. 2021. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27.

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

[49] 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*. 278–291.

[50] Frank Tip, Robert M Fuhrer, Adam Kieżun, Michael D Ernst, Ittai Balaban, and Bjorn De Sutter. 2011. Refactoring using type constraints. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 3 (2011), 1–47.

[51] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A Single-Machine Disk-Based Graph System for Interprocedural Static Analyses of Large-Scale Systems Code. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 389–404. https://doi.org/10.1145/3037697.3037744

[52] Shiyi Wei and Barbara G Ryder. 2015. Adaptive context-sensitive analysis for JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[53] John Whaley and Monica S Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation*. 131–144.

[54] Xiao Xiao and Charles Zhang. 2011. Geometric encoding: forging the high performance context sensitive points-to analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 188–198.

[55] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. 2010. Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. 218–229.