Generic Sensitivity: Generics-Guided Context Sensitivity for Pointer Analysis

Haofeng Li, Tian Tan, Yue Li, Jie Lu, Haining Meng, Liqing Cao, Yongheng Huang, Lian Li, Lin Gao, Peng Di, Liang Lin, and ChenXi Cui

***Abstract*—Generic programming has found widespread application in object-oriented languages like Java. However, existing context-sensitive pointer analyses fail to leverage the benefits of generic programming. 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 generated in a context-sensitive manner throughout the analysis process. We have implemented various variants of generic-sensitive analysis in WALA and conducted extensive experiments to compare it with state-of-the-art approaches, including both traditional and selective context-sensitivity methods. The evaluation results demonstrate that generic sensitivity effectively enhances existing context-sensitivity approaches, striking a new balance between efficiency and precision. For instance, it enables a 1-object-sensitive analysis to achieve overall better precision compared to a 2-object-sensitive analysis, with an average speedup of 12.6 times (up to 62 times).**

***Index Terms*—Pointer analysis, generic programming, context sensitivity.**

I. 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 [1], [2], [3], compiler optimization [4], security analysis [5], [6], [7], [8], 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 [9], [10], [11], [12], [13], and one of the key mechanism to improve precision is *context-sensitivity* [14], [15], [16], [17], [18], [19]. 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), types of receiver objects or types that contain the methods which allocate receiver objects (two strategies of k-type-sensitivity defined by Smaragdakis et al. [18]). For object-oriented programs, object-sensitivity is believed to be better than call-site-sensitivity in achieving precision and efficiency [14], [15], and type-sensitivity is regarded as a more efficient, but less precise alternative to object-sensitivity [18].

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 [20], [21].

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 [22] 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. And type variables can also serve as type arguments at instantiation sites. Taking the following code snippet as an example, type variable K

```
class C1<K> ... {
  void foo() {
    C2<K> c = new C2<>();
  }
}
class C2<V> {}
```

Manuscript received 15 July 2023; revised 1 March 2024; accepted 8 March 2024. Date of publication 12 April 2024; date of current version 16 May 2024. This work was supported in part by the National Key R&D Program of China under Grant 2022YFB3103900, in part by the National Natural Science Foundation of China (NSFC) under Grant 62132020, in part by the Alibaba Group through the Alibaba Innovative Research Program, and in part by the CCF-Ant Group Research Foundation. Recommended for acceptance by M. Pradel. *(Corresponding authors: Haofeng Li; Lian Li.)*

Haofeng Li and Jie Lu are with SKLP, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100049, China (e-mail: lihaofeng@ict.ac.cn).

Tian Tan and Yue Li are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China.

Haining Meng, Liqing Cao, and Yongheng Huang are with SKLP, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100049, China, and also with the University of Chinese Academy of Sciences, Beijing 101408, China.

Lian Li is with SKLP, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100049, China, also with Zhongguancun Laboratory, China, and also with the University of Chinese Academy of Sciences, Beijing 101408, China (e-mail: lianli@ict.ac.cn).

Lin Gao is with TianqiSoft, Inc., Beijing 100142, China.

Peng Di is with Ant Group, Hangzhou 310013, China.

Liang Lin and ChenXi Cui are with Alibaba Group, Beijing 100012, China.

Digital Object Identifier 10.1109/TSE.2024.3377645

is used as type argument when instantiating generic class `C2` at line 3. Therefore, both of type variables `K` and `V` represent the same concrete type. In this case, the concrete type (corresponding to type variables `K` and `V`) can distinguish different contexts of the methods constrained by `K` and `V` (i.e., the methods or their declaring classes declared with the information of generic types).

Our context customization scheme for generics is based on the observation that *generic instantiation sites*, i.e., locations instantiating generic with concrete types, can be served as key context elements, but not preserved in existing context-sensitive analyses, and thereby often leading to poor efficiency and precision. As a result, we propose *generic sensitivity*: instead of always picking the most recent traditional context elements (i.e., call-sites, objects, types), we keep generic instantiation sites as part of contexts 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., inner generic classes/objects defined within other generic classes) or generic methods (by calling other generic methods). So, the biggest challenge lies in accurately determining the generic instantiation sites of type variables. In our approach, we address this challenge by augmenting contexts with generic instantiation information, and propagate it along type variables efficiently. It should be noted that regarding the elements that form the contexts, both generic sensitivity and object sensitivity utilize allocation sites to build their respective contexts. Nevertheless, the approaches these two techniques adopt to create the contexts differ significantly. According to [14], [15], object sensitivity builds its contexts using the receiver objects at the call sites, whereas generic sensitivity builds its contexts using the sites of generic instantiation.

We have implemented our approach in WALA [23] and evaluated it against a set of 18 real-world applications, including the DACAPO benchmark suite [24] and another 7 popular open-source applications. We conduct comprehensive experiments to compare generic sensitivity with state of the arts, including both traditional and selective context-sensitivity approaches. The evaluation results show that generic sensitivity effectively facilitates existing context-sensitivity approaches in achieving a new trade-off between efficiency and precision. For examples, it 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`). Additionally, it can also contribute to enhancing the efficiency and precision of selective context-sensitivity approaches like ZIPPER [25] and ZIPPER-E [26], which are widely recognized as the state-of-the-art selective context-sensitive pointer analysis.

To summarize, the paper makes the following contributions:

- We present generic 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 generic programming.
- We explain how to apply generic sensitivity to two mainstream context-sensitive variants: k-object-sensitivity and k-type-sensitivity.
- We have implemented different variants of generic sensitive pointer analysis in WALA [23] 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 generic sensitivity effectively enhances both traditional and selective context-sensitivity approaches, striking a new trade-off between efficiency and precision.

This article extends and improves upon the paper authored by Li et al. [27], which was presented at the Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022). In comparison, this article includes significant extensions, expanding the content by approximately six additional pages in a two-column format. The key enhancements are summarized as follows:

- We introduced a local analysis which can infer actual type parameter corresponding to generic type (Section III-A).
- We presented k-generic-sensitivity which extends original 1-generic-sensitivity to support arbitrary depth of context (Section IV-C).
- We evaluated generic sensitivity with or without applying ZIPPER and ZIPPER-E (Section V-G).
- We additionally implemented k-generic-sensitivity in WALA and evaluated its precision and efficiency in our benchmarks (Section V-H).

The rest of the paper is organized as follows. Section II motivates our approach with an example and highlights its key challenges. Section III formalizes context representation of generic sensitivity. Section IV formally describes generic sensitivity and explains how it can be adapted to object-sensitivity and type-sensitivity. Section IV-C illustrates how to extend our generic sensitivity when the depth of context is more than 1. We evaluate the effectiveness and efficiency of generic sensitivity in Section V. Section VI reviews related work and Section VII concludes this paper.

II. MOTIVATION

We first give a brief introduction on context-sensitive pointer analysis (Section II-A). Then we illustrate the limitations of existing context-sensitive pointer analysis in analyzing generics with an example (Section II-B). Finally, we motivate generic sensitivity, and discuss its main challenges (Section II-C).

A. 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 instances 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 no larger 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 [28], [29]. 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 $(1 < i \leq k-1)$ 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 less precise results.

B. A Motivating Example

Let us study context-sensitive pointer analysis with an example in Fig. 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 $pts([O_1],\text{n}) = \{O_4\}$ and $pts([O_2],\text{n}) = \{O_4\}$. Then in the constructor of `Node` (lines 27-30), since O_4 is the only receiver object, we get $pts([O_4],\texttt{key}) = \{\text{“A”},\text{“B”}\}$ 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.

```
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 }
```

Fig. 1. Simplified code example of `java.util.HashMap`.

This 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 $pts([O_1],\text{n}) = \{(O_1, O_4)\}$ and $pts([O_2],\text{n}) = \{(O_2, O_4)\}$, 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 $pts([O_2, O_4],\texttt{value}) = \{O_B\}$. 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 the 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 V-A, 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>`.

For clarity, we simplify the example in Fig. 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. Various 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 [25], [30] also summarized numerous scenarios where a deeper context ($\geq$3) is required. However, as the number of contexts grows exponentially with the depth, it is often infeasible to scale 3-obj analysis to real-world applications [21], [31], [32].

C. Generic Sensitivity

Generics enable us to create standardized algorithms for processing various types of data to enhance the reusability of algorithms. Many sophisticated algorithms necessitate encapsulating and integrating numerous classes and methods for practical deployment. So these classes and methods are constrained by identical generic types within each call stack. In Fig. 1, `Node` offers support for `HashMap` and consistently upholds the equivalent generic type constraint as `HashMap`. Hence, the key to ensure precision is to keep the *instantiation location*, i.e., location instantiating generic type parameters with concrete types, as part of context for generics. As such, distinct pointer values flow into/from generic methods and generic objects can be effectively identified.

In our example in Fig. 1, line 2 and line 7 instantiate the generic class `HashMap` with actual types. So, *generic instantiation locations* of `HashMap` are O_1 and O_2 respectively. Therefore the *generic instantiation locations* 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. And then actual types are passed as type variables: $\langle K, V\rangle$ of `HashMap` to instantiate `Node` at line 17. So, *instantiation locations* O_1 and O_2 can not only differentiate the methods in class `HashMap` under different call-stack but also can distinguish the methods in `Node`. In other words, *generic instantiation locations* of the call to constructor/`getValue` of `Node` at line 17/22 are O_1 and O_2 respectively. Hence, the call to constructor/`getValue` of `Node` at different call-stacks can be distinguished only using contexts O_1 and O_2. As a result, with generic-sensitivity, we can compute the same precise result as 2-obj analysis, but with less cost by avoid computing points-to information under potentially more contexts: $pts(O_1,\texttt{key}) = \{\text{“A”}\}$, $pts(O_2,\texttt{key}) = \{\text{“B”}\}$, $pts(O_1,\texttt{value}) = \{O_A\}$, and $pts(O_2,\texttt{value}) = \{O_B\}$, where O_1 and O_2 serve as the generic instantiation locations in this case.

Someone may wonder if we can discard the receiver objects of generic classes and use their heap contexts as the contexts of target methods when the current methods are within generic classes? It seems feasible for the code in Fig. 1. For example, when we analyze the code at line 22 in `get` method whose declaring class is generic class `HashMap`, we discard the receiver object O_4 and use its heap contexts $[O_1]$ and $[O_2]$ as contexts. However, this approach does not work for generic methods. As shown in Fig. 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 (to `A`) 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`. In our generic-sensitivity approach, the instantiation location of `E`, represented by O_2 at line 3, is chosen as the context for analyzing the method call `n.bar` at line 11. However, if we were to adopt the previously mentioned approach of omitting receiver objects of generic classes, the context for `n.bar` would be determined as O_1. As a result, it would be the same context as `m.bar`, failing to distinguish between contexts.

```
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 }
```

Fig. 2. Example of generics.

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, as will be explained in Section III.

III. Preliminaries

In generic sensitivity, we precisely track propagation of type variables by augmenting context with generic instantiation locations, which are generated context-sensitively during the analysis. This section will formalize the standard context sensitivity and offer necessary extensions for generics.

A. 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 abstract 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

With Actual Type Arguments

```
1  class C {
2    void foo() {
3      Set<A> s = new HashSet<>();
4    }
5  }
```

(a)

Without Actual Type Arguments

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

(b)

Fig. 3. Generic instatiation in Java.

(Fig. 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 Fig. 3 (b), `s` is created at line 3 with type `HashSet<Object>`. At line 4, an object with type `A` is firstly created and implicitly cast to `Object`, before it is put in `s`.

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 interprocedural pre-analysis to infer actual type arguments of generics as [33], [34], [35]. 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.

In code of Java, actual type parameters are only declared in the signatures of formal parameters (i.e., $foo(Set\,\langle A\rangle\,x)$), the signatures of fields (i.e., $Set\,\langle A\rangle\,f;$) and type signature of partial local variables (i.e., $Set\,\langle A\rangle\,x = ...$). However, there are cases where it is not always possible to obtain the actual type parameters directly at the instantiation sites of generic classes or generic methods. Taking the following code snippet as an example, to get the concrete type corresponding to type variable `T` at line 4, we need to infer the actual type

```
1 void main() {
2     C<A> v1 = new C<>();
3     C v2 = v1;
4     foo(v2);
5 }
6 void <T> foo(C<T> p){}
```

parameter of variable v_2 which can only be found at signature of variable v_1. To infer the actual type parameters, we designed a local analysis. Table I shows the constraints of our local inference. We use $foo(Set\,\langle A\rangle\,x)$ to represent method foo with formal parameter x whose signature is $Set\,\langle A\rangle$. For clarity, we define method foo with one parameter only. Similarly, field signature and variable signature are defined at the rules of FIELD DECLARATION and VARIABLE DECLARATION respectively. We use R(x) to represent the inferred results (the mappings from type variables to actual type parameters) where x represents variable or field used in the current method. It is forbidden to assign several actual type parameters to a type variable. So, R(x) is singleton. Without loss of generality, we consider a simplified subset of Java with six canonical statements in Table I:

TABLE I
ACTUAL TYPE PARAMETER INFERENCE. THE GENERIC TYPE OF CLASS SET IS E

Kind	Statements in Method: $foo\ (Set\,\langle A\rangle\ \ x)$	Constraints
FORMAL PARAMETER	x	R(x) = $\{E : A\}$
FIELD DECLARATION	$Set\,\langle A\rangle\,f;$	R(f) = $\{E : A\}$
VARIABLE DECLARATION	$Set\,\langle A\rangle\,x = ...$	R(x) = $\{E : A\}$
ASSIGN	$x = y;$	R(x) = R(y)
STORE	$y.f = x;$	R(x) = R(f)
LOAD	$x = y.f;$	

- FORMAL PARAMETER, FIELD DECLARATION and VARIABLE DECLARATION. Actual type parameters can be captured directly by parsing the method signature, field signature or variable type signature if they are declared in these signatures.
- ASSIGN. If a variable can be assigned to another, both of them should contain the same actual type parameter. Otherwise, compilation errors will occur.
- STORE and LOAD. The actual type parameter can be propagated to local variables by field access statements, so, we maintain consistent actual type parameter on both sides of the statement.

According to the constraints in Table I, actual type parameters can be inferred if they are declared. However, not all actual type parameters will be explicitly declared in the Java code. To handle such conditions, we designed a local analysis to infer type parameters (Definition III.1).

Definition III.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 Fig. 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 Fig. 3(b), the actual type parameter

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	$l : x = v_0.m'\ \langle \mathcal{T} : A \rangle\ (v_1)$

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

of variable s at line 3 cannot be inferred (due to an escaped/returned variable). In such scenario, we use a pseudo type T_3 to instantiate `s`, i.e., the statement at line 3 is regarded as `Set` $< T_3 >$ `s = new HashSet()`. This allows us to label this generic context for potential context distinction. In this way, the pseudo type T_3, denoted for the current instantiation site, can be seen as a form of object sensitivity, and is employed when inferring the actual type parameters is not feasible. As such, we are effectively applying object-sensitivity-like approach in analyzing generics since each instantiation location is regarded as a distinct type even though sometimes it is not possible to deduce the actual type parameters.

B. A Simplified Java Language

Without loss of generality, we consider a simplified subset of Java, with five types of labeled statements in Fig. 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 "$x = \mathtt{new}C(...)$" in Java is modeled as "$x = \mathtt{new}C; x.\langle init \rangle\,(...)$", where $\langle init \rangle\,(...)$ 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

$$\frac{\begin{array}{c} l : x = new\ C \quad m = \mathbf{methodOf}(l) \\ ctx \in \mathbf{methodCtx}(m) \quad hctx = ctx_{k-1} \end{array}}{(O_l, hctx) \in \mathbf{pts}(x, ctx)} \quad \text{[New]}$$

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

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

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

$$\frac{\begin{array}{c} l : x = a_0.f(a_1) \quad m = \mathbf{methodOf}(l) \\ (O_0, hctx) \in \mathbf{pts}(a_0, ctx) \\ ctx' = O_0 +\!\!+ hctx \quad m' = \mathbf{dispatch}(f, O_0) \end{array}}{\begin{array}{c} ctx' \in \mathbf{methodCtx}(m') \ \ (O_0, hctx) \in \mathbf{pts}(this^{m'}, ctx') \\ \mathbf{pts}(a_1, ctx) \subseteq \mathbf{pts}(p^{m'}, ctx') \\ \mathbf{pts}(ret^{m'}, ctx') \subseteq \mathbf{pts}(x, ctx) \end{array}} \quad \text{[Call]}$$

Fig. 5. Rules for object sensitivity.

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

IV. Generic Sensitivity

In this section, we will illustrate context customization scheme formally.

A. 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 three functions:

$$\mathbf{Gen}(G, \mathcal{T}, A, O_l) = \begin{cases} \emptyset & \mathcal{T} = Nil \\ [\mathcal{T} \rightarrow O_l] & \mathcal{T} \neq Nil \land A \notin G \\ [\mathcal{T} \rightarrow G(A)] & \mathcal{T} \neq Nil \land A \in G \end{cases}$$

$$\mathbf{Append}(G, G_m, \mathcal{T}, A, O_l) = \begin{cases} G & \mathcal{T} = Nil \\ G \uplus [\mathcal{T} \rightarrow O_l] & \mathcal{T} \neq Nil \land A \notin G \uplus G_m \\ G \uplus [\mathcal{T} \rightarrow G \uplus G_m(A)] & \mathcal{T} \neq Nil \land A \in G \uplus G_m \end{cases}$$

$$\mathbf{Ctx}(ctx, O_0, G) = \langle O_0 +\!\!+ ctx, G \rangle$$

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

The first two functions are used to generate G for NEW and CALL statements, and the last function is used to construct contexts by combining standard context and generic context. Fig. 6 gives the rules for analyzing New and Call statements in Fig. 4. Except for [New] and [Call], the other 3 rules are same as standard k-obj analysis(in Fig. 5). We only list [NEW] and [CALL], since the other 3 rules are same as in Fig. 5

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

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

Fig. 6. Rules for k-Obj/k-type analysis with generic customization.

In [NEW], an abstract heap object $O_l \in \mathbb{H}$ is created. 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 generated by the **Gen** 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 set to $[\mathcal{T} \mapsto G(A)]$, enforcing that the actual generic instantiation location is always part of the context.

Note that the **Gen** 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], let O_0 be a receiver object of the method call with heap context $hctx = \langle c, G\rangle$ 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 by **Ctx** function 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 generated by the **Append** function, as follows.

- If f is a non-generic call, i.e., $\mathcal{T} = Nil$, 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 generated 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 generated 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 **Gen** 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, $ctx' \in \textbf{methodCtx}(m')$ shows how the context of a method are introduced. Initially, we have $\textbf{methodCtx}(\texttt{main}) = \{\langle [], \emptyset\rangle\}$.

Let us revisit the example in Fig. 2. A generic object O_1 is created at line 2. Hence, we have $(O_1, \langle [], \texttt{T} \mapsto O_1\rangle) \in pts(g, \langle [], \emptyset\rangle)$ ([NEW]). Line 4 invokes the generic method $\texttt{foo}\langle\texttt{E}\rangle$ 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 context $\langle [O_1], [\texttt{T} \mapsto O_1, \texttt{E} \mapsto O_2]\rangle$ ([CALL]). In `foo`, the object created at line 8 (O_3) is instantiated with type variable `T`. Hence, it has the generated 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.

B. Customizing Type Sensitivity

The k-type-sensitivity is a coarse approximation of the k-object-sensitivity, with a trade-off between precision and efficiency in favor of the latter. Similarly, we customize k-type-sensitivity to improve efficiency with sacrificing some precision.

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 **Gen**, **Append** and **Ctx** functions are defined as follows.

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

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

$$\textbf{Ctx}(ctx, O_0, G) = \langle \textbf{typeOf}(O_0) +\!\!+ ctx, G\rangle$$

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

Let us study the example in Fig. 2 again. A generic object O_1 is instantiated with actual type `A` at line 2. Hence, we have $(O_1, \langle [], \texttt{T} \mapsto \texttt{A}\rangle) \in pts(g, \langle [], \emptyset\rangle)$ ([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 context $\langle [\texttt{G}], [\texttt{T} \mapsto \texttt{A}, \texttt{E} \mapsto \texttt{B}]\rangle$ ([CALL]). In `foo`, the object created at line 8 (O_3) is instantiated with type variable `T`. Hence, it has the generated 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

```
1 public static void main(String[] args) {
2   Q<A> q1 = new Q<>();//O1
3   q1.putOrCreate("A", null);
4   Object v1 = q1.get("A");
5   A a = (A) v1;//cast may fail?
6
7   Q<B> q2 = new Q<>();//O2
8   q2.putOrCreate("B", new B());//OB
9   Object v2 = q2.get("B");
10  B b = (B) v2;//cast may fail?
11 }
12
13 class Q<E> ... {
14
15   private InnerQ<E> inner=new InnerQ();//OIQ
16   public void putOrCreate(String key, E e) {
17     return inner.putOrCreate(key, e);
18   }
19   public Object get(String key) {
20     return inner.get(key);
21   }
22   class InnerQ<E> ... {
23     private HashMap<String, Object> map = new
           HashMap<>();//OHM
24     public void putOrCreate(String key, E e) {
25       if (e == null) {
26         map.put(key, new A());//OA
27       } else {
28         map.put(key, e);
29       }
30     }
31     public Object get(String key) {
32       return map.get(key);
33     }
34   }
35 }
```

Fig. 7. Code example for k-generic sensitivity.

in our extended type analysis, the generic type and the recorded actual instantiated types form the complete instantiated type signatures for generics.

C. K-Limiting

Just like three mainstream variants of context-sensitivity (k-object sensitivity, k-type sensitivity and k-call-site sensitivity), the generic can be wrapped by another generic. So, we have to limit the depth of generic contexts to avoid unbearable number of contexts. However, we cannot increase the depth at every generic allocation since generic instantiation is less prevalent than traditional contexts in Java programs. Taking Fig. 1 as an example, the context depth of the constructor (line 17) of `Node` cannot be increased because the actual types (K, V) of `Node` are type variables, and actual instantiate locations can be retrieved according to heap context and context of current method. We will discuss how to extend our generic sensitivity when the depth of context is more than 1 by using an example in Fig. 7.

In Fig. 7, we defined a class `Q` with a generic type `E` (line 13-35). In class `Q`, there is a field `inner` which points to the `InnerQ` object $O_{\mathtt{IQ}}$ created at line 15. there are two methods `putOrCreate`(line 16-18) and `get`(line 19-21), both of which just invoke the methods in class `InnerQ`. There is a `HashMap` object $O_{\mathtt{HM}}$ being created at line 23 and assigned to field `map`. The object $O_{\mathtt{HM}}$ is instantiated with type arguments: `String` and `Object`. In method `putOrCreate`, an `A` object $O_{\mathtt{A}}$ (line 26) is created and putted in $O_{\mathtt{HM}}$ when the parameter `e` is `null`, otherwise parameter `e` is putted in $O_{\mathtt{HM}}$. The `get` method retrieves the corresponding object from `map`, then returns it(line 31-33).

Similar with the `main` method in Fig. 1, in the `main` method of Fig. 7, there are two `Q` objects: O_1 (line 2) and O_2(line 7). Because the second actual parameter of invoke `putOrCreate` at line3 is `null`, object $O_{\mathtt{A}}$ is putted into O_1 and then retrieved back via the `get` method at line 4. Similarly, object $O_{\mathtt{B}}$ is created and put into O_2 at line 8, then retrieved back at line 9. Because the second actual parameter of invoke `putOrCreate` at line 8 is not `null`, O_2 will not contain object $O_{\mathtt{A}}$. As a result, the two cast operations (line 5 and 10) will never fail. Because object $O_{\mathtt{A}}$ is created in a condition branch, we can distinguish it by using path-sensitivity.

Similar with the example in Fig. 1, line 2 and line 7 instantiate the generic class `Q` with actual types. Those actual types are passed as type variables of `Q` to instantiate `InnerQ` at line 15. Hence, the corresponding location (O_1 and O_2) should be regarded as the context in analyzing methods of class `InnerQ`. So, with 1-object-sensitivity, we can distinguish $O_{\mathtt{HM}}$ with different contexts at line 23, meanwhile, we can also distinguish $O_{\mathtt{A}}$ with different contexts at line 26. However, the type variables of $O_{\mathtt{HM}}$ are `String` and `Object`, the outermost corresponding location (O_1 and O_2) cannot be propagated into `HashMap` continuously. As a result, $O_{\mathtt{A}}$ and $O_{\mathtt{B}}$ will be confused in $O_{\mathtt{HM}}$, if the depth of context is set to 1.

If we set the context depth to 2, we combine the outer corresponding location (O_1 and O_2) and the inner corresponding location ($O_{\mathtt{HM}}$), and then propagate it into class `Node` in Fig. 1, then we can distinguish the field `value`. As a result, we can compute a precise results: $pts(\mathtt{v1}) = \{O_{\mathtt{A}}\}$, and $pts(\mathtt{v2}) = \{O_{\mathtt{B}}\}$.

If we use object sensitivity to analyze the example in Fig. 7, we have to set the context depth more than 4: 2-layer contexts can distinguish the methods in `InnerQ`, another 2-layer contexts to distinguish the methods in `Node`.

In Section III-B, we have formalized the representation and propagation of generic context. In this section, we will extend $\mathbb{G}$ and revise functions **Gen** and **Append** to adapt k-generic sensitivity where k>1.

1) Customizing Object Sensitivity: We change $\mathbb{G}$ into a array $[\mathbb{G}_k,\ldots,\mathbb{G}_1]$, where $\mathbb{G}_i := \overline{\mathbb{T} \mapsto \mathbb{H}}$ maps a type variable $\mathcal{T} \in \mathbb{T}$ to an allocation site $O_l \in \mathbb{H}$ (identified by label l). Meantime, the **Gen** and **Append** functions are changed as follows.

$$\mathbf{Gen}(G, \mathcal{T}, A, O_l) = \begin{cases} G & \mathcal{T} = Nil \\ [\mathcal{T} \to O_l] +\!\!+ G & \mathcal{T} \neq Nil \wedge \\ & A \notin G_1 \\ G_1[\mathcal{T}] \to G_1(A) & \mathcal{T} \neq Nil \wedge \\ & A \in G_1 \end{cases}$$

$$\mathbf{Append}(G, G_m, \mathcal{T}, A, O_l) = \\ \begin{cases} G & \mathcal{T} = Nil \\ G_1 \uplus [\mathcal{T} \to O_l] & \mathcal{T} \neq Nil \wedge A \notin G_1 \uplus G_{m_1} \\ G_1 \uplus [\mathcal{T} \to G_1 \uplus G_{m_1}(A)] & \mathcal{T} \neq Nil \wedge A \in G_1 \uplus G_{m_1} \end{cases}$$

For **Gen** functions, the changes are listed as follows.

- If formal type parameter $\mathcal{T}$ is Nil, we just return the generic context of current method.
- If formal type parameter $\mathcal{T} \neq Nil$ and actual type parameter $A \notin G$, the generics propagation is stopped. So, we create a new generic mapping $[\mathcal{T} \mapsto O_l]$ and append to the current generic context by using "++" operator.
- 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. We can generate the innermost generic mapping G_1 of current generic context array to $[\mathcal{T} \mapsto G(A)]$.

For **Append** functions, we just generate the innermost generic mapping by following the constraints in Section IV.

As showing in Fig. 6, we limit the depth of generic-context by G'_{n-1}. In order to distinguish from k which represents the limited depth of traditional context, we use n to represent the limited depth of generic context.

Let us revisit the example in Fig. 7. We create generic objects O_1(line 2) and O_2(line 7) respectively. Hence, we construct generic mappings $[\mathtt{E} \mapsto O_1]$ and $[\mathtt{E} \mapsto O_2]$ respectively. And then the same generic mappings can be generated for object $O_{\mathtt{IQ}}$ at line 15. At line 23, we create generic mappings $[\mathtt{K} \mapsto O_{\mathtt{HM}}, \mathtt{V} \mapsto O_{\mathtt{HM}}]$ and append the generic mappings above to generate two generic context array:$[[\mathtt{E} \mapsto O_1], [\mathtt{K} \mapsto O_{\mathtt{HM}}, \mathtt{V} \mapsto O_{\mathtt{HM}}]]$ and $[[\mathtt{E} \mapsto O_2], [\mathtt{K} \mapsto O_{\mathtt{HM}}, \mathtt{V} \mapsto O_{\mathtt{HM}}]]$. Finally, we can distinguish the methods of `Node` in Fig. 1 according to the generic context arrays above.

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

$$\mathbf{Gen}(G, \mathcal{T}, A) = \begin{cases} G & \mathcal{T} = Nil \\ [\mathcal{T} \to A] +\!\!+ G & \mathcal{T} \neq Nil \wedge \\ & A \notin G_1 \\ G_1[\mathcal{T}] \to G_1(A) & \mathcal{T} \neq Nil \wedge \\ & A \in G_1 \end{cases}$$

$$\mathbf{Append}(G, G_m, \mathcal{T}, A) = \begin{cases} G & \mathcal{T} = Nil \\ G_1 \uplus [\mathcal{T} \to A] & \mathcal{T} \neq Nil \wedge A \notin G_1 \uplus G_{m_1} \\ G_1 \uplus [\mathcal{T} \to G_1 \uplus G_{m_1}(A)] & \mathcal{T} \neq Nil \wedge A \in G_1 \uplus G_{m_1} \end{cases}$$

Let us revisit the example in Fig. 7. We create generic objects O_1(line 2) and O_2(line 7) respectively. Hence, we construct generic mappings $[\mathtt{E} \mapsto \mathtt{A}]$ and $[\mathtt{E} \mapsto \mathtt{B}]$ respectively. And then the same generic mappings can be generated for object $O_{\mathtt{IQ}}$ at line 15. At line 23, we create generic mappings $[\mathtt{K} \mapsto \mathtt{String}, \mathtt{V} \mapsto \mathtt{Object}]$ and append the generic mappings above to generate two generic context array:$[[\mathtt{E} \mapsto \mathtt{A}], [\mathtt{K} \mapsto \mathtt{String}, \mathtt{V} \mapsto \mathtt{Object}]]$ and $[[\mathtt{E} \mapsto \mathtt{B}], [\mathtt{K} \mapsto \mathtt{String}, \mathtt{V} \mapsto \mathtt{Object}]]$. Finally, we can distinguish the methods of `Node` in Fig. 1 according to the generic context arrays above.

V. Implementation and Evaluation

We evaluate generic sensitive pointer analysis by applying and comparing our context customization scheme to an array of pointer analyses at different precision. In total, there are 11 variants of standard pointer analyses. For illustration, we mainly compare two 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). Additionally, We compare the precision and efficiency of generic sensitivity with or without applying ZIPPER [25] and ZIPPER-E [26], the state-of-the-art selective context-sensitivity approaches provided by TAI-E [36].

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 [37]. 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. In other words, `Gen+k-obj` scheme uses both generic information and allocation sites of receiver objects as contexts for generic-related methods and uses allocation sites of receiver objects as contexts for the remaining methods. The same principle applies to other variants of context-sensitivity as well. The notations ending in `Z` represent applying ZIPPER to the target analysis, e.g., `1-objZ` represents applying ZIPPER to 1-obj analysis, which means that only the methods selected by ZIPPER are analyzed by `1-obj`. Similarly, The notations ending in `ZE` represent applying ZIPPER-E to the target analysis, e.g., `1-objZE` represents applying ZIPPER-E to 1-obj analysis. Note that ZIPPER-E is able to achieve significantly better efficiency than ZIPPER with comparable precision [26].

A. 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 [14], [15], [18]. We also implemented in WALA a new k-type analysis according to its original definition [18].

Sometimes, generic instantiation information may be wiped out in Java bytecode. For instance, Java tends to erase 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.

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

TABLE II
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)

B. Setting

We evaluate the 18 Benchmarks in Table II, 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 [20], [21], [25], [30], the JDK version is JDK1.6 (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 extensively 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?
- RQ5. Can generic sensitivity improve precision and efficiency over state-of-the-art selective context-sensitivity approaches, ZIPPER and ZIPPER-E?
- RQ6. How the precision and efficiency of k-generic sensitivity change as k increases?
- RQ7. Can local analysis improve precision of generic sensitivity?

C. RQ1. Generic Usages

Table II 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 II, 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.

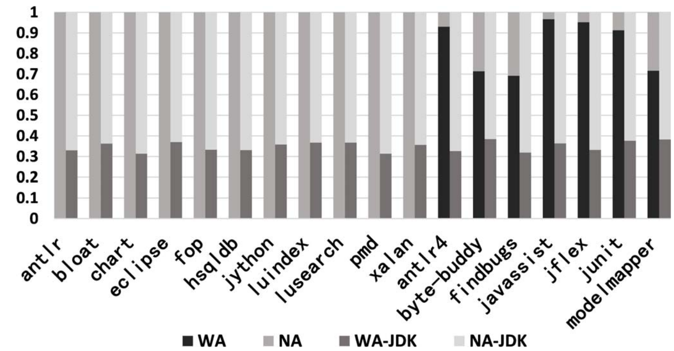


Fig. 8. Percentages of generic object allocations with actual types. WA is with actual types, NA is without actual types, WA-JDK and NA-JDK are generic object allocations in JDK with or without actual types.

Fig. 8 depicts the percentages of generic usages with actual type arguments, including those usages where our simple conservative type inference analysis (Section III) can infer actual type arguments. The `WA` is the percentage of cases that our local analysis can deal with, and the `NA` represents the other cases. The number of actual types inferred is small. As shown in Fig. 8, 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.

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.

D. RQ2. Precision

Following [30], [31], [32], we measure the precision of context-sensitive analyses using four basic metrics: #call-edges (number of call graph 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)

TABLE III
EFFICIENCY AND PRECISION METRICS OF DIFFERENT ANALYSES ON DaCapo BENCHMARKS

Program	Metrics	CI	obj					type				
			GenO	1-obj	Gen+1-obj	2-obj	Gen+2-obj	GenT	1-type	Gen+1-type	2-type	Gen+2-type
antlr	Time (s)	5.8	7.0	26.9	14.3	58.8	65.8	9.9	9.3	11.3	16.6	14.5
	#cast-may-fail	833	495	729	408	414	386	507	739	412	723	408
	#poly-call	1,513	1,248	1,374	1,184	1,167	1,166	1,252	1,418	1,211	1,275	1,178
	#reach-methods	7,324	7,020	7,171	6,953	6,962	6,935	7,020	7,201	6,966	7,142	6,936
	#call-edge	42,111	36,667	39,795	35,803	35,808	35,723	36,691	39,987	35,942	38,100	35,763
bloat	Time (s)	11.9	12.9	242.8	178.0	-	-	12.9	26.7	26.0	274.5	183.4
	#cast-may-fail	2,018	1,459	1,890	1,300	-	-	1,471	1,903	1,303	1,884	1,297
	#poly-call	2,223	1,693	1,999	1,608	-	-	1,697	2,118	1,639	1,719	1,584
	#reach-methods	9,207	8,934	9,066	8,835	-	-	8,934	9,095	8,845	9,018	8,798
	#call-edge	66,992	58,397	64,060	56,775	-	-	58,421	64,464	56,913	61,173	56,203
chart	Time (s)	11.4	12.9	282.6	33.5	2,087.6	1,855.8	12.9	27.7	15.0	191.5	42.0
	#cast-may-fail	1,795	1,175	1,668	982	979	913	1,210	1,691	998	1,661	979
	#poly-call	2,010	1,637	1,869	1,535	1,503	1,498	1,646	1,939	1,585	1,715	1,542
	#reach-methods	11,804	11,420	11,647	11,284	11,282	11,241	11,420	11,684	11,339	11,619	11,310
	#call-edge	65,360	56,387	62,180	53,928	53,780	53,603	56,522	62,696	54,651	61,112	54,313
eclipse	Time (s)	7.9	9.8	28.8	14.7	97.4	92.4	9.9	11.4	10.8	27.8	16.3
	#cast-may-fail	1,037	720	909	579	568	540	732	928	592	906	578
	#poly-call	1,268	1,013	1,123	936	920	919	1,017	1,169	977	1,053	932
	#reach-methods	7,731	7,474	7,572	7,345	7,347	7,320	7,474	7,612	7,410	7,545	7,326
	#call-edge	44,320	38,293	41,838	36,150	36,068	35,983	38,297	42,127	37,225	39,742	36,203
fop	Time (s)	8.0	8.8	17.6	12.1	68.1	70.6	9.4	10.5	10.1	17.1	14.9
	#cast-may-fail	812	485	708	397	411	383	497	718	400	702	396
	#poly-call	1,136	861	990	790	773	772	865	1,034	817	885	784
	#reach-methods	6,847	6,547	6,709	6,480	6,489	6,462	6,547	6,739	6,493	6,674	6,463
	#call-edge	37,230	32,077	35,198	31,204	31,208	31,123	32,100	35,380	31,333	33,476	31,164
hsqldb	Time (s)	7.8	8.6	19.2	11.7	53.2	61.1	8.1	9.1	9.1	17.3	12.3
	#cast-may-fail	775	453	677	371	385	357	465	687	374	671	370
	#poly-call	1,104	839	965	775	760	759	843	1,012	805	868	771
	#reach-methods	6,604	6,311	6,468	6,243	6,253	6,226	6,311	6,498	6,256	6,440	6,227
	#call-edge	36,324	31,237	34,312	30,366	30,375	30,290	31,260	34,498	30,499	32,640	30,330
jython	Time (s)	9.3	13.3	103.6	78.1	-	-	11.8	17.0	17.1	4,322.3	3,996.9
	#cast-may-fail	1,284	908	1,178	812	-	-	920	1,191	815	1,176	811
	#poly-call	1,604	1,338	1,455	1,262	-	-	1,342	1,507	1,297	1,366	1,260
	#reach-methods	8,852	8,548	8,714	8,453	-	-	8,548	8,741	8,464	8,688	8,437
	#call-edge	52,026	44,891	49,603	43,588	-	-	44,895	49,809	43,847	47,378	43,654
luindex	Time (s)	8.2	9.3	22.7	13.1	49.0	54.8	9.6	10.0	9.5	16.4	14.2
	#cast-may-fail	811	468	709	375	389	361	480	719	379	702	375
	#poly-call	1,142	869	992	802	787	786	873	1,046	831	895	798
	#reach-methods	6,924	6,635	6,789	6,568	6,577	6,550	6,635	6,818	6,580	6,760	6,551
	#call-edge	37,493	32,349	35,454	31,460	31,467	31,382	32,372	35,643	31,596	33,775	31,430
lusearch	Time (s)	8.3	10.1	26.0	14.2	71.8	70.0	8.6	10.2	10.6	20.4	15.3
	#cast-may-fail	917	517	811	417	399	371	529	823	420	772	386
	#poly-call	1,334	1,055	1,181	983	966	965	1,059	1,233	1,010	1,078	977
	#reach-methods	7,596	7,287	7,463	7,212	7,221	7,194	7,287	7,492	7,224	7,431	7,195
	#call-edge	40,787	35,267	38,706	34,328	34,333	34,248	35,290	38,893	34,454	36,878	34,288
pmd	Time (s)	8.4	10.5	27.8	14.9	78.6	77.0	10.2	13.0	11.6	24.5	16.5
	#cast-may-fail	1,260	822	1,149	727	744	712	834	1,161	730	1,143	725
	#poly-call	1,204	912	1,062	844	826	825	916	1,109	872	941	835
	#reach-methods	8,337	8,021	8,198	7,955	7,955	7,928	8,021	8,229	7,968	8,168	7,929
	#call-edge	44,572	38,754	42,490	37,873	37,870	37,778	38,777	42,690	38,016	40,624	37,821
xalan	Time (s)	6.6	8.2	19.6	12.4	54.3	56.6	9.1	10.2	9.3	15.0	13.5
	#cast-may-fail	779	461	676	374	388	360	473	686	377	670	373
	#poly-call	1,106	841	964	774	759	758	845	1,010	803	867	770
	#reach-methods	6,619	6,332	6,483	6,265	6,274	6,247	6,332	6,513	6,278	6,454	6,248
	#call-edge	36,019	30,979	34,008	30,109	30,116	30,031	31,002	34,190	30,238	32,338	30,071

is always more precise than k-obj (k-type) analysis. Table III and Table IV compare precision metrics in 2 groups: object-sensitive group (GenO, 1-obj, Gen+1-obj, 2-obj) and type-sensitive group (GenT, 1-type, Gen+1-type, 2-type). Column 4-8 in both tables show results for object-sensitive group, and results of type-sensitive group are given in Column 9-13 of both tables.

Under the given 90 minutes budget, 2-obj analysis fails to process five benchmarks: bloat, jython, antlr4, byte-buddy, and modelmaper. Both 1-obj and 2-type analyses timeout on antlr4. Those timeout cases are outlined with "-" in both of tables.

Object-Sensitivity Group. GenO, 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 GenO in those benchmarks that it is able to finish running: for #cast-may-fail, #poly-call, #reach-methods, and #call-edge, the ratio of the number reported by GenO 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.

TABLE IV
EFFICIENCY AND PRECISION METRICS OF DIFFERENT ANALYSES ON APPLICATIONS

Program	Metrics	CI	obj					type				
			GenO	1-obj	Gen+1-obj	2-obj	Gen+2-obj	GenT	1-type	Gen+1-type	2-type	Gen+2-type
antlr4	Time (s)	23.1	45.1	-	3,566.9	-	-	36.9	154.8	165.6	-	-
	#cast-may-fail	3,608	2,779	-	2,156	-	-	2,899	3,245	2,364	-	-
	#poly-call	3,507	2,888	-	2,695	-	-	2,980	3,359	2,813	-	-
	#reach-methods	21,029	20,545	-	20,430	-	-	20,548	20,900	20,453	-	-
	#call-edge	166,399	152,186	-	149,833	-	-	152,479	163,145	150,304	-	-
byte-buddy	Time (s)	14.0	21.4	1,197.3	202.1	-	-	18.0	63.0	30.2	2,393.1	752.7
	#cast-may-fail	1,715	1,095	1,582	829	-	-	1,119	1,599	856	1,549	798
	#poly-call	3,527	3,084	3,357	2,921	-	-	3,096	3,435	2,992	3,220	2,853
	#reach-methods	11,773	11,305	11,646	11,121	-	-	11,309	11,685	11,155	11,606	11,072
	#call-edge	70,635	58,776	67,552	55,848	-	-	59,443	67,866	56,511	64,337	55,459
findbugs	Time (s)	9.5	11.8	65.4	15.4	193.8	148.1	11.9	16.0	11.7	47.8	18.8
	#cast-may-fail	1,223	734	1,112	566	600	547	757	1,125	578	1,105	572
	#poly-call	1,626	1,229	1,414	1,144	1,109	1,107	1,234	1,525	1,182	1,277	1,139
	#reach-methods	9,242	8,852	9,082	8,685	8,693	8,662	8,852	9,113	8,703	9,024	8,673
	#call-edge	51,887	42,600	48,520	40,639	40,636	40,476	42,641	48,789	40,842	45,582	40,642
javassist	Time (s)	6.5	8.3	16.5	11.8	50.5	59.2	7.6	8.4	8.7	14.3	12.3
	#cast-may-fail	765	448	671	370	384	356	460	681	373	665	369
	#poly-call	1,089	824	956	766	751	750	828	1,003	796	859	762
	#reach-methods	6,519	6,229	6,381	6,159	6,168	6,141	6,229	6,413	6,174	6,352	6,142
	#call-edge	35,379	30,328	33,396	29,486	29,493	29,408	30,351	33,580	29,617	31,726	29,448
jflex	Time (s)	10.4	12.2	116.8	35.0	1,445.8	1,411.0	11.9	23.2	16.8	326.1	135.0
	#cast-may-fail	1,545	1,087	1,414	907	920	881	1,131	1,443	923	1,416	909
	#poly-call	2,004	1,631	1,854	1,547	1,532	1,507	1,673	1,923	1,602	1,783	1,563
	#reach-methods	10,708	10,356	10,544	10,265	10,247	10,206	10,364	10,577	10,286	10,522	10,254
	#call-edge	56,310	49,154	53,737	47,763	47,285	47,129	49,390	54,204	48,110	53,181	47,831
junit	Time (s)	6.8	13.1	36.8	17.5	73.2	79.2	12.9	12.1	14.6	39.0	22.8
	#cast-may-fail	962	633	862	473	502	452	648	875	489	850	470
	#poly-call	1,303	1,016	1,180	930	925	909	1,020	1,233	960	1,094	921
	#reach-methods	8,008	7,771	7,891	7,630	7,639	7,610	7,773	7,910	7,649	7,843	7,613
	#call-edge	41,715	36,473	39,663	34,867	34,905	34,764	36,530	39,839	35,030	38,021	34,817
modelmapper	Time (s)	13.0	19.8	899.5	158.4	-	-	17.7	51.8	27.0	3,058.6	843.6
	#cast-may-fail	1,678	1,094	1,546	821	-	-	1,119	1,563	847	1,513	789
	#poly-call	3,529	3,102	3,364	2,933	-	-	3,114	3,444	3,010	3,229	2,864
	#reach-methods	11,732	11,283	11,605	11,073	-	-	11,287	11,644	11,107	11,565	11,018
	#call-edge	69,389	58,014	66,347	55,040	-	-	58,684	66,669	55,716	63,237	54,633

Type-Sensitivity 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 2-type, reporting 72.98%, 96.81%, 98.27%, and 94.90% of the number reported by 2-type, respectively. This may suggest again the benefit of applying generic instantiation locations to distinguish contexts, especially for the cases where coarse context elements (like types) do not work effectively.

The context elements of type-sensitivity in Table III and Table IV are types of receiver objects (we use "bad strategy" to represent this strategy). To validate the robustness of generic sensitivity on different strategies of type-sensitivity [18], Fig. 9 shows the precision and efficiency of different analyses against 1-type-sensitivity in Table III and Table IV. We use k-typeG to represent k-type-sensitivity, the context elements of which are types that contain the methods which allocate receiver objects (we use "good strategy" to represent this strategy). The precision on all metrics of the two strategies are similar, and generic sensitivity can significantly enhance the precision of both strategies on average.

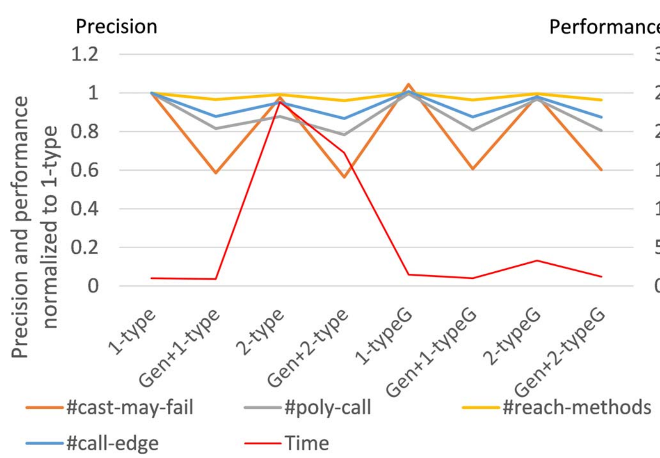


Fig. 9. Efficiency and precision metrics of different strategies of type sensitivity. Lower is better along all axes.

E. RQ3. Efficiency

As shown in Table III and Table IV, 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 speedup of 1.8 ×, despite the fact that it is simultaneously much more precise. Compared to 2-obj with similar precision, Gen+1-obj achieves a speedup of 62 × for chart, with an average speedup of 12.6 × for the 13 applications that 2-obj run to completion. Similarly, Gen+1-type also achieves noticeably better efficiency than 1-type, with an average efficiency improvement of 20%. As Fig. 9 shows, comparing with the "bad strategy", the "good strategy" is slightly slower when k=1 on average, and much faster under k=2. And generic sensitivity can generally improve the efficiency of both strategies.

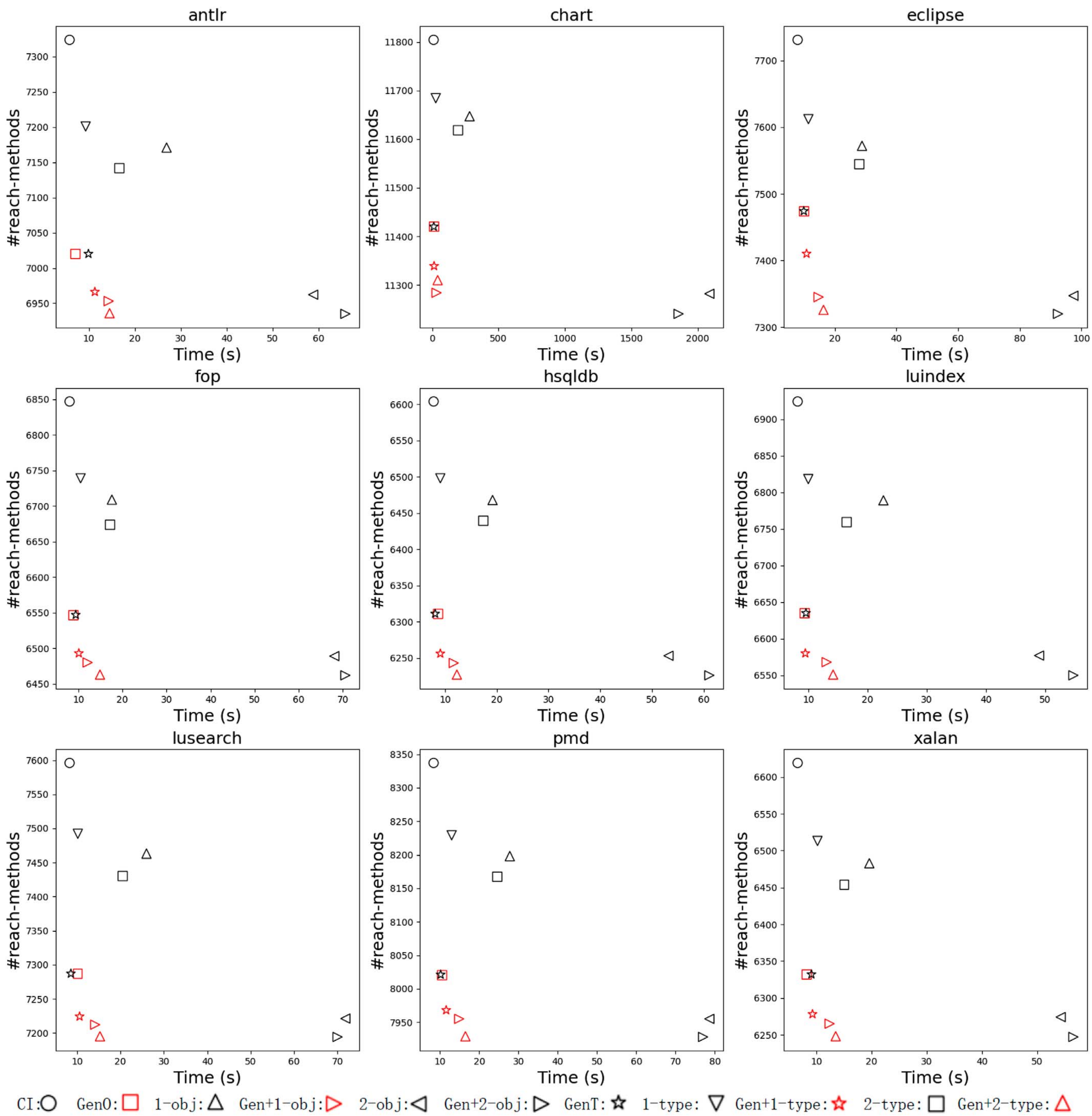


Fig. 10. Precision(#reach-methods)/efficiency spectrum for DACAPO benchmarks. Lower is better along both axes.

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

F. RQ4. Precision and Efficiency Trade-Off

Fig. 10 depicts the efficiency 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 Fig. 10, lower numbers are better on both axes. Hence, analyses in the bottom left corner are superior in both precision and efficiency. 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 efficiency. The most precise

TABLE V
EFFICIENCY AND PRECISION METRICS OF DIFFERENT ANALYSES COMBINED WITH ZIPPER AND ZIPPER-E ON DaCapo BENCHMARKS

Program	Metrics	1-objZ	Gen+1-objZ	2-objZ	Gen+2-objZ	1-objZE	Gen+1-objZE	2-objZE	Gen+2-objZE
	Time (s)	8.3	9.8	27.4	35.6	7.7	11.1	9.5	12.0
	#cast-may-fail	777	687	708	681	794	723	728	722
antlr	#poly-call	1,393	1,262	1,269	1,255	1,433	1,306	1,319	1,302
	#reach-methods	7,212	7,154	7,176	7,135	7,248	7,201	7,226	7,198
	#call-edge	40,395	38,118	38,644	38,040	40,793	38,857	39,412	38,843
	Time (s)	89.0	101.3	-	-	19.5	29.7	25.2	30.6
	#cast-may-fail	1,936	1,690	-	-	1,955	1,739	1,755	1,737
bloat	#poly-call	2,033	1,708	-	-	2,075	1,776	1,784	1,768
	#reach-methods	9,093	9,023	-	-	9,127	9,066	9,091	9,063
	#call-edge	64,599	61,174	-	-	65,107	62,134	62,369	62,015
	Time (s)	27.7	44.8	247.1	229.0	20.4	37.5	42.3	67.0
	#cast-may-fail	1,732	1,500	1,525	1,472	1,746	1,538	1,551	1,537
chart	#poly-call	1,898	1,679	1,681	1,669	1,921	1,714	1,745	1,710
	#reach-methods	11,674	11,570	11,607	11,534	11,689	11,600	11,637	11,600
	#call-edge	62,902	60,879	61,229	60,164	63,313	61,431	61,779	61,427
	Time (s)	10.7	18.6	44.6	54.6	9.7	16.4	12.0	18.0
	#cast-may-fail	979	882	899	872	995	917	918	912
eclipse	#poly-call	1,148	1,030	1,037	1,023	1,188	1,074	1,085	1,068
	#reach-methods	7,612	7,552	7,574	7,533	7,647	7,598	7,623	7,595
	#call-edge	42,401	40,159	40,700	40,051	42,826	41,296	41,861	41,261
	Time (s)	8.3	14.3	27.2	35.1	8.1	12.2	8.7	12.4
	#cast-may-fail	756	667	687	661	773	702	707	701
fop	#poly-call	1,024	884	893	879	1,056	920	933	916
	#reach-methods	6,734	6,669	6,691	6,650	6,769	6,715	6,740	6,712
	#call-edge	35,518	33,356	33,812	33,282	35,905	34,329	34,553	34,315
	Time (s)	8.8	11.3	25.7	31.8	7.7	11.6	8.5	11.5
	#cast-may-fail	724	634	655	628	741	670	675	669
hsqldb	#poly-call	991	860	869	855	1,024	897	910	893
	#reach-methods	6,493	6,434	6,457	6,416	6,528	6,480	6,506	6,478
	#call-edge	34,627	32,493	32,936	32,421	35,014	33,201	33,669	33,189
	Time (s)	45.4	51.5	-	-	12.4	15.6	15.1	20.5
	#cast-may-fail	1,234	1,129	-	-	1,241	1,159	1,163	1,157
jython	#poly-call	1,503	1,345	-	-	1,518	1,366	1,375	1,358
	#reach-methods	8,750	8,674	-	-	8,762	8,700	8,726	8,697
	#call-edge	50,092	46,605	-	-	50,504	48,287	49,310	48,264
	Time (s)	9.7	13.0	26.0	32.3	8.2	12.4	8.7	12.1
	#cast-may-fail	755	661	682	655	772	697	702	696
luindex	#poly-call	1,018	887	896	882	1,050	923	936	919
	#reach-methods	6,813	6,755	6,777	6,736	6,848	6,801	6,826	6,798
	#call-edge	35,763	33,618	34,068	33,544	36,150	34,332	34,807	34,318
	Time (s)	11.2	14.0	34.7	42.3	9.1	13.7	12.2	16.6
	#cast-may-fail	858	754	763	736	876	791	786	780
lusearch	#poly-call	1,210	1,075	1,084	1,070	1,242	1,111	1,124	1,107
	#reach-methods	7,485	7,426	7,448	7,407	7,520	7,472	7,497	7,469
	#call-edge	39,023	36,682	37,184	36,608	39,411	37,425	37,952	37,411
	Time (s)	11.5	15.6	37.7	46.9	10.5	15.1	17.8	21.1
	#cast-may-fail	1,196	1,049	1,073	1,043	1,210	1,083	1,090	1,082
pmd	#poly-call	1,092	928	937	923	1,115	955	968	951
	#reach-methods	8,225	8,162	8,184	8,143	8,251	8,199	8,224	8,196
	#call-edge	42,815	40,264	40,781	40,190	42,935	40,748	41,292	40,734
	Time (s)	9.1	12.0	25.1	30.1	7.8	11.3	8.8	11.3
	#cast-may-fail	723	632	653	626	740	668	673	667
xalan	#poly-call	994	863	872	858	1,026	899	912	895
	#reach-methods	6,508	6,450	6,472	6,431	6,543	6,496	6,521	6,493
	#call-edge	34,323	32,199	32,635	32,125	34,709	32,904	33,365	32,890

analysis is Gen+2-obj. However, its efficiency is similar to 2-obj and both are significantly slower than the other variants. For #reach-methods, Gen+2-type achieves similar precision to Gen+2-obj, with significant efficiency 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.

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

G. RQ5. Comparing With Selective Context Sensitivity

Table V and Table VI compare precision and efficiency of k-obj and Gen+k-obj with or without applying ZIPPER and

TABLE VI
EFFICIENCY AND PRECISION METRICS OF DIFFERENT ANALYSES COMBINED WITH ZIPPER AND ZIPPER-E

Program	Metrics	1-objZ	Gen+1-objZ	2-objZ	Gen+2-objZ	1-objZE	Gen+1-objZE	2-objZE	Gen+2-objZE
`antlr4`	Time (s)	829.8	1,516.3	-	-	104.2	453.1	3,342.4	3,824.7
	#cast-may-fail	3,240	3,025	-	-	3,261	3,062	3,087	3,061
	#poly-call	3,312	2,997	-	-	3,353	3,062	3,111	3,062
	#reach-methods	20,834	20,655	-	-	20,853	20,731	20,793	20,730
	#call-edge	160,391	156,775	-	-	160,862	157,928	158,552	157,918
`byte-buddy`	Time (s)	227.2	452.2	-	-	30.0	43.4	71.9	76.8
	#cast-may-fail	1,630	1,416	-	-	1,657	1,456	1,473	1,439
	#poly-call	3,399	3,083	-	-	3,427	3,149	3,164	3,139
	#reach-methods	11,666	11,495	-	-	11,684	11,574	11,623	11,569
	#call-edge	67,953	62,197	-	-	68,336	63,997	66,308	63,937
`findbugs`	Time (s)	16.0	23.8	55.1	69.0	12.0	15.8	17.7	24.2
	#cast-may-fail	1,162	956	979	952	1,172	974	995	972
	#poly-call	1,443	1,242	1,229	1,216	1,462	1,263	1,275	1,255
	#reach-methods	9,109	8,996	9,018	8,975	9,120	9,021	9,051	9,019
	#call-edge	48,855	44,845	45,050	44,707	49,188	45,609	45,923	45,559
`javassist`	Time (s)	8.0	11.8	22.3	28.7	8.6	10.1	10.3	12.9
	#cast-may-fail	714	624	645	618	722	642	655	641
	#poly-call	976	845	854	840	997	868	881	864
	#reach-methods	6,408	6,349	6,371	6,330	6,427	6,379	6,404	6,376
	#call-edge	33,682	31,557	31,993	31,483	34,035	32,203	32,679	32,189
`jflex`	Time (s)	23.5	33.3	207.2	185.8	12.5	20.2	20.3	27.1
	#cast-may-fail	1,468	1,309	1,332	1,300	1,484	1,336	1,352	1,335
	#poly-call	1,883	1,693	1,720	1,692	1,909	1,726	1,756	1,726
	#reach-methods	10,575	10,462	10,498	10,445	10,593	10,504	10,531	10,503
	#call-edge	54,146	52,335	52,663	51,837	54,554	52,852	53,140	52,848
`junit`	Time (s)	9.7	21.7	34.0	51.8	11.9	17.1	14.7	21.8
	#cast-may-fail	911	799	818	789	924	824	839	821
	#poly-call	1,200	1,019	1,032	1,010	1,230	1,055	1,064	1,047
	#reach-methods	7,917	7,854	7,866	7,834	7,936	7,874	7,902	7,870
	#call-edge	39,941	38,057	38,879	37,931	40,308	38,447	39,329	38,395
`modelmapper`	Time (s)	103.6	139.5	-	-	29.2	43.8	54.1	77.5
	#cast-may-fail	1,595	1,396	-	-	1,620	1,436	1,454	1,419
	#poly-call	3,406	3,104	-	-	3,433	3,173	3,186	3,163
	#reach-methods	11,626	11,474	-	-	11,643	11,552	11,598	11,547
	#call-edge	66,764	61,353	-	-	67,145	63,161	65,279	63,101

ZIPPER-E (two state-of-the-art selective context-sensitivity approaches) on our benchmarks. Consistent with the conclusion in [25], [26], both of ZIPPER and ZIPPER-E can achieve substantial speedup with slight loss of precision than standard k-object-sensitivity. Generic Sensitivity and selective context-sensitivity approaches complement each other, as they are able to further enhance the precision and efficiency when combined.

For examples, ZIPPER-E can significantly improve the efficiency of generic-sensitivity, i.e., `Gen+2-objZE` can analysis all benchmarks under the given time budge while `Gen+2-obj` fail to analysis five benchmarks: `bloat`, `jython`, `antlr4`, `byte-buddy` and `modelmapper`. Comparing with `1-objZ`, `Gen+1-objZ` is significant more precise: for #cast-may-fail, #poly-call, #reach-methods, and #call-edge, the ratio of the number reported by `Gen+1-objZ` against that reported by `1-objZ` is 88.03%, 87.94%, 99.06%, and 94.21% respectively. The conclusion is similar for ZIPPER-E, the ratio of the number reported by `Gen+1-objZE` against that reported by `1-objZE` is 89.83%, 88.72%, 99.26%, and 95.30%, respectively. Comparing with `2-objZ`, `Gen+1-objZ` is already able to achieve an average speedup of 2.8× with slight improvement of precision, the ratio of the number reported by `Gen+1-objZ` against that reported by `2-objZ` is 97.54%, 99.23%, 99.70%, and 98.79% for #cast-may-fail, #poly-call, #reach-methods, and #call-edge respectively.

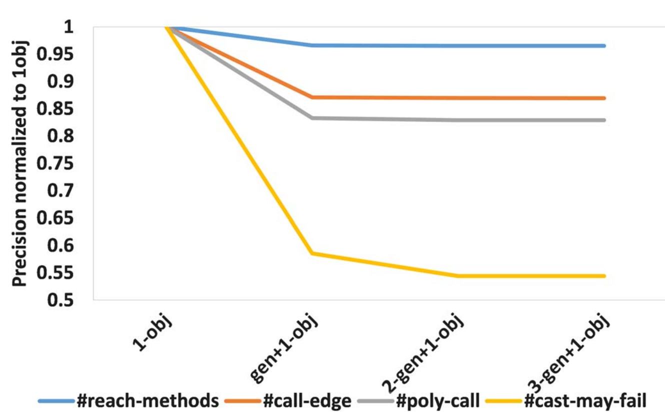


Fig. 11. Precision of k-generic sensitivity.

H. RQ6. Precision/Efficiency of K-Generic Sensitivity

Same with other variants of context sensitivity, the precision will increase as the depth of context increases. Fig. 11 compares the precision of 1-obj, `Gen+1-obj`, `2-Gen+1-obj` and `3-Gen+1-obj`. The four metrics of precision are normalized to 1-obj. Compared to 1-obj, `Gen+1-obj`, `2-Gen+1-obj` and `3-Gen+1-obj` are noticeably more precise. For #cast-may-fail metric, `2-Gen+1-obj` achieves slightly better precision than

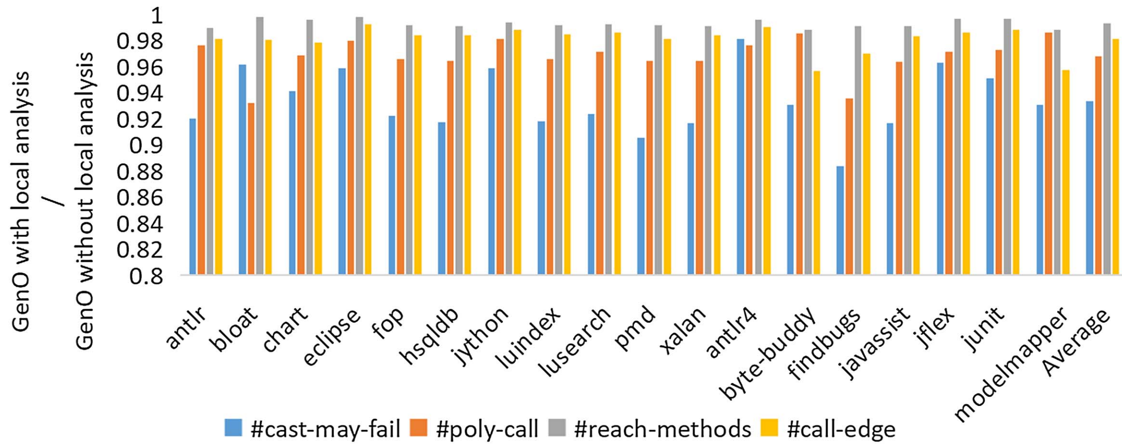


Fig. 12. The precision of `GenO` with local analysis against `GenO` without local analysis.

`Gen+1-obj`. For other metrics, the precision improvements are insignificant. For `chart`, `3-Gen+1-obj` achieves slightly better precision than `2-Gen+1-obj` on #cast-may-fail and #call-edge metrics. For other benchmarks, both of `3-Gen+1-obj` and `2-Gen+1-obj` have the same precision on all metrics.

Fig. 13 compares analysis times for 1-obj, `Gen+1-obj`, `2-Gen+1-obj` and `3-Gen+1-obj`. As shown in Fig. 13, compared to `Gen+1-obj`, `2-Gen+1-obj` and `3-Gen+1-obj` slow down 0.85× and 6.18× on average, respectively. Still, `2-Gen+1-obj` is still faster than 1-obj on average. If we disable the limitation of the depth of generic context, it will timeout on all benchmarks as other context-sensitivity approaches.

I. RQ7. Performance of Local Analysis

Section III-A introduces a local analysis that can detect actual type parameters. Additionally, we defined a degradation strategy that employs pseudo types as actual type parameters in instances where the local analysis is deemed invalid. However, this strategy may influence the precision. Taking Fig. 1 as an example, without local analysis, we cannot infer the actual type parameters of `Node` at line 17 and use T_{17} to replace them. So, the contexts of the constructor of class `Node` will be $[\texttt{V} \mapsto T_{17}]$ rather than $[\texttt{V} \mapsto O_1]$ and $[\texttt{V} \mapsto O_2]$ where we omit the context mapping of formal type parameter `K`. Fig. 12 shows the precision of `GenO` with or without local analysis for #cast-may-fail, #poly-call, #reach-methods, and #call-edge. On average, the ratio of the number reported by `GenO` with local analysis against that reported by `GenO` without local analysis is 93.31%, 96.78%, 99.26%, 98.06%, respectively. Meantime, both strategies take similar time for all benchmarks.

VI. 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 [20] proposes a hybrid approach which applies object sensitivity to instance method invocations and call-site

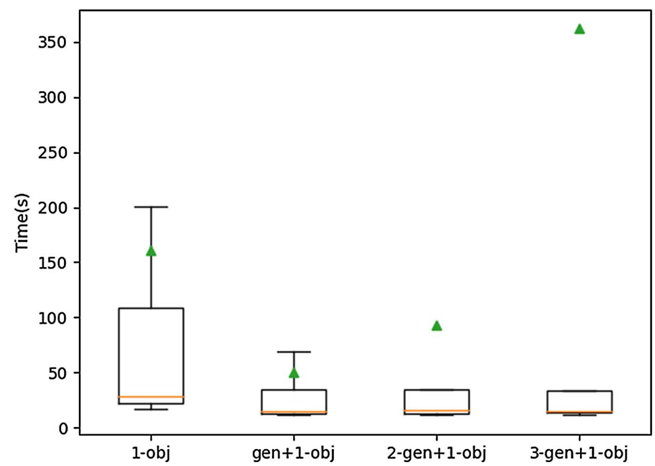


Fig. 13. Efficiency of k-generic sensitivity. The triangle represents means.

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 [38] use the points-to set of receiver object to approximate a context [38]. 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 it 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 [40], [41], [42], or learning-based approaches [43], [44], [45], [46], [47] to selectively analyze a subset of methods context-sensitivity. Scaler [21] determines whether to analyze a method context-sensitively or not based on an estimation of its potential memory consumption. Zipper [25] introduces three kinds of value-flow patterns to identify precision-critical methods, and those patterns can be computed by solving a graph reachability problem on a precision flow graph. As a result, most precision can be preserved while achieving noticeable speedup. The later

ZIPPER-E [26], as a new variant of `Zipper`, is able to significantly accelerate `Zipper` with comparable precision. EAGLE [30] performs a CFL-reachability-based pre-analysis to enable selective context-sensitivity in k-obj, while guaranteeing precision. TURNER [48] finds a sweet spot between ZIPPER and EAGLE, which enables k-obj analysis to run significantly faster than EAGLE while achieve better precision than ZIPPER. CONCH [48] finds context-dependent objects, avoiding contexts bloating. BATON [31] proposes a Unit-Relay framework by collectively integrating different context selectors. Instead of selecting which methods to be context-sensitive analyzed, BEAN [49] makes k-obj sensitive analysis more precise by skipping those unhelpful context elements. In [32], 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 precision or efficiency, or both. Generic sensitivity can also be applied with call-site sensitivity by augmenting contexts with call-sites and propagating mappings from type variables to call-sites along type variables, similar to the methodology in the paper. OBJ2CFA introduces an innovative context-tunneled k-call-site sensitive analysis which may outperform object sensitivity in a general k-limiting setting by selecting critical call-sites as contexts. Generic sensitivity and OBJ2CFA may complement each other, akin to combining generic sensitivity with Zipper. Recently, the CUT-SHORTCUT approach [50] has made significant strides in accelerating context-sensitivity-like pointer analysis without employing context sensitivity. This achievement is accomplished by leveraging various patterns within its graph manipulation principle. In light of these advancements, we anticipate that our generic-sensitivity approach can provide new insights into the identification of novel generics-related patterns within this principle. This, in turn, can facilitate the exploration of new trade-offs between precision 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 [51], [52], using binary decision diagrams (BDDs) [53], [54], [55], using geometric encoding techniques [56], or graph systems [57]. The work [58], [59] 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.

VII. 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. Our scheme is built upon the insight that generic instantiation locations can serve as crucial context elements for effectively distinguishing contexts in Java programs. Leveraging this observation, we have established formal rules and outlined the application of generic customization to two prominent context-sensitive variants: object sensitivity and type sensitivity. Extensive experimental evaluations have been conducted, demonstrating the effectiveness of generic sensitivity in improving both traditional and selective context-sensitivity approaches. Our results highlight the potential for a new trade-off between efficiency and precision in Java pointer analysis, and we expect this work to pave the way for further exploration of generics for more precise pointer analysis.

ACKNOWLEDGMENT

The authors thank all anonymous reviewers for their valuable inputs.

REFERENCES

[1] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "TriggerScope: Towards detecting logic bombs in Android applications," in *Proc. IEEE Symp. Secur. Privacy (SP),* Piscataway, NJ, USA: IEEE Press, 2016, pp. 377–396.

[2] L. Li, C. Cifuentes, and N. Keynes, "Practical and effective symbolic analysis for buffer overflow detection," in *Proc. 18th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE '10)*, New York, NY, USA: ACM, 2010, pp. 317–326, doi: 10.1145/1882291.1882338.

[3] C. Liu et al., "Detecting tensorflow program bugs in real-world industrial environment," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2021, pp. 55–66.

[4] Y. Sui, S. Ye, J. Xue, and J. Zhang, "Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation," *Softw., Pract. Experience*, vol. 44, no. 12, pp. 1485–1510, 2014.

[5] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of Android applications in droidsafe," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, vol. 15, no. 201, p. 110.

[6] S. Arzt et al., "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI '14)*, New York, NY, USA: ACM, 2014, pp. 259–269, doi: 10.1145/2594291.2594299.

[7] D. He et al., "Performance-boosting sparsification of the IFDS algorithm with applications to taint analysis," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2019, pp. 267–279.

[8] N. Grech and Y. Smaragdakis, "P/Taint: Unified points-to and taint analysis," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[9] M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *Proc. ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng.*, 2001, pp. 54–61.

[10] L. Li, C. Cifuentes, and N. Keynes, "Boosting the performance of flow-sensitive points-to analysis using value flow," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng. (ESEC/FSE '11)*, New York, NY, USA: ACM, 2011, pp. 343–353, doi: 10.1145/2025113.2025160.

[11] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang, "Level by level: Making flow-and context-sensitive pointer analysis scalable for millions of lines of code," in *Proc. 8th Annu. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2010, pp. 218–229.

[12] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Found. Trends Program. Lang.*, vol. 2, no. 1, pp. 1–69, 2015.

[13] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav, "Alias analysis for object-oriented programs," in *Aliasing in Object-Oriented Programming*. Types, Analysis and Verification, Berlin, Germany: Springer-Verlag, 2013, pp. 196–232.

[14] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to and side-effect analyses for Java," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2002, pp. 1–11.

[15] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for Java," *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, vol. 14, no. 1, pp. 1–41, 2005.

[16] M. Sharir et al., *Two Approaches to Interprocedural Data Flow Analysis*. New York Univ., Englewood Cliffs, NJ, USA: Prentice-Hall, 1978.

[17] O. G. Shivers, *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. Pittsburgh, PA, USA: Carnegie Mellon Univ., 1991.

[18] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: Understanding object-sensitivity," in *Proc. 38th Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 2011, pp. 17–30.

[19] L. Li, C. Cifuentes, and N. Keynes, "Precise and scalable context-sensitive pointer analysis via value flow graph," *ACM SIGPLAN Notices*, vol. 48, no. 11, pp. 85–96, 2013.

[20] G. Kastrinis and Y. Smaragdakis, "Hybrid context-sensitivity for points-to analysis," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 423–434, 2013.

[21] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "Scalability-first pointer analysis with self-tuning context-sensitivity," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 129–140.

[22] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "Mining billions of AST nodes to study actual and potential usage of Java language features," in *Proc. 36th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2014, pp. 779–790, doi: 10.1145/2568225.2568295.

[23] "Wala: T.J. Watson libraries for analysis." IBM. Accessed: May 2023. [Online]. Available: http://wala.sourceforge.net

[24] S. M. Blackburn et al., "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proc. 21st Annu. ACM SIGPLAN Conf. Object-Oriented Program. Syst., Lang., Appl.*, 2006, pp. 169–190.

[25] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "Precision-guided context sensitivity for pointer analysis," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 1–29, 2018.

[26] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "A principled approach to selective context sensitivity for pointer analysis," *ACM Trans. Program. Lang. Syst.*, vol. 42, no. 2, May 2020, doi: 10.1145/3381915.

[27] H. Li et al. "Generic sensitivity: Customizing context-sensitive pointer analysis for generics." Figshare. Accessed: May 2023. [Online]. Available: https://figshare.com/articles/software/Generic_Sensitivity_Customizing_Context-Sensitive_Pointer_Analysis_for_Generics/20486556

[28] O. Lhoták and L. Hendren, "Context-sensitive points-to analysis: Is it worth it?" in *Proc. Int. Conf. Compiler Construction,* Berlin, Germany: Springer-Verlag, 2006, pp. 47–64.

[29] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proc. 24th ACM SIGPLAN Conf. Object Oriented Program. Syst. Lang. Appl.*, 2009, pp. 243–262.

[30] J. Lu and J. Xue, "Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[31] T. Tan, Y. Li, X. Ma, C. Xu, and Y. Smaragdakis, "Making pointer analysis more precise by unleashing the power of selective context sensitivity," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–27, 2021.

[32] T. Tan, Y. Li, and J. Xue, "Efficient and precise points-to analysis: Modeling the heap by merging equivalent automata," in *Proc. 38th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2017, pp. 278–291.

[33] R. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller, "Efficiently refactoring Java applications to use generic libraries," in *Proc. Eur. Conf. Object-Oriented Program.*, New York, NY, USA: Springer-Verlag, 2005, pp. 71–96.

[34] W.-N. Chin, F. Craciun, S.-C. Khoo, and C. Popeea, "A flow-based approach for variant parametric types," *ACM SIGPLAN Notices*, vol. 41, no. 10, pp. 273–290, 2006.

[35] F. Tip, R. M. Fuhrer, A. Kieżun, M. D. Ernst, I. Balaban, and B. De Sutter, "Refactoring using type constraints," *ACM Trans. Program. Lang. Syst. (TOPLAS)*, vol. 33, no. 3, pp. 1–47, 2011.

[36] T. Tan and Y. Li, "Tai-e: A developer-friendly static analysis framework for Java by harnessing the good designs of classics," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2023, doi: 10.1145/3597926.3598120.

[37] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, DIKU, Univ. Copenhagen, Copenhagen, Denmark, 1994.

[38] J. Lundberg and W. Löwe, "Points-to analysis: A fine-grained evaluation." *J. Universal Comput. Sci.*, vol. 18, no. 20, pp. 2851–2878, 2012.

[39] R. Padhye and U. P. Khedker, "Interprocedural data flow analysis in soot using value contexts," in *Proc. 2nd ACM SIGPLAN Int. Workshop State Art Java Program Anal.*, 2013, pp. 31–36.

[40] B. Hassanshahi, R. K. Ramesh, P. Krishnan, B. Scholz, and Y. Lu, "An efficient tunable selective points-to analysis for large codebases," in *Proc. 6th ACM SIGPLAN Int. Workshop State Art Program Anal.*, 2017, pp. 13–18.

[41] S. Wei and B. G. Ryder, "Adaptive context-sensitive analysis for JavaScript," in *Proc. 29th Eur. Conf. Object-Oriented Program. (ECOOP),* Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Berlin, Germany: Springer-Verlag, 2015, pp. 712–734.

[42] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective analysis: Context-sensitivity, across the board," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, pp. 485–495.

[43] S. Jeong, M. Jeon, S. Cha, and H. Oh, "Data-driven context-sensitivity for points-to analysis," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[44] M. Jeon, S. Jeong, and H. Oh, "Precise and scalable points-to analysis via data-driven context tunneling," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 1–29, 2018.

[45] M. Jeon, S. Jeong, S. Cha, and H. Oh, "A machine-learning algorithm with disjunctive model for data-driven program analysis," *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 2, Jun. 2019, doi: 10.1145/3293607.

[46] M. Jeon, M. Lee, and H. Oh, "Learning graph-based heuristics for pointer analysis without handcrafting application-specific features," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[47] M. Jeon and H. Oh, "Return of CFA: Call-site sensitivity can be superior to object sensitivity even for object-oriented programs," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, pp. 1–29, 2022.

[48] D. He, J. Lu, and J. Xue, "Context debloating for object-sensitive pointer analysis," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 79–91.

[49] T. Tan, Y. Li, and J. Xue, "Making *k*-object-sensitive pointer analysis more precise with still *k*-limiting," in *Proc. Int. Static Anal. Symp.,* New York, NY, USA: Association for Computing Machinery, 2016, pp. 489–510.

[50] W. Ma, S. Yang, T. Tan, X. Ma, C. Xu, and Y. Li, "Context sensitivity without contexts: A cut-shortcut approach to fast and precise pointer analysis," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Jun. 2023, doi: 10.1145/3591242.

[51] M. Barbar and Y. Sui, "Compacting points-to sets through object clustering," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–27, 2021.

[52] O. Lhoták and L. Hendren, "Scaling Java points-to analysis using SPARK," in *Proc. Int. Conf. Compiler Construction,* Berlin, Germany: Springer-Verlag, 2003, pp. 153–169.

[53] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2004, pp. 131–144.

[54] O. Lhoták, S. Curial, and J. N. Amaral, "Using ZBDDs in points-to analysis," in *Proc. Int. Workshop Lang. Compilers Parallel Comput.*, Berlin, Germany: Springer-Verlag, 2007, pp. 338–352.

[55] O. Lhoták, S. Curial, and J. N. Amaral, "Using XBDDs and ZBDDs in points-to analysis," *Softw., Pract. Experience*, vol. 39, no. 2, pp. 163–188, 2009.

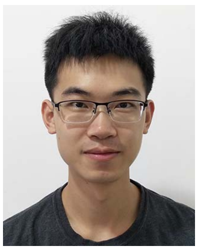
[56] X. Xiao and C. Zhang, "Geometric encoding: Forging the high performance context sensitive points-to analysis for Java," in *Proc. Int. Symp. Softw. Testing Anal.*, 2011, pp. 188–198.

[57] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. A. Sani, "Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code," in *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, *(ASPLOS '17)*, New York, NY, USA: ACM, 2017, pp. 389–404, doi: 10.1145/3037697.3037744.

[58] P. Fegade and C. Wimmer, "Scalable pointer analysis of data structures using semantic models," in *Proc. 29th Int. Conf. Compiler Construction (CC),* New York, NY, USA: ACM, 2020, pp. 39–50, doi: 10.1145/3377555.3377885.

[59] A. Antoniadis, N. Filippakis, P. Krishnan, R. Ramesh, N. Allen, and Y. Smaragdakis, "Static analysis of Java enterprise applications: Frameworks and caches, the elephants in the room," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2020, pp. 794–807.

Haofeng Li received the B.S. degree from Shandong University, in 2017, and the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, in 2023. He is currently an Assistant Professor with the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include program analysis, pointer analysis, and bug detection.



Tian Tan received the B.Eng. degree in software engineering from the Northwestern Polytechnical University, in 2013, and the Ph.D. degree in computer science from the University of New South Wales, in 2017. He was a Postdoc working with Aarhus University, Denmark, from 2017 to 2019. He is currently an Associate Research Professor with the Department of Computer Science and Technology, Nanjing University. His research interests include program analysis and programming languages.

Yue Li received the B.Eng. degree in software engineering and the M.Eng. degree in computer science from the Northwestern Polytechnical University, in 2010 and 2012, respectively, and the Ph.D. degree in computer science from the University of New South Wales (UNSW Sydney), in 2016. He was a Postdoc working with Aarhus University, Denmark, from 2017 to 2019, and UNSW Sydney, from 2016 to 2017. He is currently an Associate Professor with the Department of Computer Science and Technology, Nanjing University. His research interests include program analysis and programming languages.

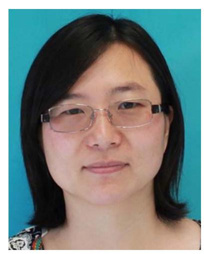
Jie Lu received the B.S. degree in computer science from Sichuan University, in 2014, and the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, in 2020. He is currently an Associate Professor with the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include program analysis, log analysis, and distributed systems.

Haining Meng received the B.S. degree in software engineering from Northwest University, China, in 2017. She is currently working toward the Ph.D. degree with the Institute of Computing Technology, Chinese Academy of Sciences. Her research interests include program analysis.

Liqing Cao received the B.S. degree in information security from Huazhong University of Science and Technology, China, in 2019. He is currently working toward the Ph.D. degree with the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include program analysis.

Yongheng Huang received the B.S. degree in information security from Huazhong University of Science and Technology, China, in 2020. He is currently working toward the Ph.D. degree with the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include program analysis and web security.

Lian Li received the B.Sc. degree in engineering physics from Tsinghua University, in 1998, and the Ph.D. degree from the University of New South Wales, in 2007. He is currently a Professor with the Institute of Computing Technology, Chinese Academy of Sciences, where he leads the Program Analysis Group. His research interests include program analysis, more specifically, on program analysis techniques, and practical tools for improving software safety and security.



Lin Gao received the B.S. degree from Peking University, in 1998, and the Ph.D. degree from the University of New South Wales, in 2009. She is currently a CTO with Beijing ZhongKe TianQi Information Technology Co., Ltd.

Peng Di received the Ph.D. degree in computer science and engineering from the University of New South Wales, in 2013. He is a Professor-Level Senior Engineer with Ant Group where he leads the Program Analysis Group. He is also an Adjunct Academic Staff with UNSW Sydney and Zhejiang University. His research interests include programming languages, compiler technology, and software engineering. He played a key role in co-founding several open-source projects, including SVF, MindSpore, CodeFuse, and others.

Liang Lin is a Collaborating Author with the Database Products Business Unit of Alibaba's Cloud Intelligence Group. He is a Researcher with the Alibaba Group and also serves as the Head with the AnalyticDB MySQL Team. The Alibaba Cloud AnalyticDB MySQL Team actively participates in industry research, continuously improves product capabilities, and is dedicated to providing the market with more agile, efficient, and secure real-time high-concurrency online analytical cloud computing services.

Liang Lin, photograph and biography not available at the time of publication.

ChenXi Cui is a Collaborating Author with the Database Products Business Unit of Alibaba's Cloud Intelligence Group. He is a Member of the AnalyticDB MySQL Engineering Efficiency & Quality team.

ChenXi Cui, photograph and biography not available at the time of publication.