Automatic Generation and Reuse of Precise Library Summaries for Object-Sensitive Pointer Analysis

Jingbo Lu[†] Dongjie He[†] Wei Li[†] Yaoqing Gao[‡] Jingling Xue[†]
[†] *School of Computer Science and Engineering, UNSW Sydney, Australia*
[‡] *Huawei Toronto Research Center, Toronto, Canada*

***Abstract*—The extensive use of libraries in modern software impedes the scalability of pointer analysis. To address this issue, library summarization can be beneficial, but only if the resulting summary-based pointer analysis is faster without sacrificing much precision in the application code. However, currently, no library summarization approaches exist that meet this design objective. This paper presents a novel approach that solves this problem by using k-object-sensitive pointer analysis, *k-obj*, for Java. The approach involves applying *k-obj*, along with a set of summary-based inference rules, to generate a k-object-sensitive library summary. By replacing the program's library with this summary and applying *k-obj*, the efficiency of the program can be significantly improved while maintaining nearly the same or better precision in the application code. We validate our approach with an implementation in SOOT and an evaluation using representative Java programs.**

I. INTRODUCTION

Programs are built increasingly on multi-level library dependencies. When performing pointer analysis on a program, which consists of the application code and the library (referred to as the *aggregate* of both the standard library and the third-party libraries in this paper), its overall precision and efficiency are dependent increasingly on the precision and efficiency achieved in analyzing the library.

Library summarization is known to improve the efficiency of pointer analysis on application code by pre-analyzing the library code and replacing it with a summary that represents its side effects on the application code. This approach avoids repeatedly analyzing the same library methods invoked from different applications or even the same application. However, the effectiveness of library summarization depends on whether it can improve performance without sacrificing precision in the application code. Currently, it is unclear how to do this automatically. In [1], the same summary is obtained for a library by merging the analysis information from all the library pointers into a single set, without considering the different precision needs of downstream analyses, including pointer analysis. As reported in [2], using this imprecise summarization technique can result in an average precision drop of 59% in a cast-check client analysis and a null-pointer client analysis. While users appreciate the performance benefits of summary-based analysis, they still expect the same high level of precision can be achieved in the application code.

We present a novel library summarization approach that improves the efficiency of pointer analysis without sacrificing precision in the application code. Our approach is instantiated by applying the k-objective-sensitive pointer analysis (*k-obj*) for Java, which is widely regarded as the best practice for analyzing object-oriented languages [3]–[7], flow-insensitively but context-sensitively by modeling context-sensitivity in terms of object-sensitivity [8]. The key innovation is to enhance *k-obj* (with a given k value) by augmenting it with a set of summary-based inference rules. These rules are applied to pre-analyze all methods in the library ($\mathcal{L}$) and derive a k-object-sensitive library summary (k-$\mathcal{L}^*$) tailored to the precision requirements of *k-obj* for the given k value. Given k-$\mathcal{L}^*$, applying *k-obj* to analyze different programs by reusing k-$\mathcal{L}^*$ (in place of the library itself) gives rise to the so-called summary-based *k-obj* (*s-k-obj*). In practice, *s-k-obj* runs faster than *k-obj* (by avoiding re-analyzing the library code in $\mathcal{L}$ repeatedly) while yielding better precision overall (albeit slightly poorer precision in some special-case scenarios) in the application code (due to a deeper heap abstraction in k-$\mathcal{L}^*$ achieved by method inlining performed due to library summarization).

In reality, various applications undergo a variety of program analyses on a daily basis [9]. However, the process of upgrading their libraries can extend over several months or even years. Therefore, it is a logical approach to automatically generate accurate library summaries through a pointer analysis algorithm. These summaries can then be reused across multiple applications.

In this paper, we make the following major contributions:

- We present a new approach to generating precise library summaries for supporting pointer analysis.
- We instantiate it by accelerating k-object-sensitive pointer analysis (*k-obj*) for Java programs and provide a prototyping open-source implementation (https://www.cse.unsw.edu.au/~corg/codesum/) in SOOT [10].
- We demonstrate its feasibility in accelerating *k-obj* while achieving even higher precision in the application code for a set of popular Java programs evaluated.

II. MOTIVATION

In *k-obj* ($k \geqslant 1$), each method is analyzed with its receiver objects used as its calling contexts, and an object o_0 is modeled context-sensitively by a *heap context* of length $k-1$, denoted as $[o_1, \cdots, o_{k-1}]$, where o_i is the receiver object of a method in which o_{i-1} is allocated. Therefore, a method with o_0 as its receiver will be analyzed context-sensitively multiple times, once for each of o_0's heap contexts $[o_1, \cdots, o_{k-1}]$, under a so-called *method context* $[o_0, \cdots, o_{k-1}]$ of length k [8].

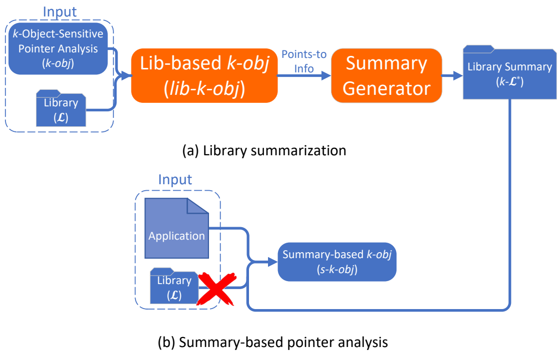

Fig. 1: Automatic library summarization for *k-obj*.

```
1  public abstract class Transformer {
2    Object content;
3    void setContent(Object content) {
4      this.content = content; }
5    Object getContent() {
6      return this.content; }
7    Object transform(Object o) {
8      this.setContent(o);
9      this.run();
10     return this.getContent(); }
11   abstract void run();
12   public static Object transformBy(Transformer t, Object o) {
13     if(t == null)
14       t = new DefaultTransformer(); // DT
15     return t.transform(o); // c } }
   // k−Object−Sensitive Summary of transformBy()
   public static Object transformBy(Transformer t, Object o) {
     return o;
     return t.transform(o); // c }
16 class DefaultTransformer extends Transformer { void run() { } }
```

Fig. 2: Summarizing a library method `transformBy()`.

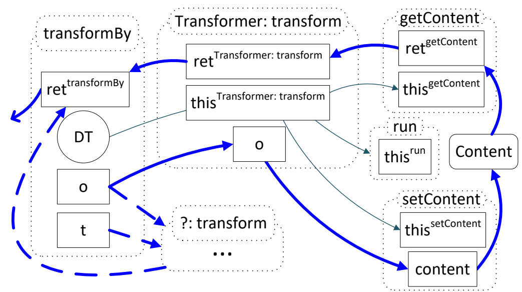

Fig. 3: The value-flows contributing to the side-effect of `transformBy()` for the program given in Figure 2 that may affect the application code, where the three dashed arrows (in blue) represent potential value-flows attributed to an unknown target method invoked by `transform()` at line 15.

A. Library Summarization

Given k-*obj*, Figure 1 illustrates our approach that first generates a k-object-sensitive library summary, denoted k-$\mathcal{L}^*$, for a library $\mathcal{L}$, independently of any application code (Figure 1(a)), and then applies k-*obj* to analyze any program with $\mathcal{L}$ replaced by k-$\mathcal{L}^*$ (Figure 1(b)), resulting in the so-called *summary-based* k-*obj*, denoted *s*-k-*obj*. *For different* k*-limited versions of* k-*obj (with different values of* k *being used), different* k*-limited library summaries* k-$\mathcal{L}^*$ *are produced.*

- **Generating k-$\mathcal{L}^*$ (Figure 1(a)).** There are two stages. First, we augment k-*obj* by adding a small set of summary-based inference rules so that we can apply the thus modified k-*obj* ("*Lib-based* k*-obj*" (denoted *lib*-k-*obj*)) to perform a pointer analysis for $\mathcal{L}$, by introducing summary objects to over-approximate the behavior of the unknown pointed-to information in $\mathcal{L}$. Second, we invoke a side-effect analyzer ("Summary Generator") to emit a k-*obj*-specific library summary (in the form of code statements), k-$\mathcal{L}^*$, based on the points-to information obtained in the first stage.
- **Performing *s*-k-*obj* (Figure 1(b)).** We apply *s*-k-*obj* to any program with its library $\mathcal{L}$ being replaced by k-$\mathcal{L}^*$ to perform the so-called summary-based pointer analysis.

B. Two Motivating Examples

We use two examples to illustrate how our approach works.

1) Over-Approximating Unknown Points-to Information with Summary Objects: In our first example shown in Figure 2, we use *lib*-k-*obj* to summarize the behavior of a library method called `transformBy()` that's invoked from the application code. Our goal is to illustrate how to use summary objects to over-approximate the behavior of unknown objects received from the application code so that context-sensitivity is irrelevant (implying that k-$\mathcal{L}^*$ remains the same even when k varies). We aim to generate a summary for `transformBy()` (shown in an orange box in Figure 2). Figure 3 depicts all the relevant value flows when `transformBy()` is invoked, with those producing cumulatively the side-effect visible to the application code being highlighted in blue. The behavior of `transformBy()` is determined mainly by the call `t.transform(o)` at line 15, whose target methods depend on the dynamic types of the receiver objects pointed to by `t`. Here, `t` may point to `DT` created directly in `transformBy()` (line 14) or any unknown objects passed into this method from the application code (line 12). Thus, this call has not only a known target, `transform()` defined in class `Transformer` (with the related value flows depicted in solid arrows), but also many other unknown targets (with the related value flows depicted in dashed arrows). In the former case, the cumulative effect of the call can be summarized by one statement, "`return o`", with the callee `transform()` in lines 7-10 inlined. In the latter case, we just have to keep the call conservatively as it is, "`return t.transform(o)`."

To obtain the summary for `transformBy()`, we apply *lib*-k-*obj* with `transformBy()` as the entry method. Let $\mathtt{t}^{\mathtt{p}}$ and $\mathtt{o}^{\mathtt{p}}$ be two summary objects abstracting all the unknown objects pointed to by `t` and `o` (the two parameters of `transformBy()`), respectively, and $\mathtt{c}^{\mathtt{r}}$ be a summary object abstracting all the unknown objects returned at callsite `c` (line 15). For each method m, we write $\mathtt{ret}^m$ to represent its unique return variable storing its return value and $\mathtt{this}^m$ to represent its *this* variable. Figure 4 gives the points-to information obtained, with only the two points-to relations highlighted in red ($\mathtt{ret}^{\mathtt{transformBy}}$ pointing to $\mathtt{o}^{\mathtt{p}}$ and $\mathtt{c}^{\mathtt{r}}$) and the call statement that is associated with an unknown object $\mathtt{c}^{\mathtt{r}}$ as its return value to be summarized. The three red circles mark the three summary objects thus introduced.

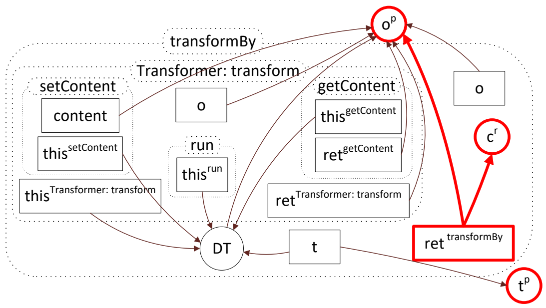

Fig. 4: The points-to information for summarizing `transformBy()` in Figure 2 with the two points-to relations in red ($\texttt{ret}^{\texttt{transformBy}}$ pointing to $\texttt{o}^{\texttt{P}}$ and $\texttt{c}^{\texttt{r}}$) and the call at callsite `c` with unknown targets to be summarized. The three summary objects used are marked by red circles.

Given the points-to information, we can obtain the summary for `transformBy()` in Figure 2 by analyzing its side-effect on the application code, identified by its return statement `t.transform(o)` at callsite `c` (line 15). Its return variable $\texttt{ret}^{\texttt{transformBy}}$ points to $\texttt{o}^{\texttt{P}}$ and $\texttt{c}^{\texttt{r}}$ (determined when `t` points to `DT` and $\texttt{t}^{\texttt{P}}$, respectively). In the former case, the cumulative effect of this call can be summarized (by method inlining) into one statement, "`return o`", with all the intermediate points-to relations that would otherwise have to be computed (repeatedly) by *k-obj* eliminated. In the latter case, the statement "`return t.transform(o)`", which may potentially have callbacks in the application code, is simply retained.

2) Making Library Method Summaries k-Object-Sensitive: Consider our second example, where we aim to summarize `A.goo()` invoked by the application code. This example is designed to illustrate the importance of customizing k-$\mathcal{L}^*$ for *k-obj* by making it k-object-sensitive in general, highlighting the relevance of context-sensitivity when k varies.

```
public class X { void foo() { ... } }        // Context-Insensitive Summary
class Y extends X { void foo() { ... } }     public static X goo() {
public class A {                               return new X(); // X1
  X id(X x) { return x; }                      return new Y(); // Y1 }
  public static X goo() {
    X x = new X(); // X1                     // k-Object-Sensitive Summary
    X y = new Y(); // Y1                     public static X goo() {
    A a1 = new A(); // A1                      return new X(); // X1 }
    A a2 = new A(); // A2
    X xx = a1.id(x);                         // Application Code
    X yy = a2.id(y);                         X x = A.goo();
    return xx; } }                           x.foo();
```

Fig. 5: k-object-sensitive method summaries for `goo()`.

In the two orange boxes, we give two summaries for `A.goo()`: (1) the context-insensitive summary obtained by applying a standard context-insensitive pointer analysis, and (2) the k-object-sensitive summary obtained by *lib-k-obj*. The former is less precise since it does not incorporate context-sensitivity in summary generation (causing the return values from `id()` to be conflated). If we want to apply *s-1-obj* to perform the summary-based pointer analysis for the program context-sensitively, *s-1-obj* achieves the same precision as the whole-program analysis counterpart *1-obj* if the former is used, but loses precision if the latter is used instead.

III. Library Summarization

Our approach, as shown in Algorithm 1, takes as input (1) $\mathcal{L}$ (a library consisting of a set of library methods) and (2) *k-obj* (a specification of *k-obj* for a given value of k), and produces as output k-$\mathcal{L}^*$ (a *k-obj*-specific summary k-$\mathcal{L}^*$), by proceeding in the two stages given in Figure 1. For each library method *mtd* $\in \mathcal{L}$, we first apply *lib-k-obj* to perform a library-based pointer analysis with *mtd* as the only entry method. Based on the points-to information $\texttt{pts}_{mtd}$ obtained, we then invoke `genSumCode()` to generate a summary *mtd** for *mtd* by performing a side-effect analysis.

Algorithm 1: Creating k-$\mathcal{L}^*$ for a library $\mathcal{L}$.

Input: ($\mathcal{L}$, *k-obj*)
Output: k-$\mathcal{L}^*$

1. k-$\mathcal{L}^* \leftarrow \varnothing$;
2. *lib-k-obj* $\leftarrow$ *k-obj* augmented with summary-based rules;
3. **foreach** *mtd* $\in \mathcal{L}$ **do**
4.   **Stage 1:** $\texttt{pts}_{mtd} \leftarrow \texttt{pts}$ returned by applying *lib-k-obj* to $\mathcal{L}$ with *mtd* as the entry method;
5.   **Stage 2:** *mtd** $\leftarrow \texttt{genSumCode}(mtd, \texttt{pts}_{mtd})$;
6. k-$\mathcal{L}^* \ni$ *mtd**;

As is standard, we use a simplified Java language, which is an IR containing eight kinds of statements in Table I, to specify *k-obj* and formalize our summarization approach.

TABLE I: Intermediate representation.

Kind	Statement	Kind	Statement
NEW	$x = \textbf{new } T$	ASSIGN	$x = y$
GLOBALSTORE	$g = x$	GLOBALLOAD	$x = g$
STORE	$x.f = y$	LOAD	$x = y.f$
CAST	$x = (T)y$	CALL	$x = y.m(a_1, ..., a_n)$

As *k-obj* is flow-insensitive, control flow statements are elided. (Instance) methods are stylized to have a single return variable. Static methods are omitted as they are analyzed by using the calling contexts of their closest callers that are instance methods on the call stack [3]–[7]. For a method m, $\texttt{ret}^m$ denotes its return variable and p_i^m its i-th parameter (starting from 0), with p_0^m representing its *this* variable.

When summarizing a library, we distinguish between *known* objects and *unknown* objects. Known objects are created explicitly in the library code, while unknown objects are passed into the library from the application code or created by an unknown target method invoked at a call statement in the library. To handle unknown objects, we introduce summary objects that over-approximate the behavior of unknown points-to information in the library, including the behavior of known objects whose fields may point to unknown objects. By over-approximating the points-to information, we ensure soundness for *lib-k-obj*. If a known object is likely to be passed to an unknown target method, it is converted into a summary object. Any field of a summary object may point to unknown objects. When dealing with summary objects with inferred upper type bounds, downcasting is handled conservatively.

We will use the standard notations in Figure 6, with the exception of those related to handling summary objects dis-

types	$T \in \mathbb{T}$
methods	$m \in \mathbb{M}$
allocation sites	$o \in \mathbb{O}$
local variables	$v, x, y \in \mathbb{V}$
global variables	$g \in \mathbb{G}$
instance fields	$f \in \mathbb{F}$
contexts	$c \in \mathbb{C} = \mathbb{O}^0 \cup \mathbb{O}^1 \cup \mathbb{O}^2 ...$
context-sensitive heap objects	$\langle o, c\rangle \in \mathbb{O} \times \mathbb{C}$
summary objects	$s \in \mathbb{S} = \mathbb{S}_{para} \cup \mathbb{S}_{glob} \cup \mathbb{S}_{load} \cup \mathbb{S}_{cast} \cup \mathbb{S}_{ret} \cup \mathbb{S}_{arg}$
abstract heap objects	$h \in \mathbb{H} = (\mathbb{O} \times \mathbb{C}) \cup \mathbb{S}$
heap references	$h.f \in \mathbb{H} \times \mathbb{F}$
abstract pointers	$p \in \mathbb{P} = (\mathbb{V} \times \mathbb{C}) \cup (\mathbb{H} \times \mathbb{F}) \cup \mathbb{G}$
ctx:	$\mathbb{M} \hookrightarrow \mathcal{P}(\mathbb{C})$
pts:	$\mathbb{P} \hookrightarrow \mathcal{P}(\mathbb{H})$
typeOf:	$\mathbb{V} \cup \mathbb{G} \cup (\mathbb{V} \times \mathbb{F}) \cup \mathbb{H} \hookrightarrow \mathbb{T}$
`isTypeExact`:	$\mathbb{H} \hookrightarrow \{\mathsf{true}, \mathsf{false}\}$
fieldRefsOf:	$\mathbb{H} \hookrightarrow \mathcal{P}(\mathbb{H} \times \mathbb{F})$
getParaObj:	$\mathbb{V} \hookrightarrow \mathbb{S}_{para}$
getGlobalObj:	$\mathbb{G} \hookrightarrow \mathbb{S}_{glob}$
getLoadObj:	$\mathbb{V} \times \mathbb{F} \times \mathbb{C} \hookrightarrow \mathbb{S}_{load}$
getCastObj:	$\mathbb{T} \times \mathbb{V} \times \mathbb{C} \hookrightarrow \mathbb{S}_{cast}$
getRetObj:	$\mathbb{V} \times \mathbb{C} \hookrightarrow \mathbb{S}_{ret}$

Fig. 6: The notations used.

cussed in Sections III-A and III-B. For now, we remark that we distinguish six categories of summary objects contained in $\mathbb{S}_{para}$, $\mathbb{S}_{glob}$, $\mathbb{S}_{load}$, $\mathbb{S}_{cast}$, $\mathbb{S}_{ret}$, and $\mathbb{S}_{arg}$, where each of the first five contains only unknown objects (created freshly by calling *getParaObj*(), *getGlobalObj*(), *getLoadObj*(), *getCastObj*(), and *getRetObj*(), respectively) and the last one contains only known objects whose fields may point to unknown objects. When analyzing a library, a *known object* $o \in \mathbb{O}$ is one that is created at an object allocation statement "... = **new** T" in the library, with an exact type $T \in \mathbb{T}$ such that $\texttt{typeOf}(o) = T$ and $\texttt{isTypeExact}(o) = \mathsf{true}$. On the other hand, an *unknown object* $s \in \mathbb{S} \setminus \mathbb{S}_{arg}$ is one that is passed to the library from the application code or created by an unknown target method invoked at a call statement in the library with an inferred upper type bound $T \in \mathbb{T}$ such that $\texttt{typeOf}(s) = T$ and $\texttt{isTypeExact}(s) = \textit{isFinalClass}(T)$, where *isFinalClass* is defined as follows:

$$\textit{isFinalClass}(T) = \textbf{if } T \text{ is a final class } \to \mathsf{true} \textbf{ else} \to \mathsf{false} \textbf{ fi}$$

A. Stage 1: Performing lib-k-obj

Figure 7 gives the rules for analyzing a library $\mathcal{L}$ by applying *lib-k-obj*, which is obtained from *k-obj* (specified by a set of standard inference rules) by adding five additional summary-based inference rules. To compute $\texttt{pts}_{mtd}$ for $mtd \in \mathcal{L}$ (Stage 1 of Algorithm 1), we simply apply *lib-k-obj* to $\mathcal{L}$ with *mtd* as the entry of the analysis.

1) Standard Inference Rules: This set of rules for performing an inclusion-based *k-obj* for Java is standard [6], [11]–[13], with `pts` recording the points-to information found. In [NEW], for an object allocation statement $x = \texttt{new}\ T$ `// o`, *heapCTXSelector(c)* returns a heap context for modeling o object-sensitively based on the current method (calling) context c used for analyzing its containing method m. Specifically, if $c = [c_0, ..., c_{k-1}]$, then *heapCTXSelector(c)* $= [c_0, ..., c_{k-2}]$. In [CALL], $metCTXSelector(h)$ returns a new method context c'' for analyzing the new method $m'' = dispatch(h, m')$ dispatched. Specifically, if $h = \langle o, [c_0, ..., c_{k-2}]\rangle$, then $metCTXSelector(h) = [o, c_0, ..., c_{k-2}]$.

When these standard rules are used to analyze a program without using a library summary, $\texttt{typeOf}(h)$ gives the exact type of h in [CAST] and $\texttt{isTypeExact}(h) = \mathsf{true}$ in [CALL] due to [NEW]. However, when used to summarize a library method, these standard rules will handle unknown objects identically as known objects, with two caveats. First, for an unknown object $h \in \mathbb{S} \setminus \mathbb{S}_{arg}$, [CAST] is still applied if $\texttt{typeOf}(h) <: T$, i.e., the upper type bound of h is a subtype of T. Otherwise, we defer to the corresponding summary-based rule to handle it. Second, [CALL] is applied only if the receiver h of the call statement is a known object, i.e., when $\texttt{isTypeExact}(h) = \mathsf{true}$. Otherwise, we will also leave it to be handled by the corresponding summary-based rule.

2) Summary-Based Inference Rules: To ensure the soundness of *lib-k-obj*, we use five summary-based rules for introducing six types of summary objects (Figure 6). These summary-based rules are applied when processing the (entry) library method summarized ([PARAINIT]), cast statements ([CAST]), global loads ([GLOBALLOAD]), loads ([LOAD]), and call statements ([CALL]). Note that [CALL] contains two sub-rules, each introducing a different kind of summary objects. For (1) object creation statements ([NEW]), (2) assignments ([ASSIGN]), (3) global stores ([GLOBALSTORE]), and (4) stores ([STORE]), no summary-based rules are needed. For (1) – (3), the corresponding standard inferences rule suffice. For (4), there is no need to update a field of a summary object since it is known to point to unknown objects.

Formal Parameters (ParaObjs in $\mathbb{S}_{para}$) When m is the entry method to be summarized (i.e., *mtd* in line 4 of Algorithm 1), we apply [PARAINIT] to capture the unknown objects received from the application code and assigned to its parameters. For each p_i^{mtd}, we introduce a distinct summary object (i.e., *ParaObj*), $s = \textit{getParaObj}(p_i^{mtd})$, where $s \in \mathbb{S}_{para}$, to abstract all unknown objects pointed to by p_i^{mtd}. As the entry of *lib-k-obj* (line 4 of Algorithm 1), the context for p_i^{mtd} is $[\,]$. We set $\texttt{typeOf}(s) = \texttt{typeOf}(p_i^{mtd})$, i.e., the upper type bound of s as the declared type of p_i^{mtd}. In addition, we set $\texttt{isTypeExact}(s) = \textit{isFinalClass}(\texttt{typeOf}(p_i^{mtd}))$.

Global Variables (GlobalObjs in $\mathbb{S}_{glob}$) A global load $x{=}g$ can also be an entry into the library via an external pointer from the application code. For a global variable g (handled context-insensitively) by [GLOBALLOAD], we introduce a distinct summary object (i.e., *GlobalObj*), $s = \textit{getGlobalObj}(g)$, where $s \in \mathbb{S}_{glob}$, to abstract all unknown objects pointed to by g and assign it to (x, c). We set $\texttt{typeOf}(s) = \texttt{typeOf}(g)$ and $\texttt{isTypeExact}(s) = \textit{isFinalClass}(\texttt{typeOf}(g))$, similarly as we do for formal parameters.

Loads (LoadObjs in $\mathbb{S}_{load}$) For a load $x = y.f$, where y points to some summary object $h \in \mathbb{S}$, $h.f$ may point to some unknown objects. According to [LOAD], we introduce context-sensitively a summary object (i.e., *LoadObj*), $s = \textit{getLoadObj}(y, f, c)$, where $s \in \mathbb{S}_{load}$, to abstract all the unknown objects pointed to by $y.f$ under context c and assign

Stmt in Method m	Standard Inference Rules for *k-obj*	Summary-Based Inference Rules Introduced	
		$\frac{\begin{array}{c} m = \mathit{mtd} \text{ is the entry method summarized} \\ T = \texttt{typeOf}(p_i^{\mathit{mtd}}),\ 0 \le i \le n \end{array}}{\begin{array}{c} s = \mathit{getParaObj}(p_i^{\mathit{mtd}}),\ s \in \texttt{pts}(p_i^{\mathit{mtd}}, [\,]) \\ \texttt{typeOf}(s) = T \quad \texttt{isTypeExact}(s) = \mathit{isFinalClass}(T) \end{array}}$	[ParaInit]
$x = \text{new T}$ // o	$\frac{c \in \texttt{ctx}(m)}{\begin{array}{c} \langle o, \mathit{heapCTXSelector}(c) \rangle \in \texttt{pts}(x, c) \\ \texttt{typeOf}(o) = T \quad \texttt{isTypeExact}(o) = \text{true} \end{array}}$		[New]
$x = y$	$\frac{c \in \texttt{ctx}(m)}{\texttt{pts}(y, c) \subseteq \texttt{pts}(x, c)}$		[Assign]
$x = (T)y$	$\frac{c \in \texttt{ctx}(m) \quad h \in \texttt{pts}(y, c) \quad \texttt{typeOf}(h) <: T}{h \in \texttt{pts}(x, c)}$	$\frac{c \in \texttt{ctx}(m) \quad h \in \texttt{pts}(y, c) \quad \texttt{typeOf}(h) \not<: T \quad \neg\texttt{isTypeExact}(h)}{\begin{array}{c} s = \mathit{getCastObj}(T, y, c), s \in \texttt{pts}(x, c) \\ \texttt{typeOf}(s) = T \quad \texttt{isTypeExact}(s) = \mathit{isFinalClass}(T) \end{array}}$	[Cast]
$x = g$	$\frac{c \in \texttt{ctx}(m)}{\texttt{pts}(g) \subseteq \texttt{pts}(x, c)}$	$\frac{c \in \texttt{ctx}(m) \quad T = \texttt{typeOf}(g)}{\begin{array}{c} s = \mathit{getGlobalObj}(g), s \in \texttt{pts}(x, c) \\ \texttt{typeOf}(s) = T \quad \texttt{isTypeExact}(s) = \mathit{isFinalClass}(T) \end{array}}$	[GlobalLoad]
$g = x$	$\frac{c \in \texttt{ctx}(m)}{\texttt{pts}(x, c) \subseteq \texttt{pts}(g)}$		[GlobalStore]
$x = y.f$	$\frac{c \in \texttt{ctx}(m) \quad h \in \texttt{pts}(y, c)}{\texttt{pts}(h.f) \subseteq \texttt{pts}(x, c)}$	$\frac{c \in \texttt{ctx}(m) \quad h \in \texttt{pts}(y, c) \quad h \in \mathbb{S} \quad T = \texttt{typeOf}(y.f)}{\begin{array}{c} s = \mathit{getLoadObj}(y, f, c), s \in \texttt{pts}(x, c) \\ \texttt{typeOf}(s) = T \quad \texttt{isTypeExact}(s) = \mathit{isFinalClass}(T) \end{array}}$	[Load]
$x.f = y$	$\frac{c \in \texttt{ctx}(m) \quad h \in \texttt{pts}(x, c)}{\texttt{pts}(y, c) \subseteq \texttt{pts}(h.f)}$		[Store]
$x = y.m'(a_1, ..., a_n)$	$\frac{\begin{array}{c} c \in \texttt{ctx}(m) \quad h \in \texttt{pts}(y, c) \quad \texttt{isTypeExact}(h) \\ m'' = \mathit{dispatch}(h, m') \quad c'' = \mathit{metCTXSelector}(h) \end{array}}{\begin{array}{c} c'' \in \texttt{ctx}(m'') \quad h \in \texttt{pts}(p_0^{m''}, c'') \\ \texttt{pts}(a_i, c) \subseteq \texttt{pts}(p_i^{m''}, c''), 1 \le i \le n \\ \texttt{pts}(\texttt{ret}^{m''}, c'') \subseteq \texttt{pts}(x, c) \end{array}}$	$\frac{\begin{array}{c} c \in \texttt{ctx}(m) \quad h \in \texttt{pts}(y, c) \\ \neg\texttt{isTypeExact}(h) \quad T = \texttt{typeOf}(x) \end{array}}{\begin{array}{c} s = \mathit{getRetObj}(x, c),\ s \in \texttt{pts}(x, c) \\ \texttt{typeOf}(s) = T \\ \texttt{isTypeExact}(s) = \mathit{isFinalClass}(T) \end{array}}$ $\quad$ $\frac{\begin{array}{c} c \in \texttt{ctx}(m) \quad h \in \texttt{pts}(y, c) \quad \neg\texttt{isTypeExact}(h) \\ h' \in \texttt{pts}(a_i, c), 1 \le i \le n \quad h' \notin \mathbb{S} \end{array}}{h' \in \mathbb{S}_{\mathit{arg}}}$	[Call]

Fig. 7: Rules for *lib-k-obj* (where *mtd* is the library method being summarized).

it to (x, c). We set $\texttt{typeOf}(s) = \texttt{typeOf}(y.f)$, which is the declared type of field f in the declared type of y, and $\texttt{isTypeExact}(s) = \mathit{isFinalClass}(\texttt{typeOf}(y.f))$.

Cast Statements (CastObjs in $\mathbb{S}_{\mathit{cast}}$) In a cast statement $x = (T)\ y$ handled by [Cast], y points to an unknown object h whose type is not exactly known (since $\texttt{isTypeExact}(h) = \text{false}$). Given $\texttt{typeOf}(h) \not<: T$, we cannot filter out h, as $\texttt{typeOf}(h)$ is only an upper type bound for h. As the actual type of h may be any subtype of T, we introduce context-sensitively a summary object (i.e., *CastObj*), $s = \mathit{getCastObj}(T, y, c)$, where $s \in \mathbb{S}_{\mathit{cast}}$, to abstract all the unknown objects pointed to by y under context c and assign it to (x, c). We set $\texttt{typeOf}(s) = T$ and $\texttt{isTypeExact}(s) = \mathit{isFinalClass}(T)$.

Call Returns (RetObjs in $\mathbb{S}_{\mathit{ret}}$) and Arguments (ArgObjs in $\mathbb{S}_{\mathit{arg}}$) We apply [Call], which consists of two sub-rules, to handle a call statement $x = y.m'(a_1, ..., a_n)$ with an unknown target method (e.g., a possible callback in the application code), denoted $\mathcal{U}$ here, since y points to an unknown receiver h whose type is not known exactly (i.e., $\texttt{isTypeExact}(h) = \text{false}$). In this case, the objects (1) returned from the return variable of $\mathcal{U}$, (2) passed into $\mathcal{U}$ via the arguments of the call, and (3) pointed to by global variables may all potentially be modified by $\mathcal{U}$. According to the first sub-rule, we handle (1) by introducing context-sensitively a summary object (i.e., *RetObj*), $s = \mathit{getRetObj}(x, c)$, where $s \in \mathbb{S}_{\mathit{ret}}$, to abstract all the unknown objects returned by $\mathcal{U}$ under context c and assign it to (x, c). According to the second sub-rule, we handle (2) by turning every non-summary object $h' \notin \mathbb{S}$ (i.e., every known object created in the library) into a summary object, where $h' \in \mathbb{S}_{\mathit{arg}}$, if it is passed via a non-receiver-variable argument into $\mathcal{U}$, since its fields may now be modified by $\mathcal{U}$ to point to unknown objects. There is no need to add any existing summary object $s \in \mathbb{S}$ passed into $\mathcal{U}$ at such a call statement via any of its arguments to $\mathbb{S}_{\mathit{arg}}$ (if it is not there yet), since its fields are known to point to unknown objects potentially, by definition. Finally, we handle (3) by doing nothing, since in every global load $x = g$, g is known to point to an unknown object represented by $\mathit{getGlobalObj}(g)$ ([GlobalLoad]).

B. Stage 2: Generating k-$\mathcal{L}^*$

Algorithm 2 emits $\mathit{mtd}^* \leftarrow \texttt{genSumCode}(\mathit{mtd}, \texttt{pts}_{\mathit{mtd}})$, where $\mathit{mtd}^* \in k$-$\mathcal{L}^*$ is a summary of $\mathit{mtd} \in \mathcal{L}$ so that mtd^* exhibits the same points-to information $\texttt{pts}_{\mathit{mtd}}$. This summary consists of a sequence of statements producing the points-to side-effect of *mtd* visible to the application code as per $\texttt{pts}_{\mathit{mtd}}$.

Conceptually, $\texttt{genSumCode}()$ is simple. We use an auxiliary function from Algorithm 3, $v = \texttt{emitStmts}(\mathit{mtd}, h)$, where $h \in \mathbb{H}$, to emit a sequence of statements for fetching h into a local variable (with its name) stored in v. The code sequence generated for h, which is uniquely identified by this local variable, is responsible for defining h and its fields transitively based on $\texttt{pts}_{\mathit{mtd}}$. In addition to some notations in Figure 6, we also use $\texttt{freshLocalVar}(T)$, where $T \in \mathbb{T}$, to return a new local variable with T as its declared type. When we write $\mathit{mtd}^* \ni$ "$v.f = v'$;", for example, we mean that we emit a store statement into *mtd**, with the variable name

Algorithm 2: $mtd^* \leftarrow \texttt{genSumCode}(mtd, \texttt{pts}_{mtd})$

Input: $(mtd, \texttt{pts}_{mtd})$
Output: mtd^*

1 $mtd^* \leftarrow \varnothing$;
2 **foreach** $h \in \texttt{pts}_{mtd}(\texttt{ret}^{mtd}, [\,])$ **do**
3 $\quad v \leftarrow \texttt{emitStmts}(mtd, h)$;
4 $\quad mtd^* \ni$ ``$\texttt{return}\ v\texttt{;}$'';
5 **foreach** $g \in \mathbb{G}$ **do**
6 $\quad$ **foreach** $h \in \texttt{pts}_{mtd}(g)$ **do**
7 $\quad\quad v \leftarrow \texttt{emitStmts}(mtd, h)$;
8 $\quad\quad mtd^* \ni$ "$g = v\texttt{;}$";
9 **foreach** $s \in \mathbb{S} \setminus \mathbb{S}_{arg}$ **do**
10 $\quad v \leftarrow \texttt{emitStmts}(mtd, s)$;
11 $\quad$ **foreach** $s.f \in \textit{fieldRefsOf}(s)$ **do**
12 $\quad\quad$ **foreach** $h \in \texttt{pts}_{mtd}(s.f)$ **do**
13 $\quad\quad\quad v' \leftarrow \texttt{emitStmts}(mtd, h)$;
14 $\quad\quad\quad mtd^* \ni$ "$v.f = v'\texttt{;}$";
15 **foreach** $x = y.m(a_1, ..., a_n)$ analyzed under context c, where $h \in \texttt{pts}_{mtd}(y, c)$, such that $\texttt{isTypeExact}(h) =$ false **do**
16 $\quad s \leftarrow \textit{getRetObj}(x, c)$;
17 $\quad x' \leftarrow \texttt{emitStmts}(mtd, s)$;
18 $\quad T \leftarrow \texttt{typeOf}(y)$;
19 $\quad y' \leftarrow \texttt{freshLocalVar}(T)$;
20 $\quad$ **foreach** $h \in \texttt{pts}_{mtd}(y, c)$ **do**
21 $\quad\quad$ **if** $\neg \textit{isTypeExact}(h)$ **then**
22 $\quad\quad\quad v \leftarrow \texttt{emitStmts}(mtd, h)$;
23 $\quad\quad\quad mtd^* \ni$ "$y' = v\texttt{;}$";
24 $\quad$ **foreach** $a_i \in \{a_1, ..., a_n\}$ **do**
25 $\quad\quad T \leftarrow \texttt{typeOf}(a_i)$;
26 $\quad\quad a'_i \leftarrow \texttt{freshLocalVar}(T)$;
27 $\quad\quad$ **foreach** $h \in \texttt{pts}_{mtd}(a_i, c)$ **do**
28 $\quad\quad\quad v \leftarrow \texttt{emitStmts}(mtd, h)$;
29 $\quad\quad\quad mtd^* \ni$ "$a'_i = v\texttt{;}$";
30 $\quad mtd^* \ni$ "$x' = y'.m(a'_1, ..., a'_n)\texttt{;}$";

contained in v, the field name in f, and the variable name in v' being used (as is standard in compiler code generation).

To understand genSumCode(), it suffices to examine four different kinds of statements reached transitively from *mtd* in the library $\mathcal{L}$ (handled by its four **for** loops), as they produce the points-to side effect of *mtd* visible to the application code:

- **Return Values (Lines 2-4).** For every object pointed to by $\texttt{ret}^{mtd}$ of *mtd*, where *mtd* is the entry method summarized (line 2), we emit the code for fetching h into a unique local variable stored in v (line 3) and "$\texttt{return}\ v$" (line 4). This will be illustrated in Section III-C1.
- **Global Stores (Lines 5-8).** If a global variable $g \in \mathbb{G}$ (line 5) is modified in a global store " $g = ...$" in the library, then $\texttt{pts}(g) \neq \varnothing$. For each of its pointed-to objects h (line 6), where $h \in \mathbb{S}_{glob}$, we generate the code required for fetching h into a unique local variable v that represents h (line 7) and an assignment $g = v$ (line 8).
- **(Local) Stores into the Fields of Unknown Objects (Lines 9-14).** For an unknown object $s \in \mathbb{S} \setminus \mathbb{S}_{arg}$ (line 9) modified in a local store "$x.f = ...$", we have $\texttt{pts}(s.f) \neq \varnothing$. We first generate the code for fetching s into a unique local variable stored in v (line 10). Then we do the following for every such a field access $s.f$ (line 11). For each object h pointed by $s.f$, we generate the code for fetching h into a unique local variable stored in v' and a store "$v.f = v'$" as desired (lines 12-14). Here, we ignore the known objects $s \in \mathbb{S}_{arg}$ since if they are accessible to the application code, say, due to being pointed by the return variable $\texttt{ret}^{mtd}$ of *mtd*, then computing the side-effect of $\texttt{ret}^{mtd}$ (lines 2-4 of Algorithm 2) will include the code responsible for its definitions in *mtd** due the last case of Algorithm 3.

Algorithm 3: emitStmts for emitting code fetching $h \in \mathbb{H}$ into a unique local variable as per $\texttt{pts}_{mtd}$.

Input: (*mtd*, *h*)
Output: *v*

1 **if** *code has already been generated for* h **then**
2 $\quad v \leftarrow$ the unique local variable for representing h
3 **else if** $h \in \mathbb{S}_{para}$ **then**
4 $\quad v \leftarrow \textit{getParaObj}^{-1}(h)$;
5 **else if** $h \in \mathbb{S}_{glob}$ **then**
6 $\quad g \leftarrow \textit{getGlobalObj}^{-1}(h)$;
7 $\quad T \leftarrow \texttt{typeOf}(g)$;
8 $\quad v \leftarrow \texttt{freshLocalVar}(T)$;
9 $\quad mtd^* \ni$ "$v = g\texttt{;}$";
10 **else if** $h \in \mathbb{S}_{load}$ **then**
11 $\quad (x, f, c) \leftarrow \textit{getLoadObj}^{-1}(h)$;
12 $\quad T \leftarrow \texttt{typeOf}(x.f)$;
13 $\quad v \leftarrow \texttt{freshLocalVar}(T)$;
14 $\quad T' \leftarrow \texttt{typeOf}(x)$;
15 $\quad v' \leftarrow \texttt{freshLocalVar}(T')$;
16 $\quad mtd^* \ni$ "$v = v'.f\texttt{;}$";
17 $\quad$ **foreach** $h' \in \textit{pts}(x, c)$ **do**
18 $\quad\quad$ **if** $h' \in \mathbb{S}$ **then**
19 $\quad\quad\quad v'' \leftarrow \texttt{emitStmts}(mtd, h')$;
20 $\quad\quad\quad mtd^* \ni$ "$v' = v''\texttt{;}$";
21 **else if** $h \in \mathbb{S}_{cast}$ **then**
22 $\quad (T, x, c) \leftarrow \textit{getCastObj}^{-1}(h)$;
23 $\quad v \leftarrow \texttt{freshLocalVar}(T)$;
24 $\quad T' \leftarrow \texttt{typeOf}(x)$;
25 $\quad v' \leftarrow \texttt{freshLocalVar}(T')$;
26 $\quad mtd^* \ni$ "$v = (T)v'\texttt{;}$";
27 $\quad$ **foreach** $h' \in \textit{pts}(x, c)$ **do**
28 $\quad\quad$ **if** $\neg \textit{isTypeExact}(h')$ **then**
29 $\quad\quad\quad v'' \leftarrow \texttt{emitStmts}(mtd, h')$;
30 $\quad\quad\quad mtd^* \ni$ "$v' = v''\texttt{;}$";
31 **else if** $h \in \mathbb{S}_{ret}$ **then**
32 $\quad T \leftarrow \texttt{typeOf}(h)$;
33 $\quad v \leftarrow \texttt{freshLocalVar}(T)$;
34 **else**
35 $\quad T \leftarrow \texttt{typeOf}(h)$;
36 $\quad v \leftarrow \texttt{freshLocalVar}(T)$;
37 $\quad mtd^* \ni$ "$v =$ new $T\texttt{;}$";
38 $\quad$ **foreach** $h.f \in \textit{fieldRefsOf}(h)$ **do**
39 $\quad\quad$ **foreach** $h' \in \textit{pts}(h.f)$ **do**
40 $\quad\quad\quad v' \leftarrow \texttt{emitStmts}(mtd, h')$;
41 $\quad\quad\quad mtd^* \ni$ "$v.f = v'\texttt{;}$";

- **Unknown Call Target Methods (Lines 15-30).** For a call with an unknown target (line 15), we must preserve it in *mtd** and include all the associated statements that may potentially modify its arguments and return values (lines 16-30), as shown in Section III-C1. For $x' = y'.m(a'_1, ..., a'_n)$ added to *mtd** (line 30) under a given context c (line 15), we also add to *mtd** the supporting code that (1) assigns its return value to x' (lines 16-17), (2) defines the receiver variable y' in terms of all unknown receiver objects (lines 18-23), and (3) defines all non-receiver-variable arguments (lines 24-29). Effectively, every call analyzed, with at least one known target method, is inlined in this entry method *mtd* context-sensitively. *Such method inlining is responsible for the precision gain in the application code achieved by the summary-based pointer analysis (Section III-D2).*

Finally, $v = \texttt{emitStmts}(\textit{mtd}, h)$ (Algorithm 3) calls itself recursively to emit a series of statements for fetching h into a unique h-specific local variable stored in v (lines 1-2). This is done by distinguishing six cases, depending on whether h is one of the five types of unknown objects (in $\mathbb{S}_{para} \cup \mathbb{S}_{glob} \cup \mathbb{S}_{load} \cup \mathbb{S}_{cast} \cup \mathbb{S}_{ret}$) or a known object (in $\mathbb{S}_{arg}$ or not).

C. Examples

We illustrate our approach by considering a few examples.

1) Example 1: For our first motivating example in Figure 2, let us obtain the summary for $\texttt{transformBy}^*$ (depicted in the orange box) automatically. Note that `return t.transform(o)`" is split into $\texttt{ret}^{\texttt{transformBy}} = \texttt{t.transform(o)}$ and $\texttt{return ret}^{\texttt{transformBy}}$.

Let us first apply *lib-k-obj* to obtain the points-to information $\texttt{pts}_{\texttt{transformBy}}$ in Figure 4, where three summary objects, $\texttt{o}^{\texttt{p}}$, $\texttt{t}^{\texttt{p}}$, and $\texttt{c1}^{\texttt{r}}$ are shown. As `transformBy()` is the entry, $\texttt{o}^{\texttt{p}} :=_{\text{def}} \textit{getParaObj}(o)$ and $\texttt{t}^{\texttt{p}} :=_{\text{def}} \textit{getParaObj}(t)$. For the call $\texttt{ret}^{\texttt{transformBy}} = \texttt{t.transform(o)}$, its receiver variable `t` may point to `DT` and $\texttt{t}^{\texttt{p}}$ since $\texttt{pts}_{\texttt{transformBy}}(t, [\,]) = \{\texttt{DT}, \texttt{t}^{\texttt{p}}\}$. We can now analyze it under $[\,]$ with the receiver being `DT` by using the (original) inference rule for *k-obj* in [Call]. We obtain $\texttt{o}^{\texttt{p}} \in \texttt{pts}_{\texttt{transformBy}}(\texttt{ret}^{\texttt{transformBy}}, [\,])$. Next, we analyze this call under $[\,]$ with the receiver object being an unknown object, $\texttt{t}^{\texttt{p}}$, by using the first summary-based sub-rule in [Call]. This time, we create the third summary object: $\texttt{c}^{\texttt{r}} :=_{\text{def}} \textit{getRetObj}(\texttt{ret}^{\texttt{transformBy}}, [\,])$. We find that $\texttt{c}^{\texttt{r}} \in \texttt{pts}_{\texttt{transformBy}}(\texttt{ret}^{\texttt{transformBy}}, [\,])$. Combining these results yields $\texttt{pts}_{\texttt{transformBy}}(\texttt{ret}^{\texttt{transformBy}}, [\,]) = \{\texttt{o}^{\texttt{p}}, \texttt{c}^{\texttt{r}}\}$, which are the two points-to relations highlighted by the two red arrows in Figure 4. Let us now apply `genSumCode()` to summarize `transformBy()`. Suppose `freshLocalVar()`, when called, will return new local variables, `tmp1`, `tmp2`, ... For this example, according to Algorithm 2, we only need to consider the two cases related to the return values of this method and its call statement $\texttt{ret}^{\texttt{transformBy}} = \texttt{t.transform(o)}$ (since `t` points to $\texttt{t}^{\texttt{p}}$). Let us consider the method returns first by applying lines 2-4 in Algorithm 2. We know already that $\texttt{ret}^{\texttt{transformBy}}$ has two pointed-to objects. Let us consider $\texttt{o}^{\texttt{p}}$ first. According to line 3 of Algorithm 2 and line 4 of Algorithm 3, we will get $v = \texttt{emitStmts}(\texttt{transformBy}, \texttt{o}^{\texttt{p}}) = \textit{getParaObj}^{-1}(\texttt{o}^{\texttt{p}}) =$ "o". Afterwards, due to line 4 of Algorithm 2, we will generate our first return statement for $\texttt{o}^{\texttt{p}}$: $\texttt{transformBy}^* = \{$"`return o;`"$\}$. Let us now consider $\texttt{c}^{\texttt{r}}$ in $\texttt{pts}_{\texttt{transformBy}}(\texttt{ret}^{\texttt{transformBy}}, [\,])$. Due to line 3 of Algorithm 2 and lines 32-33 of Algorithm 3, we generate no code but will identify $\texttt{c}^{\texttt{r}}$ by "`tmp1`": $v = \texttt{emitStmts}(\textit{transformBy}, \texttt{c}^{\texttt{r}}) = \textit{getRetObj}^{-1}(\texttt{c}^{\texttt{r}}) =$ "`tmp1`". In line 4 of Algorithm 2, we emit another return: $\texttt{transformBy}^* \ni$ "`return tmp1;`". Let us now consider $\texttt{ret}^{\texttt{transformBy}} = \texttt{t.transform(o)}$ with unknown call targets by applying lines 15-30 in Algorithm 2. As s is $\texttt{c}^{\texttt{r}}$ in line 16, processing line 17 yields $x' = \texttt{emitStmts}(\textit{transformBy}, \texttt{c}^{\texttt{r}}) = \textit{getRetObj}^{-1}(\texttt{c}^{\texttt{r}}) =$ "`tmp1`". After lines 18-19, $y' =$ "`tmp2`". After lines 20-23, $\texttt{transformBy}^* \ni$ "`tmp2 = t;`". After lines 24-29, we emit another assignment: $\texttt{transformBy}^* \ni$ "`tmp3 = o;`". In line 30, we obtain the following summary for `transformBy()`, which is identical to the one in Figure 2 (modulo the temporaries used):

$$\texttt{transformBy}^* = \left\{ \begin{array}{l} \text{“}\texttt{return o;}\text{”}, \text{“}\texttt{return tmp1;}\text{”}, \\ \text{“}\texttt{tmp2 = t;}\text{”}, \text{“}\texttt{tmp3 = o;}\text{”}, \\ \text{“}\texttt{tmp1 = tmp2.transform(tmp3);}\text{”} \end{array} \right\}$$

2) Example 2: For our second example in Figure 5, the summaries generated are the same as the ones given in the two orange boxes (modulo the temporaries introduced).

3) Example 3: Figure 8 is used to illustrate [Cast] in Figure 7 when `A.cast2A()` is summarized. In Figure 8a, `B` is a subclass of `A`. Thus, the original cast rule from *k-obj* applies, since the cast is always safe, even though the dynamic type of any object pointed by `b`, denoted $\texttt{b}^p$, is unknown (i.e., $\texttt{isTypeExact}(\texttt{b}^p) = \textsf{false}$). In Figure 8b, `B` is not a subclass of `A`, which is now an interface. This time, the original cast rule no longer holds, preventing any pointed-to object of `b` to be filtered out, since its dynamic type may be `B`'s subtype that implements `A`. In this case, the corresponding summary-based rule will come into play, as desired. However, if 'B' is final, a subclass 'C' derived from 'B' cannot exist.

```
class A {
  public static A cast2A(B b) {
    Object o = b;
    return (A) o; } }
class B extends A {}
// Application Code
class C extends B {}
B b = new C();
A a = cast2A(b);
```

(a) B is a subclass of A

```
Interface A {
  public static A cast2A(B b) {
    return (A) b;
} }
class B { }
// Application Code
class C extends B implements A {}
B b = new C();
A a = cast2A(b);
```

(b) B is not a subclass of A

Fig. 8: Applying [Cast].

4) Example 4: According to the second summary-based sub-rule in [Call] given in Figure 7, any known object passed into an unknown call target must be flagged as a summary object, since its fields may now be made to point to unknown objects. In Figure 9, `A.foo()`, which is the method to be summarized, contains a call statement `b.bar(a)` with possibly an unknown callback to the application code, where `b` points to unknown objects passed from the application code. When this call statement is analyzed, the object `A1` created at line 4 will be marked as a summary object, $\texttt{A1} \in \mathbb{S}_{arg}$, according to the second summary-based sub-rule in [Call], so that `a` is flagged to point to `A1`, which is a summary object, yielding soundly the summary, `foo*` $= \{$“`Object a = new A();`”, “`b.bar(a);`”, “`return a.f;`”$\}$ (with all the temporaries elided). If this sub-rule is ignored, `a` will not point to any summary object. As `A1.f` points to `null`, so “`return a.f`”, which is effectively “`return null`”, will appear alone in the summary (lines 2-4 of Algorithm 2), which is obviously unsound.

```
public class A {
  public Object f;
  public static Object foo(B b) {
    A a = new A(); // A1
    b.bar(a);
    return a.f; } }
public class B { void bar(A o) { } }
// Application Code
class C extends B {
  void bar(A a) {
    a.f = new Object();
} }
...
Object c = A.foo(new C());
```

Fig. 9: Applying second summary-based sub-rule in [CALL].

5) Example 5: We apply `genSumCode()` to summarize library methods across four cases (handled by corresponding **for** loops). We have shown the first case, dealing with method returns, in Example1, and the final case, addressing unknown call targets, in Examples1 and 4. Here, we focus on the third case: handling modifications to fields of unknown objects.

```
1 public class A {
2   Object f = new Object();
3   public void setF(Object o) {
4     this.f = o; } }
5 // Application Code
6 A a = new A();
7 Object o = new Object();
8 a.setF(o);
```

Fig. 10: Handling field modifcations to unknown objects.

Figure 10 illustrates a standard setter case, where `setF()` is to be summarized. In the first stage, *lib-k-obj* is performed. By applying [PARAINIT], we create two summary objects, denoted $\mathtt{this}^p$ and $\mathtt{o}^p$ (for representing unknown objects received from the application code), to make its two parameters $\mathtt{this}^{\mathtt{setF}}$ and `o` point to $\mathtt{this}^p$ and $\mathtt{o}^p$, respectively. By applying [STORE] to analyze the store at line 4, we find that $\mathtt{this}^p\mathtt{.f}$ can point to $\mathtt{o}^p$. When generating the summary for `setF()` in the second stage, we will include this modification side-effect according to the third **for** loop in Algorithm 2, since $\mathtt{this}^p$ represents an unknown object.

D. Soundness, Precision, and Time Complexity

1) Soundness: We rely on the following standard definition.

Definition III.1. *A pointer analysis is* sound *if it over-approximates the points-to information for every program.*

Informally, *lib-k-obj* specified in Figure 7 is sound for a library $\mathcal{L}$ if it over-approximates the points-to information in $\mathcal{L}$ regardless of the application code used. For the points-to information computed by *lib-k-obj* for $\mathcal{L}$, it is understood that if a variable in $\mathcal{L}$ points to an unknown object with an inferred upper-bound type T, then it may point to all possible objects of any subtype of T. This is stated formally in Lemma III.2.

Recall that $\mathtt{pts}_{mtd}$ represents the points-to relation computed for a library method $mtd \in \mathcal{L}$ by *lib-k-obj*. Let `pts` be the points-to relation computed for a program P, together with $\mathcal{L}$, by *k-obj* as a whole-program analysis (according to Figure 7). Let $\mathtt{pts}_{P,mtd}$ be `pts` restricted to the library code that is reachable from *mtd*. Let $\overline{\mathtt{pts}_{mtd}}$ and $\overline{\mathtt{pts}_{P,mtd}}$ be their context-insensitive versions with all contexts dropped. We write $\overline{\mathtt{pts}_{mtd}} \Supset \overline{\mathtt{pts}_{P,mtd}}$ to mean that $\overline{\mathtt{pts}_{mtd}}(x) \supseteq \overline{\mathtt{pts}_{P,mtd}}(x)$ holds for every variable or object field x reachable from *mtd* during the whole-program analysis.

Lemma III.2. *Let $\mathcal{P}$ be the universe of all programs that share a common library $\mathcal{L}$ summarized by lib-k-obj. Then lib-k-obj is sound if $\forall\ mtd \in \mathcal{L} : \forall\ P \in \mathcal{P} : \overline{\mathtt{pts}_{mtd}} \Supset \overline{\mathtt{pts}_{P,mtd}}$.*

Proof. Follows directly from Definition III.1. □

Lemma III.3. *For any given library $\mathcal{L}$, lib-k-obj is sound.*

Proof Sketch. According to Figure 7, *lib-k-obj* has two sets of inference rules. It uses the same set of rules from *k-obj* to handle (1) the known objects created in $\mathcal{L}$ identically as the whole-program pointer analysis counterpart *k-obj*, and (2) the unknown objects (created in either $\mathcal{L}$ or the application code) when their upper type bounds are castable ([CAST]) or exact ([CALL]). *lib-k-obj* uses another set of summary-based rules to model pointed-to unknown objects over-approximately to ensure that $\forall\ mtd \in \mathcal{L} : \forall\ P \in \mathcal{P} : \overline{\mathtt{pts}_{mtd}} \Supset \overline{\mathtt{pts}_{P,mtd}}$, where $\mathcal{P}$ is the universe of all programs that share $\mathcal{L}$. As *k-obj* is sound, *lib-k-obj* is sound by Lemma III.2. □

Lemma III.4. *Given the points-to information $\mathtt{pts}_{mtd}$ computed by lib-k-obj for a library method $mtd \in \mathcal{L}$, the summary mtd* generated by `genSumCode()` has the same points-to side effect visible to the application code according to $\mathtt{pts}_{mtd}$.*

Proof Sketch. When generating *mtd** for *mtd* according to Algorithms 2 and 3, we have considered all four possible kinds of modification side-effects visible to the application code recorded in $\mathtt{pts}_{mtd}$: (1) method returns (lines 2-4), (2) modifications to global variables (lines 5-8), (3) modifications to unknown objects (lines 9-14), and (4) modifications made in unknown target methods invoked on unknown receiver objects at a call statement (by preserving the call statement and generating the code needed for handling its arguments and method returns (lines 15-30). As `genSumCode()` generates *mtd** according to $\mathtt{pts}_{mtd}$, *mtd** is guaranteed to have the same points-to side-effect that is visible to (i.e., directly accessible by) the application code as prescribed by $\mathtt{pts}_{mtd}$. □

Theorem III.5. *s-k-obj is sound for the application code.*

Proof. Lemmas III.3 and III.4. □

2) Precision: For a program using a library $\mathcal{L}$, *s-k-obj* is sound for its application code (Theorem III.5) but does not guarantee the same precision in the application code as its whole-program pointer analysis counterpart *k-obj*. In practice, however, *s-k-obj* is usually more precise than *k-obj* due to a deeper heap abstraction provided in k-$\mathcal{L}^*$, which is made possible by method inlining performed during library summarization. However, *s-k-obj* may lose precision in two special cases due to null pointers and (also) method inlining.

Precision Loss. Let us look at the two scenarios for *s-k-obj*.

- **Null Pointers.** Consider Figure 11. If we apply *k-obj* to perform a whole-program analysis, line 9 will be ignored, since `c.copy()` at line 3 is ignored by *k-obj* due to `c = null`. However, if we summarize `copyCtor()` for *k-obj* and then apply *s-k-obj* to perform the subsequent summary-based analysis, *s-k-obj* will lose precision. When analyzing `copyCtor()`, *lib-k-obj* assumes that its parameter `c` points to a non-null unknown object, denoted $\texttt{c}^p$, of type `Ctor` ([PARAINIT]). Since `Ctor` is final, $\texttt{isTypeExact}(\texttt{c}^p)$ = true. By [CALL], `c.copy()` at line 3 is resolved to `copy()` in class `Ctor`. When `genSumCode()` is invoked, this call is inlined, yielding $\texttt{copyCtor}^*$ = {"`Ctor tmp1 = newCtor();` `// C`'', "`return tmp1;`''}. Finally, when *s-k-obj* is applied, `c` at line 9 will be made to point to `C` spuriously.

```
1 class ReflectAccess {
2   Ctor copyCtor(Ctor c) {
3     return c.copy(); } }
4 final class Ctor {
5   Ctor copy() {
6     return new Ctor(); } } // C
7 // Application Code
8 ReflectAccess r = new ReflectAccess();
9 Ctor c = r.copyCtor(null);
```

Fig. 11: Precision loss of *s-k-obj* due to null pointers.

- **Method inlining.** Consider Figure 12. Object-sensitively [8], the contexts for calling `id()` at lines 19 and 20 are [`E1`,`T1`] and [`E1`,`T2`], respectively. If we apply *2-obj* to perform a whole-program pointer analysis, we will prove that `v1` and `v2` are not aliases since `v1` and `v2` point to `O1` and `O2`, respectively. However, if we summarize this library class for *2-obj*, $\texttt{entrySet}^*$ will stay the same as `entrySet()` but $\texttt{iterator}^*$ will become "`iterator() { return new Enum(); // E1 }`", where the call at line 11 has been inlined. For the new program obtained with the library replaced by this summary, the contexts required for distinguishing the two calls to `id()` at lines 19 and 20 are now longer: [`E1`, `E2`, `T1`] and [`E1`, `E2`, `T2`], respectively. If we apply *s-2-obj* to analyze this new program, we will no longer be able to distinguish these two calls as their contexts under 2-limiting are identical: [`E1`, `E2`], causing us to conclude that `v1` and `v2` are aliases since `v1` and `v2` will both point to `O1` and `O2`. As a result, *s-2-obj* is not as precise as *2-obj*.

```
1 class Table {
2   private Enum getIterator() {
3     return new Enum();} // E1
4   Entry entrySet() {
5     return new Entry();} // E2
6 class Enum {
7   Object id(Object o) {
8     return o;}}
9 class Entry {
10  Enum iterator() {
11    return Table.this.getIterator();}}}
12 // Application Code
13 Table t1 = new Table(); // T1
14 Table t2 = new Table(); // T2
15 Object o1 = new Object(); // O1
16 Object o2 = new Object(); // O2
17 Entry e1 = t1.entrySet();
18 Entry e2 = t2.entrySet();
19 Object v1 = e1.iterator().id(o1);
20 Object v2 = e2.iterator().id(o2);
```

Fig. 12: Precision loss of *s-k-obj* due to method inlining.

Precision Gain. In general, *s-k-obj* can achieve better precision than its whole-program pointer analysis counterpart *k-obj* in the application code since method inlining performed by library summarization in $\mathcal{L}$ enables k-$\mathcal{L}^*$ to provide a deeper heap abstraction than $\mathcal{L}$. Consider Figure 13. The heap contexts for modeling the two distinct objects `B` created due to the two calls at lines 14 and 15 are [`A1`] and [`A2`], respectively. Under *1-obj*, which applies 0-limited heap, these two heap objects are conflated. As a result, `v1` and `v2` are considered to be aliases as both may point to `O1` and `O2`. However, if we summerize the library for *1-obj*, we will obtain $\texttt{foo}^*$ = {"`return new B(o); // B1`"} and $\texttt{bar}^*$ = {"`return new B(o); // B2`"}, with `baz()` inlined in its two callers. By inlining the object allocation site at line 7 in its two callers, we will end up inserting one copy in `foo()` and another copy in `bar()`. As a result, the heap contexts for modeling `B1` and `B2` are now [ ]. For the new program, *s-1-obj* (or even *s-0-obj*) can now prove that `v1` and `v2` are not aliases. Effectively, library summarization provides a more precise heap abstraction in k-$\mathcal{L}^*$ than in $\mathcal{L}$, enabling *s-1-obj* to achieve better precision in the application code than *1-obj* at the expense of having to handle slightly more objects.

```
1 class A {
2   static C foo(o) {
3     return new A().baz(o); } // A1
4   static C bar(o) {
5     return new A().baz(o); } // A2
6   B baz(Object o) {
7     return new B(o); }} // B
8 class B {
9   Object f;
10  B(Object o) { this.f = o; }}
11 // Application Code
12 Object o1 = new Object(); // O1
13 Object o2 = new Object(); // O2
14 B b1 = A.foo(o1);
15 B b2 = A.bar(o2);
16 Object v1 = b1.f;
17 Object v2 = b2.f;
```

Fig. 13: Precision gain of *s-k-obj* over *k-obj*.

In general, summarizing a library can cause its extensive method inlining, enabling *s-k-obj* to gain rather than lose precision in the application code. In addition, precision loss is negligible due to null pointers, which are rare. Thus, library summarization enables pointer analysis to run faster while achieving better precision for the application code.

3) Time Complexity: *k-obj* has a worst-case time complexity (N^3), where N is the program size [11], [14]. Our library summarization approach (Algorithm 1) has the same worst-case time complexity. In addition to the standard inference rules from *k-obj*, *lib-k-obj* (Figure 7) relies also on five summary-based rules, where [PARAINIT] is applied only once in $O(1)$ and every other rule has the same time complexity as its corresponding rule from *k-obj*. The number of summary objects created by *lib-k-obj* is linear to the number of variables in the program: $|\mathbb{S}| = \mathcal{O}(|\mathbb{V}|)$. The time-complexity of `genSumCode()` (Algorithms 2 and 3) is linear to the size of the points-to graph generated by *lib-k-obj*: $|\mathbb{H}| * (|\mathbb{H}| + |\mathbb{V}| + |\mathbb{G}|) = O(N^2)$, since $|\mathbb{H}| = |\mathbb{O}| + |\mathbb{S}|$. Thus, the overall time complexity of our approach is $\mathcal{O}(N^3)$.

IV. EVALUATION

We show that *s-k-obj* runs faster than *k-obj* while achieving better overall precision in the application code, where $1 \leqslant k \leqslant 2$. Note that *k-obj* is unscalable when $k \geqslant 3$ for large programs [3]–[7]. We address two research questions:

- RQ1: Can our approach generate k-object-sensitive library summaries efficiently with a low time overhead?
- RQ2: Can *s-k-obj* run more efficiently than *k-obj* while also achieving better overall precision (measured by several standard precision metrics) for the application code?

We have implemented our approach (open-sourced soon) in SOOT [10], where, as shown in Figure 7, *k-obj* is naturally a special case of *lib-k-obj*. The DaCapo benchmark suite [15] is commonly used in the pointer analysis literature [4], [6], [16], [17]. We have considered all 14 Java programs from its most recent edition (2018-04-06) except `jython` since its context-sensitive analyses do not scale [17]. For each program, its associated library is the aggregate of its own third-party libraries and the Java standard library (JDK1.8.0_312).

We have done our experiments on an Intel(R) Xeon E5-1660 3.2GHz machine with 256GB of RAM. For each program, the analysis time of an algorithm is the average of three runs.

A. RQ1: Overhead

Given a library $\mathcal{L}$, we produce different k-object-sensitive library summaries k-$\mathcal{L}^*$ customized for k-*obj* with different values of k. These can be obtained in parallel, since different library methods can be summarized independently. In addition, a real-world application can often take months or even years for many of its libraries to receive an update. Therefore, k-$\mathcal{L}^*$ can be reused by different applications or even the same application where k-$\mathcal{L}^*$ is applied. In this case, the time spent for obtaining k-$\mathcal{L}^*$ can be amortized across such scenarios.

TABLE II: Number of library methods summarized and ignored in the reachable methods found in the libraries used by the 13 Java programs, together with the summarization times.

Program	#Reachable		#Ignored		#Summarized		Time (secs)	
	1-obj	*2-obj*	*1-obj*	*2-obj*	*1-obj*	*2-obj*	*1-obj*	*2-obj*
avrora	7747	7622	3832	3791	3915	3831	68.71	71.66
batik	19787	16878	6819	6535	12968	10343	218.80	213.04
eclipse	10908	10685	5285	5227	5623	5458	307.46	311.94
fop	32021	31608	11549	11416	20472	20192	484.50	480.29
h2	19585	19411	7737	7689	11848	11720	157.84	177.58
luindex	14645	14485	7194	7145	7451	7340	90.95	95.58
lusearch	7748	7620	3820	3777	3928	3843	48.42	51.20
pmd	17607	17418	6141	6096	11466	11322	223.35	243.27
sunflow	13904	13754	6452	6392	7452	7362	116.37	119.70
tomcat	10585	10458	5160	5124	5425	5334	323.22	320.38
tradebeans	11818	11642	5412	5376	6406	6266	1112.90	1101.67
tradesoap	11818	11642	5412	5376	6406	6266	1106.17	1112.69
xalan	10554	10420	4040	3997	6514	6423	107.21	110.84

As a proof of concept, we compute the summaries for the methods in a library by using a simple sequential implementation of our library summarization approach. For a library method m summarized, we define *reachables*(m) to be the number of reachable methods found by *lib-k-obj*. The methods in a library tend to fall into two categories: those with a few reachable library methods and those appearing in a few strongly connected cycles (SCCs) consisting of a large number of library methods. In general, summarizing the library methods in a large SCC is not beneficial, since their summaries are more or less the same, resembling the original statements forming the SCC. Currently, we terminate the summarization process for a library method m when *reachables*$(m) > K$, where $K = 200$ empirically, and will use m directly instead.

Table II gives the number of library methods summarized for the libraries used by the 13 Java programs under *k-obj*, together with the summarization times. For *1-obj* and *2-obj*, the percentages of reachable library methods that are summarized are 56.4% and 55.9%, costing 4365.9 seconds and 4409.84 seconds, respectively, across the 13 programs.

B. RQ2: Efficiency and Precision

As revealed in Table III, *s-k-obj* runs more efficiently than *k-obj* while achieving better precision in the application code.

TABLE III: Efficiency and precision of k-*obj* and *s*-k-*obj* for the 13 Java programs (for the application code).

Program	Metrics	*1-obj*	*s-1-obj*	*2-obj*	*s-2-obj*
avrora	Time (s)	30.31	18.16	762.47	274.26
	#Reachable Meths	3495	3495	3459	3459
	#May Fail Casts	274	274	209	209
	#Poly Calls	96	96	87	87
batik	Time (s)	607.12	222.06	7019.81	2000.91
	#Reachable Meths	8268	8175	8240	8147
	#May Fail Casts	1545	1418	1254	1180
	#Poly Calls	3305	3005	3041	2960
eclipse	Time (s)	205.44	218.31	1219.90	882.12
	#Reachable Meths	3870	3818	3814	3764
	#May Fail Casts	1467	1450	1228	1209
	#Poly Calls	1671	1629	1529	1488
fop	Time (s)	7327.26	4637.04	22382.84	13815.25
	#Reachable Meths	15985	15940	15948	15909
	#May Fail Casts	3096	3086	2296	2284
	#Poly Calls	6738	6718	4519	4492
h2	Time (s)	122.42	38.04	9046.40	5429.35
	#Reachable Meths	407	407	407	406
	#May Fail Casts	34	34	28	28
	#Poly Calls	43	6	41	4
luindex	Time (s)	75.55	47.83	1019.69	377.70
	#Reachable Meths	365	365	365	365
	#May Fail Casts	30	30	24	24
	#Poly Calls	5	5	3	3
lusearch	Time (s)	20.59	11.10	656.41	262.14
	#Reachable Meths	364	364	363	363
	#May Fail Casts	31	31	25	25
	#Poly Calls	5	5	3	3
pmd	Time (s)	405.87	128.82	58193.44	31358.33
	#Reachable Meths	4308	4239	4304	4235
	#May Fail Casts	1594	1551	1436	1397
	#Poly Calls	1093	1038	954	907
sunflow	Time (s)	62.95	37.74	1547.01	887.25
	#Reachable Meths	971	971	970	970
	#May Fail Casts	90	90	54	54
	#Poly Calls	55	55	53	53
tomcat	Time (s)	34.01	18.31	854.12	278.45
	#Reachable Meths	463	463	462	462
	#May Fail Casts	40	40	34	34
	#Poly Calls	21	21	19	19
tradebeans	Time (s)	56.07	24.06	1660.67	862.95
	#Reachable Meths	394	394	394	394
	#May Fail Casts	35	35	28	28
	#Poly Calls	6	6	4	4
tradesoap	Time (s)	54.72	24.12	1684.1	879.72
	#Reachable Meths	394	394	394	394
	#May Fail Casts	35	35	28	28
	#Poly Calls	6	6	4	4
xalan	Time (s)	46.58	17.71	1064.74	358.78
	#Reachable Meths	1696	1657	1653	1648
	#May Fail Casts	179	176	107	105
	#Poly Calls	225	210	218	205

1) Precision: We measure the precision of a pointer analysis using three commonly used metrics, #Reachable Methods (number of reachable methods), #May-Fail Casts (number of casts that may fail), and #Poly Calls (number of polymorphic callsites), computed for the application code.

For each program, *s-k-obj* achieves the same precision as or higher precision than *k-obj*. By producing k-object-sensitive library summaries, *s-k-obj* can also achieve noticeable precision improvements over *k-obj* (as a nice side-effect of method inlining (Figure 13)). For #Reachable Methods, we see it reduced for `xalan` by 2.3% under *s-1-obj*. For #May-Fail Casts, *s-1-obj* reduces it by 8.2% for `batik`, and *s-2-obj* reduces it by 5.9% for `batik`, 2.7% for `pmd` and 1.8% for `xalan`. For #Poly Calls, library summarization is also beneficial. For `batik`, *s-1-obj* reduces it by 9.1%. For `eclipse`, `pmd`, and `xalan`, *s-1-obj*'s reduction rates are 2.5%, 5.0%, and 6.7%, respectively. For `h2`, we see a reduction of 86.0% under *s-1-obj* and of 90.2% under *s-2-obj*.

2) Efficiency: For a program, we obtain the analysis time of a pointer analysis under a 24-hour time budget. In Figure 14, we give the speedups of *s-k-obj* over *k-obj*. We observe that *s-1-obj* achieves an average speedup of 2.1x over *1-obj* with the largest at 3.2x for `h2`, and *s-2-obj* achieves an average speedup of 2.3x over *2-obj* with the largest at 3.5x for `batik`.

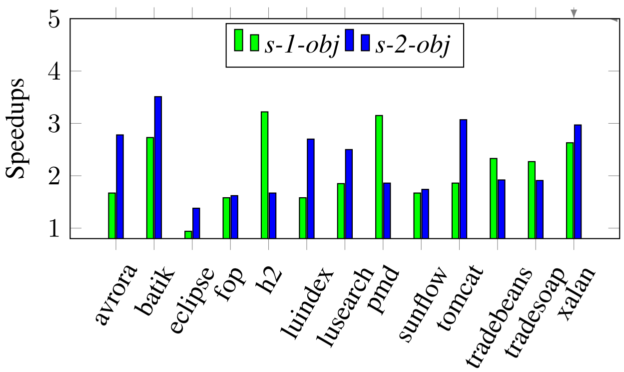

Fig. 14: The speedups of *s-k-obj* over *k-obj*, where $1 \leqslant k \leqslant 2$.

In general, *s-k-obj* is faster than *k-obj* (with one exception for `eclipse` when $k = 1$) as it avoids re-analyzing the same library methods. In addition, summarizing a library method in *s-k-obj* achieves better precision than *k-obj* in the application code due to deeper heap abstraction via method inlining, albeit with slightly more propagated objects due to object allocation site replication, as shown in Figure 13. For object-sensitivity [8], a more precise heap abstraction leads to simultaneously a more precise calling-context abstraction, resulting in slightly more calling contexts but fewer points-to facts being propagated redundantly and/or spuriously by *s-k-obj*. For `eclipse`, *s-1-obj* (218.31 secs) is 6.3% slower than *1-obj* (205.44 secs) since *s-1-obj* has to handle 6.5% more object allocation sites, costing more analysis time in handling their propagation during its analysis. In this case, the performance benefit provided by library summarization is more than offset by the analysis cost thus incurred. In return, however, *s-1-obj* becomes more precise than *1-obj* by achieving a reduction of 1.3%, 1.2% and 2.5% for #Reachable Methods, #May Fail Casts, and #Poly Calls, respectively.

V. Related Work

Most library summarization techniques [18]–[23] focus on generating "internally digested" summaries for some particular static analyses at hand. In [18], the side effect of a method is recorded by using a normalized abstract heap and then instantiated at a callsite, achieving a bottom-up pointer analysis for Java. In [19], [20], a library is summarized by reasoning about graph reachability to support program analyses formulated in terms of Context-Free Language (CFL) reachability. Other techniques make use of different analysis-specific summary functions, including the points-to and modification effects for supporting the classic MOD-REF analysis [23], the transfer functions on reduced ICFG [22], the statement-level transformers [21]. In contrast, we focus on generating precise library summaries to support pointer analysis.

Averroes [1], [24] supports an application-focused analysis by building an application-only call graph, without actually analyzing the library. It obtains a library summary per application by merging over-approximately the information from all the library pointers into a single set, without catering to the varying precision needs of different downstream analyses. This can lead to a significant precision loss, as recently validated [2]. In contrast, our approach customizes library summaries to cater to the varying precision needs of *k-obj* under varying k-limits. This way, *s-k-obj* can run faster while achieving nearly the same or better precision than *k-obj* in the application code.

The "apponly" mode in Spark [25] takes an extreme stance by disregarding library dependencies and concentrating exclusively on analyzing the application code. This sharply limits program coverage. In contrast, our approach aims to equal whole-program analysis results in precision and soundness.

Soot [10] also supports a "library mode" for its callgraph construction phase. This option is intended for the standalone analysis of libraries. Besides it is not for generating a summary that can then be plugged into the analysis of an application that uses the library, it is also very different with our lib-based pointer analysis. It primarily enhances the completeness of the call graph by supplementing new objects with any-sub-type of the declared type at some key locations. However, these added objects lack any information except for the type. In contrast, Our summary object covers the entire lifecycle of an abstract object from generation to usage, and its traceability enables precise reproduction of program behavior.

Bottom-up and top-down pointer analyses [26]–[28] use a two-phase analysis with value-flows moving in opposite directions to achieve context-sensitivity context-insensitively. Although method summaries are used in these approaches, they are closely tied to the particular framework used. These techniques tend to be less precise than inclusion-based pointer analysis counterparts [25] because the methods appearing in a strongly connected component (SCC) are usually merged and analyzed in a context-insensitive manner [25].

There are other efforts on generating method summaries. In [29], the behavior of binary code is summarized in the absence of source code. In [30], some redundant statements in methods are eliminated intra-procedurally by utilizing code patterns related to context-sensitive pointer analysis.

VI. Conclusion

We introduce a novel approach for generating precise library summaries to accelerate object-sensitive pointer analysis while achieving the same or better precision in the application code. Due to its general nature, our approach is expected to find applications in other flavours of pointer analyses, including many downstream analyses that rely on aliasing information.

Acknowledgment

This research is supported by an ARC grant DP210102409.

References

[1] K. Ali and O. Lhoták, "Averroes: Whole-program analysis without the whole program," in *European Conference on Object-Oriented Programming*. Springer, 2013, pp. 378–400.

[2] A. Utture and J. Palsberg, "Fast and precise application code analysis using a partial library," in *44th International Conference on Software Engineering*, 2022.

[3] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: understanding object-sensitivity," in *ACM SIGPLAN Notices*, vol. 46. ACM, 2011, pp. 17–30.

[4] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "Precision-guided context sensitivity for pointer analysis," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 141, 2018.

[5] J. Lu and J. Xue, "Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity," *Proc. ACM Program. Lang.*, vol. 3, Oct. 2019.

[6] D. He, J. Lu, Y. Gao, and J. Xue, "Accelerating object-sensitive pointer analysis by exploiting object containment and reachability," in *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[7] M. Jeon, S. Jeong, and H. Oh, "Precise and scalable points-to analysis via data-driven context tunneling," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 140, 2018.

[8] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 14, no. 1, pp. 1–41, 2005.

[9] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, p. 598–608.

[10] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*. IBM Corp., 2010, pp. 214–224.

[11] 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*. Springer, 2013, pp. 196–232.

[12] T. Tan, Y. Li, and J. Xue, "Making k-object-sensitive pointer analysis more precise with still k-limiting," in *International Static Analysis Symposium*. Springer, 2016, pp. 489–510.

[13] D. He, J. Lu, and J. Xue, "Context debloating for object-sensitive pointer analysis," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 79–91.

[14] D. J. Pearce, "Some directed graph algorithms and their application to pointer analysis," Ph.D. dissertation, University of London, 2005.

[15] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer *et al.*, "The dacapo benchmarks: Java benchmarking development and analysis," in *ACM Sigplan Notices*, vol. 41. ACM, 2006, pp. 169–190.

[16] 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, jan 2022. [Online]. Available: https://doi.org/10.1145/3498720

[17] R. Thiessen and O. Lhoták, "Context transformations for pointer analysis," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 263–277, 2017.

[18] Y. Feng, X. Wang, I. Dillig, and T. Dillig, "Bottom-up context-sensitive pointer analysis for java," in *Asian Symposium on Programming Languages and Systems*. Springer, 2015, pp. 465–484.

[19] H. Tang, D. Wang, Y. Xiong, L. Zhang, X. Wang, and L. Zhang, "Conditional dyck-cfl reachability analysis for complete and efficient library summarization," in *European Symposium on Programming*. Springer, 2017, pp. 880–908.

[20] H. Tang, X. Wang, L. Zhang, B. Xie, L. Zhang, and H. Mei, "Summary-based context-sensitive data-dependence analysis in presence of callbacks," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015, pp. 83–95.

[21] G. Yorsh, E. Yahav, and S. Chandra, "Generating precise and concise procedure summaries," in *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2008, pp. 221–234.

[22] A. Rountev, S. Kagan, and T. Marlowe, "Interprocedural dataflow analysis in the presence of large libraries," in *International Conference on Compiler Construction*. Springer, 2006, pp. 2–16.

[23] A. Rountev and B. G. Ryder, "Points-to and side-effect analyses for programs built with precompiled libraries," in *International Conference on Compiler Construction*. Springer, 2001, pp. 20–36.

[24] K. Ali and O. Lhoták, "Application-only call graph construction," in *European Conference on Object-Oriented Programming*. Springer, 2012, pp. 688–712.

[25] O. Lhoták and L. Hendren, "Scaling java points-to analysis using s park," in *International Conference on Compiler Construction*. Springer, 2003, pp. 153–169.

[26] E. M. Nystrom, H.-S. Kim, and W.-M. W. Hwu, "Bottom-up and top-down context-sensitive summary-based pointer analysis," in *International Static Analysis Symposium*. Springer, 2004, pp. 165–180.

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

[28] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 278–289, 2007.

[29] D. Gopan and T. Reps, "Low-level library analysis and summarization," in *International Conference on Computer Aided Verification*. Springer, 2007, pp. 68–81.

[30] Y. Smaragdakis, G. Balatsouras, and G. Kastrinis, "Set-based pre-processing for points-to analysis," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 253–270.