Cut-Shortcut Pointer Analysis: Re-imagining Context-Sensitivity without Contexts

SHENGYUAN YANG, University of Wisconsin-Madison, USA
WENJIE MA, University of California, Berkeley, USA
THOMAS REPS, University of Wisconsin-Madison, USA
TIAN TAN, Nanjing University, China
YUE LI, Nanjing University, China

Over the past decades, context sensitivity has been considered as one of the most effective ideas for improving the precision of pointer analysis for object-oriented programming languages such as Java. By analyzing each method under distinct contexts, context sensitivity separates the static representations of different dynamic instantiations of variables and heap objects, thereby reducing spurious object flows caused by method calls. However, despite great precision benefits, context sensitivity brings heavy efficiency costs because each method is essentially cloned and analyzed under different contexts. To mitigate this issue, numerous selective context-sensitive approaches have been proposed, applying context sensitivity only to selected methods while analyzing others context-insensitively. However, the selective approaches do not fully remove the efficiency bottleneck because they still rely on the fundamental idea of context sensitivity—analyzing multiple method copies based on calling contexts—which limits their ability to scale to large programs.

In this work, we present a fundamentally different approach, called Cut-Shortcut, for fast and precise pointer analysis for Java. The core insight is that the primary effect of cloning methods under different contexts is to filter spurious object flows merged within callee methods; from the vantage point of a typical pointer flow graph (PFG), such effects can be simulated by suppressing the introduction/addition of imprecise flow edges (*Cut*) and adding *Shortcut* edges that directly connect source pointers to the target ones, bypassing method boundaries. We articulate this insight into a general principle that achieves the precision benefits of context sensitivity without explicitly using contexts. This principle is instantiated by identifying and handling three well-characterized program patterns to safely suppress imprecise flows and add precise shortcut flows in the PFG. Our approach is formalized through inference rules, and we formally prove its soundness. We further present a CFL-reachability formulation of Cut-Shortcut, both to relate our technique to well-understood formal models of pointer analysis, and to show that Cut-Shortcut has the same worst-case asymptotic complexity as context-insensitive analysis. To extend our approach to modern Java programs that introduce new sources of imprecision, we propose Cut-Shortcut^S, an extension of Cut-Shortcut that lifts one of its patterns to handle precision loss caused by *Streams* and their interaction with *Lambda Expressions*. This extended pattern distinguishes object flows across different stream pipelines, thereby preserving precision without requiring context sensitivity. Our approaches are implemented on the state-of-the-art pointer analysis framework Tai-e. We evaluate Cut-Shortcut on 10 large, complex Java programs used in recent literature, and evaluate Cut-Shortcut^S on three new benchmark suites that we created, containing in total 20 programs with heavy stream usage. The evaluation results are compelling: Cut-Shortcut not only achieves precision

Some of the material in the paper has previously appeared in https://dl.acm.org/doi/10.1145/3591242.
Authors' Contact Information: Shengyuan Yang, University of Wisconsin-Madison, USA, syang686@wisc.edu; Wenjie Ma, University of California, Berkeley, USA, windsey@berkeley.edu; Thomas Reps, University of Wisconsin-Madison, USA, reps@cs.wisc.edu; Tian Tan, Nanjing University, China, tiantan@nju.edu.cn; Yue Li, Nanjing University, China, yueli@nju.edu.cn.

ACM 1558-4593/2025/11-ART11
https://doi.org/0000.0000

comparable to (selective) context-sensitive analysis (where applicable), but also consistently matches or outperforms context-insensitive analysis in efficiency across all benchmarks—running faster in 7 out of 10 cases and equally fast in the remaining 3. Meanwhile, Cut-Shortcut^S generates substantial precision improvement, while preserving scalability and efficiency for programs with heavy stream usage. To the best of our knowledge, this work is the first that has been able to achieve such a good efficiency-precision trade-off for such hard-to-analyze Java programs, including large-scale and feature-rich applications.

CCS Concepts: • **Theory of computation → Program analysis**.

Additional Key Words and Phrases: Static Analysis, Pointer Analysis, Alias Analysis, Context Sensitivity, Java

1 Introduction

Pointer analysis is a family of static-analysis techniques aimed at computing a set of abstract values that characterize what a pointer-valued variable in a program might point to during the execution of the program. The result of pointer analysis lays the groundwork for a wide range of static-analysis applications, such as bug detection [8, 9, 52], security analysis [4, 19], program optimization [71, 88], verification [16, 57], and understanding [43, 73]. Given its importance, over the past decades researchers have invested a substantial amount of effort in trying to build fast and precise algorithms for pointer analysis [67].

Andersen-style pointer analysis [2] has been considered as the standard context-insensitive pointer analysis for Java [67, 72]. We can see it as an iterative algorithm applied to an on-the-fly constructed pointer flow graph (PFG) [39, 41, 83]—where nodes represent program pointers (including variables and instance fields) and edges represent subset constraints between pointers' points-to sets. During the analysis, abstracted objects are propagated along the edges of the PFG. Because the context-insensitive approach of Andersen's algorithm does not distinguish between different call sites of the same method, incoming points-to sets are merged in callees; thus, it is extremely fast but with low precision.

To obtain higher precision, one can use a context-sensitive approach to pointer analysis. Basically, for each method call (say c) to a callee (say m), all elements in m (like variables, instance fields, and heap objects) will be cloned and analyzed under a context (say e) related to certain program elements that distinguish c from other calls. Accordingly, different forms of context sensitivity have been investigated, such as call-site sensitivity (if e is a sequence of call sites) [65, 66], object sensitivity (if e is a sequence of allocation sites) [49, 50], and type sensitivity (if e is a sequence of types) [69]. Because the object flows merged in the callees are separated under different contexts, the spurious objects flows introduced by method calls can be filtered out, thereby increasing the precision. However, context sensitivity comes with heavy efficiency costs because the analysis has to compute and maintain a huge number of context-sensitive intermediate results. Consequently, context-sensitive algorithms—including 2obj,[1] the most widely used algorithm—do not terminate in hours for several complex programs in the standard DaCapo benchmarks [32, 36, 40].

To be able to handle large and complex programs, in recent years numerous *selective* context-sensitive approaches have been proposed [20, 22, 23, 26, 30, 32, 39–41, 46, 47, 55, 56, 70, 84]. Generally, they rely on a pre-analysis to select a set of methods that are critical to improving precision but do not jeopardize scalability when analyzed using a context-sensitive approach. Then they only apply context sensitivity to those selected methods while analyzing the remaining ones context-insensitively. Although selective context sensitivity significantly improves the scalability of pointer analysis, its efficiency has not yet been fully unlocked because the core methodology of context sensitivity remains unchanged: an algorithm still must replicate methods and analyze their

[1] 2obj stands for "object sensitivity with a context length of two."

elements separately under different contexts. When many such replications occur, the analysis can become prohibitively slow and lose scalability [41, 70].

In this work, we present Cut-Shortcut, a novel approach to fundamentally change the status quo where pointer analysis for Java primarily relies on context sensitivity to acquire higher precision.

Cut-Shortcut. The key observation is that, with a context-insensitive approach, imprecision occurs when object flows are merged in a method m and then the merged flow goes outside m. In contrast with replicating the method m into multiple copies to distinguish object flows from different call sites, the approach taken in Cut-Shortcut simulates the effect of context sensitivity by suppressing the introduction/addition of imprecise flow edges (*Cut*), and instead adding only more-precise flow edges from exact source pointers to the target ones, bypassing method m (*Shortcut*). This approach allows us to achieve the benefits of context sensitivity without replicating and analyzing m under different contexts. More specifically, Cut-Shortcut eliminates the need for a context-insensitive pre-analysis phase (as typically required in selective context-sensitive approaches) and performs a single-round, context-insensitive pointer analysis on an on-the-fly constructed pointer flow graph (PFG). The analysis proceeds just like standard context-insensitive pointer analysis, except that certain imprecise edges are suppressed, and precise shortcut edges are added to our PFG. In essence, the core pointer-analysis algorithm remains unchanged—it still propagates points-to information over the PFG until a fixpoint is reached. However, it now operates on a refined graph (compared with the graph constructed by a context-insensitive analysis), where imprecision has been preemptively avoided through targeted graph construction. Here we have two key questions to answer:

- *Which edges to avoid adding?* We suppress the addition of certain edges that bring merged object flows inside a method to somewhere outside (e.g., to the variables of the caller that receive values from the callee's return variables). In other words, merging object flows within a method does not inherently reduce precision—it is only when these merged flows escape the method that precision loss occurs.
- *Where to add shortcut edges?* We identify the source nodes s that precede the merging of object flows, and introduce new edges from s to the corresponding target nodes of the edges whose addition has been suppressed, ensuring both soundness and precision of the analysis.

Although the basic idea seems natural, it is challenging both to identify *which edges to avoid adding* and *where to add shortcut edges*. For the former, we must identify edges that significantly harm precision and assess whether it is possible—and beneficial—to avoid adding them. For the latter, once certain PFG edges are not added, failing to add shortcut edges that fully capture the necessary object-flow sources can compromise soundness. To ensure soundness, rather than trying to tackle all kinds of precision loss, in this work, we articulate a guiding principle, then focus on well-characterized program patterns that align with this principle and contribute substantially to precision loss. Building on this framework, we introduce Cut-Shortcut, which systematically improves precision by targeting and handling three general program patterns.

Cut-Shortcut^S. While several whole-program pointer-analysis techniques have been proposed for Java in recent years, most remain rooted in the Java 6 language core [22, 23, 28–30, 32, 37, 40, 41, 46, 47, 70, 76–78], leaving many hard-to-analyze features introduced in newer versions of Java unaddressed. In particular, the interaction between *Streams* and *Lambda Expressions* (both introduced in Java 8) often causes substantial precision loss, because object flows become conflated across distinct stream pipelines. Although recent efforts have begun to target modern Java features (e.g., work on module-aware context-sensitive pointer analysis for the Java Platform Module System [38]), existing precision-enhancing techniques—such as context sensitivity and selective

context sensitivity—fail to provide analyses that are both precise and scalable when applied to programs with heavy use of streams.

Building on the core idea of Cut-Shortcut —suppressing imprecise flow edges and introducing precise shortcut edges—we refine and extend our approach to the stream setting. Concretely, we extend one of the existing patterns in Cut-Shortcut to also handle stream pipelines, enabling accurate tracking of input-to-output relationships within each pipeline and distinguishing object flows across pipelines without relying on context sensitivity. However, because real-world stream usage is commonly intertwined with lambda expressions, the pattern is further extended to interact with an existing static analysis of Java lambda expressions [18], thereby ensuring sound and precise handling of functional-style stream operations. This integration enables precise handling of diverse behaviors of Java streams. The resulting unified approach, Cut-Shortcut^S, takes a significant step forward in realizing highly efficient and precise pointer analysis for modern Java programs.

Contributions. In summary, the work described in this paper makes the following contributions:

- Cut-Shortcut relies on the insight that one can simulate the effect of a context-sensitive technique by modifying what edges are added to the PFG during a context-insensitive Andersen-style pointer analysis. We instantiate this principle in Cut-Shortcut by exploiting three well-characterized program patterns, designing algorithms to identify and handle them on-the-fly with pointer analysis while constructing the PFG (Section 3).
- We formalize Cut-Shortcut as a set of inference rules (Section 4) and formally prove its soundness (Section 5).
- We provide a CFL-reachability formulation to express our approach as a formal-language reachability problem (Section 6). This formulation not only connects our approach to the well-established formal-language-reachability-based formulations of pointer analysis, but also offers a perspective on the efficiency advantages of Cut-Shortcut over context-sensitive techniques.
- We present Cut-Shortcut^S, an enhanced variant of Cut-Shortcut, which extends one of the patterns used in Cut-Shortcut to accommodate modern Java features, specifically the handling of *Streams* and their interaction with *Lambda Expressions* (Section 7).
- We implement Cut-Shortcut and Cut-Shortcut^S on the state-of-the-art Java pointer-analysis framework Tai-e, and evaluate their effectiveness (Section 8).
  - Cut-Shortcut exhibits an extremely promising efficiency and precision trade-off: it is even faster than context-insensitive analysis on 7 out of 10 benchmarks, and matches its efficiency on the remaining 3, while achieving precision comparable to (selective) context-sensitive analyses—when those analyses can scale.
  - Cut-Shortcut^S further improves precision for programs with heavy stream usage, as shown on our newly proposed benchmark suites. By distinguishing object flows through different stream pipelines, Cut-Shortcut^S achieves a level of precision unattainable by any existing (selective) context-sensitivity approach, while maintaining efficiency that is consistently faster than, or on par with, context insensitivity.

This article refines and substantially extends our earlier work [48], presented at the *44th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2023)*. Whereas the conference version comprised 26 pages, this article spans 73 pages, with the following major extensions:

- Sections 2–5 reflect the core content of the conference version but have been carefully rephrased and reorganized throughout to improve clarity and precision, offering a more comprehensive and updated presentation. In addition, Section 3.1 introduces a new comparison with method-summary-based pointer analysis, highlighting key differences and advantages.

Sections 4.2–4.4 provide detailed explanations of the mechanisms for each pattern, which were not fully presented in [48] due to space constraints, along with several refinements. Finally, the soundness theorem in Section 5 is now supported by a complete and rigorous proof, whereas it was only sketched in [48].

- Section 6 introduces a CFL-reachability formulation of Cut-Shortcut, illustrated with detailed examples. This formulation demonstrates that while significantly improving precision, Cut-Shortcut preserves the asymptotic complexity of context-insensitive analysis. It not only connects our work to well-established language–reachability-based formal frameworks, but also provides a principled explanation for why Cut-Shortcut achieves high efficiency compared to context-sensitivity-based techniques.
- Section 7 introduces a new extension, Cut-Shortcut$^{\mathcal{S}}$, which abstracts stream pipelines statically and tracks their input elements and output locations on the fly, enabling the suppression of imprecise flow edges and the addition of precise ones within a single round of pointer analysis. To realize this extension, we systematically classify stream operations into categories and design dedicated handling techniques for each, including an interaction with an existing analysis of Java lambda expressions [18], to mitigate precision loss caused by functional-style stream operations.
- Section 8 presents updated experimental results for Cut-Shortcut and new evaluations for Cut-Shortcut$^{\mathcal{S}}$. For Cut-Shortcut, we report (i) results based on additional precision metrics, (ii) an additional comparison with Zipper^{e}-guided 2-type sensitive analyses [41], and (iii) the results of new experiments to isolate the contribution of each of the patterns used in Cut-Shortcut. For Cut-Shortcut$^{\mathcal{S}}$, we evaluate its effectiveness on three new benchmark suites that we created, which consist of Java programs with heavy stream usage, designed to balance operation coverage, representativeness, and realism: (i) 10 synthetic micro-benchmarks systematically covering diverse stream operations and their combinations on small inputs; (ii) 5 enterprise-backend-inspired benchmarks that mimic common web-backend data-processing and transmission workflows using streams, with an in-memory mock database; and (iii) 5 real-world programs with intensive stream usage beyond the enterprise application domain.

We provide an open-source implementation of our refined and extended version of Cut-Shortcut (including Cut-Shortcut$^{\mathcal{S}}$) at [86]. All experimental results reported in this article are fully reproducible using this artifact.

2 Motivation

We use the example in Figure 1 to illustrate the basic idea of Cut-Shortcut and show how it differs from context insensitivity and context sensitivity for pointer analysis. Class `Carton` has a field `item` and provides `setItem()` and `getItem()` to modify and retrieve the `item` value, respectively. In `main()`, two `Item` objects (o_{16}, o_{21}) are stored in the field `item` of two `Carton` objects (o_{15}, o_{20}) via method `setItem()` and retrieved later via `getItem()`. (We use o_i to denote the abstract heap object allocated at line i.) After execution, `result1` (`result2`) will only point to o_{16} (o_{21}). Below we explain how context insensitivity, context sensitivity, and our approach work for this example.

Context Insensitivity. Figure 1(a) shows the PFG fragment of the example. (An edge in the PFG that connects p to q indicates that what is pointed to by p should also be pointed to by q. We use $pt(p)$ to denote the points-to set that the algorithm computes for p.) Because a context-insensitive approach does not distinguish the two calls to `setItem()`, pt(`item1`) and pt(`item2`) will be merged into pt(`item`) (where `item` is the parameter at line 3), so that the store statement at line 4 causes o_{15}`.item` and o_{20}`.item` to have imprecise points-to sets. Moreover, the two calls of `getItem()` are

```
1  class Carton {
2    Item item;
3    void setItem(Item item){
4      this.item = item;
5    }
6    Item getItem() {
7      Item r = this.item;
8      return r;
9    }
10 }
11 class Item {}
12
13 // usage code
14 void main() {
15   Carton c1 = new Carton();//o15
16   Item item1 = new Item();//o16
17   c1.setItem(item1);
18   Item result1 = c1.getItem();
19
20   Carton c2 = new Carton();//o20
21   Item item2 = new Item();//o21
22   c2.setItem(item2);
23   Item result2 = c2.getItem();
24 }
```

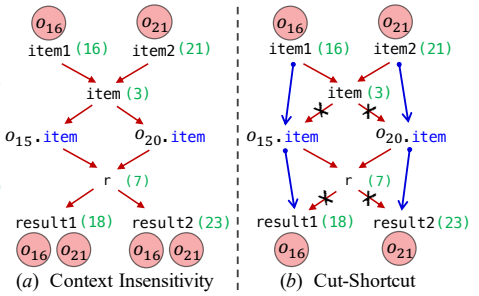


Fig. 1. An example to illustrate the basic idea of Cut-Shortcut.

handled in the same manner, so that the load statement at line 7 causes the merged objects ($\{o_{16}, o_{21}\}$) to flow to r, thereby causing the points-to sets of `result1` and `result2` to be imprecise.

Context Sensitivity. The goal of context sensitivity is to separate the flows merged at `item` (line 3) and `r` (line 7), and let o_{16} only flow to `result1` and o_{21} only flow to `result2`. To distinguish the object flows from different call sites (namely, lines 17 and 22 for `setItem()`, and lines 18 and 23 for `getItem()`), every program element in `setItem()` (i.e., `item` and `this.item`) and `getItem()` (i.e., `this.item` and `r`) should be qualified with a context and analyzed separately, which is equivalent to analyzing each of those methods twice under two different contexts. We can anticipate that the cost of computing and maintaining context-sensitive information would be very heavy in the real world, especially when the program is large or complex. Selective context sensitivity alleviates this problem, but does not fundamentally change it, because even a small set of selected methods in a complex program can threaten scalability and blow up the analysis when they are analyzed context-sensitively [41, 70].

Cut-Shortcut. Figure 1(b) depicts how Cut-Shortcut works, where the arrows with crosses indicate edges whose addition to the PFG are suppressed, and the blue arrows indicate the added shortcut edges. With a context-insensitive approach, the imprecision in $pt(o_{15}.\texttt{item})$ and $pt(o_{20}.\texttt{item})$ arises from edges $\texttt{item} \longrightarrow o_{15}.\texttt{item}$ and $\texttt{item} \longrightarrow o_{20}.\texttt{item}$ (because of the store statement at line 4), so the addition of these two edges are suppressed. Instead, we add shortcut edges from the precise source (i.e., `item1` or `item2`) directly to the target nodes. Likewise, for nodes `result1` and `result2`, the introduction of edges from `r` are suppressed, and shortcut edges $o_{15}.\texttt{item} \longrightarrow \texttt{result1}$ and $o_{20}.\texttt{item} \longrightarrow \texttt{result2}$ are added. In our new PFG, by applying the same algorithm as context insensitivity (propagating the points-to results along the PFG edges), we derive points-to results as precise as context sensitivity for this example.

3 The Cut-Shortcut Approach, Informally

We introduce the overall principle lying behind our approach (Section 3.1), on which the three program patterns, the field-access pattern(Section 3.2), the container-access pattern(Section 3.3) and the local-flow pattern(Section 3.4) are based.

3.1 Overview

Cut-Shortcut addresses the precision loss inherent in context-insensitive pointer analysis by adhering to a general principle: suppress the addition of imprecise flow edges that would carry merged object flows out, and instead add precise shortcut edges as their substitutes in the pointer flow graph (PFG). To elucidate this principle, we first introduce some key terminology.

Definition 3.1 (Local/Non-local pointers). A pointer p is *local* to a method m if p is a variable declared in m. Otherwise, p is *non-local* to m (i.e., p is declared outside m or is an instance field.)

Definition 3.2 (Conflated path, Entrance and Exit). A path p in PFG is a *conflated path* if: (1) it begins at a confluence point, where two (or more) pointers *non-local* to a method m merge into a pointer s that is *local* to m, and (2) it ends at a divergence point, where a pointer e *local* to a method n leads to two (or more) pointers *non-local* to n. We say that s is the *start node* of p, and that e is the *end node* of p. We also say that m is p's Entrance, and that n is p's Exit.

For example, the path `item` $\longrightarrow o_{15}$.`item` $\longrightarrow$ `r` in Figure 1(a) is a conflated path. Here, `item` and `r` are the start and end nodes, respectively, with `setItem()` and `getItem()` serving as its Entrance and Exit, respectively. Similarly, the single-node paths `item` (line 3) and `r` (line 7) are also conflated paths.

Definition 3.3 (Target and Source pointers). Given a conflated path p, its Target pointer (Target for short) is any successor of p's end node in PFG. A Target receives the merged object flow leaving Exit. A Source pointer (Source for short) of a given Target t is any predecessor of the start node of p whose points-to set carries (part of) the objects that flow to t (i.e., contributes to the points-set of t) during dynamic execution. A Target may have multiple Sources, and multiple Targets of the same conflated path may share one or more Sources. In context-insensitive analysis, object flows from distinct Sources of different Targets are merged inside Entrance and propagated together to all Targets.

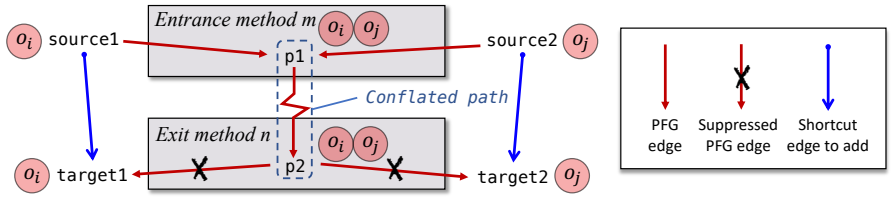


Fig. 2. Illustration of the insight that is the basis of Cut-Shortcut, about which edges to suppress and where to add shortcut edges in the PFG.

Now we explain the insight that lies behind our principle using Figure 2. `p1`, `p2`, `target1`, `target2`, `source1`, and `source2` are PFG nodes that represent program pointers (i.e., variables or instance fields), and o_i and o_j represent two abstract heap objects. Suppose that `source1` and `source2` are *non-local* to method m, and `target1` and `target2` are *non-local* to method n. If there exists a PFG path from `p1` to `p2`, this path qualifies as a conflated path by Definition 3.2. At runtime, suppose that `target1` only points to o_i, which originates from `source1`, and that `target2` only points to to o_j, which originates from `source2`. With a context-insensitive approach, however, these object flows are conflated, leading both `target1` and `target2` to point to both o_i and o_j. In many cases of real programs, precise points-to results can be easily identified by human intuition with the comprehension of semantic information that are ignored by traditional pointer analysis. Cut-Shortcut emulates such a scheme: it identifies certain conflated paths where Sources for each Target can be soundly identified (by leveraging certain semantic clues to better capture the dynamic behavior of the program) so that we can effectively improve precision by suppressing the edges connecting the end node of the conflated path to the Targets (e.g., `p2` $\longrightarrow$ `target1`/`target2`) and adding shortcut edges to connect corresponding Sources for each Target, like `source1` $\longrightarrow$ `target1` and `source2` $\longrightarrow$ `target2`. Note that Figure 2 shows a simple case with only two Targets, each

having a single Source. In real programs, a Target may have multiple Sources, all of which must be connected via shortcut edges to maintain soundness.

To enhance precision, we aim to discover as many conflated paths as possible and determine minimal Source sets that fully cover each Target. However, generalizing this process is difficult due to the semantic diversity of real-world code [39, 41] and the challenge of ensuring soundness when the addition of some edges are suppressed in the PFG. To increase precision, while maintaining soundness, Cut-Shortcut defines three concrete program patterns prevalent in practice—field-access pattern, container-access pattern, and local-flow pattern—which instantiate our general principle and account for a large portion of precision loss. The following subsections detail how we identify conflated paths, their Entrances and Exits, and the corresponding Targets and Sources for each pattern, allowing us to safely suppress the addition of imprecise edges and introduce precise shortcut edges.

Finally, we emphasize that, although the PFG built by our approach is different from the original context-insensitive PFG, Cut-Shortcut does not physically modify an existing graph during the analysis. Rather, it builds the graph on-the-fly with guidance from pattern-specific rules. Thus, our pointer analysis remains a single-phase, monotonic process without any graph rewriting—imprecise edges are never added, and precise edges are introduced as needed during construction.

Comparison with Selective Context Sensitivity. Cut-Shortcut is fundamentally different from selective context sensitivity approaches [28, 30, 39–41, 70, 76], which select methods by different heuristics and apply contexts directly to them (equivalent to cloning the methods and analyzing them). The main cost of these analyses is from cloning the methods and analyzing them multiple times. In contrast, to avoid the overhead of cloning, Cut-Shortcut neither selects methods nor analyzes them context-sensitively; instead, it simulates the effect of context sensitivity by suppressing imprecise edges and adding precise edges in the PFG. (The challenge is to reduce imprecision while not compromising soundness.) No contexts are applied to any methods in Cut-Shortcut.

Comparison with Method-Summary-Based Pointer Analysis. Method summarization is a widely used technique in inter-procedural program analysis, and has been applied to pointer analysis [11, 15, 54, 85]. A summary encodes how a method manipulates pointers—what objects its parameters may point to, how it updates pointer relationships internally, and what pointers it returns or makes accessible to callers. In terms of how these issues were handled in the motivating example in Section 2, Cut-Shortcut may seem to resemble method summarization by summarizing pointer flow at caller's invocation site. However, as we detail in later sections, Cut-Shortcut differs from summary-based pointer analysis in several key aspects:

- *No Context Overhead*: Summary-based pointer analysis approaches typically rely on context sensitivity to improve precision, often reanalyzing or reusing specialized summaries for each context [15, 85]. Cut-Shortcut, however, never duplicates methods during analysis, and hence achieves high efficiency.
- *Granularity*: Summary-based approaches encapsulate the entire method as a single unit, whereas Cut-Shortcut does not generate whole-method summaries. Instead, it operates at a finer granularity by selectively rerouting imprecise object flows for a method (or a series of methods, depending on the specific pattern) while leaving unaffected parts intact.
- *Top-Down Manner*: Summary-based approaches compute method summaries upfront and instantiate them as constraints when the summarized method is called. In contrast, Cut-Shortcut applies standard top-down context-insensitive pointer analysis on a refined PFG; the analysis still proceeds from callers to callees in a top-down way (with some callees' out-flow being ignored).

- *Pattern-Based Selectivity*: Summary-based approaches treat all methods uniformly, generating summaries for every procedure regardless of necessity. Cut-Shortcut, however, selectively identifies specific program patterns that commonly cause precision loss, making it more lightweight and focused.
- *Cross-Invocation Flow Rerouting*: The principle of Cut-Shortcut enables re-routing of object flows across multiple call sites. This capability is exemplified in the container-access pattern, where the input source of a container instance is directly connected to its retrieval sites by shortcut edges. Such cross-invocation flow rerouting is difficult to achieve using traditional method-summarization techniques.

3.2 Field-Access Pattern

There are two kinds of field accesses in Java, field store and field load (in this pattern, we focus on instance fields). Java programs often wrap field accesses in methods (e.g., setter/getter), so the client code needs to access the fields by calling these methods. However, in a context-insensitive analysis, object flows involved in the accesses to fields of different objects will be merged inside these methods, leading to imprecision. In this pattern, we tackle such precision loss via the overarching principle described in Section 3.1. To aid understanding, we first introduce the basic cases for handling field stores (Section 3.2.1) and loads (Section 3.2.2) using our motivating example, provided in Figure 1. We then describe a more general case in Section 3.2.3 to show how our semantics-based scheme tracks the value flows from the caller (and the caller of the caller ...), where the values are finally stored in the target fields through a series of invocation parameters.

3.2.1 Handling of Store. We address the imprecision in the points-to results of instance fields. For a field-store statement $s: x.f = y$, if objects pointed to by both x and y come from the parameters (including the `this` variable) of the method m that contains s, then incoming object flows from arguments of different calls (of m) are merged inside m, which may cause imprecision in $pt(o_i.f)$ where $o_i \in pt(x)$. Line 4 in Figure 1 gives such an example: `this` and `item` are parameters of method `setItem()`. When analyzing `setItem()` context-insensitively, objects from call sites at lines 17 and 22 are merged in it, leading to an imprecise result, i.e., $pt(o_{15}.\texttt{item}) = pt(o_{20}.\texttt{item}) = \{o_{16}, o_{21}\}$.

In Cut-Shortcut, our idea is to suppress the addition of certain store edges that propagate the merged object flows to instance fields (i.e., Targets), and find Sources at the call-sites that call the method containing those field stores. We search for store statements of the form $s: x.f = y$, in which both x and y are parameters of method m (that contains s) and not redefined in m. (This condition guarantees that the values of both x and y all come from arguments of m's call-sites.) If such an s is found, then multiple calls of m result in a conflated path (with only one node y, e.g., `item` in Figure 1(a)), with m being both Entrance and Exit. Accordingly, Targets are the instance fields being accessed by $x.f$ and stored at s (e.g., o_{15}.`item` and o_{20}.`item` stored at line 4). To prevent merged object flows from being propagated to the Targets, we suppress the addition of store edges to them in the PFG. For example, the addition of $\texttt{item} \longrightarrow o_{15}.\texttt{item}$ and $\texttt{item} \longrightarrow o_{20}.\texttt{item}$ are suppressed in Figure 1.

Because the values of both x and y come from arguments of m's call-sites, we can match the Source for Target (s) at each call-site—namely, the Source is the argument passed to y (e.g., the argument is `item1` at line 17 and y is the parameter `item` at line 3) and the Target is $o_i.f$, where $o_i \in pt(a)$ and a is the argument passed to x (e.g., the argument a is `c1` at line 17 and x is the parameter `this`). For example, at line 17, `item1` is the Source for Target o_{15}.`item` (as $pt(\texttt{c1}) = \{o_{15}\}$), and likewise, `item2` is the Source for Target o_{20}.`item` at line 22 (because $pt(\texttt{c2}) = \{o_{20}\}$). Thus, we add shortcut edges $\texttt{item1} \longrightarrow o_{15}.\texttt{item}$ and $\texttt{item2} \longrightarrow o_{20}.\texttt{item}$, which in this example achieves a precise result, i.e., $pt(o_{15}.\texttt{item}) = \{o_{16}\}$ and $pt(o_{20}.\texttt{item}) = \{o_{21}\}$ (depicted in the top half of Figure 1(b)).

3.2.2 Handling of Load. We address the imprecision in the points-to results for left-hand side (LHS) variables whose values are returned from the methods that load instance fields. For a field-load statement of the form $s: x = y.f$, if objects pointed to by y come from a parameter (including the `this` variable) of the method m that contains s, and x is the return variable of m, then the objects loaded from $y.f$ (where y points to the incoming objects from arguments of different calls of m) are merged inside m, which may cause imprecision in points-to sets for the LHS variables of m's call sites. Line 7 in Figure 1 gives such an example: `this` is a parameter of method `getItem()` and `r` is its return variable. When analyzing `getItem()` context-insensitively, objects from the call sites at lines 18 and 23 (i.e., o_{15} and o_{20}) are merged in it, and objects loaded from o_{15} and o_{20} (i.e., o_{16} and o_{21}) are also merged at `r` and propagated to `result1` and `result2`, leading to the imprecise result $pt(\texttt{result1}) = pt(\texttt{result2}) = \{o_{16}, o_{21}\}$.

In Cut-Shortcut, our idea is to suppress the addition of these return edges from m (that return values loaded from fields), which propagate merged object flows to the LHS variables (i.e., Targets), and find Sources at m's call sites. We search for load statements of the form $s: x = y.f$ in which (i) the values of both x and y are directly related to the variables of the call sites of m (that contains s), i.e., y is a parameter of m and not redefined in m (this condition guarantees that the value of y all come from arguments of m's call sites), and (ii) x is the return variable of m. If such s and m are found, then multiple calls of m result in a conflated path (with only one node x, e.g., `r` in Figure 1(a)), with m being both Entrance and Exit. Hence, the addition of return edges from m to the LHS variable (i.e., the Target) of each call-site that calls m are suppressed to avoid propagation of merged flows, and the Source at each call-site is identified, i.e., $o_i.f$ where $o_i \in pt(a)$ and a is the argument passed to y (e.g., a is `c1` at line 18 and y is `this`). For example, for field load at line 7 in Figure 1, the edges `r` $\longrightarrow$ `result1` and `r` $\longrightarrow$ `result2` are suppressed. For `result1`, we find its Source, o_{15}.`item`, at line 18 (because $pt(\texttt{c1}) = \{o_{15}\}$), and add shortcut edge o_{15}.`item` $\longrightarrow$ `result1`. Similarly, we add o_{20}.`item` $\longrightarrow$ `result2` at line 23. By this means, in this example we obtain a precise result: $pt(\texttt{result1}) = \{o_{16}\}$ and $pt(\texttt{result2}) = \{o_{21}\}$ (see the bottom half of Figure 1(b)).

3.2.3 Handling of Nested Calls for Field Accesses. In real-world programs, field accesses become complicated when nested method calls are involved. For instance, when a constructor is called, it may call another constructor or setter method where the store statement is finally executed. In addition, when inheritance is used, it becomes common for field stores to involve nested calls. For example, in Figure 3, `A.set()` is a setter called by `A`'s constructor (line 3). There are two nested calls to it, 8 $\longrightarrow$ 3 $\longrightarrow$ `A.set()` and 10 $\longrightarrow$ 3 $\longrightarrow$ `A.set()` (a call site is represented by its line number).

```
1 class A {                    6 // usage code
2   T f;                       7 T t1 = new T(); //o7
3   A(T t){this.set(t);}       8 A a1 = new A(t1); //o8
4   set(T p){this.f = p;}      9 T t2 = new T(); //o9
5 }                           10 A a2 = new A(t2); //o10
```

Fig. 3. An example of nested calls for field store.

Here, if we add shortcut edges `t` $\longrightarrow o_8$.`f` and `t` $\longrightarrow o_{10}$.`f`, at the call-site of `A.set()` (line 3), we still lose precision because both `this` and `t` are parameters of `A()`, and object flows from call sites at lines 8 and 10 are merged here, leading to the imprecise result $pt(o_8.\texttt{f}) = pt(o_{10}.\texttt{f}) = \{o_7, o_9\}$. To handle nested calls for field stores, we extend our approach in a more general manner to obtain better precision. Specifically, for a method that contains a field store, we analyze its nested calls (from the method itself to its caller, caller's caller, and so on, if possible) to add shortcut edges at proper call-sites. For the field store in `A.set()`, we trace back to the call-sites at lines 8 and 10, add shortcut edges `t1` $\longrightarrow o_8$.`f` and `t2` $\longrightarrow o_{10}$.`f`, and derive the precise result $pt(o_8.\texttt{f}) = \{o_7\}$ and $pt(o_{10}.\texttt{f}) = \{o_9\}$. For the details of our approach to handling nested calls for field stores, see Section 4.2, where we formally define this pattern using a set of inference rules.

3.3 Container-Access Pattern

Containers (e.g., `ArrayList`) are pervasively used in Java programs. Because numerous objects flow in and out of containers, how they are handled has a crucial effect on the precision of a pointer-analysis algorithm. With a context-insensitive approach, objects stored in different containers may flow to—and get merged—in the same container methods, and thus there may be horrendous precision loss when the merged objects flow out.

To separate object flows in different containers, conventional approaches improve precision by analyzing container methods context-sensitively. However, as pointed out by recent work [3, 14], applying context sensitivity to container methods will bring heavy efficiency costs, especially for large programs. As a result, to analyze containers precisely while reducing analysis overhead, researchers [3, 14, 21] have proposed various techniques. He et al. [21] perform a pre-analysis to identify context-dependent objects based on container-usage patterns, and then use a context-debloating technique to speed up object sensitivity by forming contexts with context-dependent objects only. Fegade and Wimmer [14] and Antoniadis et al. [3] propose to manually rewrite container implementations for composing code models that are simpler than the original implementations, but still preserve the side effects of containers needed for pointer analysis; then, the rewritten container code is analyzed using a context-sensitive approach. In Cut-Shortcut, we also need to handle containers for obtaining good precision. However, we do it differently by adopting a scheme that uses the principle from Section 3.1, without relying on context sensitivity at all.

3.3.1 Handling of Containers. Figure 4 shows an `ArrayList` example of a container. (For now, just focus on lines 1-9.) We create two `ArrayLists`, o_1 (line 1) and o_6 (line 6), and add two objects o_2 (line 3) and o_7 (line 8) to them, respectively. At lines 4 and 9, we retrieve the elements from o_1 and o_6 and assign them to `x` and `y` separately. Clearly, `x` (`y`) only points to o_2 (o_7) at run time.

```
1  ArrayList l1 = new ArrayList(); // o1, pt_H(l1) = {h1}
2  Object a = new Object();        // o2
3  l1.add(a);                      // call to Entrance, a is Source
4  Object x = l1.get(0);           // call to Exit, x is Target
5
6  ArrayList l2 = new ArrayList(); // o6, pt_H(l2) = {h2}
7  Object b = new Object();        // o7
8  l2.add(b);                      // call to Entrance, b is Source
9  Object y = l2.get(0);           // call to Exit, y is Target
10
11 Iterator it1 = l1.iterator();   // pt_H(it1) = {h1}
12 Object r1 = it1.next();         // call to Exit, r1 is Target
13 Iterator it2 = l2.iterator();   // pt_H(it2) = {h2}
14 Object r2 = it2.next();         // call to Exit, r2 is Target
```

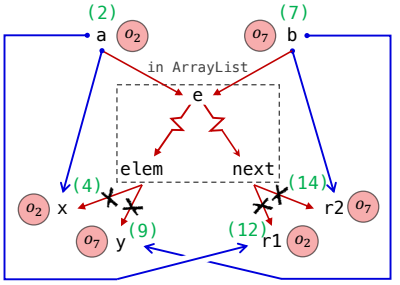


Fig. 4. An example of the container-access pattern.

Container implementations typically offer APIs for adding elements into, and retrieving elements from, containers. When a program creates multiple instances of the same container class (e.g., two `ArrayList` container instances in Figure 4) and uses these APIs separately on each instance, context-insensitive pointer analysis fails to distinguish the flows associated with different containers. Specifically, the input objects added to each container instance are merged and propagated to all retrieval sites, leading to significant imprecision. Consider Figure 4: with a context-insensitive analysis, both o_2 and o_7 are merged into the internal variable `e` (the parameter of `add()`), flow through a series of edges (representing the internal working of `ArrayList`, shown in the PFG in Figure 4) to the return value `elem` of `get()`, and then further flow to both `x` and `y`. As a result, both $pt(x)$ and $pt(y)$ contain o_2 and o_7, which is imprecise.

Following the over-arching principle of Cut-Shortcut, we identify these in-container conflated paths for each container class, which start at the parameters of addition APIs and end at the return values of retrieval APIs. These internal conflated paths (e.g., `e` $\rightsquigarrow$ `elem` in Figure 4) have their Entrance being the addition method and Exit being the retrieval method. Let Entrance_{cs} and Exit_{cs} denote call sites to the addition and retrieval APIs, respectively. At an Entrance_{cs}, the relevant input argument (say $\text{Entrance}^{arg}_{cs}$, e.g., `a` at line 3) is treated as a Source, while at an Exit_{cs}, the LHS variable (say Exit^{lhs}_{cs}, e.g., `x` at line 4) is treated as a Target.

The key challenge that remains is matching Sources and Targets —i.e., determining which pairs correspond to the same container instance. To handle this task uniformly—not only for simple cases like the example above, but also for more complex scenarios such as element retrieval via iterators or via `Map`'s `keySet()` and `values()`—we introduce a static abstraction for container instances, which we call *hosts*, and a mapping called the *pointer-host map*, denoted as pt_H. In our analysis, each dynamic container instance is abstracted statically by a host h generated at its allocation site, analogous to a standard allocation-site-based heap abstraction, but specialized for containers. The map pt_H then associates pointers with the set of hosts they may relate to. For example, because the `ArrayList` object o_1 is assigned to `l1`, we have $pt_H(\texttt{l1}) = \{h_1\}$, where h_1 denotes the host abstracting the first `ArrayList` instance. Similarly, we have $pt_H(\texttt{l2}) = \{h_2\}$. To match Sources and Targets, we compare the pt_H sets of the receiver variables at each pair of addition-method and retrieval-method call-sites (denoted by $\text{Entrance}^{rv}_{cs}$ and Exit^{rv}_{cs}). If these sets overlap—i.e., they can share a common host—the corresponding $\text{Entrance}^{arg}_{cs}$ and Exit^{lhs}_{cs} are considered a match, and we add a shortcut edge from the $\text{Entrance}^{arg}_{cs}$ (the Source) to the Exit^{lhs}_{cs} (the Target). In our above example, both `add()` and `get()` are called on `l1`, whose pt_H includes h_1. Therefore, we add the shortcut edge `a` $\longrightarrow$ `x`. Similarly, we add `b` $\longrightarrow$ `y` for the second host. More straightforwardly, this process can be viewed from the perspective of each host h maintaining a *source set* and a *target set*, consisting of the Sources and Targets associated with h via Entrance and Exit call-sites. We then connect each Source in h's *source set* to each Target in its *target set* via shortcut edges. As a result, for this example the analysis obtains precise points-to for the variables `x` and `y`, namely, $pt(x) = \{o_2\}$ and $pt(y) = \{o_7\}$. In this context, pt_H serves a role analogous to a traditional points-to relation. We will revisit its other uses shortly.

3.3.2 Handling of Container-Dependent Objects. Lines 11-14 in Figure 4 show a common usage of containers, i.e., accessing elements via iterators. At static time, each iterator depends on an abstract container instance (i.e., a host), and the elements retrieved from the iterator are the ones stored in that container instance (i.e., the Sources of that host). We call the iterator objects the *container-dependent objects*.[2] With a context-insensitive approach, when elements are retrieved through `Iterator.next()` methods (at line 12 and 14), o_2 and o_7 will flow from `e` (the parameter of method `add()`) to `next` (the return value of method `next()`) through a series of in-container PFG edges, and both flow to `r1` and `r2`, leading to an imprecise result. To handle containers comprehensively for better precision, we need to take such usages into account.

To handle container-dependent objects, our principle remains the same as the one in Section 3.3.1: the addition of return edges from Exits to the Exit^{lhs}_{cs} should be suppressed, and shortcut edges should be added to connect matched $\text{Entrance}^{arg}_{cs}$ and Exit^{lhs}_{cs} if $pt_H(\text{Entrance}^{rv}_{cs})$ and $pt_H(\text{Exit}^{rv}_{cs})$ overlap. The difference is that we need to expand the set of Entrances and Exits, as well as the rules for deriving pt_H. In our example, elements of containers are retrieved by the method `next()` (lines 12 and 14); hence, the additional conflated path here is the path `e` $\rightsquigarrow$ `next` (shown in

[2]Container-dependent objects include not only iterators, but also some other objects that depend on certain containers, such as the collection views of `Map` (e.g., the return value of `Map.keySet()`), an example of which is shown in Figure 12 in Section 4.3.

the PFG of Figure 4), with its Entrance still being `add()`, but Exit being `next()`. As for pt_H, we need to expand this mapping relation by (1) expanding the pt_H for pointers that point to container-dependent objects (we call these pointers p_{dep}, e.g., `it1` and `it2` are p_{dep} because they point to `Iterator` objects), and (2) for each p_{dep}, adding the abstract container instance(s), i.e., host(s), that they depend on (say h), into their pt_H (e.g., we want $pt_H(\texttt{it1}) = \{h_1\}$ and $pt_H(\texttt{it2}) = \{h_2\}$). Now the question left is, how does one identify h for different p_{dep} pointers? To answer this question, we first define a new set of methods called *host-propagation methods*. If a container-dependent object (e.g., an `Iterator` object) is created and returned by invoking a method m declared in the container class (the class of a container-dependent object is typically an inner class of its corresponding container class, and hence can access the container's content). We call m a *host-propagation method*. In our example, `iterator()` at line 11 is a host-propagation method because it is a method declared in the `ArrayList` class, and invoking it creates/returns an `Iterator` object that is pointed to by p_{dep} (`it1` at line 11). Thus, to map a p_{dep} to h, for each call-site of a host-propagation method, we propagate the hosts at its receiver variable (e.g., hosts in $pt_H(\texttt{l1})$, at call-site line 11) to its LHS variable (e.g., `it1` at line 11). As a result, for this example we have $pt_H(\texttt{it1}) = \{h_1\}$ because $pt_H(\texttt{it1}) \supseteq pt_H(\texttt{l1}) = \{h_1\}$.

To account for aliasing when building the pt_H relation (for example, when two pointers refer to the same `ArrayList` heap object, then both should map to the host abstracting that object), its derivation relies on the points-to information obtained during pointer analysis. (The detailed rules are presented in Section 4.3.) In turn, pt_H guides the pointer analysis in adding precise *shortcut* edges. Consequently, the construction of pt_H and the pointer analysis are performed in a mutually recursive fashion, making its derivation inherently on-the-fly: our analysis proceeds in a single round, where pt_H and pt are computed together until a fixed point is reached.

To sum up, by statically abstracting container instances as hosts and identifying their Sources (input elements) and Targets (output locations) on-the-fly, we can add *shortcut* PFG edges that directly connect its Sources and Targets within the pointer analysis. This approach allows us to avoid relying on imprecise interprocedural return flows—where return values from different container instances may be merged—to determine the points-to sets of container outputs. As a result, our approach improves analysis precision without impairing scalability.

Our scheme requires specifying a small set of container-related APIs in advance—such as Entrances, Exits, and *host-propagation methods*—while all remaining behaviors are handled automatically by the analysis. Additional mechanisms needed to achieve full precision are omitted here for brevity and deferred to Section 4.3. For generality, we focus on JDK container classes, whose APIs are stable and well-documented. Because the classification of relevant APIs is typically clear from their method names, one author was able to complete the necessary specification in about five hours. For custom or third-party containers, we ensure soundness by conservatively disabling the suppression of corresponding PFG edges—a mechanism detailed in Appendix A. Compared to past work [3, 14], which requires to rewrite code implementations for related container APIs, our scheme is more general and simpler. Moreover, unlike prior approaches—which still rely on context sensitivity for analyzing container methods [3, 14, 21]—our analysis is entirely context-insensitive.

3.4 Local-Flow Pattern

The local-flow pattern is a lightweight pattern that addresses the case of precision loss when the values of a method's parameters flow to the return variable via a series of local assignments. When such a method is called from multiple call-sites, their argument values are merged into the method, and propagated together to the LHS variable at each call-site, leading to imprecise points-to results. We use the example in Figure 5 to illustrate this pattern. In method `select()`, the values of its parameters `p1` and `p2` will flow to return variable `r`. It has two call-sites at lines 12 and 16. With a context-insensitive approach, objects o_{10}, o_{11}, o_{14} and o_{15} will be passed to and merged in `select()`,

```
1 A select(A p1,A p2){ 9 // usage code
2   A r;              10 A a1 = new A(); //o10
3   if (…)            11 A a2 = new A(); //o11
4       r = p1;       12 A r1 = select(a1, a2);
5   else              13
6       r = p2;       14 A a3 = new A(); //o14
7   return r;         15 A a4 = new A(); //o15
8 }                   16 A r2 = select(a3, a4);
```

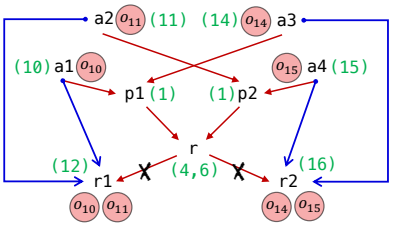


Fig. 5. An example of the local-flow pattern.

and then flow together to both r1 and r2 imprecisely: in any execution, r1 (r2) only point to o_{10} or o_{11} (o_{14} or o_{15}).

Following the over-arching principle of Cut-Shortcut, for this pattern, multiple calls to such a method m (whose parameter values flow to the return variable) forms a conflated path (paths) that starts from the specific method parameter(s) and ends at the return value (e.g., p1 ⟶ r and p2 ⟶ r are two conflated paths), with Entrance and Exit both being m. We focus on the cases that the return values must all come from the method parameters via (potentially a series of) local assignments (excluding other sources of return values, such as field loads and method calls), and the detection of such cases relies on an intraprocedural value-flow analysis. Hence, we can safely use arguments (which correspond to the parameters) as the Sources, and add shortcut edges from them to the Targets, i.e., the LHS variables of the call-sites. In this example, the addition of the return edges from r are suppressed, and we add shortcut edges a1 ⟶ r1 and a2 ⟶ r1 (a3 ⟶ r2 and a4 ⟶ r2) for the call-site at line 12 (16). As a result, r1 (r2) points to only $\{o_{10}, o_{11}\}$ ($\{o_{14}, o_{15}\}$) in Cut-Shortcut, which is precise. We formally define this analysis pattern in Section 4.4.

Limitations of Cut-Shortcut. Note that in Cut-Shortcut, the over-arching principle (Figure 2) is instantiated as three analysis patterns that are designed for the analysis of Java programs. Consequently, Cut-Shortcut cannot be directly applied to other programming languages; however, the high-level idea, namely suppressing the addition of imprecise flow edges and adding semantics-preserving precise flow edges in the PFG might inspire techniques for improving the precision of pointer analysis for other languages. For example, other programming languages may have precision-loss patterns similar to the patterns we proposed. Regarding the applicability of Cut-Shortcut to other abstractions like context sensitivity, our current approach is not designed to be pluggable into other (selective) context-sensitive analyses. Rather than selecting methods and analyzing them context-sensitively, Cut-Shortcut directly guides PFG construction without applying any contexts. However, it could be beneficial to have such a combination. For instance, the methods whose PFG edges are not affected by our approach, but considered by other selective context-sensitivity approaches [28, 39–41, 70], could be analyzed context-sensitively for possibly better precision. As for flow sensitivity, like almost all whole-program pointer analyses for Java in the literature [31, 46, 67, 72, 76], our approach is also flow-insensitive. It would be non-trivial to apply Cut-Shortcut to a flow-sensitive analysis, but it would be interesting to try it in the future.

4 Formalizing Cut-Shortcut

In this section, we formalize our Cut-Shortcut pointer-analysis core (Section 4.1) and its three patterns (Sections 4.2–4.4).

4.1 Pointer Analysis with Cut-Shortcut

In this section, we formalize how the Cut-Shortcut principle is integrated into a context-insensitive pointer analysis. We first give in Figure 6 definitions of the domains and notation

variables	$x, y \in \mathbb{V}$	pointers	$p \in \mathbb{P} = \mathbb{V} \cup (\mathbb{O} \times \mathbb{F})$
heap objects	$o_i, o_j \in \mathbb{O}$	points-to sets	$pt : \mathbb{P} \to \mathcal{P}(\mathbb{O})$
methods	$m \in \mathbb{M}$	PFG	$\mathcal{G} = (\mathcal{N} \subseteq \mathbb{P}, \mathcal{E} \subseteq \mathbb{P} \times \mathbb{P})$
fields	$f, g \in \mathbb{F}$	PFG edges	$a \longrightarrow b \in \mathcal{E}$
instruction labels	$i, j \in \mathbb{L}$	call-graph edges	$\to_{call} \subseteq \mathbb{L} \times \mathbb{M}$

$\texttt{arg}_k^i$	the k-th argument of call site i
$\texttt{LHS}^i$	the LHS variable (if exists) of call site i
$\texttt{param}_k^m$	the k-th parameter of method m
$\texttt{ret}^m$	the return variable of method m
def_x	the set of statements that define variable x

Fig. 6. Domains and notations.

used. The top section of Figure 6 lists relevant domains and specifies core functions and relations, while the bottom section provides a set of useful abbreviations. Most of the notation is standard or self-explanatory. Here, $pt(x)$ denotes the points-to set of pointer x, mapping it to a subset of heap objects (with $\mathcal{P}(\mathbb{O})$ denoting the power set of $\mathbb{O}$). A call-graph edge $i \to_{call} m$ indicates that call-site i invokes method m. The call-graph is constructed on-the-fly during pointer analysis, because precise resolution of dynamic dispatch relies on the points-to information of receiver variables at call-sites. The notation $\texttt{param}_k^m$ and $\texttt{arg}_k^i$ denote method parameters of method m and invocation arguments of call site at i (using instruction label i as its identifier), in the program, respectively. For the special cases where $k = 0$, $\texttt{param}_0^m$ denotes the `this` variable of m, and $\texttt{arg}_0^i$ denotes the receiver variable of call-site i. To simplify notation, we overload the membership operator $\in$ to also express containment within methods: $i \in m$ means instruction i appears in method m, and $x \in m$ indicates that x is a local variable of m.

Pointer Analysis, in a PFG View. Figure 7 presents our formalization of pointer analysis (for now, ignore rule [Shortcut] and $\boxed{\text{premises in black boxes}}$, which are introduced specifically for Cut-Shortcut). Essentially, our formalism is equivalent to ones appearing in the existing literature [50, 67, 72] because they all express Andersen-style pointer analysis for Java. To incorporate the Cut-Shortcut approach, we explicitly express the *subset constraints* among pointers by the edges of the PFG (pointer flow graph) [72, 81]. Specifically, all nodes of the PFG are pointers, and by [Propagate], an edge $a \longrightarrow b$ in the PFG indicates that the objects pointed to by a should also be pointed to by b (i.e., a subset constraint). Other rules describe how to generate PFG edges for different kinds of statements: object allocation ([New], where allocation-site-based heap abstraction is used, namely, o_i denotes all heap objects allocated by instruction i), local assignment ([Assign]), instance field load ([Load]) and store ([Store]), and method invocation ([Call], [Param], [Return]). The rules for static members and arrays are elided, because accessing static members is straightforward, and array accesses are analyzed with all elements collapsed into a single field of the array object.

Cut-Shortcut. Cut-Shortcut improves precision over context-insensitive pointer analysis by adhering to a single core principle: suppress the addition of PFG edges that introduce imprecision, and add shortcut edges that propagate precise object flows—captured before flow merging occurs. In the inference rules shown in Figure 7, the boxed premises encode the *cut* mechanism, while the rule [Shortcut] encodes the *shortcut* mechanism, as detailed below:

- $\boxed{i \notin cutStore}$ in [Store]: Here, $cutStore \subset \mathbb{L}$ is a set of store statements (identified by their labels). By rule [Store], if a store statement j is recorded in $cutStore$, then the store edges that would normally be generated for j are suppressed and not added to the PFG.

$$\frac{\texttt{i: x = new A()}}{o_i \in pt(x)}\ [\textsc{New}] \qquad \frac{\texttt{x = y}}{y \longrightarrow x}\ [\textsc{Assign}] \qquad \frac{a \longrightarrow b}{pt(a) \subseteq pt(b)}\ [\textsc{Prop}] \qquad \frac{a \twoheadrightarrow b}{a \longrightarrow b}\ [\textsc{Shortcut}]$$

$$\frac{\texttt{x = y.f} \qquad o_i \in pt(y)}{o_i.f \longrightarrow x}\ [\textsc{Load}] \qquad \frac{\texttt{i: x.f = y} \qquad o_i \in pt(x) \qquad \boxed{i \notin cutStore}}{y \longrightarrow o_i.f}\ [\textsc{Store}]$$

$$\frac{\texttt{i:r = b.m(a1,a2,...,an)} \qquad o_j \in pt(b) \qquad m' = \text{dispatch}(m, o_j)}{i \rightarrow_{call} m' \qquad o_j \in pt(\texttt{this}^{m'})}\ [\textsc{Call}]$$

$$\frac{\texttt{i:r = b.m(a1,a2,...,an)} \qquad i \rightarrow_{call} m'}{\forall 1 \le k \le n : \texttt{arg}_k^i \longrightarrow \texttt{param}_k^{m'}}\ [\textsc{Param}] \qquad \frac{i \rightarrow_{call} m \qquad \boxed{\texttt{ret}^m \notin cutRet}}{\texttt{ret}^m \longrightarrow \texttt{LHS}^i}\ [\textsc{Return}]$$

Fig. 7. Rules for pointer analysis, with Cut-Shortcut.

- $\boxed{i \notin cutRet}$ in [Return]: $cutRet \subset \mathbb{V}$ is a set of return variables. If a return variable $\texttt{ret}^m$ is in *cutRet*, then [Return] prevents adding return edges from $\texttt{ret}^m$ to the left-hand-side (LHS) variable at its call-sites (i.e., $\texttt{LHS}^i$ for each i such that $i \rightarrow_{call} m$).
- [Shortcut]: $a \twoheadrightarrow b$ denotes a shortcut edge between pointers a and b. By [Shortcut], each such shortcut edge contributes a corresponding PFG edge $a \longrightarrow b$, preserving intended precise flow.

The edges suppressed by Cut-Shortcut are of two types: store edges and return edges, which are controlled by the sets *cutStore* and *cutRet*, respectively. The current three patterns of Cut-Shortcut handle imprecision introduced by these two types of edges. Extensions to Cut-Shortcut, such as Cut-Shortcut^S (discussed in Section 7), may introduce new patterns that require suppressing additional types of edges. We emphasize that the derivation of *cutStore* and *cutRet* is performed prior to the pointer analysis. This pre-computation ensures that, once the main analysis begins, it is already known which edges should be suppressed—guaranteeing monotonicity of the analysis (i.e., no edge removals occur during the fix-point iteration). Our main analysis itself is then a single-round process, where PFG construction, call-graph construction (i.e., inference of $\rightarrow_{call}$), and the derivation of the *pointer-host map* (pt_H) used in the container-access pattern (detailed in Section 4.3) are performed on-the-fly in a mutually recursive fashion with the context-insensitive pointer analysis.

Next, we describe how the sets *cutStore* and *cutRet* are computed, and how shortcut edges are inferred, for each of the three patterns in Sections 4.2–4.4.

4.2 Field-Access Pattern

In this section, we formalize the field-access pattern of Cut-Shortcut.

4.2.1 Handling of Store. Figure 8 gives the rules for deriving *cutStore* and shortcut edges for handling field stores in the field-access pattern, including support for nested calls of stores. Intuitively, we suppress the addition of (imprecise) PFG edges introduced by a store statement and instead add precise shortcut edges at the call site (of the method containing the store statement)—when the base and right-hand-side (RHS) variables of the store (e.g., x and y in $x.f = y$) derive their values solely from method parameters (including `this`). To facilitate shortcut-edge inference, we define an auxiliary set called *tempStore*, which contains triples of the form $\langle base, f, from \rangle$ to represent (potential) store operations. To support handling nested calls of field stores (as discussed in Section 3.2.3), we construct and propagate *temp stores* along call chains—starting from the innermost callee (which contains the actual store statement) and continuing through its callers—until the value of *base* or *from* depends (transitively) on something other than the parameter of a method (e.g., a local allocation, or the return value from a method invocation).

$$\frac{\texttt{i: x.f = y} \quad i \in m \quad \texttt{param}^m_{k_1} = x \quad \texttt{param}^m_{k_2} = y \quad def_x = \emptyset \quad def_y = \emptyset}{i \in cutStore} \text{ [CutStore]}$$

$$\frac{i \rightarrow_{call} m \quad \texttt{param}^m_k = x \quad def_x = \emptyset}{x \hookleftarrow \texttt{arg}^i_k} \text{ [Arg2Var]}$$

$$\frac{((\texttt{i: x.f = y} \wedge i \in cutStore) \vee \langle x, f, y\rangle \in tempStore) \quad x \hookleftarrow \texttt{arg}^j_{k_1} \quad y \hookleftarrow \texttt{arg}^j_{k_2}}{\langle \texttt{arg}^j_{k_1}, f, \texttt{arg}^j_{k_2}\rangle \in tempStore} \text{ [PropStore]}$$

$$\frac{\langle base, f, from\rangle \in tempStore \quad base, from \in m \quad o_j \in pt(base) \quad ((def_{base} \neq \emptyset) \vee (def_{from} \neq \emptyset) \vee (\nexists k : \texttt{param}^m_k = base) \vee (\nexists k : \texttt{param}^m_k = from))}{from \longrightarrow\!\!\!\!\!\rightarrow o_j.f} \text{ [ShortcutStore]}$$

Fig. 8. Rules for handling field stores in the field access pattern.

Cut. We compute *cutStore* using rule [CutStore]. For a store statement $i : x.f = y$, if both x and y receive their values from method parameters and are not redefined within the method, then i is added to *cutStore*, indicating that the corresponding store edge should not be added into the PFG. This inference is performed prior to the main pointer analysis and is straightforward to compute, because it does not rely on any intermediate analysis results.

Shortcut. To simplify the inference rules, we introduce the auxiliary notation $\hookleftarrow$, defined by [Arg2Var]. The judgment $x \hookleftarrow \texttt{arg}^i_k$ states that variable x in method m receives its value from the k^{th} argument of call site i and is not redefined in m—implying that x can only point to objects originating from $\texttt{arg}^i_k$. We define two rules, [PropStore] and [ShortcutStore], to derive shortcut edges for field stores. Rule [PropStore] describes the construction and propagation of *temp stores*—either from store statements marked in *cutStore* or from existing *temp stores*—along call chains. Rule [ShortcutStore] determines when such propagation should terminate: if the disjunction in its premise holds, then the current *temp store* cannot be propagated further, and shortcut edges should be generated instead.[3]

4.2.2 Handling of Load. Figure 9 shows the rules for handling field loads in field-access pattern. Specifically, if return values of a method m are loaded from the objects that are passed to m as arguments, then precision loss can occur when arguments from different call sites of m are merged inside m. For such cases, the PFG edges from return variables of m to LHS variables of its call sites should be suppressed. Similar to the handling of stores, we define a set *tempLoad* to help identify what shortcuts should be added; *tempLoad* contains triples of the form $\langle to, base, f\rangle$, which represents a load operation $to = base.f$.

$$\frac{\texttt{i: x = y.f} \quad i \in m \quad \texttt{ret}^m = x \quad y = \texttt{param}^m_k \quad |def_x| = 1 \quad def_y = \emptyset}{\texttt{ret}^m \in cutRet \quad \langle x, y, f\rangle \in tempLoad} \text{ [CutLoad]}$$

$$\frac{\langle to, base, f\rangle \in tempLoad \quad base \hookleftarrow \texttt{arg}^i_k \quad o_i \in pt(\texttt{arg}^i_k)}{o_i.f \longrightarrow\!\!\!\!\!\rightarrow \texttt{LHS}^j} \text{ [ShortcutLoad]}$$

Fig. 9. Rules for handling field loads in the field access pattern.

[3] In principle, if a callee contains x field stores and is invoked at y distinct call sites, a naïve construction of shortcut-store edges could introduce up to $x \times y$ additional edges. In our implementation, we impose a cap of at most 10 field-store statements per handled method. Methods exceeding this threshold (i.e., with more than 10 field stores that satisfy the premise of [CutStore]) are excluded from edge suppression and shortcut-edge insertion. Consequently, the number of shortcut edges added is bounded by a small constant (10) times the number of call sites. Similar caps are applied to our handling of field loads and to the local-flow patterns discussed later.

Cut. [CutLoad] defines how to derive *cutRet*. For a load statement $i: x = y.f$, if (i) x is the return variable $\texttt{ret}^m$ and is only defined by this load statement (i.e., $|def_x| = 1$), and (ii) y obtains its value from method parameter and is not re-defined, then we add $\texttt{ret}^m$ to *cutRet*, indicating that the return edges from $\texttt{ret}^m$ to the LHS variables of call sites of m are to be suppressed. In addition, we record the triple $\langle x, y, f \rangle$ in *tempLoad*. This inference is performed prior to the main pointer analysis.

Shortcut. In [ShortcutLoad], for each invocation of a method that contains a load statement recorded in *tempLoad*, the argument ($\texttt{arg}_k^j$) flowing to the base of the actual load statement is used as the new base to generate shortcut edges from instance fields to the LHS variable ($\texttt{LHS}^j$) of call site j.

Compared with our earlier conference paper [48], the handling of *field loads* has been refined to consider only a single layer of calls, excluding nested calls. This change was motivated by two observations. First, experimental results indicated that handling nested calls of field loads is costly in time and space, but does not yield significant precision improvements. Second, supporting such handling would require performing strong updates to remove existing pointer-flow edges, which would cause the analysis to be non-monotonic.

4.3 Container-Access Pattern

JDK offers a powerful but complex *collection framework*. To apply the Cut-Shortcut principle to reduce the precision loss caused by containers, we introduce a series of concepts that model the key behaviors of containers.

We first introduce the notation that we use to explain how the container-access pattern is handled in Cut-Shortcut. (A table of the notation used is given in Figure 10.) A type $\tau \in \mathbb{T}$ denotes a Java class or interface, and $\mathcal{T}$ maps pointers/objects to types. A kind $c \in \mathbb{K}$ classifies container elements as Coll (collection elements), Mkey (map keys), or Mval (map values), which separates keys from values for analyzing maps.[4] Hosts $\mathbb{H}$ are defined as $\mathbb{L} \times \mathbb{K}$, where $\mathbb{L}$ are instruction labels and $\mathbb{K}$ are kinds—a host h_c^i abstracts the container instance(s) allocated at line i that holds elements of kind c (subscripts and superscripts are omitted when clear from context). pt_H is the pointer-host mapping introduced in Section 3.3.1, mapping each pointer to a set of hosts. <: denotes the standard subtyping relation in Java. $\Leftarrow$ and $\Rightarrow$ relate hosts to their Sources and Targets (as introduced in Section 3.3.1): $h \Leftarrow s$ means that objects pointed to by s are stored as elements in the container instance(s) abstracted by h (we say s is a Source of h); conversely, $h \Rightarrow t$ means that pointer t is a Target of host h, i.e., it receives elements stored in the container(s) abstracted by h. Finally, we define $\lhd$ as a preorder over hosts, referred to as *host dependency*. Intuitively, $h' \lhd h$ means that the elements stored in the container instance(s) abstracted by h' are also regarded as elements of those abstracted by h. In other words, the Sources of h depend on those of h'. This relation facilitates the propagation of Sources across hosts, especially for *bulk-entrance methods* like `ArrayList.addAll`.

types	$\tau \in \mathbb{T}$	subtyping	$<: \subseteq \mathbb{T} \times \mathbb{T}$
kinds	$c \in \mathbb{K} = \{\text{Coll}, \text{Mkey}, \text{Mval}\}$	host sources	$\Leftarrow \subseteq \mathbb{P} \times \mathbb{H}$
hosts	$h \in \mathbb{H} = \mathbb{L} \times \mathbb{K}$	host targets	$\Rightarrow \subseteq \mathbb{P} \times \mathbb{H}$
pointer-host map	$pt_H : \mathbb{P} \to \mathcal{P}(\mathbb{H})$	host dependency	$\lhd \subseteq \mathbb{H} \times \mathbb{H}$

Fig. 10. Additional notation for the container-access pattern.

We now define four sets that specify which methods are container-manipulation operations, and of what kind. In effect, these sets configure Cut-Shortcut to implement the container-access pattern for the indicated methods. (These configuration sets are typically populated by the analysis

[4] We also support map entries via an additional kind, omitted here for brevity as it is not central to the core design.

$$\frac{\langle m, \_\rangle \in \textsc{ContExit}}{\texttt{ret}^m \in \mathit{cutRet}}\ [\textsc{CutCont}] \qquad \frac{h \Leftarrow s \quad h \Rightarrow t}{s \dashrightarrow t}\ [\textsc{ShortcutCont}]$$

$$\frac{\texttt{i: x = new A()} \quad \texttt{A} <: \text{Collection}}{h^i_{\text{Coll}} \in pt_H(x)}\ [\textsc{ColHost}] \qquad \frac{\texttt{i: x = new A()} \quad \texttt{A} <: \text{Map}}{h^i_{\text{Mkey}} \in pt_H(x) \quad h^i_{\text{Mval}} \in pt_H(x)}\ [\textsc{MapHost}]$$

$$\frac{\langle m, k, c\rangle \in \textsc{ContEntr} \quad i \rightarrow_{call} m \quad h^l_c \in pt_H(\texttt{arg}^i_0)}{h^l_c \Leftarrow \texttt{arg}^i_k}\ [\textsc{HostSource}] \qquad \frac{\langle m, c\rangle \in \textsc{ContExit} \quad i \rightarrow_{call} m \quad h^l_c \in pt_H(\texttt{arg}^i_0)}{h^l_c \Rightarrow \texttt{LHS}^i}\ [\textsc{HostTarget}]$$

$$\frac{\langle m, c\rangle \in \textsc{ContProp} \quad i \rightarrow_{call} m \quad h^l_c \in pt_H(\texttt{arg}^i_0)}{h^l_c \in pt_H(\texttt{LHS}^i)}\ [\textsc{HostProp-I}] \qquad \frac{a \longrightarrow b \quad h \in pt_H(a) \quad \neg(a = \texttt{ret}^m \wedge \langle m, \_\rangle \in \textsc{ContProp})}{h \in pt_H(b)}\ [\textsc{HostProp-II}]$$

$$\frac{i \rightarrow_{call} m \quad h^l_c \in pt_H(\texttt{arg}^i_0) \quad \langle m, k, c\rangle \in \textsc{ContBulkEntr} \quad h^{l'}_c \in pt_H(\texttt{arg}^i_k)}{h^{l'}_c \lhd h^l_c}\ [\textsc{HostBulkEtr}] \qquad \frac{h' \lhd h \quad h' \Leftarrow s}{h \Leftarrow s}\ [\textsc{HostDep}]$$

Fig. 11. Rules for handling the container-access pattern.

designer, based on the JDK, but could be added to by a user so that user-defined container classes are treated according to the container-access pattern by Cut-Shortcut.)

- ContExit: a set of pairs $\langle m, c\rangle$, where m is an Exit method that returns elements of kind c.
- ContEntr: a set of triples $\langle m, k, c\rangle$, where m is an Entrance method, and its k^{th} parameter stands for an input element of kind c, to be stored as an element of the container.
- ContProp: a set of pairs $\langle m, c\rangle$, where a *host-propagation method* m (introduced in Section 3.3.2) propagates hosts of kind c from the receiver at call site i ($\texttt{arg}^i_0$) to its LHS variable ($\texttt{LHS}^i$).
- ContBulkEntr: a set of triples $\langle m, k, c\rangle$, where m is a *bulk-entrance method* that adds all elements of kind c from one container (the k^{th} parameter) into another container represented by the receiver. For example, $\langle$`ArrayList.addAll(int,Collection)`, 1, Coll$\rangle$ specifies that all elements (of kind Coll) from its second parameter (a collection) are added into the receiver.

Given the above sets, the container-access pattern is handled according to the inference rules listed in Figure 11.

Cut. By [CutCont], all return edges from the methods specified by ContExit must be suppressed. The inferences with respect to this rule are performed prior to the main pointer analysis, ensuring monotonicity.

Shortcut. [ShortcutCont] describes the addition of shortcut edges by connecting Sources and Targets associated with the same host. Specifically, whenever both $h \Leftarrow s$ (pointer s is a Source of host h) and $h \Rightarrow t$ (pointer t is a Target of host h) hold, a shortcut edge $s \longrightarrow t$ is introduced.

The remaining rules compute pt_H and derive tuples in the relations $\Leftarrow$ and $\Rightarrow$:

[ColHost] and [MapHost] introduce hosts at container-allocation sites. When the allocated object is a subtype of Collection, a Coll host is added to the pt_H of the receiving variable. When it is a subtype of Map, two distinct hosts of kinds Mkey and Mval are created for the same object o_i, thereby distinguishing keys from values.

[HostSource] derives the relation $\Leftarrow$ at each call site i invoking a container Entrance method m. If the receiver variable ($\texttt{arg}^i_0$) at i is associated with a host of kind c (specified in the triple $\langle m, k, c\rangle$), then the k^{th} argument at the call site ($\texttt{arg}^i_k$) is marked as a Source of that host. Similarly,

[HostTarget] derives $\Rrightarrow$ for each call-site i invoking a container Exit method m—for each host of kind c found in pt_H of the receiver variable, the LHS variable is marked as its Target.

[HostProp-I] handles call sites that invoke *host-propagation methods*. If a method m is specified to propagate hosts of kind c, then these hosts are propagated from the receiver variable ($\texttt{arg}_0^i$) to the LHS variable ($\texttt{LHS}^i$) at each invocation i of m. We illustrate this mechanism with an example involving the `keySet` view of a map (see Figure 12). Line 1 allocates a `Map`, resulting in two hosts (of kinds Mkey and Mval) added to $pt_H(\texttt{x})$ via [MapHost]. Line 2 invokes an Entrance method `put()`. By [HostSource], the arguments `k` and `v` are associated as Sources of the respective hosts in $pt_H(\texttt{x})$ (due to the specified triples $\langle \texttt{Map.put}, 1, \text{Mkey}\rangle$ and $\langle \texttt{Map.put}, 2, \text{Mval}\rangle$ in ContEntr). Line 3 propagates only the Mkey host to `s` because we specify $\langle \texttt{Map.keySet}, \text{Mkey}\rangle \in$ ContProp. Line 4 further propagates the Mkey host from `s` to `it`, because `iterator` is specified to propagate any hosts (specified as $\langle \texttt{Collection.iterator}, \_\rangle$ in ContProp). Line 5 invokes the Exit method `next()`, associating `r` as a Target of the MKey host. Finally, a shortcut edge $\texttt{k} \longrightarrow\!\!\!\!\!\rightarrow \texttt{r}$ is introduced by [ShortcutCont], linking the Source and Target of this host. This example also highlights the role of host kinds in handling maps precisely.

```
1 Map x = new Map();          // o1, pt_H(x) = {h^1_Mkey, h^1_Mval}
2 x.put(k, v);                // h^1_Mkey ⇐ k , h^1_Mval ⇐ v
3 Set s = x.keySet();         // pt_H(s) = {h^1_Mkey}
4 Iterator it = s.iterator();// pt_H(it) = {h^1_Mkey}
5 Object r = it.next();       // h^1_Mkey ⇛ r
```

Fig. 12. An example of Map container usage.

[HostProp-II] derives pt_H by propagating hosts along the PFG, which is essential for handling aliasing. If $pt_H(x) \neq \emptyset$ and y is an alias of x, then $pt_H(y)$ should include the hosts associated with x, because later invocations of Entrance or Exit methods may use y as the receiver. To ensure this behavior, we allow hosts to propagate along PFG edges, analogous to how heap objects flow in standard pointer analysis, allowing $pt_H(p)$ to be derived for every pointer p that may reference a container instance. However, such propagation is excluded for one case: return edges of a host-propagation method m. The reason propagation is suppressed is because (1) [HostProp-I] already derives pt_H for $\texttt{LHS}^i$ when $i \rightarrow_{call} m$, and (2) $\texttt{ret}^m$ may merge multiple hosts in its pt_H, so blindly propagating them out of the called method m would cause imprecision.

[HostBulkEtr] handles *bulk-entrance methods*. In Figure 13, line 5 calls `addAll()` to add elements from container o_1 into o_3. For soundness, the analysis must recognize not only `a` but also `b` as Sources of host h^1_{Coll}. To ensure this behavior, we define a *host-dependency* preorder $\lhd$ whose semantics is given by [HostDep]. [HostBulkEtr] infers such dependencies between hosts of the receiver and argument containers at bulk-entrance call-sites.[5] Because the standard configuration of the set ContBulkEntr contains the tuple $\langle \texttt{Collection.addAll}, 1, \text{Coll}\rangle$, the [HostBulkEtr] rule causes all Coll hosts in $pt_H(\texttt{l2})$ (the first, i.e., k=1 argument) to be related to the receiver's Coll hosts in $pt_H(\texttt{l1})$. Therefore, in this case we obtain $h^3_{\text{Coll}} \lhd h^1_{\text{Coll}}$, which then triggers [HostDep] to propagate Sources from h^3_{Coll} to h^1_{Coll}, thereby making `b` a Source of the latter. Consequently, two shortcut edge $\texttt{a} \longrightarrow\!\!\!\!\!\rightarrow \texttt{r}$ and $\texttt{b} \longrightarrow\!\!\!\!\!\rightarrow \texttt{r}$ are derived by [ShortcutCont] once `r` is identified as a Target of h^1_{Coll} at line 6.

```
1 ArrayList l1 = new ArrayList(); // o1, pt_H(l1) = {h^1_Coll}
2 l1.add(a);                      // h^1_Coll ⇐ a
3 ArrayList l2 = new ArrayList(); // o3, pt_H(l2) = {h^3_Coll}
4 l2.add(b);                      // h^3_Coll ⇐ b
5 l1.addAll(l2);                  // h^3_Coll ◁ h^1_Coll, hence h^1_Coll ⇐ b
6 Object r = l1.get(0);           // h^1_Coll ⇛ r
```

Fig. 13. An example of bulk operation for containers.

[5] The $\lhd$ preorder, by definition it is a transitive relation. We omit the rules that define the transitivity property of $\lhd$.

Compared to the earlier conference paper [48], we provide more detailed explanations of the rules, illustrated with examples, including the treatment of *bulk-entrance methods*, which were previously omitted due to space constraints. We also extend the discussion of conservative fallback mechanisms (see Appendix A), covering sound handling for custom containers and the propagation of hosts to `this` variables. Moreover, the mechanisms of the container-access patternhave been refined to reduce the amount of information that the analysis designer has to specify, and improving the efficiency of the analysis.

4.4 Local-Flow Pattern

Figure 14 gives the rules to derive *cutRet* and shortcut edges for the local-flow pattern.

$$\frac{def_{\mathtt{param}_k^m} = \varnothing}{\langle m, k\rangle \rightarrowtail \mathtt{param}_k^m} \text{ [Param2Var]} \qquad \frac{\texttt{i: x = y'} \quad \langle m, k\rangle \rightarrowtail y' \quad (\forall j \in def_x : (\texttt{j: x = y}) \wedge (\langle m, \_\rangle \rightarrowtail y))}{\langle m, k\rangle \rightarrowtail x} \text{ [Param2VarRec]}$$

$$\frac{\langle m, \_\rangle \rightarrowtail \mathtt{ret}^m}{\mathtt{ret}^m \in cutRet} \text{ [CutLFlow]} \qquad \frac{i \rightarrow_{call} m \quad \langle m, k\rangle \rightarrowtail \mathtt{ret}^m}{\mathtt{arg}_k^i \longrightarrow\!\!\!\!\!\rightarrow \mathtt{LHS}^i} \text{ [ShortcutLFlow]}$$

Fig. 14. Rules for local-flow pattern.

To formalize the local-flow pattern, we introduce the notation $\rightarrowtail$, as defined by rules [Param2Var] and [Param2VarRec]. The judgment $\langle m, k\rangle \rightarrowtail x$ states that x is a local variable in method m that obtains its value from the k^{th} parameter of m. Moreover, [Param2Var] and [Param2VarRec] together ensure that when $\langle m, k\rangle \rightarrowtail x$ holds, the values of x flow solely from m's parameters—possibly from multiple parameters (e.g., both $\langle m, k\rangle \rightarrowtail x$ and $\langle m, k'\rangle \rightarrowtail x$ may hold at the same time)—via zero or more local assignments. Crucially, this property guarantees that x does not receive values from other sources, such as field loads or method calls. With this notation in place, the edges to be suppressed—and shortcut edges to add—for local-flow pattern can be defined cleanly and precisely:

Cut. By [CutLFlow], if we find that the values of return variable $\mathtt{ret}^m$ all come from m's parameter(s), then we record $\mathtt{ret}^m$ in *cutRet*. The inference of [Param2Var], [Param2VarRec] and [CutLFlow] is performed prior to the main pointer analysis.

Shortcut. By [ShortcutLFlow], if $\langle m, k\rangle \rightarrowtail \mathtt{ret}^m$ holds, then for each call site i of method m, we add a shortcut edge from the k^{th} argument of i (i.e., $\mathtt{arg}_k^i$) to its LHS variable $\mathtt{LHS}^i$.

5 Soundness

In this section, we formally prove the soundness of Cut-Shortcut, based on the inference rules defined in Sections 4.1-4.4.

We begin by introducing notation. Let $\mathcal{A}$ denote a pointer-analysis instance over an input program, with subscripts identifying specific approaches (e.g., $\mathcal{A}_{ci}$ for context-insensitive analysis, $\mathcal{A}_{csc}$ for Cut-Shortcut). For analysis $\mathcal{A}_a$, we denote its corresponding pointer-flow graph (PFG) as $\mathcal{G}_a = (\mathcal{N}_a, \mathcal{E}_a)$. The set $\mathcal{E}^\times$ is defined as the collection of PFG edges included in $\mathcal{E}_{ci}$, but explicitly suppressed by the boxed premises of rules [Store] and [Return] in Figure 7. Formally:

$$e \in \mathcal{E}^\times \Longleftrightarrow \Big[\big(e = (y, o_j.f) \ \wedge \ i : x.f = y \ \wedge \ o_j \in pt(x) \ \wedge \ i \in cutStore\big) \bigvee \big(e = (\mathtt{ret}^m, \mathtt{LHS}^i) \ \wedge \ i \rightarrow_{call} m \ \wedge \ \mathtt{ret}^m \in cutRet\big)\Big] \tag{1}$$

Let $T = \{t \mid (s, t) \in \mathcal{E}^\times\}$, and define the set $shortcutSrc(t) = \{s \mid s \longrightarrow\!\!\!\!\!\rightarrow t\}$, where $\longrightarrow\!\!\!\!\!\rightarrow$ is derived from the four inference rules for shortcut edges in Figures 8, 9, 11, and 14.

Intuitively, we prove soundness as follows: for any object o that reaches a target $t \in T$ dynamically via a path P in $\mathcal{G}_{ci}$, suppose the ending edge of this path is suppressed in $\mathcal{G}_{csc}$. Then, along path

P, we show that there exists a corresponding source $s \in \mathit{shortcutSrc}(t)$, such that o can still reach t via a new path in $\mathcal{G}_{csc}$, by replacing the sub-path from s to t in path P with the shortcut edge connecting s to t.

To formalize reachability from objects to arbitrary pointers, we first address a subtlety in our PFG formulation. Following Sridharan et al. [72], when the pointer-analysis algorithm constructs the call graph on-the-fly, the value flow from a receiver to `this` is not explicitly represented by an edge. At call site $i : r = b.m(...)$, the callee is resolved dynamically via $o_j \in pt(b)$, and only o_j is added to $pt(\texttt{this}^{m'})$ with $m' = \mathrm{dispatch}(m, o_j)$. Thus, no edge from b to $\texttt{this}^{m'}$ exists in the PFG. While this way of constructing the PFG avoids imprecise propagation of $o_k \in pt(b)$ for $k \neq j$, it complicates reasoning about reachability on the PFG. To help present our proof, we define an auxiliary relation $\mathcal{E}^\dagger \subseteq \mathbb{P} \times \mathbb{P}$ ($\mathbb{P}$ is the pointer domain) such that $(\texttt{arg}_0^i, \texttt{this}^m) \in \mathcal{E}^\dagger \iff i \rightarrow_{call} m$. This relation makes the receiver-to-`this` flow explicit.

Definition 5.1 (Pointer-Flow Path). Given analysis $\mathcal{A}_a$ with edge sets $\mathcal{E}_a$ and $\mathcal{E}_a^\dagger$, a *pointer-flow path* for object o_i is a sequence $[p_0, p_1, \ldots, p_n]$, where p_0 is the variable receiving o_i at its allocation site and $\forall i \geq 0$, $(p_i, p_{i+1}) \in \mathcal{E}_a \cup \mathcal{E}_a^\dagger$. (When such a pointer-flow path exists, we say that "o_i reaches p_n under $\mathcal{A}_a$.") We say such a path exists dynamically if each edge corresponds to a runtime value flow, which we sometimes express as "o reaches p_n under dynamic execution." Consider a sound but imprecise analysis $\mathcal{A}_a$: every dynamic pointer-flow path must be included in the over-approximated path space under $\mathcal{A}_a$. The converse does not always hold, due to the possibility of spurious flows in an imprecise analysis.

We now prove the core soundness theorem and its supporting lemma, under Assumption 1 and 2.

Assumption 1. *We assume the soundness of the context-insensitive analysis $\mathcal{A}_{ci}$.*

Assumption 2. *We assume that the input sets* ContExit, ContEntr, ContProp, and ContBulkEntr *for the container-access pattern of $\mathcal{A}_{csc}$ are complete with respect to the container classes we consider.*

Theorem 5.2 (Soundness of $\mathcal{A}_{csc}$). *For any object o and pointer p, if o reaches p under dynamic execution, then o also reaches p under $\mathcal{A}_{csc}$.*

Proof. We construct a family of analyses $\mathcal{A}_{csc}^l$ ($0 \leq l \leq |\mathcal{E}^\times|$), where each $\mathcal{A}_{csc}^l$ denotes an analysis that suppresses only a subset of the edges in $\mathcal{E}^\times$. Specifically, consider $\mathcal{E}^\times$ as a fixed set (computed once by running $\mathcal{A}_{csc}$), and assign each $e \in \mathcal{E}^\times$ a unique index from 1 to $|\mathcal{E}^\times|$. The analysis $\mathcal{A}_{csc}^l$ suppresses only those edges with index less than or equal to l.[6]

For fixed l, we define the predicate $Q^l(p)$ over a pointer p as follows:

$$Q^l(p) \triangleq \textit{For any object } o \textit{ that reaches } p \textit{ dynamically, } o \textit{ also reaches } p \textit{ under } \mathcal{A}_{csc}^l.$$

We now prove, by induction on l, that for all l, $\forall p.Q^l(p)$ holds.

i) *Base Case.* If $l = 0$, $\mathcal{A}_{csc}^0$ degenerates to $\mathcal{A}_{ci}$. By assumption 1, $\mathcal{A}_{ci}$ is sound, which directly gives us $\forall p.Q^0(p)$.

ii) *Inductive Step.* Suppose that $0 \leq l < |\mathcal{E}^\times|$, and assume that $\forall p.Q^l(p)$ holds. We need to prove that $\forall p.Q^{l+1}(p)$. Let the suppressed edge indexed by $l+1$ be the edge (u, v). For an arbitrary pointer p, we distinguish three cases:

   a) If $p = v$, we must prove $Q^{l+1}(v)$.

[6] To define $\mathcal{A}_{csc}^l$ more formally: we keep all inference rules unchanged, except for the suppression conditions in [Store] and [Return] (Figure 7). We replace the boxed premise in [Store] with $\mathcal{L}(y, o_i.f) > l$, and that in [Return] with $\mathcal{L}(\texttt{ret}^m, \texttt{LHS}^i) > l$, where $\mathcal{L} : \mathbb{P} \times \mathbb{P} \to \mathbb{N}$ maps each edge in $\mathcal{E}^\times$ to its index, and assigns all other edges a constant greater than $|\mathcal{E}^\times|$, ensuring only edges with index $\leq l$ are suppressed in the PFG of $\mathcal{A}_{csc}^l$.

b) If $p \neq v$ and $pt(p)$ depends on $pt(v)$ (directly or indirectly). Because $Q^l(p)$ holds by the inductive hypothesis, if we can show $Q^{l+1}(v)$, then $Q^{l+1}(p)$ follows.
c) If $pt(p)$ does not depend on $pt(v)$, then the suppression of (u, v) does not affect $pt(p)$. Because $Q^l(p)$ holds, $Q^{l+1}(p)$ also holds.

Thus, it suffices to prove $Q^{l+1}(v)$. To that end, we invoke Lemma 5.3, which guarantees that for any object o that reaches v dynamically along a path ending in (u, v), there exists some $s \in \mathit{shortcutSrc}(v)$ on that path. (Among possible candidates, we choose s closest to o to ensure s precedes v when cycles are involved.) Because the suppression of (u, v) does not affect whether o reaches s, and o reaches s under $\mathcal{A}^l_{csc}$ (by the inductive hypothesis), the dynamic path from o to s plus the shortcut edge (s, v)—added by $\mathcal{A}^{l+1}_{csc}$—form a valid path from o to v under $\mathcal{A}^{l+1}_{csc}$. Therefore, $Q^{l+1}(v)$ holds.

By induction, for all $l \in [0, |\mathcal{E}^\times|]$ and all pointers p, any object o that reaches p dynamically also reaches p under $\mathcal{A}^l_{csc}$. Setting $l = |\mathcal{E}^\times|$ yields $\mathcal{A}^l_{csc} = \mathcal{A}_{csc}$, completing the proof. □

The key lemma used in the proof of Theorem 5.2 is stated and proved below.

Lemma 5.3. *Assume that $\forall p.Q^l(p)$ holds for a given l, where $0 \leq l < |\mathcal{E}^\times|$. Suppose that the suppressed edge indexed by $l+1$ is (u, v). Then, for any object that dynamically reaches v via a pointer-flow path P that ends with (u, v), there exists some $s \in \mathit{shortcutSrc}(v)$ that precedes v on P, under Assumption 2.*

Proof. We split the proof into cases based on the kind of pattern in Cut-Shortcut that is responsible for suppressing the edge (u, v).

i) *Store-Edge Case.* Suppose that (u, v) corresponds to a field-store edge—i.e., the first disjunct in Equation (1)—and takes the form $(y, o_j.f)$, where $x.f = y$ is a store statement recorded in *cutStore* and $o_j \in pt(x)$. By [CutStore], both x and y are formal parameters of the enclosing method m. Therefore, the path P must traverse an argument $\mathtt{arg}^i_{k_1}$ (at call site $i \rightarrow_{call} m$), passed to y. If $\mathtt{arg}^i_{k_1}$ is still a method parameter and not redefined, then by repeated applications of [PropStore], we can trace back along the call chain to find the outermost arguments s_y for y, and s_x for x, respectively (choosing the one nearest to o in the case of a cycle). Then, by [ShortcutStore],[7] a shortcut edge is added from s_y to $o_j.f$, because $o_j \in pt(s_x)$ under $\mathcal{A}^{l+1}_{csc}$ (guaranteed by $Q^l(s_x)$ and the fact that suppressing $(y, o_j.f)$ does not affect $pt(s_x)$). Thus, we find $s_y \in \mathit{shortcutSrc}(o_j.f)$, which precedes v in P.
ii) *Return-Edge Case.* Suppose that (u, v) is a return edge, i.e., the second disjunct in Equation (1), of the form $(\mathtt{ret}^m, \mathtt{LHS}^i)$, where $i \rightarrow_{call} m$ and $\mathtt{ret}^m \in \mathit{cutRet}$. We consider three sub-cases:degenerates
   a) If $\mathtt{ret}^m \in \mathit{cutRet}$ is derived from [CutLoad] by the *load* part of the field-access pattern. Then $\mathtt{ret}^m = x$ where $x = y.f$ and y is the k^{th} parameter of m and is not redefined. Hence, y originates from $\mathtt{arg}^i_k$ at the call site $i \rightarrow_{call} m$. By [ShortcutLoad], a shortcut edge is added from $o_j.f$ to $\mathtt{LHS}^i$ for any $o_j \in pt(\mathtt{arg}^i_k)$ under $\mathcal{A}^{l+1}_{csc}$ (guaranteed by the inductive

[7] Deriving this shortcut edge requires that the relevant call-graph edge $i \rightarrow_{call} m$ (and any along the call chain) be derivable under $\mathcal{A}^{l+1}_{csc}$. This is ensured by the inductive hypothesis $\forall p.Q^l(p)$, which guarantees the soundness of receiver's points-to set under $\mathcal{A}^l_{csc}$ (for each receiver involved in that call chain). Moreover, suppressing $(y, o_j.f)$ does not affect the derivation of these specific call-graph edges, as dispatch precedes callee-body analysis. Crucially, this also justifies the need for induction on the index of suppressed edges: earlier suppressed edges may influence the derivability of later call-graph edges (i.e., if the call graph were built imprecisely due to an imprecise receiver). By proceeding incrementally, we ensure that every call-graph edge involved in shortcut edges derivation of each inductive step remains soundly constructed. This reasoning applies to all shortcut-edge-derivation cases, and we omit further repetition for brevity.

hypothesis $Q^l(\texttt{arg}_k^i)$ and the fact that suppressing the specific return edge $(\texttt{ret}^m, \texttt{LHS}^i)$ does not affect $pt(\texttt{arg}_k^i)$). Therefore, we find $o_j.f \in \mathit{shortcutSrc}(\texttt{LHS}^i)$ along path P.

b) If $\texttt{ret}^m \in \mathit{cutRet}$ is derived from [CutCont] by the container-access pattern, then container elements are retrieved at call site i, from a container instance o_κ (or a container-dependent object that depends on o_κ). To ensure that a suitable shortcut edge can be identified, we first need to establish that the host abstraction h_c^κ for the container instance o_κ exists and is available at call site i—that is, $h_c^\kappa \in pt_H(\texttt{arg}_0^i)$. By [ColHost] and [MapHost], such a host h_c^κ is created at the allocation site of o_κ and associated with the LHS variable x referring to it. This host then propagates to $\texttt{arg}_0^i$ through one of two mechanisms: (1) via [HostProp-I], if $\texttt{arg}_0^i$ refers to a container-dependent object, or (2) via [HostProp-II], if $\texttt{arg}_0^i$ is an alias of x. Once h_c^κ reaches $\texttt{arg}_0^i$, [HostTarget] derives $h_c^\kappa \Rrightarrow \texttt{LHS}^i$. We then need to find a suitable s along path P that explains how the object entered the container as an input element, and $s \in \mathit{shortcutSrc}(\texttt{LHS}^i)$ can be derived. There are two cases for such an s: (1) a call site j invokes an Entrance, where s is an argument passed in (i.e., $s = \texttt{arg}_k^j$); (2) a *bulk-entrance method* inserts all elements of a host h' (where $h' \lhd h_c^\kappa$), and $h' \Lleftarrow s$. In both cases, we find that a shortcut edge that connects s to $\texttt{LHS}^i$ is indeed derivable. For the first case, similar to the reason why h_c^κ is available at call site i, h_c^κ is also be available at call site j, and we derive $h_c^\kappa \Lleftarrow s$ by [HostSource]. For the second case, we still derive $h_c^\kappa \Lleftarrow s$, but by applying [HostBulkEtr] at the call site of the *bulk-entrance method*, followed by [HostDep], which propagates Sources of h' to h_c^κ. Therefore, [ShortcutCont] will always add a shortcut edge from s to $\texttt{LHS}^i$, establishing that $s \in \mathit{shortcutSrc}(\texttt{LHS}^i)$, as desired.

c) If $\texttt{ret}^m$ comes from [CutLFlow] by the local-flow pattern, then $\langle m, \_\rangle \rightarrowtail \texttt{ret}^m$ indicates that $\texttt{ret}^m$ depends only on parameters of m. The dynamic path P must thus come from some $\texttt{arg}_k^i$ passed into m. By [ShortcutLFlow], a shortcut edge is added from $\texttt{arg}_k^i$ to $\texttt{LHS}^i$, and we have $\texttt{arg}_k^i \in \mathit{shortcutSrc}(\texttt{LHS}^i)$. □

6 A CFL-Reachability Formulation of Cut-Shortcut

In this section, we show how Cut-Shortcut can be formulated as a CFL-reachability problem. Although our implementation is not CFL-reachability-based, this formulation situates our technique within the broader landscape of formal-language-reachability problems. It also clarifies why Cut-Shortcut achieves such high efficiency compared with context-sensitivity-based techniques. Specifically, we establish two points: (1) Cut-Shortcut can be transformed from the standard *single Dyck-k*[8] CFL-reachability formulation of context-insensitive, field-sensitive pointer analysis, without introducing an *additional Dyck-k* language to pair method entry and exits as required for full context sensitivity. Thus, Cut-Shortcut remains solvable under the same formal model as context-insensitive analysis, without assuming additional computational power. (2) Relative to the context-insensitive baseline, Cut-Shortcut incurs only linear growth in grammar and graph size, thereby preserving asymptotic complexity when solved by a black-box CFL-reachability solver. In contrast, context-sensitivity-based techniques—even regularized ones such as k-CFA-based context sensitivity—replicate program nodes and edges under different contexts and can cause significant blowup.

Section 6.1 reviews the standard CFL-reachability formulation for Java pointer analysis. Sections 6.2–6.3 present a two-step transformation that yields the CFL-reachability formulation of Cut-Shortcut. Finally, Section 6.4 discusses what the CFL-reachability formulation of Cut-Shortcut says about its computational complexity.

[8] In a Dyck-k language, k denotes the number of distinct types of parentheses defined in the alphabet.

6.1 The L_F-Based CFL-Reachability Formulation of Java Pointer Analysis

Many program-analysis problems can be expressed as formal-language reachability problems, including pointer analysis [59]. Suppose that graph G is a directed graph with edge labels drawn from an alphabet Σ, and L is a formal language over Σ. Then each path p in G can be seen as labeled with a string $s(p) \in \Sigma^*$ obtained by concatenating—in order—the edge labels along the path. We say that p is an L-path if $s(p) \in L$. When such a path exists from node u to node v, we say that v is L-reachable from u, denoted by $u\ L\ v$. An L-reachability problem is a CFL-reachability problem when L is a context-free language. In that case, for any non-terminal S in the grammar of L, if $s(p)$ belongs to the language generated from S, we say that $u\ S\ v$, and refer to p as an S-path.

For Java pointer analysis, Andersen-style (context-insensitive) analysis can be formulated as a CFL-reachability problem over a pointer assignment graph (PAG) [74]—a graph representation slightly different from the PFG used in our approach. In the PAG, nodes represent variables and heap objects, and edges capture various types of value assignments. The associated language is denoted by L_F, where the subscript F reflects the fact that field sensitivity is enforced by the grammar. The PAG edges for L_F are constructed according to the rules shown in Figure 15. Unlike the PFG, PAG nodes do not include instance fields; instead, field accesses are encoded as edge labels—specifically, store[f] and load[f] label the edges generated by rules [L_F-Store] and [L_F-Load], respectively.

$$\frac{\texttt{i: x = new A()}}{o_i \xrightarrow{\text{new}} x}\ [L_F\text{-New}] \qquad \frac{\texttt{x = y}}{y \xrightarrow{\text{assign}} x}\ [L_F\text{-Assign}]$$

$$\frac{\texttt{x.f = y}}{y \xrightarrow{\text{store[f]}} x}\ [L_F\text{-Store}] \qquad \frac{\texttt{x = y.f}}{y \xrightarrow{\text{load[f]}} x}\ [L_F\text{-Load}]$$

$$\frac{\texttt{i:r = b.m(a1,a2,...,an)} \qquad i \rightarrow_{call} m'}{b \xrightarrow{\text{assign}} \texttt{this}^{m'} \qquad \forall 1 \le k \le n : \texttt{arg}_k^i \xrightarrow{\text{assign}} \texttt{param}_k^{m'} \qquad \texttt{ret}^{m'} \xrightarrow{\text{assign}} r}\ [L_F\text{-Call}]$$

Fig. 15. Rules for building the pointer assignment graph (PAG) for language L_F.

One important distinction from the PFG-based pointer analysis rules introduced earlier in Figure 7 is that the traditional CFL-reachability formulation of Andersen-style pointer analysis assumes a pre-built call graph—needed to derive the premise $i \rightarrow_{call} m'$ in [L_F-Call]. Such a call graph can be computed separately using techniques like Class Hierarchy Analysis (CHA) [13]. While recent work has explored CFL formulations that support on-the-fly call-graph construction [25], we follow the traditional formulation for conciseness and clarity. The grammar of L_F is as follows:

$$\begin{aligned}
\mathit{flowsTo} &\longrightarrow \text{new}\ \mathit{flows}^* \\
\mathit{flows} &\longrightarrow \text{assign} \mid \text{store[f]}\ \mathit{alias}\ \text{load[f]} \\
\mathit{alias} &\longrightarrow \overline{\mathit{flowsto}}\ \mathit{flowsTo} \\
\overline{\mathit{flowsTo}} &\longrightarrow \overline{\mathit{flows}}^*\ \overline{\text{new}} \\
\overline{\mathit{flows}} &\longrightarrow \overline{\text{assign}} \mid \overline{\text{load[f]}}\ \mathit{alias}\ \overline{\text{store[f]}}
\end{aligned}$$

Note that the inverse edge of a PAG edge $x \xrightarrow{l} y$ is defined as $y \xrightarrow{\bar{l}} x$ (where l denotes the label on the original edge). The generation of inverse edges is not shown explicitly in Figure 15, but can be understood as adding an inverse edge for each forward edge. We define u and v to be aliases if there exists an object o_i such that $u\ \overline{\mathit{flowsTo}}\ o_i\ \mathit{flowsTo}\ v$.

The solution of the L_F-reachability problem over the PFG is a mapping from variables to (abstract) heap objects: in the mapping, a variable v points to an object o_i iff there exists a *flowsTo*-path from o_i to v.

Because L_F enforces field sensitivity by matching store[f] and load[f] as balanced parentheses, it corresponds to a *Dyck-k* language over the family of fields (where k is the number of distinct fields). This language is a standard context-free language of balanced parentheses, and can be recognized by a single-stack pushdown automaton. A formulation of context-sensitive pointer analysis based on formal-language theory typically introduces an additional context-free language L_C to enforce context sensitivity by matching method entries and exits [71]. This extension effectively adds a second *Dyck-k′* language over method contexts (where k' is the number of distinct calling contexts). Solving a pointer analysis that is both context- and field-sensitive thus becomes an *interleaved Dyck-reachability problem*: determining paths that simultaneously satisfy both L_F and L_C, which requires recognizing potentially crossing pairs, as in the sequence "store[f] entry[c] load[f] exit[c]" (where c denotes a calling context). However, it has been shown that precise interleaved Dyck-reachability is undecidable [60]. In practice, over-approximations are obtained by regularizing L_C [71] and/or L_F [25, 74]. In contrast, in the next subsections we show that Cut-Shortcut can be formulated as a CFL-reachability problem without introducing a second Dyck language over calling contexts. Instead, its semantics can be expressed within a *single Dyck-k* CFL, L_{csc}, which can be seen as a modification of L_F. Thus, it can be solved by the same formal model that recognizes L_F, without requiring additional computational power.

Our formulation of L_{csc} is presented as a two-step transformation from the baseline L_F. Step One addresses the local-flow pattern, the field-access pattern, and the *cut* mechanism of the container-access pattern, which can be expressed entirely by refining edges in the PAG (i.e., modifying terminals of the language). This processing does not require points-to information, and can therefore be performed prior to the CFL-reachability solving procedure. Step Two, in contrast, handles the *shortcut* mechanism of the container-access pattern, which depends on querying alias information between variables that may reference container objects. Its formulation must therefore be carried out on-the-fly, alongside the pointer analysis. To capture this approach, we extend the L_F grammar with additional host-tracking nonterminals. Splitting the formulation of Cut-Shortcut in these two steps cleanly separates the points-to-independent pre-processing step from the grammar extension, making the correctness clearer and the complexity overhead easier to analyze.

6.2 Formulating Cut-Shortcut, Step One

Assuming a pre-built call graph, we can express the local-flow pattern, the field-access pattern, and the *cut* mechanism of the container-access pattern in Cut-Shortcut using the rules shown in Figure 16. These rules derive binary relations over $\mathbb{V} \times \mathbb{V}$ that represent imprecise assignments (which identify PAG edges to be suppressed) and precise shortcut assignments. Crucially, this derivation is independent of any points-to information, and can thus be performed before solving CFL-reachability on the PAG.

Given the binary relations derived in Figure 16, along with the edge relations defined for L_F (also on $\mathbb{V} \times \mathbb{V}$) in Figure 15, we now define the new PAG edge relations used in L_{csc} by applying set subtraction and union as follows (we use subscript F to denote relations from L_F, and the relations with no subscripts, such as cutStore[f], are as computed by the rules in Figure 16):

$$\begin{aligned}
&\mathsf{new}_{csc} := \mathsf{new}_F \\
&\mathsf{assign}_{csc} := (\mathsf{assign}_F \setminus \mathsf{cutReturn}) \cup \mathsf{shortcutAssign} \\
&\mathsf{store[f]}_{csc} := (\mathsf{store[f]}_F \setminus \mathsf{cutStore[f]}) \cup \mathsf{shortcutStore[f]} \\
&\mathsf{load[f]}_{csc} := \mathsf{load[f]}_F \cup \mathsf{shortcutLoad[f]}
\end{aligned} \tag{2}$$

Field-Access Pattern

$$\frac{\texttt{i: x.f = y} \quad i \in m \quad \texttt{param}^m_{k_1} = x \quad \texttt{param}^m_{k_2} = y \quad def_x = \emptyset \quad def_y = \emptyset}{y\ \texttt{cutStore[f]}\ x}\ [L_{csc}\text{-CutStore}]$$

$$\frac{\langle base, f, from\rangle \in tempStore \quad base, from \in m \quad ((def_{base} \neq \emptyset) \vee (def_{from} \neq \emptyset) \vee (\nexists k : \texttt{param}^m_k = base) \vee (\nexists k : \texttt{param}^m_k = from))}{from\ \texttt{shortcutStore[f]}\ base}\ [L_{csc}\text{-ShortcutStore}]$$

$$\frac{\texttt{i: x = y.f} \quad i \in m \quad \texttt{ret}^m = x \quad y = \texttt{param}^m_k \quad |def_x| = 1 \quad def_y = \emptyset \quad j \rightarrow_{call} m}{\texttt{ret}^m\ \texttt{cutReturn LHS}^j \quad \langle x, y, f\rangle \in tempLoad}\ [L_{csc}\text{-CutLoad}]$$

$$\frac{\langle to, base, f\rangle \in tempLoad \quad base \leftarrow\!\!\!\shortmid \texttt{arg}^j_k}{\texttt{arg}^j_k\ \texttt{shortcutLoad[f] LHS}^j}\ [L_{csc}\text{-ShortcutLoad}]$$

Local-Flow Pattern

$$\frac{\langle m, \_\rangle \rightarrowtail \texttt{ret}^m \quad i \rightarrow_{call} m}{\texttt{ret}^m\ \texttt{cutReturn LHS}^i}\ [L_{csc}\text{-CutLFlow}] \qquad \frac{i \rightarrow_{call} m \quad \langle m, k\rangle \rightarrowtail \texttt{ret}^m}{\texttt{arg}^i_k\ \texttt{shortcutAssign LHS}^i}\ [L_{csc}\text{-ShortcutLFlow}]$$

Container-Access Pattern (Cut)

$$\frac{\langle m, \_\rangle \in \textsc{ContExit} \quad i \rightarrow_{call} m}{\texttt{ret}^m\ \texttt{cutReturn LHS}^i}\ [L_{csc}\text{-CutCont}]$$

Fig. 16. Rules for deriving the binary relations for imprecise assignments and precise shortcut assignments on the PAG, for the field-access pattern, local-flow pattern, and the *cut* mechanism of the container-access pattern. For the field-access pattern, $[L_{csc}$-CutStore$]$ and $[L_{csc}$-ShortcutStore$]$ are adapted from [CutStore] and [ShortcutStore] in Figure 8, with modifications to operate on the PAG rather than on PFG edges; similarly, $[L_{csc}$-CutLoad$]$ and $[L_{csc}$-ShortcutLoad$]$ are adapted from Figure 9. For the local-flow pattern, $[L_{csc}$-CutLFlow$]$ and $[L_{csc}$-ShortcutLFlow$]$ are adapted from their counterparts in Figure 14. Other rules from Figures 8, 9 and 14 that do not involve PAG edges remain valid under the CFL-reachability formulation and are omitted here for brevity. Finally, for the container-access pattern, $[L_{csc}$-CutCont$]$ corresponds to [CutCont] in Figure 11; the remaining rules for this pattern will be reformulated in the next step (Section 6.3).

Step One of the transformation from L_F to L_{csc} thus involves the following substeps:

i) Generate the relations for the unmodified PAG via Figure 15
ii) Generate the relations for the three Cut-Shortcut patterns via Figure 16
iii) Combine the information computed in the previous substeps via rule set (2)
iv) Define a new *single Dyck-k* language L_{csc} as follows, based on the rules defined and edges computed in the previous steps:

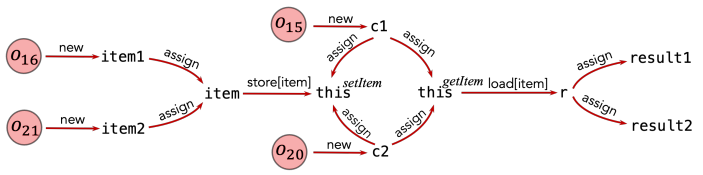
$$\begin{aligned}
flowsTo_{csc} &\longrightarrow \texttt{new}_{csc}\ flows_{csc}^{*} \\
flows_{csc} &\longrightarrow \texttt{assign}_{csc} \mid \texttt{store[f]}_{csc}\ alias_{csc}\ \texttt{load[f]}_{csc} \mid shortcutCont \\
alias_{csc} &\longrightarrow \overline{flows_{csc}}\ flows_{csc} \\
\overline{flowsTo_{csc}} &\longrightarrow \overline{flows_{csc}}^{*}\ \overline{\texttt{new}_{csc}} \\
\overline{flows_{csc}} &\longrightarrow \overline{\texttt{assign}_{csc}} \mid \overline{\texttt{load[f]}_{csc}}\ alias_{csc}\ \overline{\texttt{store[f]}_{csc}} \mid \overline{shortcutCont}
\end{aligned} \tag{3}$$

The definition of non-terminal *shortcutCont* is deferred to Step Two (Section 6.3).

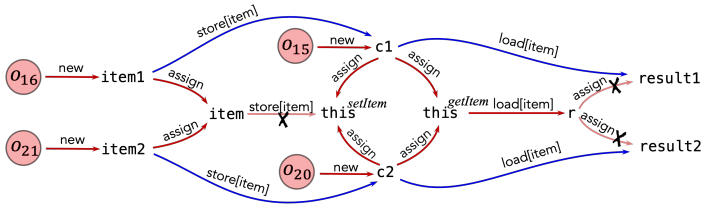
The points-to relations of interest are the solution to the $flowsTo_{csc}$-reachability problem in the modified PAG, with respect to grammar (3) augmented by a grammar defined in Section 6.3.

Similar to L_F, in which *flowsTo*-paths represent standard points-to relations. In L_{csc}, a path $o\ flowsTo_{csc}\ v$ indicates that variable v may point to heap object o. The structure of L_{csc} closely mirrors that of L_F, except that: (1) each edge relation from L_F has been replaced with its *csc*-refined version; and (2) a new non-terminal, *shortcutCont*, has been added to account for the *shortcut* mechanism of the container-access pattern, which we will elaborate on in Section 6.3.

```
1  class Carton {
2    Item item;
3    void setItem(Item item){
4      this.item = item;
5    }
6    Item getItem() {
7      Item r = this.item;
8      return r;
9    }
10 }
11 class Item {}
12
13 // usage code
14 void main() {
15   Carton c1 = new Carton();//o15
16   Item item1 = new Item();//o16
17   c1.setItem(item1);
18   Item result1 = c1.getItem();
19
20   Carton c2 = new Carton();//o20
21   Item item2 = new Item();//o21
22   c2.setItem(item2);
23   Item result2 = c2.getItem();
24 }
```



(a) PAG for L_F, constructed for the example shown on the left.



(b) PAG for L_{csc}, where blue arrows denote shortcut edges and arrows marked with ✗ denote suppressed edges.

Fig. 17. An example illustrating the CFL-reachability formulation of Cut-Shortcut. Subscripts of edge labels (i.e., F in (a), *csc* in (b)) are omitted for brevity.

To concretize this formulation, Figure 17 revisits the same motivating example from Section 2 and demonstrates how Cut-Shortcut is captured under the CFL-reachability framework up to this point. In Figure 17(a), the store[item] edge from `item` to $\texttt{this}^{\texttt{setItem}}$ brings a merged flow from both `item1` and `item2` into $\texttt{this}^{\texttt{setItem}}$. Similarly, the assign edges from `r` to `result1` and `result2` bring merged flows from $\texttt{this}^{\texttt{getItem}}$ to both result variables. As a result, there are two L_F-paths ending at `result1`:

$$o_{16} \xrightarrow{\text{new}} \texttt{item1} \xrightarrow{\text{assign}} \texttt{item} \xrightarrow{\text{store[item]}} \texttt{this}^{\texttt{setItem}} \xrightarrow{\textit{alias}} \texttt{this}^{\texttt{getItem}} \xrightarrow{\text{load[item]}} \texttt{r} \xrightarrow{\text{assign}} \texttt{result1}$$

$$o_{21} \xrightarrow{\text{new}} \texttt{item2} \xrightarrow{\text{assign}} \texttt{item} \xrightarrow{\text{store[item]}} \texttt{this}^{\texttt{setItem}} \xrightarrow{\textit{alias}} \texttt{this}^{\texttt{getItem}} \xrightarrow{\text{load[item]}} \texttt{r} \xrightarrow{\text{assign}} \texttt{result1}$$

where the *alias*-path between $\texttt{this}^{\texttt{setItem}}$ and $\texttt{this}^{\texttt{getItem}}$ is given by:

$$\texttt{this}^{\texttt{setItem}} \xrightarrow{\overline{\text{assign}}} \texttt{c1} \xrightarrow{\overline{\text{new}}} o_{15} \xrightarrow{\text{new}} \texttt{c1} \xrightarrow{\text{assign}} \texttt{this}^{\texttt{getItem}}$$

These paths imply that `result1` points to both o_{16} and o_{21}—an imprecise result. A similar imprecision occurs for `result2`.

In contrast, Figure 17(b) shows the PAG for the CFL formulation, where three imprecise edges are suppressed (crossed arrows), and four shortcut edges (blue arrows) are added, relative to Figure 17(a). As a result, only one L_{csc}-path leads to `result1`:

$$o_{16} \xrightarrow{\text{new}} \texttt{item1} \xrightarrow{\text{store[item]}} \texttt{c1} \xrightarrow{\text{load[item]}} \texttt{result1}$$

Thus, `result1` precisely points to o_{16}. Similarly, only one L_{csc}-path leads to `result2`, giving a precise analysis result. Note that all three suppressed edges in Figure 17(b) are necessary. For example, if the store[item] edge from `item` to $\texttt{this}^{\texttt{setItem}}$ was not suppressed, an imprecise *flowsTo*-path from o_{21} to `result1` would be present, namely,

$$o_{21} \xrightarrow{\text{new}} \texttt{item2} \xrightarrow{\text{store[item]}} \texttt{this}^{\texttt{setItem}} \xrightarrow{\textit{alias}} \texttt{c1} \xrightarrow{\text{load[item]}} \texttt{result1}$$

If the assign edge from `r` to `result1` was not suppressed, then the PAG would contain the path

$$o_{21} \xrightarrow{\text{new}} \texttt{item2} \xrightarrow{\text{store[item]}} \texttt{c2} \xrightarrow{\text{assign}} \texttt{this}^{\texttt{getItem}} \xrightarrow{\text{load[item]}} \texttt{r} \xrightarrow{\text{assign}} \texttt{result1}$$

Similarly, not suppressing the assign edge to `result2` causes `result2` to imprecisely point to o_{16}.

6.3 Formulating Cut-Shortcut, Step Two

In this step, we formalize the *shortcut* mechanism of the container-access pattern. Because this mechanism depends on aliasing information for variables that may reference container objects, it must be integrated into the pointer analysis itself and executed in an on-the-fly manner. In a pointer flow graph (PFG)-based formulation, this on-the-fly behavior is realized by propagating hosts—our abstraction of container instances—along PFG edges during pointer analysis, in a manner similar to heap-object propagation (as discussed in Section 4.3). To express this behavior in terms of CFL-reachability, we introduce a new non-terminal *hostFlows*, which captures the paths that a host node may take to reach a variable—analogous to how the non-terminal *flows* in L_F captures the paths that an object may take to reach a variable. Non-terminal *hostFlows* is used to define a central non-terminal *shortcutCont*, which characterizes shortcut paths that objects may take to reach variables. Before presenting the full grammar for them, we first introduce the additional PAG edge-construction rules shown in Figure 18, which are used in the grammar of these non-terminals.

$$\frac{\texttt{i:x = new A()} \quad \texttt{A} <: \text{Collection} \vee \text{Map}}{h_i \xrightarrow{\text{newHost}} x} \; [L_{csc}\text{-NewHost}] \qquad \frac{i \rightarrow_{call} m \quad \langle m, k, \_\rangle \in \text{ContBulkEntr}}{\texttt{arg}_k^i \xrightarrow{\text{hostDepend}} \texttt{arg}_0^i} \; [L_{csc}\text{-HostDep}]$$

$$\frac{i \rightarrow_{call} m \quad \langle m, k, \_\rangle \in \text{ContEntr}}{\texttt{arg}_k^i \xrightarrow{\text{source}} \texttt{arg}_0^i} \; [L_{csc}\text{-HostSource}] \qquad \frac{i \rightarrow_{call} m \quad \langle m, \_\rangle \in \text{ContExit}}{\texttt{arg}_0^i \xrightarrow{\text{target}} \texttt{LHS}^i} \; [L_{csc}\text{-HostTarget}]$$

$$\frac{i \rightarrow_{call} m \quad \langle m, \_\rangle \in \text{ContProp}}{\texttt{arg}_0^i \xrightarrow{\text{propagate}} \texttt{LHS}^i} \; [L_{csc}\text{-HostProp}] \qquad \frac{i \rightarrow_{call} m \quad \langle m, \_\rangle \in \text{ContProp}}{\texttt{ret}^m \; \text{PropExcept} \; \texttt{LHS}^i} \; [L_{csc}\text{-PropEx}]$$

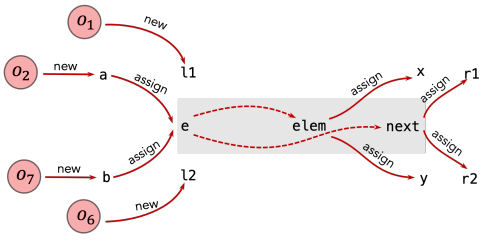
Fig. 18. Additional PAG edge-construction rules used to define the grammar of the non-terminals involved in the *shortcut* mechanism of the container-access pattern. [L_{csc}-NewHost] is adapted from [ColHost] and [MapHost] in Figure 11; likewise, [L_{csc}-HostDep] corresponds to [HostBulkEtr]; [L_{csc}-HostSource] to [HostSource]; [L_{csc}-HostTarget] to [HostTarget]; and [L_{csc}-HostProp] to [HostProp-I]. Finally, [L_{csc}-PropEx] captures the exceptional case excluded by the premise of [HostProp-II].

The rules in Figure 18 encode[9] those in Figure 11. The first five rules directly introduce additional PAG edges. The last one, [L_{csc}-ProEx], is used to define a new PAG edge type: hostAssign := $\text{assign}_{csc} \setminus$ PropExcept, which consists of assign edges, excluding those captured by the `PropExcept` relation. With these new edges in place, we define the following non-terminals:

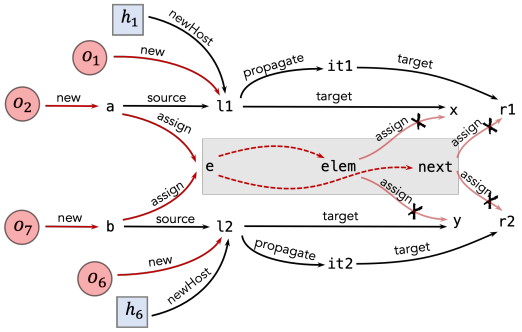
$$\begin{aligned}
\textit{hostFlows} &\longrightarrow \text{hostAssign} \mid \text{store[f]}_{csc}\ \textit{alias}_{csc}\ \text{load[f]}_{csc} \mid \text{propagate} \mid \textit{shortcutCont} \\
\overline{\textit{hostFlows}} &\longrightarrow \overline{\text{hostAssign}} \mid \overline{\text{load[f]}_{csc}}\ \textit{alias}_{csc}\ \overline{\text{store[f]}_{csc}} \mid \overline{\text{propagate}} \mid \overline{\textit{shortcutCont}} \\
\textit{sourceOfHost} &\longrightarrow \text{source}\ (\overline{\textit{hostFlows}} \mid \text{hostDep})^*\ \overline{\text{newHost}} \\
\overline{\textit{sourceOfHost}} &\longrightarrow \text{newHost}\ (\overline{\text{hostDep}} \mid \textit{hostFlows})^*\ \overline{\text{source}} \\
\textit{targetOfHost} &\longrightarrow \overline{\text{target}}\ \overline{\textit{hostFlows}}^*\ \overline{\text{newHost}} \\
\overline{\textit{targetOfHost}} &\longrightarrow \text{newHost}\ \textit{hostFlows}^*\ \text{target}
\end{aligned} \tag{4}$$

[9]For clarity, we do not distinguish the three host kinds in this CFL formulation, although doing so is straightforward: each rule in Figure 18 could be instantiated once per kind. We omit this distinction here to avoid unnecessary repetition.

```
1  ArrayList l1 = new ArrayList();//o1, h1
2  Object a = new Object();        //o2
3  l1.add(a);                      //⟨add, 1, Coll⟩ ∈ ContEntr
4  Object x = l1.get(0);           //⟨get, Coll⟩ ∈ ContExit
5
6  ArrayList l2 = new ArrayList();//o6, h6
7  Object b = new Object();        //o7
8  l2.add(b);                      //⟨add, 1, Coll⟩ ∈ ContEntr
9  Object y = l2.get(0);           //⟨get, Coll⟩ ∈ ContExit
10 // retrieve from container-dependent objects
11 Iterator it1 = l1.iterator();   //⟨iterator, _⟩ ∈ ContProp
12 Object r1 = it1.next();         //⟨next, _⟩ ∈ ContExit
13 Iterator it2 = l2.iterator();   //⟨iterator, _⟩ ∈ ContProp
14 Object r2 = it2.next();         //⟨next, _⟩ ∈ ContExit
```



(a) PAG fragment of L_F for the above code example.



(b) PAG fragment of L_{csc}, where black arrows denote new edges, and arrows marked with ✗ denote suppressed edges.

Fig. 19. An example to illustrate the CFL-reachability formulation of the container-access pattern. Black arrows in (b) represent additional PAG edges introduced via the rules in Figure 18. Dashed arrows indicate the presence of *flows* paths in (a) and $flows_{csc}$ paths in (b). Gray regions mark boundaries of JDK library internals. The subscripts F (in (a)) and *csc* (in (b)) on edge labels are omitted. For brevity, `AssignHost` edges in (b) are not shown, because they are not relevant to this example.

Intuitively, *hostFlows* captures the paths that a host node can take to reach host-related variables, such as a list or its iterator. The non-terminal *sourceOfHost* defines valid paths from a Source of host h to the host node that represents h, while *targetOfHost* defines paths from a Target of h to h. A subtle point: in *sourceOfHost*-paths, an edge labeled hostDep is allowed, but such edges do not appear in *targetOfHost*-paths. This asymmetry reflects the preorder $\lhd$ defined in Section 4.3, capturing Source dependencies between hosts. Intuitively, the hostDep edges allow the Sources of host h to "jump" into the *hostFlows* of another host h' when $h \lhd h'$.

Finally, the *shortcutCont* non-terminal is defined by concatenating a *sourceOfHost* path with a $\overline{\textit{targetOfHost}}$ path, matching the Sources and Targets of the same host:

$$\begin{aligned} \textit{shortcutCont} &\longrightarrow \textit{sourceOfHost}\ \overline{\textit{targetOfHost}} \\ \overline{\textit{shortcutCont}} &\longrightarrow \textit{targetOfHost}\ \overline{\textit{sourceOfHost}} \end{aligned} \tag{5}$$

In Figure 19, we revisit the example introduced earlier in Section 3.3 to illustrate the CFL-reachability formulation of the container-access pattern. In Figure 19 (a), under a context-insensitive pointer analysis, there exist two L_F-paths leading to `x`. (Note that the *flows*-path from `e` to `elem` models the internal behavior of the `ArrayList` class.)

$$o_2 \xrightarrow{\text{new}} \texttt{a} \xrightarrow{\text{assign}} \texttt{e} \xrightarrow{\textit{flows}} \texttt{elem} \xrightarrow{\text{assign}} \texttt{x} \quad \text{and} \quad o_7 \xrightarrow{\text{new}} \texttt{b} \xrightarrow{\text{assign}} \texttt{e} \xrightarrow{\textit{flows}} \texttt{elem} \xrightarrow{\text{assign}} \texttt{x}$$

These paths give an imprecise result: `x` is considered to point to both o_2 and o_7. Similarly, `y` is imprecisely inferred to point to both objects. In addition, for variable `r1`, retrieved via the iterator of `l1`, the following two L_F-paths exist:

$$o_2 \xrightarrow{\text{new}} \texttt{a} \xrightarrow{\text{assign}} \texttt{e} \xrightarrow{\textit{flows}} \texttt{next} \xrightarrow{\text{assign}} \texttt{r1} \quad \text{and} \quad o_7 \xrightarrow{\text{new}} \texttt{b} \xrightarrow{\text{assign}} \texttt{e} \xrightarrow{\textit{flows}} \texttt{next} \xrightarrow{\text{assign}} \texttt{r1}$$

which similarly yield imprecise results for r1 (and likewise for r2). In contrast, under Cut-Shortcut, as shown in Figure 19 (b), imprecise edges are suppressed and additional host-handling edges are introduced. Now only one L_{csc}-path exists for x:

$$o_2 \xrightarrow{\text{new}} \text{a} \xrightarrow{\textit{shortcutCont}} \text{x}$$

where the *shortcutCont* path from a to x expands to

$$\text{a} \xrightarrow{\textit{sourceOfHost}} h_1 \xrightarrow{\overline{\textit{targetOfHost}}} \text{x}$$

which in turn expands to

$$\text{a} \xrightarrow{\text{source}} \text{l1} \xrightarrow{\overline{\text{newHost}}} h_1 \xrightarrow{\text{newHost}} \text{l1} \xrightarrow{\text{target}} \text{x}$$

Likewise, only one L_{csc}-path leads to y, yielding precise results for the points-to sets of both x and y. For iterators, the only L_{csc}-path for r1 expands to

$$o_2 \xrightarrow{\text{new}} \text{a} \xrightarrow{\text{source}} \text{l1} \xrightarrow{\overline{\text{newHost}}} h_1 \xrightarrow{\text{newHost}} \text{l1} \xrightarrow{\text{propagate}} \text{it1} \xrightarrow{\text{target}} \text{r1}$$

This path, together with the absence of any L_{csc}-path to r1 from any other variable, gives the precise result that r1 points only to o_2. Similarly, the only L_{csc}-path for r2 yields the precise result that r2 points to o_7.

6.4 Complexity Discussion

From the perspective of CFL-reachability, the core insight of Cut-Shortcut remains consistent: compared to L_F, certain edges are suppressed in the PAG for L_{csc}—ensuring that some imprecise L_F-paths are no longer present—while new edges and non-terminals are introduced to enable the *shortcut* mechanism.

We emphasize that because the desired language L_{csc} for our language-reachability problem is expressed directly *using a context-free grammar* (i.e., grammars (3), (4) and (5)), it follows that L_{csc} is a context-free language. In contrast, if it were an interleaved-Dyck language, we would not be able to express it with a context-free grammar. Crucially, L_{csc} avoids introducing a second *Dyck-k* language for matching method entries and exits, so that reachability queries under L_{csc} maintain comparable complexity to those under L_F, despite its improved precision.

Concretely, we will first show in Lemma 6.1 that transforming L_F to L_{csc} incurs only linear growth in grammar and graph size, and prove in Theorem 6.2 that L_{csc} preserves the asymptotic complexity of L_F under a black-box CFL solver. Finally, we compares this overhead with k-CFA [65], a common regularization for context-sensitive analysis: while k-CFA bounds the context length, it can still suffer from a potential cost blow up, where the blow-up factor is exponential in k.

Lemma 6.1 (Transformation Overhead). *Moving from L_F to L_{csc} introduces only linear growth in the size of the underlying graph (nodes and edges) and the grammar.*

Proof. Let $|\mathbb{F}|$ denote the number of fields and $|\mathbb{H}|$ the number of hosts. We analyze the effect on graph and grammar through Step One and Step Two of our transformation:

(1) *Graph Nodes.* Step One adds no new nodes to the underlying PAG. Step Two introduces one additional node per host (i.e., each container allocation site), contributing $|\mathbb{H}|$ nodes. If the original PAG has n nodes, then the new PAG has $n + |\mathbb{H}|$ nodes. Because $|\mathbb{H}|$ is itself bounded by the number of allocation sites, hence by $O(n)$, the total number of nodes is still $O(n)$.

(2) *Graph Edges.* Step One refines the PAG of L_F by suppressing some edges and introducing shortcut edges. As noted in the footnote of Section 4.2.1, we impose a cap on the handling of the field-access and local-flow patterns, ensuring that the number of shortcut edges introduced is bounded by a small constant times the number of call sites in the program. Because the number of call sites is already $O(m)$, where m denotes the number of edges in the PAG of L_F, Step One contributes only $O(m)$ edges overall. Step Two introduces a small number of new edge kinds (e.g., newHost, hostDep, etc.), each generated one-for-one from either allocation sites or container-API call sites (cf. Figure 18). Because both allocation sites and call sites are bounded by $O(m)$, Step Two likewise adds only linearly many edges. Hence, the total number of edges in the transformed PAG remains $O(m)$.

(3) *Grammar.* The baseline grammar $\mathcal{G}_F$ has size $O(|\mathbb{F}|)$, because it contains one family of productions parameterized by fields. Step One rewrites $\mathcal{G}_F$ by adding subscripts to existing nonterminals and introducing a placeholder nonterminal *shortcutCont*. Step Two extends the grammar with a constant number of new nonterminals (*hostFlows*, *sourceOfHost*, *targetOfHost*, *shortcutCont*) and a fixed schema of productions—these reuse existing terminals and introduce one additional $\Theta(|\mathbb{F}|)$ family of field-parameterized productions inside *hostFlows*, of the form store[f] $alias_{csc}$ load[f] (and its symmetric variant). Instantiating such a schema for each $\mathsf{f} \in \mathbb{F}$ adds at most $2|\mathbb{F}|$ concrete productions. Formally, the grammar size grows from $|\mathcal{G}_F| = O(|\mathbb{F}|)$ to $|\mathcal{G}_{csc}| = O(|\mathbb{F}|) + c + 2 \cdot O(|\mathbb{F}|) = O(|\mathbb{F}|)$ where c is a constant. Crucially, no cross-product over fields and hosts arises, so there is no $O(|\mathbb{F}| \cdot |\mathbb{H}|)$ blow-up.

To sum up: if the original L_F instance has graph size (n, m) and grammar size $|\mathcal{G}_F|$, then the transformed L_{csc} instance has size $(O(n), O(m))$ and $O(|\mathcal{G}_F|)$ respectively—at most linear growth. □

Theorem 6.2. *L_{csc} preserves the asymptotic complexity of the baseline L_F.*

Proof. Let $T(n, m, |\mathcal{G}|)$ and $S(n, m, |\mathcal{G}|)$ denote the abstract time and space cost functions of a single-stack black-box CFL-reachability solver on a graph with n nodes, m edges, and grammar size $|\mathcal{G}|$. By Lemma 6.1, if the original L_F instance has graph node, graph edge, and grammar size $(n, m, |\mathcal{G}|)$, then the transformed L_{csc} instance has size $(O(n), O(m), O(|\mathcal{G}|))$. Hence the cost of solving L_{csc} can be expressed as

$$\text{Time}_{csc} = O(|\text{PAG}_F|) + T\big(O(n),\ O(m),\ O(|\mathcal{G}|)\big), \quad \text{Space}_{csc} = S\big(O(n),\ O(m),\ O(|\mathcal{G}|)\big).$$

The additive term $O(|\text{PAG}_F|)$ in Time_{csc} corresponds to the cost of refining the PAG in Step One, which is proportional to the size of the original PAG size (i.e., L_F). □

Compare with k-CFA-based context-sensitivity. We now compare this complexity with context-sensitivity–based techniques. As noted earlier, fully context- and field-sensitive pointer analysis is an interleaved-Dyck-reachability problem, and hence undecidable. A common regularization is to use k-CFA [65], which bounds method contexts by call strings of length k. This approach transforms the underlying PAG into a product graph: each node is paired with a call string of length $\leq k$, and each original edge (u, v) is lifted to edges between (u, c_1) and (v, c_2), where c_1 and c_2 are contexts (which may coincide or differ depending on the edge kind). The resulting graph is thus a blown-up version of the original PAG. Let γ be the number of call sites and $\mathbb{C}$ the set of allowed contexts. A standard bound is

$$|\mathbb{C}| \leq 1 + \gamma + \gamma^2 + \cdots + \gamma^k = O(\gamma^k).$$

If the original L_F instance has size $(n, m, |\mathcal{G}|)$, then the product graph under k-CFA has at most $n \cdot O(\gamma^k)$ nodes and $O(m \cdot \gamma^k)$ edges, while the grammar size remains unchanged. Hence, for the same

abstract single-stack CFL solver that handles L_F and L_{csc}, solving the k-CFA-based context-sensitive and field-sensitive pointer analysis requires

$$\text{Time}_{k\text{-CFA}} = T\big(n \cdot O(\gamma^k),\ O(m \cdot \gamma^k),\ |\mathcal{G}|\big), \quad \text{Space}_{k\text{-CFA}} = S\big(n \cdot O(\gamma^k),\ O(m \cdot \gamma^k),\ |\mathcal{G}|\big).$$

Thus, k-CFA can be solved by the same black-box CFL solver that solves L_F and L_{csc}, but at the cost of a potential blowup in the size of the analysis graph that is exponential in k. In practice, k-CFA is inefficient/less-scalable [36] than context insensitivity or Cut-Shortcut.

7 Cut-Shortcut^S

In this section, we present Cut-Shortcut^S, an extension of Cut-Shortcut that addresses imprecision caused by pervasive use of Java streams, by modeling various stream pipeline operations within pointer analysis.

7.1 Motivation

7.1.1 Challenge Posed by Java Streams. Pointer analysis for Java has seen substantial research progress in improving precision, efficiency, and scalability. However, most existing whole-program pointer analyses are built around core Java features and evaluated on Java 6 benchmarks [22, 23, 28–30, 32, 37, 40, 41, 46, 47, 70, 76–78].

However, according to a report by New Relic [1], the most-used Java version in industry as of 2023 is Java 11, with Java 8 being a close second. In practice, applying existing pointer-analysis approaches to Java 8 (or higher-version) programs will face significant challenges [18]: the precision, scalability, or even soundness may not be preserved when hard-to-analyze language features are used. In the past few years, effort has been made towards more sound and precise handling of those features, including Java reflection [42, 68], invoke-dynamic along with Java lambda expressions [18], and native codes [87]. That said, Java streams, a widely used feature introduced in Java 8, are still an obstacle for precise whole-program pointer analysis.

Java streams are a pervasively used language feature in modern Java programs, and expose a set of APIs for developers to process collections of data using functional-style operations, to generate elegant code. In programs, streams are used as pipelines, which typically contain a source, zero or multiple intermediate operations, and one terminal operation. If a typical context-insensitive Andersen-style analysis [2] is used, the wide use of stream pipelines can pose a significant challenge to the precision, because the analysis will mix together elements (i.e., heap objects) that pass through different pipelines.

7.1.2 A Motivating Example. Figure 20(a) shows a motivating example. The class `Stud` has two `String` fields: `name` and `id`. In `main()`, four `Stud` objects are created and assigned to variables `a1` to `a4`; hence we observe $pt(\texttt{a}_i) = \{o_i\}$ for each $\texttt{a}_i$. Figure 20(b) illustrates how these objects propagate in the first stream pipeline at runtime. Pointers `a1` and `a2` are passed as arguments for method invocation of `Stream.of`, which generates a new instance of `Stream` type that takes `a1` and `a2` as input elements. A `Stream.filter` operation then drops students whose `name.length() <= 4`, so only o_1 survives. Then, an invocation of `Stream.findFirst` would find the first object in that pipeline, and wrap it into an `Optional` type (a wrapper type in Java to represent potentially `null` objects), which is finally unwrapped through `Optional.get`, with o_1 assigned to variable `r1`. Therefore, `r1` would only point to o_1 but not o_2 after dynamic execution. Similarly, `r2` would only point to o_3.

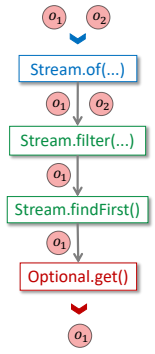
Static Over-Approximation Goal. The exact behavior of `Stream.filter` is determined by the lambda expression[10] used as the argument at this call site. The semantics of an arbitrary lambda

[10] In Java, each lambda expression is a reference to an object with a type annotated as `FunctionalInterface`, indicating that it is a "functional object." More details of Java lambda expressions will be discussed in Section 7.4.

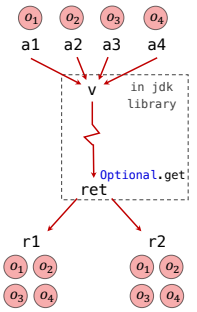
```
Stud {
  String name, id;
  Stud(String name,
       String id) {
    this.name = name;
    this.id = id;
  }
  String getName() {
    return this.name
  }
  String getID() {
    return this.id
  }
}
```

```
main() {
  Stud a1 = new Stud("1", "Alice");   // o1
  Stud a2 = new Stud("2", "Bob");     // o2
  Stud a3 = new Stud("3", "Charlie"); // o3
  Stud a4 = new Stud("4", "Dean");    // o4

  Stud r1 = Stream.of(a1, a2)
            .filter(a -> a.name.length > 4)
            .findFirst()
            .get();

  Stud r2 = Stream.of(a3, a4)
            .filter(a -> a.name.length > 4)
            .findFirst()
            .get();
}
```

(a) A stream use example.



(b) Dynamic behavior of the first pipeline instance.



(c) PFG fragment under context insensitivity

Fig. 20. A motivating example for Java streams.

expression is difficult to model statically, because it could function as any `Stud` $\mapsto$ `Boolean` function in this case. However, we can still aim for a sound over-approximation: a reasonable static analysis result would be $pt(\texttt{r1}) = \{o_1, o_2\}$ and $pt(\texttt{r2}) = \{o_3, o_4\}$, meaning that the analysis can tell the difference between the objects passed through different stream pipelines.

Context Insensitivity. Figure 20(c) shows the PFG fragment under context insensitivity. The arguments of the first `Stream.of` call-site, namely `a1` and `a2`, are passed to the parameter `v` of the callee (which is a JDK library method). Due to context-insensitivity, the analysis does not distinguish between the two `Stream.of` invocations; hence `a3` and `a4` are also passed to `v`, resulting in the merging of object flows from both call-sites. The merged flow then propagates through the library's internal methods and returns to both `r1` and `r2`, yielding $pt(\texttt{r1}) = pt(\texttt{r2}) = \{o_1, o_2, o_3, o_4\}$.

Limitations of Context Sensitivity. While deep context sensitivity can, in theory, recover the desired precision, it comes at exponential cost. In practice, even for our motivating example—which consists of only two stream pipelines—selective context-sensitive analyses that target stream-related methods fail to yield precise results. For instance, selective 5call-4h (5 layers of call-site sensitivity with 4 layers of context-sensitive heap abstraction) is unable to distinguish the two pipelines, and increasing the context depth further leads to out-of-memory errors (exceeding 100GB). Similarly, with object sensitivity, selective 3obj-2h (3 layers of object sensitivity with 2 layers of context-sensitive heap abstraction) also fails to distinguish the pipelines, and increasing the context depth provides no additional precision benefits for this program.

Our Approach. Instead of relying on context sensitivity via method cloning, we adopt the Cut-Shortcut philosophy—suppressing imprecise PFG edges and introducing precise shortcut edges. We extend our container-access pattern to handle Java streams by modeling stream pipelines as containers. The key insight is that each stream pipeline's outputs can only derive from its own inputs. In our container-access pattern, we abstract container instances as hosts and associate each with its input elements (Sources) and output locations (Targets), then suppress imprecise return edges and introduce shortcut edges from Sources to Targets of the same host. This idea naturally extends to streams: pipelines are abstracted as a new kind of host, with Sources and Targets identified based on modeled stream operations. This uniform treatment lets us reuse some of the existing analysis rules, rather than introducing a completely separate mechanism for streams. It also cleanly accounts for conversions between streams and containers: by treating pipelines as a new kind of host, distinct from those for collections or maps, such conversions become

transformations between host kinds. At the same time, streams introduce challenges beyond those of standard containers. Pipeline operations such as `map` and `flatMap` transform element values rather than merely transferring references, and have no direct analog in collections. These higher-order operations apply user-supplied callbacks (lambdas/method references) to elements, which are not handled "for free" by container abstractions and require careful modeling. Thus, while our reuse of the container-access pattern provides a clean starting point, capturing the distinct semantics of streams operations ensures that this extension is a non-trivial generalization.

In the following subsections, we define the scope and classification of stream operations (Section 7.2) and describe our modeling strategies in detail (Sections 7.3–7.5).

7.2 Classify Stream Operations

The Java stream APIs consist of a set of publicly accessible classes and methods that developers can use to process elements in a pipeline style. Figure 21 provides an overview of the stream APIs, focusing specifically on operations relevant to pointer analysis. To the best of our knowledge, we are the first to systematize Java stream operations from the perspective of pointer analysis, highlighting and classifying the different ways in which these operations affect object flow.

As shown, Java defines four interfaces to abstract streams over different types. `IntStream`, `LongStream`, and `DoubleStream` represent streams over primitive types (`int`, `long`, and `double`, respectively), while the `Stream` interface handles reference types. Because pointer analysis is concerned only with heap objects—which are reference types—we restrict our analysis to streams over reference types. It is worth noting that some primitive-type stream interfaces provide operations to convert to reference-type streams, such as `IntStream.mapToObj`; hence such operations must also be modeled in our analysis. In addition to the `Stream` interface, other classes may interact with reference-type streams. One example is the `Optional` class. As shown in the code snippet in Figure 20(a), calling `Stream.findFirst` does not return an element in that stream pipeline directly; instead, it returns an `Optional` object that wraps the element, which is later retrievable via `Optional.get`. Another example is the `Collection` interface, which provides the method `Collection.stream` to create a stream instance whose input elements are the elements of a `Collection` instance. In general, aside from methods declared in the `Stream` and `Optional` classes, any method in the JDK library that creates and returns an object of type `Stream` or `Optional` should be considered in our handling, because it may participate in the life-cycle of a stream pipeline.

We refer to all such methods as stream pipeline-related operations (stream operations, for short), and classify them into three main categories, corresponding to the three stages of a stream pipeline life-cycle: *creation*, *process*, and *output*. Representative examples of each category are shown in Figure 21, along with their method signatures.[11] Beyond life-cycle stages, stream operations can also be classified based on how they are parameterized, which indicates how they behave—and in turn determines how our analysis deals with them. Accordingly, each of the three main categories–*creation*, *process*, and *output*—is further divided into two or three subcategories, to reflect such behavioral characteristics. This classification is visually represented in Figure 21, where the color indicates the life-cycle stage of each operation, and the depth of color encodes its behavioral characteristics. The three behavioral subcategories are defined as follows:

[11]Note that our classification is based on how operations affect object flow, and thus differs slightly from the Java documentation, which uses the terms *source*, *intermediate* operation, and *terminal* operation. For example, `Stream.findFirst` is classified as a *terminal* operation in the official documentation, but we treat it as a *process* operation, instead of an *output* operation. This approach is warranted because the returned value is not a stream element but an `Optional` object that wraps it, requiring further unwrapping. Consequently, our analysis handles such operations similarly to *process* operations like `Stream.filter` or `Stream.sorted`, while those unwrapping operations, like `Optional.get`, are classified as *output* operations.

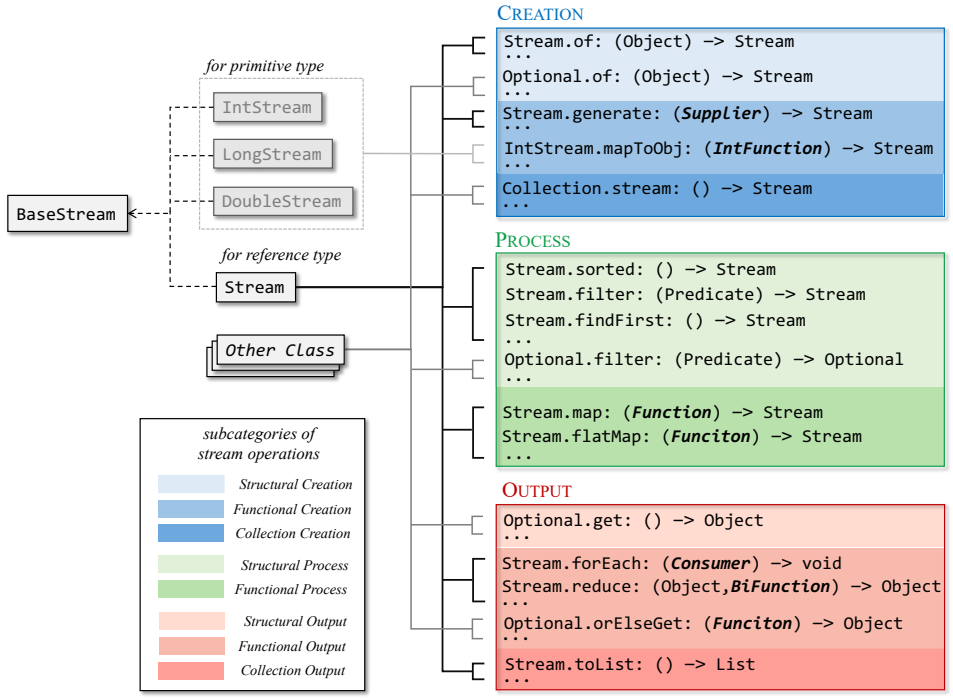


Fig. 21. Overview of pointer-analysis-related stream-pipeline operations

(a) *Structural.* These operations have fixed semantics and do not take user-specified behavior (e.g., lambda expressions) as arguments. Examples include `Stream.of` (a *structural creation* operation), `Stream.findFirst` (a *structural process* operation), and `Optional.get` (a *structural output* operation). They are well-characterized and have predictable effect, and hence can be handled uniformly, either by specifying them in an input set already defined by the container-access pattern or by introducing a single new analysis rule. Details are given in Section 7.3.

(b) *Functional.* These operations take functions (i.e., lambda expressions/method references) that encode how stream elements are processed or transformed as invocation arguments.[12] For example, `Stream.map` (a *functional process* operation) takes a `Function` argument (which represents the function to be applied to each pipeline element), and produces a new `Stream` object as its return value. The elements in the resulting `Stream` object depend on how the functional argument behaves. Similarly, `Stream.generate` and `Stream.forEach` are *functional creation* operation and *functional output* operation, respectively. Each *functional* operation has its unique semantics (regarding how the functional arguments may interact with stream elements) even within the same sub-category. For instance, `Stream.map` and `Stream.flatMap` are both *functional process* operations, but they affect stream elements in different ways. Consequently, we define a separate rule for each *functional* operation to model their different semantics. Section 7.4 details our handling of these operations, which includes integrating with an existing static analysis for Java's functional features [18].

[12] An exception to this category is `Stream.filter`, which, while technically functional (taking a function as its argument), only removes elements from the stream rather than producing new ones. Because our analysis focuses on tracking and distinguishing stream elements (not deletions), we classify it as a *structural process* operation, and handle it accordingly.

(c) *Collection-Interacting.* These operations bridge the Java stream and collection domains. For example, `Collection.stream` creates a `Stream` object from a `Collection` instance (hence a *collection creation* operation), and `Stream.toList` collects stream elements into a new `List` instance (hence a *collection output* operation). It is worth noting that *process* operations do not directly interact with collections—hence, there is no corresponding subcategory. The key to handling these operations is to model the transformations between streams and collections, which can be naturally expressed using the existing mechanisms of our container-access pattern. Detail of this handling is provided in Section 7.5.

The complete list of stream operations modeled in our analysis is provided in Appendix B.

7.3 Modeling Stream Pipelines

In this section, we extend the container-access pattern to statically abstract stream pipeline instances and handle *structural* pipeline operations uniformly. This extension enables the identification of input elements (Sources) and output locations (Targets) for each abstract stream instance on-the-fly, allowing the reuse of existing rules to support the suppression of imprecise PFG edges and addition of precise shortcut edges, for achieving our precision goal in analyzing Java streams.

7.3.1 Abstracting Stream Pipelines. We begin by introducing how abstract stream instances are represented in our analysis. Recall from Sections 3.3 and 4.3 that a *host* abstracts container instances (objects of type `Collection` or `Map`), and is defined on $\mathbb{L} \times \mathbb{K}$. Hosts are created per allocation site, such that all dynamic container objects allocated at the same site are mapped to the same (or the same two, for `Map`) host(s). To model streams, we introduce a new host kind Strm into the domain $\mathbb{K}$ to represent abstract stream pipeline instances. However, unlike the container abstraction, stream pipelines are not abstracted per allocation site. This difference in treatment is motivated by the fact that many stream *creation* operations delegate `Stream` object creation to a shared factory method hidden within the JDK library, and statically we will observe only a single allocation site (inside that factory method) for all such creations. Consequently, different *creation* operations—and multiple invocations of the same *creation* operation—appear to return the same abstract heap object, making them indistinguishable.

To more accurately capture the intuition of what constitutes a dynamic stream pipeline, we adopt a finer-grained abstraction: *one Strm host per call-site of a creation operation.* Specifically, the instruction label of each call-site for a *creation* operation is used to uniquely identify a Strm host. Figure 22(a) shows the intermediate three-address code representation of the two stream pipelines in our motivating example. (Three-address code is a commonly used IR for static analysis, and our analysis is also based on it.) In this IR, the two call-sites labeled `l1` and `l2` correspond to separate invocations of `Stream.of`, and are abstracted as distinct Strm hosts: $h^{l_1}_{\text{Strm}}$ and $h^{l_2}_{\text{Strm}}$, respectively.

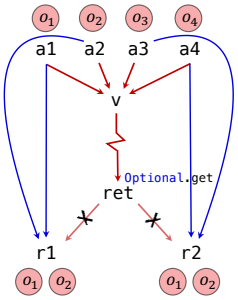
7.3.2 Modeling Structural Operations. With Strm hosts representing abstract stream instances, we now explain how to model structural operations using the extended container-access pattern:

- *Structural Creation.* For stream creation, we introduce rule [S-Create] (Figure 23) and define an input set StruCre, where each pair $\langle m, k\rangle$ specifies a *creation* operation m and its k-th argument as the source of input elements. At each call-site i of such a method, we: (i) create a Strm host h^i_{Strm} and derive $h^i_{\text{Strm}} \in pt_H(\texttt{LHS}^i)$, to associate it with the left-hand-side variable receiving the created `Stream` object, and (ii) record $\texttt{arg}^i_k$ as a Source of this host, denoted $h^i_{\text{Strm}} \Leftarrow \texttt{arg}^i_k$. For example, in Figure 22(a), for lines l_1 and l_2 (highlighted with blue), we have $h^{l_1}_{\text{Strm}} \in pt_H(\texttt{t1})$, with `a1` and `a2` being its Sources. Similarly, $h^{l_2}_{\text{Strm}} \in pt_H(\texttt{t4})$, with `a3` and `a4` being its Sources.
- *Structural Process.* A *structural process* operation m, such as `Stream.filter` or `Stream.findFirst` in Figure 22(a), highlighted in green, is registered in ContProp via $\langle m, \text{Strm}\rangle$, enabling rule

StruCre `Stream t1 = Stream.of(a1, a2);` // line l1, $pt_H(\texttt{t1}) = \{h^{l_1}_{\text{Strm}}\}$, $h^{l_1}_{\text{Strm}} \Leftarrow \texttt{a1/a2}$
StruPro `Stream t2 = t1.filter(lambda);` // $pt_H(\texttt{t2}) = \{h^{l_1}_{\text{Strm}}\}$
StruPro `Optional t3 = t2.findFirst();` // $pt_H(\texttt{t3}) = \{h^{l_1}_{\text{Strm}}\}$
StruOut `Stud r1 = t3.get();` // $h^{l_1}_{\text{Strm}} \Rrightarrow \texttt{r1}$

StruCre `Stream t4 = Stream.of(a3, a4);` // line l2, $pt_H(\texttt{t4}) = \{h^{l_2}_{\text{Strm}}\}$, $h^{l_2}_{\text{Strm}} \Leftarrow \texttt{a3/a4}$
StruPro `Stream t5 = t4.filter(lambda);` // $pt_H(\texttt{t5}) = \{h^{l_2}_{\text{Strm}}\}$
StruPro `Optional t6 = t5.findFirst();` // $pt_H(\texttt{t6}) = \{h^{l_2}_{\text{Strm}}\}$
StruOut `Stud r2 = t6.get();` // $h^{l_2}_{\text{Strm}} \Rrightarrow \texttt{r2}$

(a) IR for the two stream pipelines in motivating example



(b) PFG fragment under CUT-SHORTCUT^S

Fig. 22. Example for handling structural operations.

[HostProp-I] (in Figure 11) to propagate the associated Strm host from the receiver to the LHS. This models stream chaining accurately while retaining host identity. For the first pipeline in Figure 22(a), because $h^{l_1}_{\text{Strm}} \in pt_H(\texttt{t1})$, we derive $h^{l_1}_{\text{Strm}} \in pt_H(\texttt{t2})$ and subsequently derive $h^{l_1}_{\text{Strm}} \in pt_H(\texttt{t3})$. Similarly, for the second pipeline, we derive $h^{l_2}_{\text{Strm}} \in pt_S(\texttt{t5})$ and $h^{l_2}_{\text{Strm}} \in pt_H(\texttt{t6})$.

- *Structural Output.* A *structural output* operation m, like `Optional.get` in Figure 22(a), highlighted in red, is registered in ContExit with $\langle m, \text{Strm}\rangle$, enabling (1) suppression of imprecise return edges by rule [CutCont] (in Figure 11), and (2) identification of Targets for Strm hosts via rule [HostTarget] (in Figure 11). For example, the return variable of `Optional.get` is included in *cutRet* by [CutCont], and imprecise return edges from it to `r1` and `r2` are suppressed by the boxed premise of [Return] in Figure 7, as depicted in Figure 22(b). In addition, we derive $h^{l_1}_{\text{Strm}} \Rrightarrow \texttt{r1}$ and $h^{l_2}_{\text{Strm}} \Rrightarrow \texttt{r2}$ by [HostTarget].

$$\frac{i \rightarrow_{call} m \quad \langle m, k\rangle \in \text{StruCre}}{h^i_{\text{Strm}} \in pt_H(\text{LHS}^i) \quad h^i_{\text{Strm}} \Leftarrow \text{arg}^i_k} \text{ [S-Create]}$$

Fig. 23. The rule for *structural creation* stream operations.

Finally, once both Sources and Targets are identified for each Strm host, rule [ShortcutCont] (in Figure 11) is applied to add shortcut edges that connect sources to targets of the same host (e.g., from `a1`/`a2` to `r1`, and `a3`/`a4` to `r2`), as illustrated in Figure 22(b).

Summary of Extensions. To support precisely abstracting stream pipelines and handling *structural* operations, we extend the container-access pattern in the following ways: (1) we extend the domain of host kinds $\mathbb{K}$ to include a new kind Strm for abstract stream instances; (2) a rule [S-Create] is introduced to generate Strm hosts and locate their Sources; (3) for each *structural process* operation m, we specify $\langle m, \text{Strm}\rangle \in$ ContProp; (4) for each *structural output* operation m, we specify $\langle m, \text{Strm}\rangle \in$ ContExit; and (5) the rule [HostProp-II] in Figure 11 is updated by modifying the negation in its premise to $\neg(a = \texttt{ret}^m \wedge \langle m, \_\rangle \in \text{ContProp} \cup \text{StruCre})$. The modification of [HostProp-II] prevents a Strm host that is generated within a *structural creation* operation from being propagated along the return edge.[13] Additional mechanisms for soundly handling potential unmodeled operations introduced in newer JDK versions are discussed in Appendix D.

[13]This blocking is necessary because many *structural creation* internally delegate stream creation to other such operations. Without this propagation blocking, multiple call-sites of the same *creation* operation (who calls another *structural creation* to create a stream instance) could result in the same Strm host (generated at the inner *structural creation* call-site) propagated out to all outer call-sites, undermining our precision goal.

7.4 Handling Functional Operations

In this section, we describe how to precisely handle *functional* stream operations, i.e., operations that take function-like values (such as lambdas or method references) as arguments. Such handling requires extending the container-access pattern by integrating it with an existing Java lambda analysis [18] to model a variety of operations. We begin by reviewing the necessary background on Java lambdas and method references, illustrated with a motivating stream example. We then outline how the existing lambda analysis models the creation and invocation of Java lambdas (or method references), and demonstrate why its results alone are not precise enough. Finally, we explain how the our stream handling interacts with this lambda analysis to obtain more precise results.

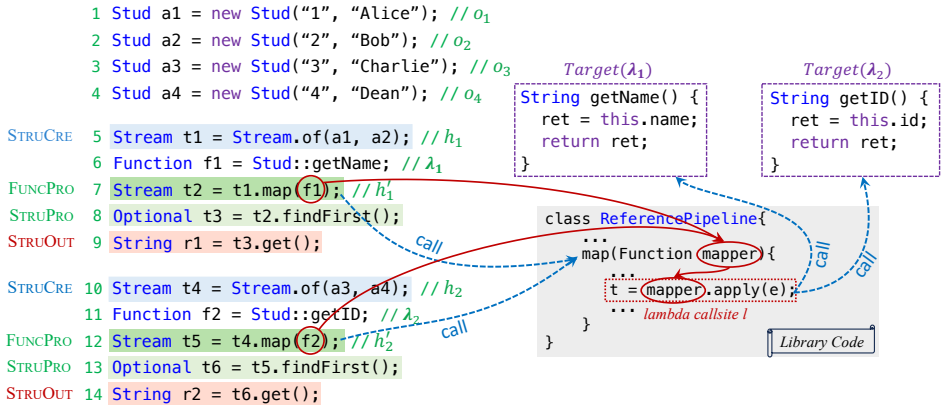


Fig. 24. Example of using *functional* operations in stream pipelines.

7.4.1 Lambdas and Method References. Java 8 introduced lambdas and method references to support functional-style programming, especially in conjunction with stream APIs. *Functional* stream operations (e.g., `Stream.map(Function)`) accept arguments of *functional interface* types—annotated interfaces that declare a special abstract method. These interfaces act as typed "function pointers," allowing users to pass executable behavior as arguments. For example, in Figure 24, lines 7 and 12 call the *functional* operation `Stream.map(Function)`, passing in arguments `f1` and `f2`, both of type `Function<Student,String>`.[14] A "function pointer" in Java is therefore a reference to a "functional object" (i.e., an instance of a functional interface) created using either a lambda expression (e.g., `a -> a.getName()`) or a method reference (e.g., `Stud::getName`). Both forms result in a functional object whose *target method* (also called its *implementation method*) encodes the behavior to execute. In our example, lines 6 and 11 use method references `Stud::getName` and `Stud::getID` to create two functional objects (whose targets are the methods `Stud.getName` and `Stud.getID`, respectively), and then assign the two functional objects to pointers `f1` and `f2`, respectively.

Internally, both lambdas and method references rely on the Java `invokedynamic` framework, using programmable dynamic linking to generate functional objects. Each functional object is associated with (1) a method handle to its *target method*, and (2) any captured environment values, if applicable. The key difference between lambdas and method references lies in how these two pieces of metadata are encoded in their syntax. In our example in Figure 24, only method references are used—chosen for their simplicity, because in this case they do not capture environment values and their target methods are easily identified.

[14]The `Function` type is a standard functional interface provided in JDK, representing functions that take one argument and return a value. (The types of the argument and the return value are specified with generic typing.) Java provides other such interfaces, like `Consumer` (takes one argument, returns nothing) and `BiFunction` (takes two arguments, returns one).

When executing a *functional* stream operation, the function-pointer argument is passed into the internal stream-library code, where it is invoked multiple times—once for each element in the stream. In Figure 24, both `f1` and `f2` are passed to the parameter `mapper`, which is later invoked through the interface method `apply`—the special abstract method defined in the `Function` interface. This call site, labeled l, is thus identified as a *lambda call site*, and its actual callee is dynamically resolved to the *target method* of the functional object referred to by `mapper`. As a result, call site l resolves to `Stud.getName` for the first stream pipeline and to `Stud.getID` for the second. At runtime, the argument `e` at the lambda call site refers to each input element of the stream (i.e., the `Stud` objects), and serves as the receiver of the call (i.e., is bound to `this` in the target method). For instance, in the first pipeline, the lambda call site l is invoked twice—once on o_1 and once on o_2—with `getName()` executed on each object. The return value of each invocation at the lambda call site is assigned to the left-hand-side variable `t`, which is then propagated through the stream pipeline as the mapped element. After execution, `r1` points to `"Alice"` (the result of invoking `getName()` on `a1`), and `r2` points to `"3"` (the result of invoking `getID()` on `a3`).

7.4.2 Lambda Analysis. Standard pointer analysis (with on-the-fly call-graph construction) is insufficient for resolving the dynamic targets of lambda call-sites. Consequently, a dedicated analysis is required to statically model functional call-graph edges. The lambda analysis proposed in [18] provides a comprehensive static modeling of Java's `invokedynamic`-based functional features, including both lambda expressions and method references. In their work, method references are treated as a syntactically constrained subset of lambdas, allowing both constructs to be handled uniformly via a single rule set. For consistency, we refer to this unified approach as the *lambda analysis* throughout the article. This analysis interacts with the core pointer analysis in two key phases: (1) *Functional object creation*—where the analysis models the allocation of functional-interface instances and records key metadata, including its *target method* and any captured environment variables; and (2) *Lambda call-site resolution*—where the analysis resolves invocations on functional objects to their target methods, and models argument passing and return behavior.

$$\frac{\texttt{i:r = b.m(...)} \quad \lambda \in pt(b) \quad \text{Target}(\lambda) = m}{i \rightarrow_\lambda m}\ [\text{L-Call}] \qquad \frac{i \rightarrow_\lambda m}{\texttt{ret}^m \longrightarrow \texttt{LHS}^i}\ [\text{L-Ret}]$$

$$\frac{i \rightarrow_\lambda m \quad m \notin \mathit{Static} \quad \#\text{cap}^\lambda = 0}{\forall k \geq 1 : \texttt{Larg}_k^{i,\lambda} \longrightarrow \texttt{param}_{k-1}^m}\ [\text{L-InsLArg}] \qquad \frac{i \rightarrow_\lambda m \quad \boxed{i \notin \text{StmLCallSite}}}{\forall k \geq 1 : \texttt{arg}_k^i \longrightarrow \texttt{Larg}_k^{i,\lambda}}\ [\text{Arg2LArg}]$$

Fig. 25. Selected core rules of the lambda analysis. The boxed premise in [Arg2LArg] is added when integrated with the our stream handling.

Figure 25 presents the inference rules relevant to our motivating example.[15] To align with our formalism, we rephrase the inference rules from [18] using the notational conventions adopted in this paper. We first introduce several new notations: $\lambda \in \Lambda$ denotes a functional object generated by a lambda or method reference; $\text{Target}(\lambda)$ maps a functional object λ to its target method; and $\#\text{cap}_k^\lambda$ denotes the number of variables captured from environment by λ. These three quantities are predicates derived during the functional-object creation phase (whose rules are omitted). Although our analysis only interacts with the lambda analysis during its invocation phase (i.e., lambda call-site resolution), it is important to emphasize that both analyses are implemented on-the-fly and

[15] For brevity, the rules for functional objects creation are omitted, as they rely on lower-level modeling of `invokedynamic`, which is orthogonal to our analysis. We also omit rules related to captured variables and static target methods, as our illustrative examples focus on instance targets with no captured variables. The handling of dynamic dispatch of instance target methods is also not shown for simplicity. The complete rules of these complex scenarios are presented in Appendix B.

operate in a mutually recursive manner with the pointer-analysis core—there is no predetermined order. In addition, we introduce the notation $\texttt{Larg}_k^{i,\lambda}$ to denote a mock variable that abstracts the k^{th} (k starts from 1) actual argument for a lambda call site i invoked on λ. The use of such mock variables simplifies the presentation of the interaction between the container-access pattern and the lambda analysis.

We now explain how these rules interact on-the-fly with pointer analysis to build the pointer flow graph (PFG) illustrated in Figure 26, based on the code example in Figure 24, and highlight where imprecision arises along the way. The rule [L-Call] constructs call-graph edges for lambda call sites (*lambda-call-edges* in short). Given an invoke instruction of the form `i:r=b.m(...)`, if the base pointer `b` points to a functional object λ whose target method is m, a lambda-call-edge $i \rightarrow_\lambda m$ is created. In our example (Figure 24), the functional objects λ_1 and λ_2 flow to the pointer `mapper` at the lambda call site l (inside the library code). Thus, two lambda-call-edges, $l \rightarrow_{\lambda_1} \texttt{getName}$ and $l \rightarrow_{\lambda_2} \texttt{getID}$, are generated.

The rule [Arg2LArg] specifies how the mock variables $\texttt{Larg}_k^{i,\lambda}$ receive their value flow in the lambda analysis. For now, we can set aside the boxed premise $m \notin \textsc{StmLCallSite}$—this condition is introduced later to integrate the lambda analysis with our stream modeling. In essence, the lambda analysis identifies the k^{th} argument of the lambda call-site i (i.e., $\texttt{arg}_k^i$) as the value source for $\texttt{Larg}_k^{i,\lambda}$ and adds edges in PFG to connect them accordingly. In our example, both $\texttt{Larg}_1^{l,\lambda_1}$ and $\texttt{Larg}_1^{l,\lambda_2}$ receive their values from $\texttt{arg}_1^l$. Because $\texttt{arg}_1^l$ corresponds to the argument `e` at line l, which receives elements from both stream pipelines, the points-to set of `e` contains all objects referenced by pointers `a1`-`a4` (i.e., o_1-o_4). Consequently, these objects merge and flow into both $\texttt{Larg}_1^{l,\lambda_1}$ and $\texttt{Larg}_1^{l,\lambda_2}$, as is shown in the PFG in Figure 26. Note that this merging causes imprecision: despite the lambda call site l being invoked separately on functional objects λ_1 and λ_2 (each corresponding to a distinct stream pipeline), the analysis merges their input elements in the argument pointer `e`.

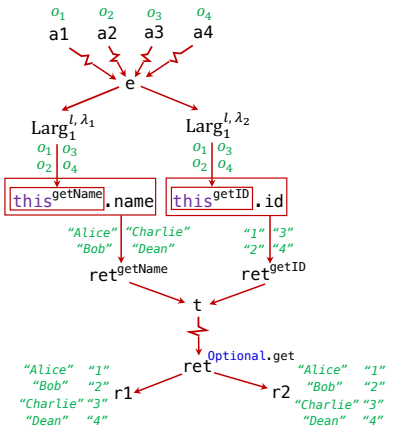


Fig. 26. Selected PFG for the example in Figure 24 (without integration with stream handling).

The rule [L-InsLArg] subsequently propagates mock actual arguments to parameters for each lambda-call-edge $i \rightarrow_\lambda m$, provided that the target method m is an instance method and no variables are captured by λ. Specifically, it adds PFG edges from each actual argument $\texttt{Larg}_k^{i,\lambda}$ to the corresponding formal parameter $\texttt{param}_{k-1}^m$. Note the index shift by -1: this adjustment is necessary to correctly pass the first actual argument to $\texttt{this}^m$ of an instance method m, denoted as $\texttt{param}_0^m$ in the rule. For example, the first argument $\texttt{Larg}_1^{l,\lambda_1}$ of the lambda-call-edge $l \rightarrow_{\lambda_1} \texttt{getName}$ flows to $\texttt{this}^{\texttt{getName}}$ (similarly, $\texttt{Larg}_1^{l,\lambda_2}$ flows to $\texttt{this}^{\texttt{getID}}$), as depicted by the corresponding PFG edges in Figure 26.

Finally, rule [L-Ret] propagates the return value of the target method (i.e., $\texttt{ret}^m$) back to the left-hand-side variable (i.e., $\texttt{LHS}^i$) of the lambda call site. In our example, during pointer analysis $\texttt{ret}^{\texttt{getName}}$ points to the `name` field loaded from each receiver object (four `Stud` instances), and similarly, $\texttt{ret}^{\texttt{getID}}$ points to the corresponding `id` fields. These returned objects flow to the LHS variable `t` at call site l, subsequently propagating to the returned value of the *structural output* operation `Optional.get()`, and eventually reaching both `r1` and `r2`. As a result, the points-to sets of `r1` and `r2` both contain eight string objects (four student names and their four corresponding IDs).

In general, input elements of distinct streams are merged into a single lambda call-site argument within the implementation of *functional* stream operations, resulting in imprecise propagation to the parameters (or `this`) of target methods of functional objects used in different stream pipelines.

7.4.3 Integration with Stream Handling. The central idea of integrating our extended container-access pattern with the lambda analysis is to replace the overly conservative argument flow (originally from lambda-call-site arguments, which merges elements from multiple stream pipelines) by precise shortcut edges derived from the Sources of Strm hosts. Consequently, we add the boxed premise $i \notin$ StmLCallSite to the rule [Arg2LArg] in Figure 25. Here, StmLCallSite denotes the set of lambda call-sites that are part of any *functional* stream operation implementation (e.g., the call site l in Figure 24). [16] By adding this premise, only lambda invocations that are unrelated to any *functional* stream operation will have their mock actual arguments $\texttt{Larg}_k^{i,\lambda}$ populated via the standard flow from the call-site arguments $\texttt{arg}_k^i$—in such cases, no precise source (e.g., identified stream Sources) is available to provide a more precise abstraction. On the other hand, for lambda call-sites involved in *functional* stream operations, we instead introduce explicit shortcut edges to improve precision. As discussed previously (Section 7.2), each *functional* stream operation has distinct semantics regarding how it transforms or consumes stream elements. Therefore, separate inference rules are required for each operation. The full set of rules is provided in Appendix C.

$$\frac{i \rightarrow_{call} m \quad m = \texttt{Stream.map(Function)} \quad \lambda \in pt(\texttt{arg}_1^i) \quad h^j_{\text{Strm}} \in pt_H(\texttt{arg}_0^i) \quad \text{Target}(\lambda) = m' \quad h^i_{\text{Strm}} \in pt_H(\texttt{LHS}^i)}{h^j_{\text{Strm}} \Rrightarrow \texttt{Larg}_1^{*,\lambda} \quad h^i_{\text{Strm}} \Lleftarrow \texttt{ret}^{m'}} \text{ [StmMap]}$$

Fig. 27. Rule for handling `Stream.map`.

The specific rule for handling the `Stream.map` operation is shown in Figure 27. First, we will create hosts of kind Strm for *structural-creation* operations at lines 5 (create h^5_{Strm}, henceforth denoted by h_1 for simplicity) and line 10 (create h^{10}_{Strm}, henceforth denoted by h_2) of Figure 24, and identify h_1's Sources as `a1` and `a2`, and h_2's Sources as `a3` and `a4`. At lines 7 and 12, where `Stream.map` is invoked, the rule [StmMap] applies: first, for each functional object λ referenced by the first argument $\texttt{arg}_1^i$ (e.g., λ_1 for `f1` at line 7, and λ_2 for `f2` at line 12), and for each host h of kind Strm associated with the receiver variable $\texttt{arg}_0^i$ (e.g., h_1 for `t1`, h_2 for `t4`), the first mock actual argument of any lambda invocation invoked on this λ is recorded as a Target of h (e.g., $h_1 \Rrightarrow \texttt{Larg}_1^{l,\lambda_1}$ and $h_2 \Rrightarrow \texttt{Larg}_1^{l,\lambda_2}$). Second, a new Strm host h^i_{Strm} is generated to represent the resulting stream pipeline after the mapping operation. This host is associated with the LHS variable (e.g., h'_1 with `t2` and h'_2 with `t5`), and takes the return value of the lambda's target method as its Source (e.g., $h'_1 \Lleftarrow \texttt{ret}^{\texttt{getName}}$, $h'_2 \Lleftarrow \texttt{ret}^{\texttt{getID}}$).

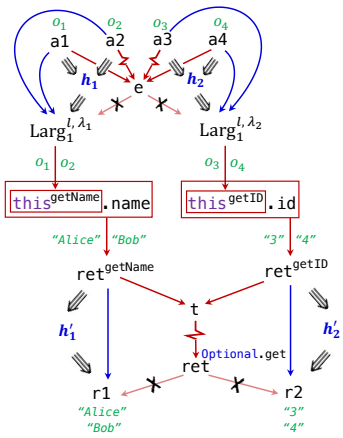


Fig. 28. Fragment of the PFG for the example in Figure 24 (with integration with stream handling).

Figure 28 illustrates the resulting precise Pointer Flow Graph (PFG). The top half of the figure shows precise flows to each target method's `this` variable, achieved via the shortcut edges (blue edges) generated by rule [ShortcutCont] from Figure 11 (which connect Sources and Targets for each host). The bottom half of Figure 28 demonstrates the precise routing of lambda target return values to their corresponding stream outputs (`r1` and

[16] This set is predetermined by the analysis designer via a lightweight local pre-analysis of the stream library code

r2). Ultimately, our approach provides much more precise points-to sets results (r1 points to only "Alice", "Bob", and r2 points only to "3", "4"), compared to the merged, imprecise results shown in Figure 26.

We emphasize that [StmMap] is specifically tailored to the Stream.map operation, reflecting its semantics of transforming a stream by applying a function to each element. Similar customized rules are required for other *functional* stream operations, which are detailed in Appendix B. Additionally, a conservative fallback mechanism for ensuring soundness is discussed in Appendix D.

7.5 Handling Collection-Interacting Operations

In this section, we introduce how the transformations between Java streams and collections can be naturally modeled within our container-access pattern. Figure 29 illustrates an example of such transformation: a HashSet object is first allocated and then transformed into a Stream by invoking Collection.stream. (HashSet implements the Collection interface in Java.) The generated stream pipeline then performs a sorting operation via Stream.sorted, and subsequently stores the sorted elements into a List object. This kind of stream-to-collection interaction accounts for a large portion of real-world stream-usage patterns in Java applications [53]. According to the classification introduced in Section 7.2, the three stream operations in this example correspond to a *collection creation*, a *structural process*, and a *collection output*, respectively, each visually distinguished by subcategory-specific colors.

```
          l1 Set set = new HashSet();   // pt_H(set) = {h^{l1}_Coll}
             ...
CollCre   l2 Stream t1 = set1.stream(); // pt_H(t1) = {h^{l2}_Strm}, h^{l1}_Coll ◁ h^{l2}_Strm
StruPro   l3 Stream t2 = t1.sorted();   // pt_H(t2) = {h^{l2}_Strm}
CollOut   l4 List list = t2.toList();   // pt_H(list) = {h^{l4}_Coll}, h^{l2}_Strm ◁ h^{l4}_Coll
             ...
```

Fig. 29. An example of the interaction between streams and collections.

The inference rules in Figure 30 formalize this interaction. Rule [C-Create] handles *collection creation* operations, which transform a container into a stream pipeline; rule [C-Output] handles the reverse transformation—i.e., converting a stream pipeline back into a container. In [C-Create], when a call site i invokes a method $m \in$ CollCre (the set of all *collection creation* stream operations), we (1) generate a host of kind Strm at line i (i.e., h^i_{Strm}), associating it with the LHS variable by adding it to $pt_H(\text{LHS}^i)$; and (2) identify the host(s) of kind Coll (i.e., h^l_{Coll}) that are associated with the receiver variable arg^i_0, and add a preorder $\lhd$ (recall the semantics of $\lhd$ given by rule [HostDep-I] and [HostDeo-II] in Figure 11) between it and the newly generated Strm host (i.e., $h^l_{\text{Coll}} \lhd h^i_{\text{Strm}}$), indicating that all Sources of the former are also Sources of the later. Conversely, in [C-Output], for a call site i that invokes a method $m \in$ CollOut (the set of all *collection output* stream operations), we (1) generate a new host of kind Coll (i.e., h^i_{Coll}) to abstract the container instance that receives the output elements of the stream pipeline, associating it with LHS^i, and (2) locate the Strm host(s) h^l_{Strm} associated with the receiver variable arg^i_0, and derive $h^l_{\text{Strm}} \lhd h^i_{\text{Coll}}$, indicating the Sources are propagated from the Strm host to the newly generated Coll host.

$$\frac{i \rightarrow_{call} m \quad m \in \text{CollCre} \quad h^l_{\text{Coll}} \in pt_H(\text{arg}^i_0)}{h^i_{\text{Strm}} \in pt_H(\text{LHS}^i) \quad h^l_{\text{Coll}} \lhd h^i_{\text{Strm}}}\ [\text{C-Create}] \qquad \frac{i \rightarrow_{call} m \quad m \in \text{CollOut} \quad h^l_{\text{Strm}} \in pt_H(\text{arg}^i_0)}{h^i_{\text{Coll}} \in pt_H(\text{LHS}^i) \quad h^l_{\text{Strm}} \lhd h^i_{\text{Coll}}}\ [\text{C-Output}]$$

Fig. 30. Rules for handling the interaction between streams and collections.

We now illustrate how the rules are applied to each line for the code snippet in Figure 29. At line l_1, a Coll host is created by [ColHost] in Figure 11, and associated with variable `set`. At line l_2, a *collection creation* stream operation is invoked, resulting in the creation of a Strm host $h^{l_2}_{\text{Strm}}$, which is associated with the LHS variable `t1`. Additionally, $h^{l_1}_{\text{Coll}} \lhd h^{l_2}_{\text{Strm}}$ is derived. Line l_3 corresponds to a *structural process* operation. Rule [HostProp-I] (in Figure 11) is applied to propagate the Strm host from `t1` to `t2`. Finally, line l_4 invokes a *collection output* stream operation. According to [C-Output], a new Coll host is created to abstract the resulting collection instance and is associated with the LHS variable `t4` (i.e., $h^{l_4}_{\text{Coll}} \in pt_H(\texttt{t4})$), with $h^{l_2}_{\text{Strm}} \lhd h^{l_4}_{\text{Coll}}$.

Transformations through Collectors. Among all stream operations, one API that requires special attention is `Stream.collect(Collector)`, which performs a reduction using a `Collector` instance. The JDK class `java.util.stream.Collectors` provides a variety of factory methods to generate default collectors for different purposes—for example, `Collectors.toSet()` produces a collector that accumulates stream elements into a `Set`, while `Collectors.groupingBy(Function)` generates a collector that groups stream elements by a classification function and returns the result in a `Map`. In our analysis, we model such transformations precisely only when the collector is constructed via `Collectors.toSet`, `Collectors.toList`, or `Collectors.toCollection`. These collectors are widely used and have straightforward semantics—namely, accumulating all stream elements into a newly generated `Collection` instance—which aligns naturally with our handling in [C-OutPut] (by specifying `Stream.collect` in CollOut). In contrast, more complex collectors such as `Collectors.toMap`, `Collectors.groupingBy` or custom collectors exhibit dynamic behaviors that are difficult to reason about statically. Furthermore, empirical studies of stream-usage patterns [35, 62, 63] indicate that these complex collectors account for fewer than 10% of all collector usages in real-world programs. To balance soundness and practicality, we handle these cases conservatively, with detailed rules deferred to Appendix D.

8 Evaluation

In this section, we investigate the following research questions for evaluation.

RQ1. How does Cut-Shortcut compare to context-insensitive pointer analysis?
RQ2. How does Cut-Shortcut compare to conventional context-sensitive pointer analysis?
RQ3. How does Cut-Shortcut compare to state-of-the-art selective context-sensitive pointer analysis techniques that have similar goal (i.e., high efficiency with good precision)?
RQ4. How does each pattern of Cut-Shortcut affect the precision and efficiency?
RQ5. Is Cut-Shortcut^S effective in analyzing programs with heavy stream usage?

Implementation. We implemented Cut-Shortcut and Cut-Shortcut^S on top of the state-of-the-art Java static analysis framework Tai-e [75] (version 0.5.1, the latest stable release at the time this article was written). Tai-e is an imperative, highly extensible framework equipped with a powerful pointer analysis system that achieves high efficiency, precision, and soundness. [17] The core implementation of Cut-Shortcut is about 2,000 lines of Java code. We release and will maintain

[17] In our earlier conference paper [48], we also implemented the previous version of Cut-Shortcut on another pointer analysis framework, Doop [7] and reported experimental results on it. However, we no longer maintain that version and have not ported the new features (the optimizations in Cut-Shortcut and the Cut-Shortcut^S extension) to it, because: (1) Doop uses a declarative Datalog-based solver that is significantly slower than Tai-e, which makes compared analyses far less scalable—for example, 2obj exhausting 128GB of memory or failing to terminate within 2 hours across all 10 benchmarks; and (2) its declarative nature makes extensive extensions difficult to debug. For these reasons, in this article we focus our evaluation on Tai-e, which provides a more scalable and efficient foundation, and thereby enables fair comparisons among Cut-Shortcut, Cut-Shortcut^S, and state-of-the-art (selective) context-sensitive analyses.

Table 1. Program sizes of benchmark programs, measured by the number of classes, methods, variable pointers, call sites, and statements in the intermediate representation (IR) after whole-program world building in the Tai-e analysis framework.

Program	Application (without JDK dependencies)					Whole-Program (with JDK dependencies)				
	#class	#method	#var-pointer	#callsite	#statement	#class	#method	#var-pointer	#callsite	#statement
soot	3,419	34,294	245,506	170,763	447,826	8,932	82,479	519,049	346,422	1,206,198
freecol	1,703	13,737	116,663	85,610	270,173	10,156	89,844	592,183	385,803	1,618,682
briss	1,728	15,315	137,974	92,441	408,859	8,649	78,970	518,207	343,183	1,525,911
gruntspud	1,460	9,540	72,440	50,702	148,100	9,233	77,584	487,043	319,432	1,345,960
columba	4,777	38,499	238,449	162,386	523,377	12,388	106,410	650,498	430,980	1,709,493
jython	1,185	10,820	71,886	43,452	163,756	6,805	59,653	349,729	221,220	931,206
jedit	453	3,508	27,483	20,566	74,207	6,852	62,061	366,253	264,397	1,067,931
eclipse	2,529	25,145	199,838	130,898	518,812	8,086	73,400	474,402	306,804	1,279,765
hsqldb	492	5,696	53,757	37,843	124,664	6,159	55,794	339,936	222,872	911,043
findbugs	1,767	12,225	79,684	55,658	167,365	8,290	69,169	422,000	270,284	1,093,506

the source code at [86]. All experiments were run on an Intel i7-13700K machine with 128 GB of running memory.

We use the same experimental settings for RQ1–RQ4, introduced below. For RQ5, different compared analyses, benchmarks, and precision metrics are employed, which are described in Section 8.5.

Compared Analyses. To investigate RQ1-RQ4, we compare Cut-Shortcut to three types of pointer analyses: context insensitivity (CI), mainstream context sensitivity, and state-of-the-art selective context sensitivity. CI is the de facto fastest pointer analysis for Java. For conventional context sensitivity, we select the commonly-used 2-object-sensitive (2obj) [49, 50] and 2-type-sensitive (2type) [69] analyses. 2obj is highly-precise, and 2type often achieves much better scalability than 2obj while yielding comparable precision. For selective context-sensitivity, we consider a state-of-the-art approach Zipper^e [41], which exhibits a better efficient and precision trade-off than many other selective approaches. We use Zipper^e-2obj and Zipper^e-2type to denote the selective 2-object-sensitive and 2-type-sensitive analyses guided by Zipper^e. Zipper [39]—often evaluated as state-of-the-art in recent literature, with comparable precision to Zipper^e but much less efficient—is not considered here because it fails to scale to most of the benchmark programs used in our experiments.

Benchmarks. To investigate RQ1-RQ4, we consider all the large and complex Java programs used for evaluation in recent related literature [28, 29, 32, 39–41, 46, 76], except for those that cannot be executed—because we cannot perform recall experiments on them for evaluating soundness. We analyze the programs with a large Java library, JDK1.6, which is commonly used in recent work [22, 23, 30, 31, 46]. In Table 1, we report program sizes in terms of key program elements. Specifically, we record the number of classes, methods, variable pointers, call sites, and total statements. Here, a "statement" refers to an instruction in the intermediate representation (IR) used by the Tai-e framework. All counts are obtained from the IR that the analysis framework builds for the entire program, rather than from the source code, to more accurately reflect the relationship between number of program elements and analysis performance. Table 1 reports both the size of the application part (excluding JDK dependencies but including other external libraries) and the size of the whole program (including the relevant portions of the JDK on which the application depends). All analyses are performed in a whole-program setting, where the pointer analysis fully analyzes the relevant JDK portions.

Precision Metrics. To measure precision for RQ1-RQ4, we use the total number of objects pointed to by program variables, i.e, $\Sigma_v|pt(v)|$ (abbreviated as #var-pts), along with four independently useful clients that are widely-used as precision metrics in the pointer-analysis literature [28, 30, 32, 39–41, 46, 69, 70, 78]: a cast-resolution analysis (metric: the number of casts that may fail—#fail-cast), a method-reachability analysis (metric: the number of reachable methods—#reach-mtd), a devirtualization analysis (metric: the number of virtual call sites that cannot be disambiguated into monomorphic calls—#poly-call) and a call-graph-building analysis (metric: the number of call graph edges—#call-edge). In the rest of this section, we report all average speed-ups and average percent precision improvement using geometric means, as recommended by [17], because geometric means provide a more reliable summary of normalized numbers and help avoid misleading conclusions.

8.1 RQ1: Cut-Shortcut vs. Context Insensitivity (CI)

In this section, we examine how Cut-Shortcut fares against the fastest pointer analysis, CI. But in order to trust the reliability of Cut-Shortcut's efficiency benefit, we must first confirm its soundness. In addition to the theoretical soundness proofs in Section 5.2, below we further validate its soundness by a recall experiment.

Soundness (Recall). We execute all the evaluated programs with their default tests (e.g., for DaCapo benchmarks [6]) or the ones we input (e.g., for GUI programs, we click to interact with them and the results are recorded and will be provided in our artifact), then dynamically record their reachable methods and call-graph edges during execution, and examine how many of them can be recalled (over-approximated) by Cut-Shortcut and other analyses in our evaluation. (It is hard to instrument programs to gather other dynamic information, such as dynamic points-to relations.) The results show that Cut-Shortcut can recall all true reachable methods and call-graph edges discovered by other theoretically sound analyses [18]. Thus, in addition to the theoretical-soundness proof, we can trust that the remaining efficiency and precision results are reliable. The detailed results from the recall experiment are provided in Appendix E.

Efficiency. Figure 31 shows graphically the elapsed time of all analyses across the 10 benchmarks, and Table 2 presents the efficiency and precision results in full detail. As expected, CI is substantially faster than (selective) context-sensitive analyses. However, Cut-Shortcut achieves even better performance: it is faster than CI on 7 out of 10 benchmarks, and as fast as CI on the remaining 3. Cut-Shortcut's speed advantage is more pronounced as program size increases. For example, on the largest benchmark `columba` (which has the highest number of program elements in whole-program setting, as reported in Table 1), Cut-Shortcut outperforms CI by more than 20 seconds (26% faster). On relatively smaller benchmarks (`hsqldb`, `jython`, `findbugs`), Cut-Shortcut performs on par with CI. Overall, Cut-Shortcut is on average 1.18× faster than CI across 10 benchmarks.

The speed of Cut-Shortcut arises from both its methodology and its precision improvements. First, unlike context sensitivity—which improves precision by cloning methods and heap objects under different contexts—Cut-Shortcut enhances precision by suppressing imprecise PFG edges and adding precise ones, which generally introduces little overhead (see Section 6.2 for a formal discussion). Second, in contrast to existing selective context-sensitivity approaches, Cut-Shortcut requires no pre-analysis phase (a more detailed comparison is provided in Section 8.3); instead, it runs the context-insensitive pointer analysis algorithm on an on-the-fly constructed PFG. Third,

[18] For a pointer analysis to be practical on real-world Java programs, it must handle dynamic features. However, some of these dynamic features, like reflection, cannot be soundly analyzed in general at static time [42], and therefore no existing pointer analysis is fully sound in their presence. The goal of our recall experiment is to demonstrate that the implementation of Cut-Shortcut does not introduce *additional* unsoundness: the set of dynamically recorded reachable methods and call-graph edges recalled by other compared analyses can also be recalled by Cut-Shortcut.

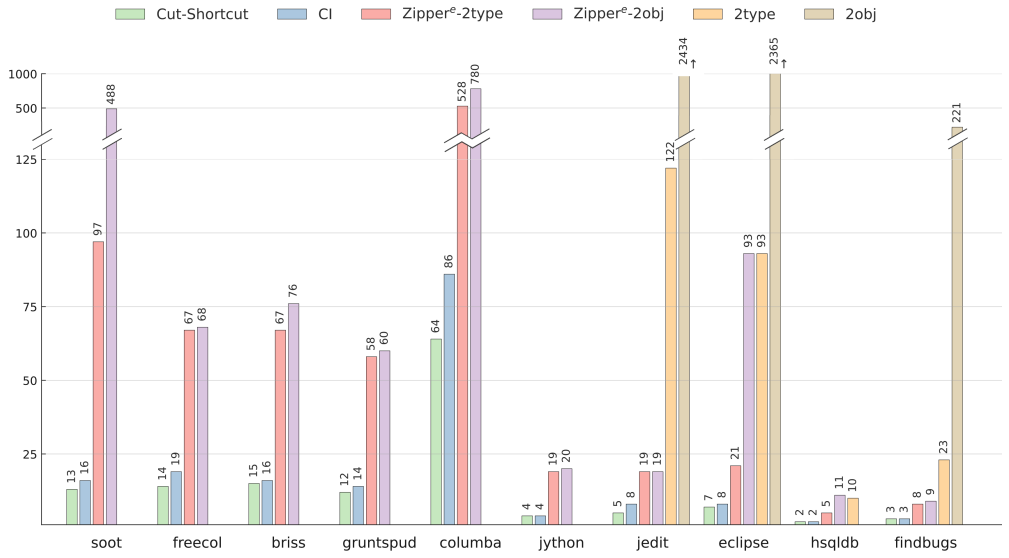


Fig. 31. Analysis time (in seconds) of Cut-Shortcut, context-insensitive (CI), Zipper^e-guided selective 2-type and 2-object sensitive, and conventional context-sensitive (2type and 2obj) pointer analyses. Cut-Shortcut (green bars) is consistently as fast as or faster than all compared analyses, including CI, across all benchmarks.

Cut-Shortcut prevents a substantial amount of spurious points-to information from propagating across the program. Because the cost of suppressing imprecise edges and adding precise ones is negligible compared to the efficiency gains from improved precision, Cut-Shortcut can in practice outperform even CI.

Precision. As shown in Table 2, Cut-Shortcut consistently achieves better precision than CI across all benchmarks and all precision metrics, with particularly notable improvements for #fail-cast, #call-edge, and #var-pts. These gains arise from guiding PFG construction using the proposed patterns, demonstrating that the three patterns underlying Cut-Shortcut are effective in improving precision [19].

8.2 RQ2: Cut-Shortcut vs. Conventional Context Sensitivity

In this section, we investigate how Cut-Shortcut performs compared to mainstream conventional context-sensitive pointer analyses, namely 2obj and 2type.

Efficiency. As shown in Table 2, 2obj runs out of 128GB of memory for 7 out of 10 programs. 2type performs slightly better, but still runs out of memory on 6 programs. In contrast, Cut-Shortcut successfully completes all benchmarks. On the programs on which 2obj and 2type do succeed, Cut-Shortcut is remarkably faster: for the three program on which 2obj succeeds, Cut-Shortcut is 229.7× faster, while Cut-Shortcut is 10.6× on the 4 programs on which 2type succeeds. In summary, Cut-Shortcut provides dramatically better scalability and efficiency than conventional context sensitivity.

[19] Relative to the results reported earlier in [48]: (1) we now include results for an additional compared analysis, Zipper^e-2type; (2) all analyses show improved efficiency on the updated Tai-e framework, due to its recent performance enhancements; and (3) with the refinements we introduced to the field-access and container-access patterns, Cut-Shortcut is now more efficient, albeit with slightly reduced precision on some metrics. We find this trade-off worthwhile—our refinements make both patterns more principled—and therefore make Cut-Shortcut easier to adopt, extend, and maintain in practice.

Table 2. Efficiency and precision results of context-insensitive (CI), conventional context-sensitive (2obj and 2type), Zipper^e-guided selective 2-object and 2-type sensitive analyses, and Cut-Shortcut across 10 benchmarks. For all precision metrics, smaller values indicate better precision. "OOM" indicates that the analysis terminated with an out-of-memory error (unscalable under our experimental settings).

Program	Analysis	Time(s)	#fail-cast	#reach-mtd	#poly-call	#call-edge	#var-pts
soot	CI	16	16,640	32,913	16,385	415,710	84,491,342
	2obj	OOM	—	—	—	—	—
	2type	OOM	—	—	—	—	—
	Zipper^e-2obj	488	9,392	32,165	13,837	281,289	14,701,551
	Zipper^e-2type	97	11,683	32,275	14,724	353,082	23,224,281
	Cut-Shortcut	13	10,321	32,659	15,188	328,689	30,248,396
freecol	CI	19	9,739	46,915	15,489	322,837	71,774,368
	2obj	OOM	—	—	—	—	—
	2type	OOM	—	—	—	—	—
	Zipper^e-2obj	68	6,885	46,058	13,501	277,883	15,650,971
	Zipper^e-2type	67	8,269	46,282	13,735	281,618	19,079,170
	Cut-Shortcut	14	6,434	46,448	14,210	293,814	21,925,124
briss	CI	16	7,788	41,853	12,763	294,125	65,924,173
	2obj	OOM	—	—	—	—	—
	2type	OOM	—	—	—	—	—
	Zipper^e-2obj	76	5,998	41,426	11,579	260,792	23,098,275
	Zipper^e-2type	67	6,752	41,468	11,709	263,373	25,024,647
	Cut-Shortcut	15	5,531	41,515	11,931	265,592	27,606,506
gruntspud	CI	14	6,749	39,863	12,348	274,755	49,608,582
	2obj	OOM	—	—	—	—	—
	2type	OOM	—	—	—	—	—
	Zipper^e-2obj	60	5,239	39,200	10,981	229,902	16,225,651
	Zipper^e-2type	58	5,774	39,266	11,082	233,617	17,836,307
	Cut-Shortcut	12	4,759	39,465	11,699	230,641	18,822,052
columba	CI	86	10,453	56,737	17,680	426,134	168,420,335
	2obj	OOM	—	—	—	—	—
	2type	OOM	—	—	—	—	—
	Zipper^e-2obj	780	8,070	55,809	15,897	341,394	42,395,433
	Zipper^e-2type	528	8,904	55,970	16,205	351,855	48,735,371
	Cut-Shortcut	64	7,416	56,151	16,724	363,149	60,650,004
jython	CI	4	2,402	13,086	2,952	121,489	12,107,477
	2obj	OOM	—	—	—	—	—
	2type	OOM	—	—	—	—	—
	Zipper^e-2obj	20	1,851	12,612	2,598	111,632	8,093,191
	Zipper^e-2type	19	1,987	12,648	2,642	111,964	8,115,225
	Cut-Shortcut	4	1,903	12,956	2,813	117,943	9,731,365
jedit	CI	8	4,090	25,318	6,358	150,742	25,849,267
	2obj	2,434	2,566	24,229	4,944	122,037	1,645,511
	2type	122	3,083	24,278	5,108	122,889	1,939,458
	Zipper^e-2obj	19	2,910	24,331	5,207	123,645	2,420,703
	Zipper^e-2type	19	3,272	24,395	5,288	124,632	2,656,387
	Cut-Shortcut	5	2,851	24,691	5,869	133,691	5,735,162
eclipse	CI	8	5,078	23,973	10,664	184,875	25,095,353
	2obj	2,365	3,621	23,212	9,734	164,462	2,626,243
	2type	93	4,181	23,355	9,920	166,097	3,102,358
	Zipper^e-2obj	93	4,056	23,593	9,948	170,952	12,066,219
	Zipper^e-2type	21	4,442	23,623	10,108	173,328	12,859,903
	Cut-Shortcut	7	3,915	23,783	10,329	175,241	13,545,299
hsqldb	CI	2	1,653	11,586	1,830	64,398	1,850,474
	2obj	OOM	—	—	—	—	—
	2type	10	1,066	11,171	1,480	56,624	489,923
	Zipper^e-2obj	11	927	11,215	1,511	56,930	729,197
	Zipper^e-2type	5	1,100	11,258	1,100	57,475	769,173
	Cut-Shortcut	2	1,151	11,429	1,689	60,695	1,065,182
findbugs	CI	3	3,374	17,352	4,480	107,391	7,326,707
	2obj	221	1,962	16,821	3,578	89,003	886,778
	2type	23	2,338	16,874	3,770	90,073	994,799
	Zipper^e-2obj	9	2,428	17,093	3,932	95,278	2,592,066
	Zipper^e-2type	8	2,672	17,115	4,102	96,048	2,720,052
	Cut-Shortcut	3	2,112	17,182	4,226	97,461	2,123,513

Precision. When 2obj succeeds (on 3 out of 10 programs), it achieves the highest precision across all metrics, outperforming Cut-Shortcut and all other analyses. When 2type succeeds (on 4 out of 10 programs), it is generally more precise than Cut-Shortcut, however, it performs worse on the #fail-cast metric for 3 of the 4 programs. For #var-pts, 2obj and 2type typically yield smaller sets than Cut-Shortcut, reflecting their finer-grained context distinctions. On average for the precision improvement over CI on #var-pts, Cut-Shortcut retains 70.1% of 2obj's improvement (over the 3 programs on which 2obj succeeds) and 67.7% of 2type's improvement (over the 4 programs on which 2type succeeds). These results highlight the high precision of conventional context-sensitive analyses; however, their poor scalability makes them impractical as a general solution for large programs.

8.3 RQ3: Cut-Shortcut vs. State-of-the-Art selective Context Sensitivity

In this section, we compare Cut-Shortcut to Zipper^e [41], a state-of-the-art selective context sensitivity approach. We use the default configuration of Zipper^e as in [41], which is well-tuned and achieves a very good trade-off between efficiency and precision. We compare Cut-Shortcut to 2 variants, Zipper^e-guided 2-object-sensitive (Zipper^e-2obj) and Zipper^e-guided 2-type sensitive (Zipper^e-2type) analyses.

Efficiency. A Zipper^e-guided analysis consists of three stages: (1) a context-insensitive pre-analysis (a standard CI analysis, necessary to provide information for Zipper^e 's method-selection strategy), (2) running Zipper^e itself (to select a set of methods according to its strategy), and (3) the main analysis (a context-sensitive analysis applied only to the selected methods). Thus, the elapsed time of a Zipper^e-guided analysis is the sum of all three stages. As shown in Table 2, Zipper^e is substantially more scalable (completing on all benchmarks) and faster than conventional 2obj and 2type analyses. Nonetheless, Cut-Shortcut remains more efficient, achieving an average speedup of 6.9× over Zipper^e-2obj and 4.3× over Zipper^e-2type. To compare Zipper^e and Cut-Shortcut thoroughly, Table 3 breaks down the elapsed time of the Zipper^e-guided analyses. For Zipper^e, the "Total time" column lists the total analysis time (as reported in Figure 31 and Table 2); "Pre-analysis" shows the combined time of the CI pre-analysis and the method-selection phase (accounts for 44.0% of the total time of Zipper^e-2obj and 70.0% for Zipper^e-2type, on average); and "Main analysis" reports the time of Zipper^e-guided context-sensitive analysis.

Precision. Overall, Cut-Shortcut is comparable in precision to Zipper^e-2obj and Zipper^e-2type. For #var-pts, Cut-Shortcut achieves on average 83.7% of the precision improvement of Zipper^e-2obj (over CI) and 87.8% of that of Zipper^e-2type (over CI). Regarding the four client metrics, Cut-Shortcut and Zipper^e exhibit different strengths: for #fail-cast, Cut-Shortcut outperforms Zipper^e-2obj on all programs except `hsqldb`, `soot`, and `jython`, and outperforms Zipper^e-2type on all programs except `hsqldb`. For the remaining metrics, the Zipper^e-guided analyses generally outperform Cut-Shortcut across most benchmarks, while the results of Cut-Shortcut remain competitive—for instance, Cut-Shortcut outperforms both Zipper^e-2obj and Zipper^e-2type on #var-pts for `findbugs`, and outperforms Zipper^e-2type on #call-edge for `soot` and `gruntspud`. We anticipate that incorporating additional program patterns following the principle used in Cut-Shortcut could further enhance its precision.

Although Zipper^e and Cut-Shortcut improve precision in fundamentally different ways, a natural question is whether the methods selected by Zipper^e overlap with those involved in Cut-Shortcut 's PFG edge suppression and addition. Table 3 provides a detailed answer. On average, Cut-Shortcut considers 5,281 methods (i.e., methods involved in PFG edge suppression and addition), accounting for 17.2% of all reachable methods per program, while Zipper^e selects 3,770 methods. Among them, on average only 34.5% of the methods identified by Cut-Shortcut overlap

Table 3. Detailed comparison between Zipper^e-guided selective context sensitivity and Cut-Shortcut.

Program	Cut-Shortcut		Selective Context-sensitivity					Overlapped methods
	Time	#involved-methods	Analysis	Pre-analysis	Main-analysis	Total time	#selected-methods	
soot	13	5,585	Zipper^e-2obj Zipper^e-2type	23	465 74	488 97	12,102	71.91%
freecol	14	7,818	Zipper^e-2obj Zipper^e-2type	55	13 12	68 67	3,845	27.31%
briss	15	6,891	Zipper^e-2obj Zipper^e-2type	49	27 18	76 67	3,211	26.43%
gruntspud	12	6,922	Zipper^e-2obj Zipper^e-2type	47	13 11	60 58	3,503	26.77%
columba	64	9,415	Zipper^e-2obj Zipper^e-2type	462	318 66	780 528	4,069	21.19%
jython	4	2,270	Zipper^e-2obj Zipper^e-2type	17	3 2	20 19	1,611	35.29%
jedit	5	5,095	Zipper^e-2obj Zipper^e-2type	16	3 3	19 19	2,278	26.09%
eclipse	7	4,160	Zipper^e-2obj Zipper^e-2type	13	80 8	93 21	3,404	46.54%
hsqldb	2	1,754	Zipper^e-2obj Zipper^e-2type	4	7 1	11 5	1,491	47.43%
findbugs	3	2,902	Zipper^e-2obj Zipper^e-2type	6	3 2	9 8	2,182	40.80%

with those selected by Zipper^e (see column "Overlapped methods"). This limited overlap highlights the orthogonality of the two approaches, and suggests that there may be opportunities for future schemes that combine their strengths.

8.4 RQ4: Effect of Each Pattern of Cut-Shortcut

We instantiated the Cut-Shortcut philosophy via three patterns, so it is natural to ask which pattern contributes most to the precision of the algorithm. To answer this question, we conducted experiments running Cut-Shortcut with different combinations of patterns to isolate their impact.

Precision. We use CI as a baseline (i.e., no Cut-Shortcut pattern enabled). We measure the impact of each pattern by computing the precision improvement over CI for different pattern combinations, and normalize the results with respect to the overall improvement achieved by enabling all three patterns together. Figure 32 shows the results for the metric #var-pts. On average, using only the field-access pattern achieves an improvement of 22.7%, while combining the field-access and container-access patterns yields 95.8% improvement (with all three patterns together achieving 100%). For the other four metrics, the relative contributions vary. For example, on average, the field-access pattern; the field-access and container-access combination; and all three patterns together improve precision by 70.5%, 95.1%, and 100%, respectively, for #reach-mtd; and by 11.4%, 88.3%, and 100%, respectively, for #fail-cast.

Efficiency. Table 4 reports the elapsed time of Cut-Shortcut under different combinations of the three patterns. Including CI (which corresponds to enabling no patterns), the overall trend is that enabling more patterns makes the analysis run faster. With only the field-access pattern enabled, Cut-Shortcut runs faster than CI on 5 out of 10 programs, as fast as CI on 3 programs,

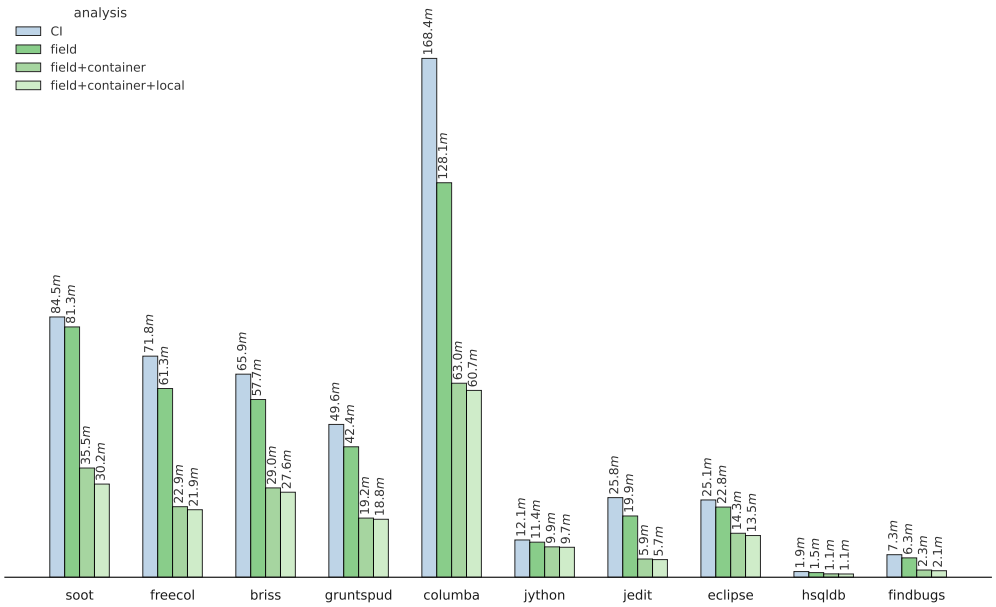


Fig. 32. #var-pts (in millions) for different combinations of the three patterns (the field-access pattern, abbreviated as "field"; the container-access pattern, abbreviated as "container"; and the local-flow pattern, abbreviated as "local") of Cut-Shortcut. Small value indicate better precision.

Table 4. The corresponding elapsed time (in seconds) of the analyses in Figure 32.

	soot	freecol	briss	gruntspud	columba	jython	jedit	eclipse	hsqldb	findbugs
CI	16.5	18.9	15.8	13.9	86.3	4.4	8.0	7.8	1.9	3.0
Field	15.2	18.0	16.8	14.2	67.6	4.4	7.3	7.7	1.9	3.0
Field+Container	15.4	15.4	16.4	13.5	72.4	4.7	5.6	7.5	1.9	2.9
Field+Container+Local	13.3	13.6	15.3	12.3	64.3	4.4	5.5	7.2	1.9	2.7

and slightly slower on 2 programs. Adding the container-access pattern further reduces analysis time (or leave it unchanged) on 8 out of 10 programs, while the remaining 2 are slightly slower than with field-access alone. Finally, enabling all three patterns (field-access, container-access, and local-flow) further reduces analysis time on 9 out of 10 programs, with the last one unchanged compared to field + container. While one might expect that enabling more patterns—thus adding more analysis logic—would slow the analysis down, in practice, the opposite occurs: each additional pattern filters out more spurious points-to information, reducing the propagation of points-to facts during pointer analysis. The overhead of identifying suppressed and added PFG edges is negligible compared to the performance gain from reduced propagation. These results also suggest that incorporating additional patterns in the future could further improve efficiency.

8.5 RQ5: Effectiveness of Cut-Shortcut^S

Cut-Shortcut^S extends Cut-Shortcut to streams by statically abstracting pipelines, tracking their input elements and output locations, and modeling stream–lambda interactions on the fly, thereby improving precision without relying on context sensitivity. To evaluate Cut-Shortcut^S comprehensively—not only on isolated operations but also in realistic usage—we constructed three

complementary benchmark suites (20 programs in total). These suites, released in our artifact, balance *operation coverage and controllability*, *representativeness*, and *realism*:

(a) *Synthetic* (10 programs): micro-benchmarks that systematically cover combinations of stream operations on small inputs, each designed to isolate and highlight a specific aspect of stream behavior, with high controllability and enabling easy manual verification of analysis results.

(b) *Workflow-mimicking* (5 programs): enterprise-backend-inspired benchmarks built with hand-crafted in-memory mock databases. This suite adapts and extends the BSS benchmark [82], originally generated by the S2S tool [64], which translates SQL queries into Java stream code. The resulting programs exercise end-to-end data-processing pipelines (e.g., filter–map–aggregate, PO/VO conversions), thereby bridging the gap between toy micro-benchmarks and production code while avoiding external dependencies. We introduce this suite because real enterprise backends rely heavily on streams for such data-processing and transmission tasks. However, analyzing real enterprise backends would additionally require modeling database queries, framework-specific behaviors (e.g., Spring DI & AOP), and other challenges that demand specialized solutions in pointer analysis and are orthogonal to our contribution. Thus, although current pointer analysis techniques do not yet fully address the complexities of enterprise applications, this benchmark suite serves as a proxy: it demonstrates the degree to which our stream-handling technique can improve precision once those obstacles are addressed, offering a forward-looking perspective on its applicability to real-world enterprise software.

(c) *Real-world* (5 programs): open-source Java projects with intensive stream usage, providing realistic code bases outside the enterprise domain.

Together, the three suites strike a balance: the synthetic suite provides broad operation coverage with high controllability, the workflow-mimicking suite captures realistic backend-style usage, and the real-world suite validates effectiveness on existing projects beyond the enterprise domain. Detailed descriptions of each suite appear in the following subsections. Each of these benchmarks is analyzed with a large Java library, JDK 1.8.0_312.

Precision Metrics. To measure precision while separating the parts of a program affected by stream computations from the rest, we introduce two new metrics tailored for RQ5: (1) The total number of objects pointed to by variables in application code ($\Sigma_{v \notin \text{JDK}} |pt(v)|$), denoted by #app-vpt. We use this metric instead of #var-pts to focus on the application-level portion of the program, avoiding dilution from pointers in the Java 8 library, which account for a large share of #var-pts. (2) The total number of objects pointed to by stream-output pointers, denoted by #stm-out. This metric includes both the outputs of stream pipelines (i.e., the Target pointers of h_{Strm}) and the outputs of collections directly derived from stream pipelines.

Compared Analyses. We compare Cut-Shortcut$^{\mathcal{S}}$ (CSC-S) with four baselines: context insensitivity (CI), conventional context sensitivity, selective context sensitivity, and Cut-Shortcut without the stream extension (CSC). CI serves as a highly efficient precision baseline. For conventional context sensitivity, we use 1-object sensitivity (1obj), because 2obj does not scale (running out of 128GB memory) even on toy Java 8 programs due to the complexity of the JDK 8 library. For selective context sensitivity, we use selective 4-object sensitivity with a 3-layer context-sensitive heap abstraction targeting stream-related libraries (JDK collections, maps, streams, and the `Optional` class), denoted by stm-4obj3h. We fix the context depth at 4obj3h, because deeper settings do not yield additional precision improvements for stream-output pointers. We do not report the results for selective call-site sensitivity, because under the maximum depth that such an analysis algorithm can succeed in our experimental setting, it performs worse than stm-4obj3h in both run time and precision across all suites.

Table 5. Efficiency and precision results of context-insensitive analysis (CI), conventional context sensitivity (1obj), selective 4-object sensitivity with 3 context-sensitive heap abstractions on stream-related library (stm-4obj3h), Cut-Shortcut (CSC), and Cut-Shortcut^S (CSC-S) across 10 synthetic micro-benchmarks.

Micro-benchmark	Dyn #stm-out	#pipeline	Analysis	Time(s)	#app-vpt	#stm-out	Micro-benchmark	Dyn #stm-out	#pipeline	Analysis	Time(s)	#app-vpt	#stm-out
simpleStruc	6	6	CI	2	13,905	46	stmReduce	34	8	CI	1	2,987	216
			1obj	4	9,014	46				1obj	4	1,752	216
			stm-4obj3h	24	5,550	46				stm-4obj3h	13	1,979	216
			CSC	2	8,716	46				CSC	1	937	216
			CSC-S	2	3,112	22				CSC-S	1	339	52
buildStm	6	10	CI	1	8,951	66	stmCollect	30	4	CI	1	8,876	128
			1obj	4	257	66				1obj	4	7,236	128
			stm-4obj3h	14	3,263	66				stm-4obj3h	13	434	108
			CSC	1	503	66				CSC	1	512	108
			CSC-S	1	103	13				CSC-S	1	230	32
simpleFunc	12	6	CI	2	10,527	92	stmCollInter	10	6	CI	1	8,656	84
			1obj	4	7,270	92				1obj	4	6,953	84
			stm-4obj3h	20	4,065	92				stm-4obj3h	16	1,956	84
			CSC	2	7,027	92				CSC	1	3,931	60
			CSC-S	2	3,163	18				CSC-S	1	235	42
stmMap	16	7	CI	1	2,707	119	primStm	4	4	CI	1	1,375	8
			1obj	4	502	119				1obj	4	99	8
			stm-4obj3h	13	2,511	119				stm-4obj3h	15	1,263	8
			CSC	1	766	119				CSC	1	263	8
			CSC-S	1	159	18				CSC-S	1	54	6
stmFlatMap	20	4	CI	2	1,700	116	conservStm	9	5	CI	2	11,729	107
			1obj	4	1,454	112				1obj	4	5,192	89
			stm-4obj3h	25	752	96				stm-4obj3h	14	971	102
			CSC	2	684	94				CSC	2	3,308	106
			CSC-S	2	532	22				CSC-S	2	2,236	106

8.5.1 Synthetic Micro-Benchmarks. To comprehensively evaluate Cut-Shortcut^S on stream APIs, we constructed a suite of 10 synthetic micro-benchmarks. Each benchmark targets specific aspects of stream usage, covering a wide range of operations and their combinations. For example, `buildStm` focuses on stream builders and concatenation; `simpleStruc` covers simple structural operations (and their combinations); `simpleFunc` addresses simple functional operations (and their combinations); `stmCollInter` captures bidirectional interactions between streams and collections; `primStm` examines primitive-type streams and their conversions to object streams and vice versa; and `conservStm` evaluates conservatively modeled operations. The remaining benchmarks are self-explanatory by their name (see the "Micro-benchmark" column in Table 5).

Each benchmark contains 4–10 stream pipelines (see the "#pipeline" column in Table 5), with the number of input heap objects per pipeline ranging from 1 to 5. To provide a baseline for evaluating precision, we manually recorded the abstract heap objects pointed to by stream-output pointers during dynamic execution (results available in our artifact) and reported their totals for each benchmark (see the "Dyn #stm-out" column). To validate soundness, we verified that every heap object observed dynamically for stream-output pointers is also captured by the static analyses, including both Cut-Shortcut and Cut-Shortcut^S (achieving 100% recall). This measurement confirms that the precision results of Cut-Shortcut^S are trustworthy.

Efficiency. The red lines in Figure 33 highlight the efficiency advantage of Cut-Shortcut^S: across all 10 micro-benchmarks, its runtime is comparable to CI and Cut-Shortcut. Because these benchmarks are relatively small, the efficiency gains of Cut-Shortcut^S (and Cut-Shortcut) over

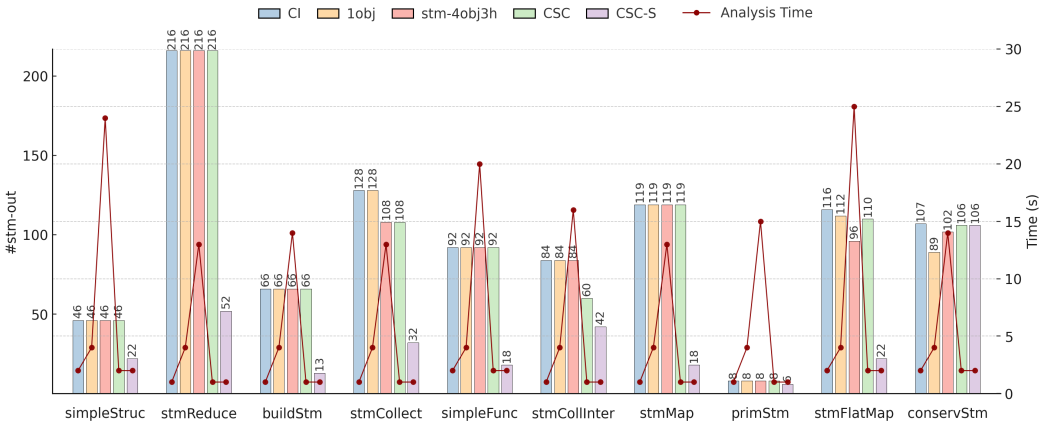


Fig. 33. Evaluation results across 10 micro-benchmarks corresponding to Table 5. Bars (with numbers) indicate #stm-out (left axis, lower is more precise), and red lines show analysis time in seconds (right axis).

CI are not pronounced; more substantial advantages will appear in the evaluation of real-world Java 8 programs (see later subsections). Nonetheless, Cut-Shortcut$^{\mathcal{S}}$ clearly outperforms conventional and selective context-sensitive analyses, achieving an average speedup of 3.0× over 1obj and 12.3× over stm-4obj3h.

Precision. The colored bars in Figure 33 present the precision results for #stm-out across all 10 micro-benchmarks. Cut-Shortcut$^{\mathcal{S}}$ consistently achieves the highest precision, except on `conservStm`, which consists solely of operations conservatively modeled in our approach. The reason Cut-Shortcut$^{\mathcal{S}}$ attains superior precision is because it can distinguish between different stream pipelines—an effect unattainable with conventional or selective context sensitivity in practice. In principle, full call-site sensitivity could yield similar precision, but the deepest selective configuration that succeeded on our machine (3-call-site sensitivity with 2-layer heap abstraction) still failed to match the precision of Cut-Shortcut$^{\mathcal{S}}$ and was far less efficient than stm-4obj3h; thus, it is omitted from Table 5. Overall, across the 10 benchmarks, Cut-Shortcut$^{\mathcal{S}}$ reduces #stm-out by 55.9% on average compared to all four baselines.

While conventional and selective context sensitivity fail to accurately distinguish objects propagated through different streams, they nonetheless yield notable precision gains for other pointers used within or related to pipelines, reflected in improvements in the #app-vpt metric. Summarizing, for #app-vpt, Cut-Shortcut$^{\mathcal{S}}$ achieves average precision improvements of 86.2% over CI, 67.2% over 1obj, 31.9% over stm-4obj3h, and 57.9% over Cut-Shortcut.

8.5.2 Micro-Benchmarks Mimicking Backend Workflows. In real-world software development, the Java stream API is widely used in backend systems to support declarative, functional-style processing of collections. It enables concise and elegant implementations of common tasks such as filtering, mapping, grouping, and data transformation—operations central to business logic, PO/VO/BO/DTO conversions, and query-result aggregation. However, static analysis of such enterprise backend code remains challenging due to obstacles such as modeling Java/database interactions and identifying application entry points. To enable evaluation, we designed a benchmark suite that mimics backend workflows using a hard-coded in-memory mock database, avoiding the need for actual database connections.

The workflowX Micro-Benchmarks. This suite builds on and extends the BSS benchmark [82], originally generated by the S2S tool [64], which translates SQL queries into Java stream code.

Table 6. Efficiency and precision results of context-insensitive analysis (CI), conventional context sensitivity (1obj), selective 4-object sensitivity with 3 context-sensitive heap abstractions on stream-related library code (stm-4obj3h), Cut-Shortcut (CSC), and Cut-Shortcut^S (CSC-S) across 5 micro-benchmarks mimicking backend workflows (left half), and their 5 string-outputting variants (right half).

Micro-benchmark	Dyn #stm-out	Analysis	Time(s)	#app-vpt	#stm-out	Micro-benchmark	Dyn #stm-out	Analysis	Time(s)	#app-vpt	#stm-out
workflow1	65	CI	2	14,433	202	workflow1-str	63	CI	8	89,068	31,326
		1obj	5	12,501	202			1obj	155	67,451	26,544
		stm-4obj3h	16	8,597	202			stm-4obj3h	22	72,620	26,550
		CSC	3	12,297	202			CSC	7	84,066	30,072
		CSC-S	2	3,896	99			CSC-S	7	15,246	85
workflow2	32	CI	3	29,908	357	workflow2-str	12	CI	9	165,794	31,038
		1obj	5	14,210	323			1obj	157	125,906	26,514
		stm-4obj3h	28	10,422	357			stm-4obj3h	58	116,839	26,242
		CSC	3	13,612	357			CSC	7	138,749	29,814
		CSC-S	2	5,021	90			CSC-S	7	11,409	46
workflow3	35	CI	3	47,747	507	workflow3-str	35	CI	10	181,839	31,542
		1obj	5	18,167	507			1obj	170	87,926	26,952
		stm-4obj3h	28	14,496	507			stm-4obj3h	67	84,395	26,712
		CSC	3	18,163	507			CSC	7	95,920	30,228
		CSC-S	3	7,773	228			CSC-S	7	8,251	228
workflow4	44	CI	3	75,642	382	workflow4-str	44	CI	10	276,354	32,052
		1obj	5	36,368	382			1obj	174	162,678	27,474
		stm-4obj3h	115	17,819	382			stm-4obj3h	247	88,176	27,288
		CSC	3	26,970	382			CSC	8	167,770	30,768
		CSC-S	3	11,374	188			CSC-S	8	32,335	188
workflow5	40	CI	4	70,720	290	workflow5-str	82	CI	10	281,520	32,070
		1obj	7	28,580	273			1obj	202	149,626	27,474
		stm-4obj3h	42	18,678	290			stm-4obj3h	102	139,656	27,306
		CSC	3	24,460	290			CSC	8	159,372	30,774
		CSC-S	3	18,783	177			CSC-S	8	90,930	10,184

To adapt BSS for static analysis, we replaced its database connection with an in-memory mock database, modified and expanded the original 8 tasks into 30, then grouped into five benchmarks (`workflow-1` through `workflow-5`, see the left half of Table 6). Each benchmark contains 6 tasks (6 stream pipelines), representing distinct backend business logic tasks. The numbering (1–5) reflects increasing task complexity, from simple querying and mapping to advanced grouping and data transformation. Our mock database consists of 8 tables, each with 5–100 rows, where each row corresponds to one abstract heap object. For each task, rows are queried from the database, used as inputs to stream pipelines, processed according to the specified business logic, and the resulting outputs mimic the objects returned to the application frontend.

For each benchmark, we manually recorded the abstract heap objects pointed to by stream-output variables during dynamic execution (results available in our artifact) and reported their totals in the "Dyn #stm-out" column in Table 6 (the left half). To validate soundness, we confirmed that every heap object observed dynamically at stream-output pointers is also captured by all static analyses, including both Cut-Shortcut and Cut-Shortcut^S (achieving 100% recall). This measurement confirms the reliability of the precision improvements obtained from Cut-Shortcut^S.

The workflowX-str Micro-Benchmarks. To more closely reflect real-world backend scenarios, we further adapted each benchmark `workflow1`–`workflow5` into a corresponding variant, `workflow1-str` to `workflow5-str`, in which the final outputs of the stream pipelines are of type `String`. This extension is motivated by the ubiquity of string processing in enterprise applications: many

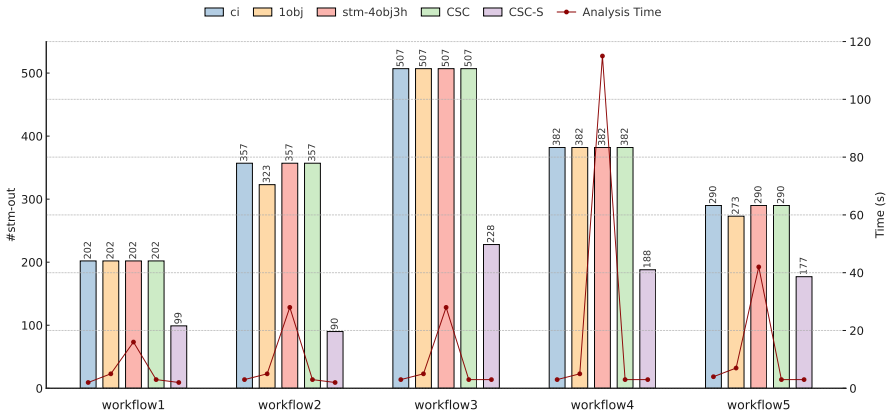


Fig. 34. Evaluation results across the 5 micro-benchmarks mimicking backend workflows (corresponding to the left half of Table 6). Bars (with numbers) indicate #stm-out (left axis, lower is more precise), while red lines show analysis time in seconds (right axis).

database fields are stored as strings, and backend systems frequently convert diverse data types into strings prior to transmission, serialization, or logging. Capturing this scenario is therefore essential for evaluating pointer analyses on realistic workloads.

More importantly, these benchmarks expose a key weakness of conventional analyses. As with other types, when pipelines are `String`-typed, all compared analyses other than Cut-Shortcut^S merge the input elements of different pipelines, leading to imprecision. For `String`, however, this problem is especially acute due to the prevalence of string-related pipelines in the JDK 8 library. Critically, this imprecision cannot be mitigated by a trivial post-hoc type filter on stream outputs: once merged, the spurious `String` objects propagate throughout the analysis, contaminating both library and application-level results. By contrast, Cut-Shortcut^S preserves the separation of pipelines regardless of element type, and thus remains robust in this setting. We note that analogous issues also arise for other generic-typed pipelines, such as those over `Object`. However, we restrict our evaluation to `String`-typed pipelines, because they are both pervasive in practice and sufficient to demonstrate the precision degradation of existing approaches. The results for these extended benchmarks are reported in the right half of Table 6.

Efficiency. The red lines in Figure 34 illustrate the efficiency advantage of Cut-Shortcut^S. Across the five `workflowX` benchmarks, its elapsed time is consistently on par with or faster than CI and Cut-Shortcut. This efficiency gain arises because Cut-Shortcut^S reduces the amount of spurious points-to information propagated during analysis, while the overhead introduced by its stream-handling extension is negligible. Compared to 1obj, Cut-Shortcut^S achieves an average speedup of 2.1×; compared to stm-4obj3h, the speedup is 14.1×.

For the five `workflowX-str` benchmarks, the analysis times are shown in the "Time(s)" column in the right half of Table 6. For these benchmarks, we configured the Tai-e framework to distinguish `String` constants in pointer analysis in order to evaluate precision (whereas the default configuration merges all `String` constants that are not used in reflection into a single abstract heap object). This configuration substantially increases the analysis time relative to the non-string variants (left half of Table 6). Nevertheless, Cut-Shortcut^S and Cut-Shortcut remain the most efficient analyses, achieving on average 1.3× speedup over CI, 23.1× over 1obj, and 10.0× over stm-4obj3h.

Precision. The colored bars in Figure 34 show the #stm-out precision results across the five `workflowX` benchmarks. As expected, Cut-Shortcut^S consistently achieves the highest precision

Table 7. Efficiency and precision results of context-insensitive analysis (CI), conventional context sensitivity (1obj), selective 4-object sensitivity with 3 context-sensitive heap abstractions on stream-related library (stm-4obj3h), Cut-Shortcut (CSC), and Cut-Shortcut^S (CSC-S) across 5 real-world stream-intensive programs.

Program	#pipeline	Analysis	Time(s)	#app-vpt	#stm-out
ws4j	23 (25)	ci	6	1,422,873	5,772
		1obj	14	730,580	3,738
		stm-4obj3h	49	860,197	4,301
		CSC	4	374,182	4,209
		CSC-S	3	287,807	1,540
finmath	66 (91)	ci	8	3,181,920	5,148
		1obj	35	923,423	4,495
		stm-4obj3h	116	1,351,276	4,956
		CSC	6	1,127,181	4,678
		CSC-S	6	1,098,442	1,763
jbayes	75 (84)	ci	105	21,828,769	10,873
		1obj	—	—	—
		stm-4obj3h	285	15,812,207	8,774
		CSC	32	14,294,470	10,656
		CSC-S	36	14,205,333	6,440
amazon-sqs	382 (399)	ci	143	45,846,018	404,354
		1obj	730	23,449,094	232,277
		stm-4obj3h	1,157	29,087,035	338,019
		CSC	76	14,673,724	255,103
		CSC-S	70	12,167,567	135,274
mnemonics	14 (24)	ci	5	179,135	14,440
		1obj	122	90,024	12,153
		stm-4obj3h	21	84,788	11,512
		CSC	4	116,763	13,536
		CSC-S	3	108,316	8,939

by distinguishing objects flowing through different stream pipelines, which is beyond the capabilities of the other analyses. Overall, Cut-Shortcut^S improves #stm-out precision by 52.6% on average compared to all four baselines. For #app-vpt, it achieves average improvements of 79.5% over CI, 57.0% over 1obj, 36.0% over stm-4obj3h, and 50.6% over Cut-Shortcut.

For the five `workflowX-str` benchmarks, the precision degradation of the four baseline analyses is substantial, whereas Cut-Shortcut^S retains its high precision. (See the right half of Table 6.) The only exception is `workflow5-str`, in which 2 of the 6 pipelines include conservatively modeled stream operations; nonetheless, Cut-Shortcut^S still delivers the best precision among all analyses. On average, across these benchmarks, Cut-Shortcut^S improves #stm-out precision by 91.4% compared to the baselines. For #app-vpt, it achieves average improvements of 84.9% over CI, 72.5% over 1obj, 67.7% over stm-4obj3h, and 75.0% over Cut-Shortcut.

Forward Looking. Recent work has advanced the static analysis of enterprise applications [3, 44, 45], addressing challenges such as identifying application entry points [3], handling database connections [44], and modeling dependency injection, RPC, and message-based communication in microservices [45]. Our workflow-mimicking suite provides evidence of the potential benefits of precise stream handling by emulating enterprise backend workflows, thereby serving as a proxy to demonstrate the extent of precision improvements achievable in real-world scenarios. Nonetheless, integrating Cut-Shortcut^S with these emerging techniques and applying it to real enterprise systems remains non-trivial. We anticipate that such integration would enable highly precise, scalable analyses of enterprise applications in the future.

8.5.3 Real-World Stream-Intensive Programs. This benchmark suite consists of 5 real-world Java programs with intensive use of streams. Unlike the workflow-mimicking suite, these are not enterprise applications, making them complementary to the other benchmark suites. The `mnemonics`

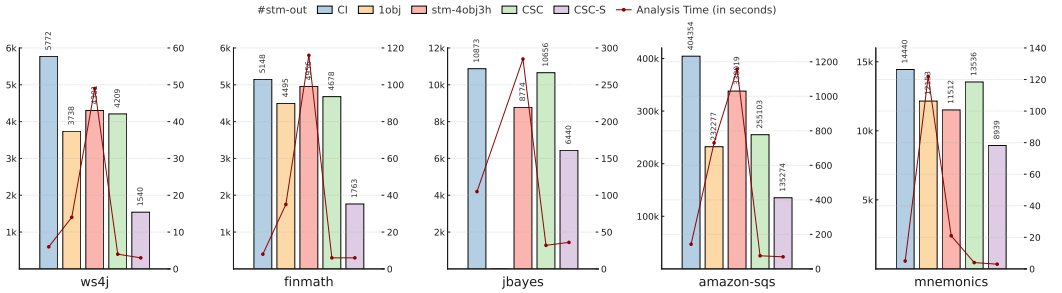


Fig. 35. Evaluation results across 5 real-world programs corresponding to Table 7. Bars (with numbers) indicate #stm-out (left axis, lower is more precise), and red lines show analysis time in seconds (right axis).

benchmark is drawn from the Renaissance suite [58], a widely used benchmark suite targeting advanced Java features. The remaining four benchmarks are selected from Rosales et al. [62], which surveyed Java projects with heavy stream usage. From their collection, we chose four projects that avoid complex framework-specific features (e.g., database queries, Spring AOP) that require additional handling in pointer analysis. Evaluation results on these five benchmarks are reported in Table 7. The "#pipeline" column indicates the number of abstract stream pipelines identified in each program, (numbers in parentheses include conservatively modeled ones).

Efficiency. The red lines in Figure 35 illustrate the efficiency advantage of Cut-Shortcut^S. Across all five programs, Cut-Shortcut is consistently faster than CI. Compared to Cut-Shortcut, Cut-Shortcut^S is even faster for three programs and equally fast in one program; on the remaining one (`finmath`), it is only slightly slower (by about one second) yet still faster than CI. On average, Cut-Shortcut^S achieves a 1.9× speedup over CI, a 10.4× speedup over 1obj (for the 4 program that 1obj can scale), and an 12.4× speedup over stm-4obj3h.

Precision. The colored bars in Figure 35 show the #stm-out precision results across the five programs. As with the previous benchmarks, Cut-Shortcut^S consistently delivers the best precision. While (selective) context sensitivity and Cut-Shortcut cannot distinguish objects flowing through different pipelines, they can still yield some precision improvements for #stm-out by reducing the size of the points-to sets of stream inputs. On average, Cut-Shortcut^S improves #stm-out precision by 54.9% over CI, 44.6% over 1obj (on the four programs where 1obj scales), 43.0% over stm-4obj3h, and 47.8% over Cut-Shortcut.

For #app-vpt, Cut-Shortcut^S achieves the highest precision for 3 out of 5 programs. Overall, Cut-Shortcut^S improves precision by 55.6% over CI, 11.3% over 1obj (for the four benchmarks on which 1obj succeeded, on two of which 1obj obtained better precision than Cut-Shortcut^S), 20.0% over stm-4obj3h (including one on which stm-4obj3h obtained better precision), and 5.4% over Cut-Shortcut. These gains, though substantial, are somewhat smaller than in the synthetic and backend workflow-style benchmarks. The reason is that the five real-world Java programs, while stream-intensive, do not fully capture the backend workflow patterns—such as query transformations or large-scale data aggregation—for which Cut-Shortcut^S offers its greatest advantages.

9 Related Work

9.1 Context-Sensitivity

Context sensitivity plays an essential role for whole-program Java pointer analysis, and we have discussed some of the related work in earlier sections. The other relevant research is covered below.

In recent years, many selective context-sensitivity approaches have been proposed, which provide different efficiency and precision trade-offs, especially for large and complex Java programs. We can

understand "selective" in two ways: select more effective context elements to distinguish different invocations of the same method (*Select context elements*), versus select a set of program methods to which a context-sensitive analysis is to be applied (*Select program methods*). Below, we discuss these two types of selective context-sensitivity approaches.

Select Context Elements. Conventional (k-limiting) context sensitivity uses k consecutive context elements, e.g., $[c_k..., c_2, c_1]$ to analyze a method m, where c_1 is m's call site and c_2 is the call site of the method containing c_1, etc. However, Tan et al. [77] found that this approach may result in many context elements that are not useful for improving precision while occupying a portion of the limited number of k slots. Tan et al. [77] presented an approach to recognize such redundant context elements by exploiting the so-called object-allocation graph that they proposed. Similarly, Jeon et al. [29] developed a machine-learning scheme called context tunneling to select such precision-useless context elements for more effective analysis. Furthermore, building on context tunneling, Jeon and Oh [31] proposed an interesting approach that transforms object sensitivity to call-site sensitivity, and showed that the latter can simulate the former (but not vice versa). In addition, He et al. [24] first proposed an IFDS-based [61] approach that identifies "context-independent" objects and "debloats" unnecessary context, thereby significantly reducing overhead in object-sensitive pointer analyses while retaining almost all precision. Later, they refined this concept by focusing on container-usage patterns [21], identifying context-independent objects even more accurately. Recently, Li et al. [37] presented generic sensitivity, which preserves generic instantiation sites as key context elements, leveraging a type-variable lookup map that is updated during a context-sensitive analysis.

Select Program Methods. Conventional context sensitivity uniformly applies contexts to every method in a given program. However, for certain program methods, context sensitivity does not help improve precision: it only incurs extra analysis cost. As a result, researchers have proposed ways to select a set of methods for which it is (likely to be) beneficial to analyze precisely; the analyzer only performs a context-sensitive analysis to those methods, and analyzes the remaining methods in a context-insensitive manner. To make a good efficiency/precision trade-off, the overall principle is to select the methods that are precision-critical [39] but not scalability-threatening [41]. To do so, various selection strategies exist: they rely on parameterized heuristics (whose thresholds are based on expert experience) [20, 70], machine-learning approaches [28, 30, 32], abstracted memory capacity [40], CFL-reachability-based pre-analysis [47], program patterns [39, 41]. Tan et al. [76] gave a scheme that supports (i) applying multiple selection strategies—with different variants of context sensitivity—to different program parts, combining their precision gains, and (ii) iteratively applying multiple strategies to the same program part to jointly enhance precision. In addition, context sensitivity can also be applied to selected variables and objects (rather than methods) [22, 23, 46], leveraging a pre-analysis in which object reachability is formulated (and solved) as a CFL-reachability problem [59] on the so-called pointer-assignment graph.

Hybrid Context Sensitivity. In some pointer analyses, the context elements for the same method may vary. Kastrinis and Smaragdakis [34] present a hybrid context-sensitivity approach in which a method can be analyzed under both call-site sensitivity and object sensitivity, and in some cases, such a combination can lead to more effective results than using a single context type. Thakur and Nandivada [80] mix object-sensitivity contexts and level-summarized relevant-value contexts (a context abstraction proposed in their earlier work [79]) to analyze each method, and are sometimes able to obtain more precise results than with either individual approach alone.

All of the approaches described above rely on the core idea of context sensitivity to replicate (a set of) methods and analyze the program elements in them separately under different (types of) contexts. Our approach is different in that it simulates the effect of context sensitivity by

suppressing (i.e., not generating) edges that would cause a loss of precision, and adding shortcut edges directly on the PFG, as explained throughout the paper.

9.2 Java Streams

Previous research on Java streams mainly falls into two categories: one for empirical study of stream usage in real-world code [35, 53, 62, 63], and the other for re-writing and optimizing stream performance [5, 51]. In contrast, the static modeling of streams in Cut-Shortcut^S is the first technique that aims at improving the precision of pointer analysis for programs that use Java streams.

9.3 Other Related Work

In Section 3.1, we discussed how Cut-Shortcut differs from summary-based pointer analysis techniques. However, it is worth noting that Cut-Shortcut employ a high-level idea similar to the use of summary edges in interprocedural slicing [27] and the later stages of interprocedural data-flow analysis [61, 65]. In the interprocedural-slicing algorithm proposed by Horwitz et al. [27], when marking vertices that can reach a given input vertex set s, their algorithm avoids descending into called procedures and instead uses *transitive flow dependence edges* (connecting actual-in to actual-out vertices) to discover paths that reach s via a procedure call—conceptually similar to how our shortcut edges in the PFG enable precise propagation without following full interprocedural flows. A particularly close parallel is the use of summary relations by Reps et al. [61] and Sharir and Pnueli [65]: once new relations are identified (shortcut edges in Cut-Shortcut and Cut-Shortcut^S, summary edges linking invocation arguments to returns in [61], and tabulated ϕ functions in [65]), the analysis proceeds using these facts, while method-return facts are effectively ignored.

Although these techniques share the same high-level idea, Cut-Shortcut and Cut-Shortcut^S differ in both their target problem (pointer analysis) and concrete methodology. Cut-Shortcut is a pattern-based approach, with shortcut-edge generation specifically tailored to each pattern. This design allows Cut-Shortcut and Cut-Shortcut^S to operate at a finer granularity by selectively suppressing imprecise flows and introducing precise shortcut flows for a certain method (or sequences of methods) while leaving unrelated parts of that method unchanged. Furthermore, the container-access pattern in Cut-Shortcut and the stream modeling in Cut-Shortcut^S enables cross-call-site object-flow routing—i.e., directly connecting the Sources of a container/stream instance to its Targets —a capability that is difficult to achieve using traditional summarization. Finally, unlike the phased approach of the interprocedural slicing and data-flow analysis, Cut-Shortcut and Cut-Shortcut^S are implemented to run on-the-fly alongside the pointer analysis itself, rather than performing a separate stage to compute summary relations.

In addition, Zhang et al. [89] present a hybrid top-down and bottom-up analysis. Top-down and bottom-up approaches are general ideas that many static analyses adopt. Like other mainstream pointer analyses for Java [67], Cut-Shortcut also performs in a top-down manner; if we treat the identification of cut and shortcut spots in the PFG for the field access pattern as a separate process, it shares a flavor of bottom-up summarization. However, our pattern-specific imprecise flow identification is on-the-fly, and neither the target problem, nor its underlying concrete methodology of Cut-Shortcut is similar to [89].

A different approach to avoiding spurious points-to information during the analysis of Java programs was presented by De and D'Souza [12]. In addition to using context sensitivity, their method provides a flow-sensitive approach to partially performing strong updates to heap objects. They accomplish this task by expressing points-to relations via access paths [16, 33]. Their approach

is different from Cut-Shortcut and many other schemes, which express points-to information via PFG-like graphs.

Finally, there are efforts to accelerate context-sensitive Java pointer analysis by merging heap objects [10, 78]; these approaches are orthogonal to ours.

10 Conclusion

For the past 20 years, context sensitivity has been considered as virtually the most useful technique for increasing the precision of Java pointer analysis. However, it imposes substantial efficiency costs, especially for large and complex programs. Selective context-sensitivity approaches partially alleviate this issue, but have not solved the efficiency bottleneck: it is hard to select the correct methods that are precision-critical, but do not threaten scalability. As a result, selective context-sensitivity approaches still run the risk of computing and maintaining a large number of contexts to distinguish spurious object flows, which limits their ability to analyze large programs successfully.

To address this dilemma, we developed a fundamentally different approach, called Cut-Shortcut, which tries to simulate the effect of context sensitivity without applying contexts. This effect is achieved by suppressing in the pointer flow graph the addition of edges that bring precision loss, and adding precise shortcut edges. We instantiated this high-level principle by exploiting three program patterns, and designing rules based on them in accordance with our principle. Then we formalized it and prove its soundness. We implemented Cut-Shortcut on the state-of-the-art Java pointer analysis framework Tai-e. The experimental results are extremely promising: Cut-Shortcut consistently matches or outperforms context insensitivity in terms of analysis time across all benchmarks, while obtaining significantly better precision—precision that obtained via context-sensitive approaches in some cases.

Building on this foundation, we extended Cut-Shortcut to modern Java language features, focusing on Java streams, which introduce new challenges for pointer analysis. We systematically classified stream operations into categories and developed a static modeling of Java streams that aligns with the principles of Cut-Shortcut, yielding the Cut-Shortcut^S extension. This extension improves precision by associating the input sources and output targets of stream pipeline instances, without relying on context sensitivity. Our evaluation on both micro-benchmarks and real-world programs shows that Cut-Shortcut^S enhances precision while preserving scalability and efficiency—consistently outperforming or matching context-insensitive analyses—and successfully handles stream-intensive programs where (selective) context-sensitive techniques fail to provide analyses that are both precise and scalable.

Together, these contributions demonstrate the flexibility and scalability of the Cut-Shortcut approach, both for classic Java programs and for modern constructs like streams. Given the encouraging outcomes, more program patterns or insights are expected to be explored on top of our approach, and we hope that our approach could provide new perspectives for developing more effective pointer analysis for Java in the future.

Acknowledgments

This work was supported, in part, by a gift from Rajiv and Ritu Batra, and by NSF under grants CCF-2211968 and CCF-2212558. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring entities.

References

[1] 2023. https://newrelic.com/resources/report/2023-state-of-the-java-ecosystem

[2] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph. D. Dissertation. University of Copenhagen.

[3] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. 2020. Static Analysis of Java Enterprise Applications: Frameworks and Caches, the Elephants in the Room. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 794–807. doi:10.1145/3385412.3386026

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

[5] Matteo Basso, Filippo Schiavio, Andrea Rosà, and Walter Binder. 2022. Optimizing Parallel Java Streams. In *2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS)*. 23–32. doi:10.1109/ICECCS54210.2022.00012

[6] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA). ACM Press, New York, NY, USA, 169–190. doi:10.1145/1167473.1167488

[7] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 243–262. doi:10.1145/1640089.1640108

[8] Yuandao Cai, Peisen Yao, and Charles Zhang. 2021. Canary: Practical Static Detection of Inter-Thread Value-Flow Bugs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1126–1140. doi:10.1145/3453483.3454099

[9] Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: A Powerful Approach to Weakest Preconditions. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 363–374. doi:10.1145/1542476.1542517

[10] Yifan Chen, Chenyang Yang, Xin Zhang, Yingfei Xiong, Hao Tang, Xiaoyin Wang, and Lu Zhang. 2021. Accelerating Program Analyses in Datalog by Merging Library Facts. In *Static Analysis: 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17–19, 2021, Proceedings* (Chicago, IL, USA). Springer-Verlag, Berlin, Heidelberg, 77–101. doi:10.1007/978-3-030-88806-0_4

[11] Ben-Chung Cheng and Wen-Mei W. Hwu. 2000. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 57–69. doi:10.1145/349299.349311

[12] Arnab De and Deepak D'Souza. 2012. Scalable Flow-Sensitive Pointer Analysis for Java with Strong Updates. In *ECOOP 2012 – Object-Oriented Programming*, James Noble (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 665–687.

[13] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, Mario Tokoro and Remo Pareschi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 77–101.

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

[15] Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. 2015. Bottom-Up Context-Sensitive Pointer Analysis for Java. In *Programming Languages and Systems*, Xinyu Feng and Sungwoo Park (Eds.). Springer International Publishing, Cham, 465–484.

[16] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.* 17, 2, Article 9 (May 2008), 34 pages. doi:10.1145/1348250.1348255

[17] Philip J. Fleming and John J. Wallace. 1986. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM* 29, 3 (March 1986), 218–221. doi:10.1145/5666.5673

[18] George Fourtounis and Yannis Smaragdakis. 2019. Deep Static Modeling of invokedynamic. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:28. doi:10.4230/LIPIcs.ECOOP.2019.15

[19] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: unified points-to and taint analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 102 (Oct. 2017), 28 pages. doi:10.1145/3133926

[20] Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. 2017. An Efficient Tunable Selective Points-to Analysis for Large Codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis* (Barcelona, Spain) *(SOAP 2017)*. Association for Computing Machinery, New York, NY, USA, 13–18. doi:10.1145/3088515.3088519

[21] Dongjie He, Yujiang Gui, Wei Li, Yonggang Tao, Changwei Zou, Yulei Sui, and Jingling Xue. 2023. A Container-Usage-Pattern-Based Context Debloating Approach for Object-Sensitive Pointer Analysis. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 256 (Oct. 2023), 30 pages. doi:10.1145/3622832

[22] Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. 2021. Accelerating Object-Sensitive Pointer Analysis by Exploiting Object Containment and Reachability. In *35th European Conference on Object-Oriented Programming (ECOOP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 16:1–16:31. doi:10.4230/LIPIcs.ECOOP.2021.16

[23] Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. 2023. Selecting Context-Sensitivity Modularly for Accelerating Object-Sensitive Pointer Analysis. *IEEE Transactions on Software Engineering* 49, 2 (2023), 719–742. doi:10.1109/TSE.2022.3162236

[24] Dongjie He, Jingbo Lu, and Jingling Xue. 2023. IFDS-based Context Debloating for Object-Sensitive Pointer Analysis. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 101 (May 2023), 44 pages. doi:10.1145/3579641

[25] Dongjie He, Jingbo Lu, and Jingling Xue. 2024. A CFL-Reachability Formulation of Callsite-Sensitive Pointer Analysis with Built-In On-The-Fly Call Graph Construction. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 18:1–18:29. doi:10.4230/LIPIcs.ECOOP.2024.18

[26] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Selective conjunction of context-sensitivity and octagon domain toward scalable and precise global static analysis. *Softw. Pract. Exp.* 47, 11 (2017), 1677–1705. doi:10.1002/spe.2493

[27] Susan Horwitz, Thomas W. Reps, and David W. Binkley. 1990. Interprocedural Slicing Using Dependence Graphs. *ACM Trans. Program. Lang. Syst.* 12, 1 (1990), 26–60. doi:10.1145/77606.77608

[28] Minseok Jeon, Sehun Jeong, Sungdeok Cha, and Hakjoo Oh. 2019. A Machine-Learning Algorithm with Disjunctive Model for Data-Driven Program Analysis. *ACM Trans. Program. Lang. Syst.* 41, 2, Article 13 (jun 2019), 41 pages. doi:10.1145/3293607

[29] Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and Scalable Points-to Analysis via Data-Driven Context Tunneling. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 140 (oct 2018), 29 pages. doi:10.1145/3276510

[30] Minseok Jeon, Myungho Lee, and Hakjoo Oh. 2020. Learning Graph-Based Heuristics for Pointer Analysis without Handcrafting Application-Specific Features. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 179 (nov 2020), 30 pages. doi:10.1145/3428247

[31] Minseok Jeon and Hakjoo Oh. 2022. Return of CFA: Call-Site Sensitivity Can Be Superior to Object Sensitivity Even for Object-Oriented Programs. *Proc. ACM Program. Lang.* 6, POPL, Article 58 (jan 2022), 29 pages. doi:10.1145/3498720

[32] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 100 (Oct. 2017), 28 pages. doi:10.1145/3133924

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

[34] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 423–434. doi:10.1145/2491956.2462191

[35] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Baishakhi Ray. 2020. An Empirical Study on the Use and Misuse of Java 8 Streams. In *Fundamental Approaches to Software Engineering*, Heike Wehrheim and Jordi Cabot (Eds.). Springer International Publishing, Cham, 97–118.

[36] Ondřej Lhoták and Laurie Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *Compiler Construction*, Alan Mycroft and Andreas Zeller (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 47–64.

[37] Haofeng Li, Jie Lu, Haining Meng, Liqing Cao, Yongheng Huang, Lian Li, and Lin Gao. 2022. Generic sensitivity: customizing context-sensitive pointer analysis for generics. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1110–1121. doi:10.1145/3540250.3549122

[38] Haofeng Li, Chenghang Shi, Jie Lu, Lian Li, and Zixuan Zhao. 2025. Module-Aware Context Sensitive Pointer Analysis . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1819–1831. doi:10.1109/ICSE55347.2025.00227

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

[40] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 129–140. doi:10.1145/3236024.3236041

[41] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 42, 2, Article 10 (may 2020), 40 pages. doi:10.1145/3381915

[42] Yue Li, Tian Tan, and Jingling Xue. 2019. Understanding and Analyzing Java Reflection. *ACM Trans. Softw. Eng. Methodol.* 28, 2, Article 7 (feb 2019), 50 pages. doi:10.1145/3295739

[43] Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. 2016. Program Tailoring: Slicing by Sequential Criteria. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 15:1–15:27. doi:10.4230/LIPIcs.ECOOP.2016.15

[44] Yufei Liang, Teng Zhang, Ganlin Li, Tian Tan, Chang Xu, Chun Cao, Xiaoxing Ma, and Yue Li. 2025. Pointer Analysis for Database-Backed Applications. *Proc. ACM Program. Lang.* 9, PLDI, Article 204 (June 2025), 25 pages. doi:10.1145/3729307

[45] Yufei Liang, Teng Zhang, Ganlin Li, Tian Tan, Chang Xu, Chun Cao, Xiaoxing Ma, and Yue Li. 2025. Pointer Analysis for Database-Backed Applications. *Proc. ACM Program. Lang.* 9, PLDI, Article 204 (June 2025), 25 pages. doi:10.1145/3729307

[46] Jingbo Lu, Dongjie He, and Jingling Xue. 2021. Eagle: CFL-Reachability-Based Precision-Preserving Acceleration of Object-Sensitive Pointer Analysis with Partial Context Sensitivity. *ACM Trans. Softw. Eng. Methodol.* 30, 4, Article 46 (jul 2021), 46 pages. doi:10.1145/3450492

[47] Jingbo Lu, Dongjie He, and Jingling Xue. 2021. Selective Context-Sensitivity for k-CFA with CFL-Reachability. In *Static Analysis: 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17–19, 2021, Proceedings* (Chicago, IL, USA). Springer-Verlag, Berlin, Heidelberg, 261–285. doi:10.1007/978-3-030-88806-0_13

[48] Wenjie Ma, Shengyuan Yang, Tian Tan, Xiaoxing Ma, Chang Xu, and Yue Li. 2023. Context Sensitivity without Contexts: A Cut-Shortcut Approach to Fast and Precise Pointer Analysis. *Proc. ACM Program. Lang.* 7, PLDI, Article 128 (June 2023), 26 pages. doi:10.1145/3591242

[49] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2002. Parameterized Object Sensitivity for Points-to and Side-Effect Analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Roma, Italy) *(ISSTA '02)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/566172.566174

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

[51] Anders Møller and Oskar Haarklou Veileborg. 2020. Eliminating abstraction overhead of Java stream pipelines using ahead-of-time program optimization. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 168 (Nov. 2020), 29 pages. doi:10.1145/3428236

[52] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 308–319. doi:10.1145/1133981.1134018

[53] Joshua Nostas, Juan Pablo Sandoval Alcocer, Diego Elias Costa, and Alexandre Bergel. 2021. How Do Developers Use the Java Stream API?. In *Computational Science and Its Applications – ICCSA 2021*, Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Chiara Garau, Ivan Blečić, David Taniar, Bernady O. Apduhan, Ana Maria A. C. Rocha, Eufemia Tarantino, and Carmelo Maria Torre (Eds.). Springer International Publishing, Cham, 323–335.

[54] Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. 2004. Bottom-Up and Top-Down Context-Sensitive Summary-Based Pointer Analysis. In *Static Analysis*, Roberto Giacobazzi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 165–180.

[55] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective Context-Sensitivity Guided by Impact Pre-Analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 475–484. doi:10.1145/2594291.2594318

[56] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2015. Selective X-Sensitive Analysis Guided by Impact Pre-Analysis. *ACM Trans. Program. Lang. Syst.* 38, 2, Article 6 (dec 2015), 45 pages. doi:10.1145/2821504

[57] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. 2012. Statically checking API protocol conformance with mined multi-object specifications. In *2012 34th International Conference on Software Engineering (ICSE)*. 925–935. doi:10.1109/ICSE.2012.6227127

[58] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: benchmarking

suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 31–47. doi:10.1145/3314221.3314637

[59] Thomas Reps. 1998. Program analysis via graph reachability1An abbreviated version of this paper appeared as an invited paper in the Proceedings of the 1997 International Symposium on Logic Programming [84].1. *Information and Software Technology* 40, 11 (1998), 701–726. doi:10.1016/S0950-5849(98)00093-7

[60] Thomas Reps. 2000. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 162–186. doi:10.1145/345099.345137

[61] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 49–61. doi:10.1145/199448.199462

[62] Eduardo Rosales, Matteo Basso, Andrea Rosà, and Walter Binder. 2023. Large-scale characterization of Java streams. *Software: Practice and Experience* 53, 9 (2023), 1763–1792.

[63] Eduardo Rosales, Andrea Rosà, Matteo Basso, Alex Villazón, Adriana Orellana, Ángel Zenteno, Jhon Rivero, and Walter Binder. 2022. Characterizing Java Streams in the Wild. In *2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS)*. 143–152. doi:10.1109/ICECCS54210.2022.00025

[64] Filippo Schiavio, Andrea Rosà, and Walter Binder. 2022. SQL to Stream with S2S: An Automatic Benchmark Generator for the Java Stream API. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Auckland, New Zealand) *(GPCE 2022)*. Association for Computing Machinery, New York, NY, USA, 179–186. doi:10.1145/3564719.3568699

[65] Micha Sharir and Amir Pnueli. 1981. *Two approaches to interprocedural data flow analysis.* Prentice-Hall, Chapter 7, 189–234.

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

[67] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (April 2015), 1–69. doi:10.1561/2500000014

[68] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More Sound Static Handling of Java Reflection. In *Programming Languages and Systems*, Xinyu Feng and Sungwoo Park (Eds.). Springer International Publishing, Cham, 485–503.

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

[70] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective Analysis: Context-Sensitivity, across the Board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 485–495. doi:10.1145/2594291.2594320

[71] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 387–400. doi:10.1145/1133981.1134027

[72] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. *Alias Analysis for Object-Oriented Programs.* Springer Berlin Heidelberg, Berlin, Heidelberg, 196–232. doi:10.1007/978-3-642-36946-9_8

[73] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. 2007. Thin Slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 112–122. doi:10.1145/1250734.1250748

[74] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 59–76. doi:10.1145/1094811.1094817

[75] Tian Tan and Yue Li. 2023. Tai-e: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1093–1105. doi:10.1145/3597926.3598120

[76] Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. 2021. Making Pointer Analysis More Precise by Unleashing the Power of Selective Context Sensitivity. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 147 (oct 2021), 27 pages. doi:10.1145/3485524

[77] Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-Object-Sensitive Pointer Analysis More Precise with Still k-Limiting. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9837)*, Xavier Rival (Ed.). Springer, 489–510. doi:10.1007/978-3-662-53413-7_24

[78] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 278–291. doi:10.1145/3062341.3062360

[79] Manas Thakur and V. Krishna Nandivada. 2019. Compare Less, Defer More: Scaling Value-Contexts Based Whole-Program Heap Analyses. In *Proceedings of the 28th International Conference on Compiler Construction* (Washington, DC, USA) *(CC 2019)*. Association for Computing Machinery, New York, NY, USA, 135–146. doi:10.1145/3302516.3307359

[80] Manas Thakur and V. Krishna Nandivada. 2020. Mix Your Contexts Well: Opportunities Unleashed by Recent Advances in Scaling Context-Sensitivity. In *Proceedings of the 29th International Conference on Compiler Construction* (San Diego, CA, USA) *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 27–38. doi:10.1145/3377555.3377902

[81] Paolo Tonella and Alessandra Potrich. 2005. *Reverse Engineering of Object Oriented Code*. Springer. doi:10.1007/b102522

[82] USI-DAG. 2022. BSS: Benchmark Suite for the Stream API. https://github.com/usi-dag/BSS.

[83] WALA. 2006. Watson Libraries for Analysis. http://wala.sf.net.

[84] Shiyi Wei and Barbara G. Ryder. 2015. Adaptive Context-sensitive Analysis for JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 37)*, John Tang Boyland (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 712–734. doi:10.4230/LIPIcs.ECOOP.2015.712

[85] Robert P. Wilson and Monica S. Lam. 1995. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation* (La Jolla, California, USA) *(PLDI '95)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/207110.207111

[86] YangShengYuan. 2025. Cut-Shortcut-Artifact. https://github.com/YangShengYuan/cut-shortcut-artifact.

[87] Chenxi Zhang, Yufei Liang, Tian Tan, Chang Xu, Shuangxiang Kan, Yulei Sui, and Yue Li. 2025. Interactive Cross-Language Pointer Analysis For Resolving Native Code in Java Programs . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 612–612. doi:10.1109/ICSE55347.2025.00075

[88] Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. 2013. Fast Algorithms for Dyck-CFL-Reachability with Applications to Alias Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 435–446. doi:10.1145/2491956.2462159

[89] Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. 2014. Hybrid top-down and bottom-up interprocedural analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 249–258. doi:10.1145/2594291.2594328

A Conservative Fallback Mechanisms for Container-Access Pattern

In this appendix, we introduce additional mechanisms for conservative fallbacks for the container-access pattern, including sound handling of custom containers (i.e., containers not defined in the JDK and therefore lacking specified APIs) and the treatment of hosts for `this` variables. We first extend the domain of hosts to $\mathbb{H} = (\mathbb{L} \cup \hat{l}) \times \mathbb{C}$, where $\hat{l}$ is a special mock label, introduced to handle the `this` variable in the pointer–host mapping (pt_H). For clarity of formalism, we define container types as $\tau_C = \text{Collection} \mid \text{Map}$, representing the standard JDK container interfaces. Host-relevant types are then defined as $\tau_{\mathcal{H}} = \tau_C \mid \text{Iterator} \mid \text{ListIterator} \mid \text{Enumeration} \mid \text{Map.Entry}$, thereby extending τ_C with container-dependent types.

$$\frac{\mathcal{T}(o_i) = \tau \quad \tau \notin \mathit{JDK}}{h_c^i \in \mathit{conservH}}\ [\textsc{CustomCont}] \qquad \frac{h' \lhd h \quad h' \in \mathit{conservH}}{h \in \mathit{conservH}}\ [\textsc{HostDep-II}]$$

$$\frac{\begin{array}{c} i \rightarrow_{call} m \quad m \in \tau \\ \exists \tau_{\mathcal{H}}.\tau <: \tau_{\mathcal{H}} \quad c \in \mathbb{K} \end{array}}{h_c^i \in pt_H(\texttt{this}^m) \quad h_c^i \in \mathit{conservH}}\ [\textsc{MockHost}] \qquad \frac{\begin{array}{c} i \rightarrow_{call} m \quad \texttt{ret}^m \in \mathit{cutRet} \\ h \in pt_H(\texttt{arg}_0^i) \quad h \in \mathit{conservH} \end{array}}{\texttt{ret}^m \longrightarrow \texttt{LHS}^i}\ [\textsc{ConservHost}]$$

Fig. 36. Rules for handling container access pattern.

The four rules above formalize mechanisms employed to ensure soundness. The central idea is to define a set *conservH*, which records certain hosts—those whose underlying container instances require conservative treatment during element retrieval. Rule [ConservHost] governs this conservative behavior: while our analysis normally suppresses imprecise edges from Exit methods' return variables to the Targets of a host, this suppression is deliberately disabled for hosts in *conservH*. Specifically, the method return edges that would otherwise be suppressed (as per the boxed condition in rule [Return], Figure 7) are added back to the PFG (such inference takes place during the main pointer analysis, because it is part of the on-the-fly construction of the PFG), preserving soundness for those container usages. The next three rules, [CustomCont], [HostDep-II], and [MockHost], describe how hosts are added to *conservH*:

- [CustomCont] adds any host created from a custom container class (i.e., a class of τ such that τ is not defined in the JDK) to *conservH*. Because such custom classes lack pre-specified Entrance/Exit specifications, the analysis cannot reason about them and must fall back on conservative handling.
- [HostDep-II] propagates conservativeness via the host dependency preorder $\lhd$. If host h' is in *conservH* and $h' \lhd h$, then h is also added to *conservH*. For example, if a JDK container receives elements via `addAll()` from a custom container, then the JDK container must be conservatively treated, because its Sources may now include elements that the analysis cannot track.
- [MockHost] addresses a subtle case: the analysis does not propagate hosts from a receiver variable to the `this` variable during method calls. This omission is deliberate and replaced by conservative handling. For any call site i that invokes a method m declared in a container or a container-dependent class τ (i.e., $\exists\, \tau_{\mathcal{H}}$ such that $\tau <: \tau_{\mathcal{H}}$[20]), we construct a set of mock hosts $\{h_c^i \mid c \in \mathbb{K}\}$, assign them to $pt_H(\texttt{this}^m)$, and also record them in *conservH*. These mock hosts—defined using the special mock-instruction label $\hat{l}$—are shared across all `this` variables of relevant methods. This design avoids the cost of propagating hosts from receivers to `this` at each call site, while maintaining soundness. It also has little impact on precision, because most Exit invocations on `this` do not cause merged flows to leak beyond the container.

[20]Note that because hosts can also propagate to container-dependent objects that are not subtypes of `Collection` or `Map`, the type $\tau_{\mathcal{H}}$ must include all top-level types of such container-dependent classes.

B Stream Operations

A complete list of stream operations modeled in our analysis is shown in Figure 37. (The signatures of methods inherited by subclassing? are omitted.) Operations marked with circled letters indicate cases where, beyond the sub-category-specific rules introduced in Sections 7.3–7.5, the following special-case handling is used:

Ⓐ: requires special handling to record array elements as host Sources;

Ⓑ: `Stream.concat` concatenates two stream pipelines into a new one. Handling this requires propagating Sources from multiple Strm hosts to another, which is easily achieved using the host-dependency preorder $\lhd$ (defined in Section 4.3).

Ⓒ: requires augmenting the Sources of an existing Strm host, which can be handled straightforwardly by registering these operations in the ContEntr set (defined in Section 4.3).

The detailed rules for such special-case handling are omitted for brevity, because they are not central to the core design of our stream modeling. The complete inference rules for all *functional* stream operations are provided in Appendix C.

Structure Creation — handled by [S-Create]

```
Stream.of: (Object) -> Stream
Stream.of: (Object[]) -> Stream Ⓐ
Stream.builder: () -> Stream$Builder
Stream.empty: () -> Stream
Stream.concat: (Stream,Stream) -> Stream Ⓑ
Arrays.stream: (Object[]) -> Stream Ⓐ
Arrays.stream: (Object[],int,int) -> Stream Ⓐ
Optional.of: (Object) -> Stream
Optional.empty: () -> Optional
Optional.of: (Object) -> Optional
Optional.ofNullable: (Object) -> Optional
```

Structure Process — handled by [S-Process]

```
Stream.distinct: () -> Stream
Stream.filter: (Predicate) -> Stream
Stream.findAny: (Predicate) -> Optional
Stream.findFirst: (Predicate) -> Optional
Stream.limit: long -> Stream
Stream.min: (Comparator) -> Optional
Stream.max: (Comparator) -> Optional
Stream.sorted: () -> Stream
Stream.sorted: (Comparatpr) -> Stream
Stream.skip: long -> Stream
BaseStream.sequential: () -> BaseStream
BaseStream.parallel: () -> BaseStream
BaseStream.onClose: (Runnable) -> BaseStream
BaseStream.unordered: () -> BaseStream
Stream$Builder.build: () -> Stream
Stream$Builder.add: (Object) -> Stream$Builder Ⓒ
Stream$Builder.accept: (Object) -> Stream$Builder Ⓒ
Optional.filter: (Predicate) -> Optional
```

Structure Output — handled by [S-Output]

```
Optional.get: () -> Object
Optional.orElse: (Object) -> Object Ⓒ
Optional.orElseThrow: (Supplier) -> Object
```

handled by operation-specific rules (in Appendix B)

Functional Creation

```
Stream.generate: (Supplier) -> Stream
Stream.iterate: (Object,UnaryOperator) -> Stream
IntStream.boxed: () -> Stream
IntStream.mapToObj: (IntFunction) -> Stream
LongStream.boxed: () -> Stream
LongStream.mapToObj: (LongFunction) -> Stream
DoubleStream.boxed: () -> Stream
DoubleStream.mapToObj: (DoubleFunction) -> Stream
```

Functional Process

```
Stream.flatMap: (Function) -> Stream
Stream.map: (Function) -> Stream
Stream.reduce: (BinaryOperator) -> Optional
Optional.flatMap: (Function) -> Optional
Optional.map: (Function) -> Optional
Stream.peek: (Consumer) -> Stream
```

Functional Output

```
Stream.forEach: (Consumer) -> void
Stream.forEachOrdered: (Consumer) -> void
Stream.collect: (Supplier,BiConsumer,BiConsumer) -> Object
Stream.reduce: (Object,BinaryOperator) -> Object
Stream.reduce: (Object,BiFunction,BinaryOperator) -> Object
Optional.orElseGet: (Supplier) -> Object
Optional.ifPresent: (Consumer) -> void
```

Collection Creation — handled by [C-Create]

```
Collection.stream: () -> Stream
Collection.parallelStream: () -> Stream
StreamSupport.stream: (Spliterator,int,boolean) -> Stream
```

Collection Output — handled by [C-Output]

```
Stream.collect: (Collector) -> Object
Stream.toList: () -> List
BaseStream.spliterator: () -> Spliterator
BaseStream.iterator: () -> Iterator
```

Fig. 37. The list of modeled stream operations.

C Handling Functional Stream Operations

In Section 7.4, we presented a selected subset of rules (in Figure 25) from the lambda analysis proposed by Fourtounis and Smaragdakis [18]. The complete rule set used to handle lambda call edges is shown in Figure 38. For clarity and consistency, we re-formalize these rules in our own notation.

$$\frac{\texttt{i: r = b.m(a1, ..., an)} \quad \lambda \in pt(b)}{\langle i, \lambda \rangle \in LCallSites} \text{ [L-CallSite]}$$

$$\frac{\langle i, \lambda \rangle \in LCallSites \quad \#\mathsf{cap}^{\lambda} = 0 \quad \mathrm{Target}(\lambda) = m \quad m \notin Static}{\langle i, \lambda, \mathsf{Larg}_1^{i,\lambda} \rangle \in LRecvs} \text{ [L-Recv-I]}$$

$$\frac{\langle i, \lambda \rangle \in LCallSites \quad \#\mathsf{cap}^{\lambda} \neq 0 \quad \mathrm{Target}(\lambda) = m \quad m \notin Static}{\langle i, \lambda, \mathsf{cap}_1^{\lambda} \rangle \in LRecvs} \text{ [L-Recv-II]}$$

$$\frac{\langle i, \lambda \rangle \in LCallSites \quad \mathrm{Target}(\lambda) = m \quad m \notin Static \quad \langle i, \lambda, x \rangle \in LRecvs \quad o_j \in pt(x) \quad m' = \mathrm{dispatch}(m, o_j)}{o_j \in pt(\mathsf{param}_0^{m'}) \quad i \rightarrow_{\lambda} m'} \text{ [L-InsCall]}$$

$$\frac{i \rightarrow_{\lambda} m \quad m \notin Static \quad \#\mathsf{cap}^{\lambda} > 1}{\forall k \geq 2 : \mathsf{cap}_k^{\lambda} \longrightarrow \mathsf{param}_{k-1}^{m}} \text{ [L-InsCap]}$$

$$\frac{i \rightarrow_{\lambda} m \quad m \notin Static \quad \#\mathsf{cap}^{\lambda} = 0}{\forall k \geq 2 : \mathsf{Larg}_k^{i,\lambda} \longrightarrow \mathsf{param}_{k-1}^{m}} \text{ [L-InsLArg-I]}$$

$$\frac{i \rightarrow_{\lambda} m \quad m \notin Static \quad \#\mathsf{cap}^{\lambda} = n > 0}{\forall k \geq 1 : \mathsf{Larg}_k^{i,\lambda} \longrightarrow \mathsf{param}_{k+n-1}^{m}} \text{ [L-InsLArg-II]}$$

$$\frac{\langle i, \lambda \rangle \in LCallSites \quad \mathrm{Target}(\lambda) = m \quad m \in Static}{i \rightarrow_{\lambda} m} \text{ [L-StaCall]}$$

$$\frac{i \rightarrow_{\lambda} m \quad m \in Static \quad \#\mathsf{cap}^{\lambda} \neq 0}{\forall k \geq 1 : \mathsf{cap}_k^{\lambda} \longrightarrow \mathsf{param}_k^{m}} \text{ [L-StaCap]}$$

$$\frac{i \rightarrow_{\lambda} m \quad m \in Static \quad \#\mathsf{cap}^{\lambda} = n \geq 0}{\forall k \geq 1 : \mathsf{Larg}_k^{i,\lambda} \longrightarrow \mathsf{param}_{n+k}^{m}} \text{ [L-StaLArg]}$$

$$\frac{i \rightarrow_{\lambda} m}{\mathsf{ret}^{m} \longrightarrow \mathsf{LHS}^{i}} \text{ [L-Ret]}$$

$$\frac{i \rightarrow_{\lambda} m \quad \boxed{i \notin \textsc{StmLCallSite}}}{\forall k \geq 1 : \mathsf{arg}_k^{i} \longrightarrow \mathsf{Larg}_k^{i,\lambda}} \text{ [Arg2LArg]}$$

Fig. 38. Rules for handling functional call-graph edges. The boxed premise in [Arg2LArg] is added when integrated with stream modeling.

Compared to its original formulation in [18], this version introduces slight modifications to support more precise handling of dynamic dispatch and receiver passing for *target methods* of lambda expressions and method references. that are instance methods (see [L-Recv-I], [L-Recv-II], and [L-InsCall]). The primary complexity of this rule set lies in uniformly modeling receiver passing and environment-captured variable passing for both method references and lambdas. While method references have a constrained syntax that makes their target methods straightforward to identify, lambdas can capture local variables from the surrounding scope. To handle this issue, the compiler generates a synthetic method that encapsulates the lambda body, with any captured variables passed as parameters—this generated method becomes the target method of the functional object created by the lambda. Notably, the rules for handling a *target method* being an instance method require shifting the argument list to accommodate the receiver. This issue is reflected in the "-1" shift in [L-InsCap], [L-InsLArg-I], and [L-InsLArg-II], where captured variables (if any) are followed by the lambda's actual arguments when passed to the parameters of the target method. In contrast, static methods do not require this shift ([L-StaCap] and [L-StaAct]). These distinctions ensure accurate

modeling of function-invocation semantics in the presence of lambda expressions and method references.

$$\frac{m \in \{\texttt{Stream.reduce(Object,BinaryOperator)},\ \texttt{Stream.reduce(Object,BiFunction,BinaryOperator)},\ \texttt{Stream.collect(Supplier,BiConsumer,BiConsumer)},\ \texttt{Optional.orElseGet(Supplier)}\}}{\texttt{ret}^{m} \in \mathit{cutRet}}\ [\textsc{Cut-FStm}]$$

$$\frac{i \rightarrow_{call} m \quad \lambda \in pt(\texttt{arg}_2^i) \quad \texttt{Target}(\lambda) = m' \quad m = \texttt{Stream.iterate(Object,UnaryOperator)}}{\texttt{arg}_1^i \longrightarrow \texttt{Larg}_1^{*,\lambda} \quad \texttt{ret}^{m'} \longrightarrow \texttt{Larg}_1^{*,\lambda} \quad h^i_{\texttt{Strm}} \in pt_H(\texttt{LHS}^i) \quad h^i_{\texttt{Strm}} \Leftarrow \texttt{arg}_1^i \quad h^i_{\texttt{Strm}} \Leftarrow \texttt{ret}^{m'}}\ [\textsc{Iterate}]$$

$$\frac{i \rightarrow_{call} m \quad \lambda \in pt(\texttt{arg}_1^i) \quad \texttt{Target}(\lambda) = m' \quad m \in \{\texttt{Stream.generate}, \texttt{IntStream.mapToObj}, \texttt{LongStream.mapToObj}, \texttt{DoubleStream.mapToObj}\}}{h^i_{\texttt{Strm}} \in pt_H(\texttt{LHS}^i) \quad h^i_{\texttt{Strm}} \Leftarrow \texttt{ret}^{m'}}\ [\textsc{Generate}]$$

$$\frac{i \rightarrow_{call} m \quad m \in \{\texttt{Stream.map}, \texttt{Optional.map}\} \quad \lambda \in pt(\texttt{arg}_1^i) \quad \texttt{Target}(\lambda) = m' \quad h^j_{\texttt{Strm}} \in pt_H(\texttt{arg}_0^i)}{h^j_{\texttt{Strm}} \Rrightarrow \texttt{Larg}_1^{*,\lambda} \quad h^i_{\texttt{Strm}} \in pt_H(\texttt{LHS}^i) \quad h^i_{\texttt{Strm}} \Leftarrow \texttt{ret}^{m'}}\ [\textsc{Map}]$$

$$\frac{i \rightarrow_{call} m \quad m \in \{\texttt{Stream.flatMap}, \texttt{Optional.flatMap}\} \quad \lambda \in pt(\texttt{arg}_1^i) \quad \texttt{Target}(\lambda) = m' \quad h^j_{\texttt{Strm}} \in pt_H(\texttt{arg}_0^i) \quad h^k_{\texttt{Strm}} \in pt_H(\texttt{ret}^{m'})}{h^j_{\texttt{Strm}} \Rrightarrow \texttt{Larg}_1^{*,\lambda} \quad h^i_{\texttt{Strm}} \in pt_H(\texttt{LHS}^i) \quad h^k_{\texttt{Strm}} \lhd h^i_{\texttt{Strm}}}\ [\textsc{FlatMap}]$$

$$\frac{i \rightarrow_{call} m \quad m = \texttt{Stream.reduce(BinaryOperator)} \quad \lambda \in pt(\texttt{arg}_1^i) \quad \texttt{Target}(\lambda) m' \quad h^j_{\texttt{Strm}} \in pt_H(\texttt{arg}_0^i)}{h^j_{\texttt{Strm}} \Rrightarrow \texttt{Larg}_1^{l,\lambda} \quad h^j_{\texttt{Strm}} \Rrightarrow \texttt{Larg}_2^{*,\lambda} \quad h^i_{\texttt{Strm}} \in pt_H(\texttt{LHS}^i) \quad h^j_{\texttt{Strm}} \lhd h^i_{\texttt{Strm}} \quad h^i_{\texttt{Strm}} \Leftarrow \texttt{ret}^{m'} \quad \texttt{ret}^{m'} \longrightarrow \texttt{Larg}_1^{*,\lambda}}\ [\textsc{Reduce-I}]$$

$$\frac{i \rightarrow_{call} m \quad m = \texttt{Stream.reduce(Object,BinaryOperator)} \quad \lambda \in pt(\texttt{arg}_2^i) \quad \texttt{Target}(\lambda) = m' \quad h^j_{\texttt{Strm}} \in pt_H(\texttt{arg}_0^i)}{h^j_{\texttt{Strm}} \Rrightarrow \texttt{Larg}_2^{*,\lambda} \quad \texttt{arg}_1^i \longrightarrow \texttt{LHS}^i \quad \texttt{arg}_1^i \longrightarrow \texttt{Larg}_1^{*,\lambda} \quad \texttt{ret}^{m'} \longrightarrow \texttt{Larg}_1^{*,\lambda} \quad \texttt{ret}^{m'} \longrightarrow \texttt{LHS}^i}\ [\textsc{Reduce-II}]$$

$$\frac{\begin{array}{c} i \rightarrow_{call} m \quad \lambda_1 \in pt(\texttt{arg}_2^i) \quad \texttt{Target}(\lambda_1) = m' \quad \lambda_2 \in pt(\texttt{arg}_3^i) \quad \texttt{Target}(\lambda_2) = m'' \quad h \in pt_H(\texttt{arg}_0^i) \\ m = \texttt{Stream.reduce(Object,BiFunction,BinaryOperator)} \end{array}}{\begin{array}{c} \texttt{arg}_1^i \longrightarrow \texttt{LHS}^i \quad \texttt{ret}^{m'} \longrightarrow \texttt{LHS}^i \quad \texttt{ret}^{m''} \longrightarrow \texttt{LHS}^i \\ \texttt{arg}_1^i \longrightarrow \texttt{Larg}_1^{*,\lambda_1} \quad \texttt{ret}^{m'} \longrightarrow \texttt{Larg}_1^{*,\lambda_1} \quad \texttt{ret}^{m''} \longrightarrow \texttt{Larg}_1^{*,\lambda_1} \quad h \Rrightarrow \texttt{Larg}_2^{*,\lambda_1} \\ \texttt{arg}_1^i \longrightarrow \texttt{Larg}_1^{*,\lambda_2} \quad \texttt{ret}^{m'} \longrightarrow \texttt{Larg}_1^{*,\lambda_2} \quad \texttt{ret}^{m''} \longrightarrow \texttt{Larg}_1^{*,\lambda_2} \\ \texttt{arg}_1^i \longrightarrow \texttt{Larg}_2^{*,\lambda_2} \quad \texttt{ret}^{m'} \longrightarrow \texttt{Larg}_2^{*,\lambda_2} \quad \texttt{ret}^{m''} \longrightarrow \texttt{Larg}_2^{*,\lambda_2} \end{array}}\ [\textsc{Reduce-III}]$$

$$\frac{i \rightarrow_{call} m \quad h^j_{\texttt{Strm}} \in pt_H(\texttt{arg}_0^i) \quad \lambda_1 \in pt(\texttt{arg}_1^i) \quad \texttt{Target}(\lambda_1) = m' \quad \lambda_2 \in pt(\texttt{arg}_2^i) \quad \texttt{Target}(\lambda_2) = m'' \quad \lambda_3 \in pt(\texttt{arg}_3^i) \quad \texttt{Target}(\lambda_3) = m''' \quad m = \texttt{Stream.collect(Supplier,BiConsumer,BiConsumer)}}{h^j_{\texttt{Strm}} \Rrightarrow \texttt{Larg}_2^{*,\lambda_2} \quad \texttt{ret}^{m'} \longrightarrow \texttt{LHS}^i \quad \texttt{ret}^{m'} \longrightarrow \texttt{Larg}_1^{*,\lambda_2} \quad \texttt{ret}^{m'} \longrightarrow \texttt{Larg}_1^{*,\lambda_3} \quad \texttt{ret}^{m'} \longrightarrow \texttt{Larg}_2^{*,\lambda_3}}\ [\textsc{Collect}]$$

$$\frac{i \rightarrow_{call} m \quad m = \texttt{Optional.orElseGet(Supplier)} \quad \lambda \in pt(\texttt{arg}_1^i) \quad \texttt{Target}(\lambda) = m' \quad h^j_{\texttt{Strm}} \in pt_H(\texttt{arg}_0^i)}{\texttt{ret}^{m'} \longrightarrow \texttt{LHS}^i \quad h^j_{\texttt{Strm}} \Rrightarrow \texttt{LHS}^i}\ [\textsc{OrElse}]$$

$$\frac{i \rightarrow_{call} m \quad \lambda \in pt(\texttt{arg}_1^i) \quad h^j_{\texttt{Strm}} \in pt_H(\texttt{arg}_0^i) \quad m \in \{\texttt{Stream.forEach}, \texttt{Stream.forEachOrdered}, \texttt{Stream.peek}, \texttt{Optional.ifPresent}\}}{h^j_{\texttt{Strm}} \Rrightarrow \texttt{Larg}_1^{*,\lambda}}\ [\textsc{Foreach}]$$

Fig. 39. Rules for *functional* stream operations, where "∗" indicates a wildcard.

The complete rules for each *functional* stream operation are provided in Figure 39. The rule [Cut-FStm] plays a role similar to [CutCont] in Figure 11, both should be performed prior to the pointer analysis to determine the set *cutRet*. The remaining rules in Figure 39 handle each *functional* stream operation. Some operations can be covered by a single rule because, in terms of our analysis, their semantics are equivalent. Notably, `IntStream.boxed`, `LongStream.boxed`, and `DoubleStream.boxed` are

implicitly handled by the rule [Generate], because these methods internally invoke the corresponding `mapToObj` method. The most intricate cases are the rules for the three variants of `Stream.reduce`, which differ in their argument counts and semantics. These require careful modeling to capture how the arguments interact with each other, and with the host representing the stream pipeline instance. For example, the rule [Reduce-III] models the most complex variant, `Stream.reduce(Object, BiFunction, BinaryOperator)`, which takes three arguments:

- *identity*: This argument is the initial value of the reduction. We model its flow by connecting it to actual arguments of the two lambda functions that take in partial-reduction results (λ_1 and λ_2), i.e., $\texttt{arg}_1^i \rightarrow \texttt{Larg}_1^{*,\lambda_1}$, $\texttt{arg}_1^i \rightarrow \texttt{Larg}_1^{*,\lambda_2}$, and $\texttt{arg}_1^i \rightarrow \texttt{Larg}_2^{*,\lambda_2}$. It also serves as the default result if the stream is empty, so $\texttt{arg}_1^i \rightarrow \texttt{LHS}^i$.
- *accumulator*: This function takes two arguments—a partial-reduction result and the next stream element. The stream element is passed via $h \Rrightarrow \texttt{Larg}_2^{*,\lambda_1}$. The function returns a new partial result, which is connected back to wherever a partial-reduction result is needed: to *accumulator* in subsequent reduction steps ($\texttt{ret}^{m'} \rightarrow \texttt{Larg}_1^{*,\lambda_1}$), to the *combiner*'s arguments ($\texttt{ret}^{m'} \rightarrow \texttt{Larg}_1^{*,\lambda_2}$ and $\texttt{ret}^{m'} \rightarrow \texttt{Larg}_2^{*,\lambda_2}$), and to the overall result ($\texttt{ret}^{m'} \rightarrow \texttt{LHS}^i$).
- *combiner*: This function merges two partial results when the reduction is parallelized. The return value of it is also a partial result, and flows into wherever a partial result is needed: it is connected back to the final result ($\texttt{ret}^{m''} \rightarrow \texttt{LHS}^i$), to the *accumulator* ($\texttt{ret}^{m''} \rightarrow \texttt{Larg}_1^{*,\lambda_1}$), and recursively to the *combiner* itself ($\texttt{ret}^{m''} \rightarrow \texttt{Larg}_1^{*,\lambda_2}$ and $\texttt{ret}^{m''} \rightarrow \texttt{Larg}_2^{*,\lambda_2}$).

Among the remaining rules, [FlatMap] is the most interesting. Here, the return value of the lambda's target method is itself a stream, and we need to identify all hosts of kind Strm that are associated with that return value ($h^k_{\text{Strm}} \in pt_H(\texttt{ret}^{m'})$). We then add a preorder $\lhd$ between each of these hosts and the respective result host associated with the LHS variable ($h^i_{\text{Strm}} \in pt_H(\texttt{LHS}^i)$), indicating that we propagate the Sources of the former to the newly generated host representing the result stream, so that the "flattened" semantics is accurately modeled.

D Ensuring Soundness for Handling Streams

In this appendix, we introduce additional mechanisms and conservative fallbacks to ensure soundness for our handling of Java streams in Cut-Shortcut^S.

Unmodeled Operations. Although our implementation fully models all relevant stream operations up to Java 8, newer APIs (e.g., `Stream.mapMulti` introduced in Java 16) may not be covered. To maintain soundness, we conservatively handle such cases using rule [UnModeled] in Figure 40. If a method is not included in the set of modeled operations (ModeledStmOps), but its return type is a subtype of `Stream` or `Optional`, we generate a placeholder host h^i_{Strm} and record it in *conservH*, indicating that it must be treated conservatively. Rule [ConservHost] in Figure 36 then ensures those otherwise-suppressed return edges for the relevant stream pipelines are preserved, and soundness is maintained.

$$\frac{i \rightarrow_{call} m \quad m \notin \text{ModeledStmOps} \quad \mathcal{T}(\texttt{ret}^m) <: \texttt{Stream} \vee \texttt{Optional}}{h^i_{\text{Strm}} \in pt_H(\texttt{LHS}^i) \quad h^i_{\text{Strm}} \in \mathit{conservH}} \text{ [UnModeled]}$$

Fig. 40. The rule for handling unmodeled stream operations.

Functional Operations. To ensure that the integration of our handling of streams with lambda analysis [18] is sound, we introduce the rule [ConservFunc] in Figure 41 to disable PFG edge suppression for *functional* stream operations when the associated host is conservatively handled. Specifically, [ConservFunc] add edges for the actual arguments of lambdas involved in *functional*

stream operations, only when the invocation receiver is associated with a host recorded in *conservH*. These edges connect call-site arguments ($\texttt{arg}_k^{i'}$) to the mock actual arguments ($\texttt{Larg}_k^{i',\lambda}$), and would otherwise be suppressed due to the boxed premise in [Arg2LArg] in Figure 25.

$$\frac{i \rightarrow_{call} m \quad \lambda \in pt(\texttt{arg}_{\_}^{i}) \quad i' \rightarrow_{\lambda} m' \quad h \in pt_H(\texttt{arg}_0^{i}) \quad h \in conservH}{\forall k \geq 1 : \texttt{arg}_k^{i'} \longrightarrow \texttt{Larg}_k^{i',\lambda}} \text{[ConservFunc]}$$

Fig. 41. The rule to disable PFG edge suppression for *functional* stream operations.

Transformation through Complex Collectors. Finally, we conservatively handle cases where `Stream.collect(Collector)` is invoked with a *complex* collector. Specifically, if the collector argument is not constructed via `Collector.toSet`, `Collector.toList`, or `Collector.toCollection` (determined by checking whether its allocation site belongs to SimpleCollectorAlloc, a set of instruction labels gathered by a naive pre-analysis over the `Collector` class), we conservatively generate one host per $c \in \mathbb{K}$ and associate them with the LHS variable. This association signals that suppression of the corresponding PFG edges must be disabled.

$$\frac{i \rightarrow_{call} m \quad m = \texttt{Stream.collect(Collector)} \quad o_j \in \texttt{arg}_1^{i} \quad j \notin \text{SimpleCollectorAlloc} \quad c \in \mathbb{K}}{h_c^{i} \in conservH} \text{[ConservCollect]}$$

Fig. 42. Rules for conservatively handling `Stream.collect` through complex collectors.

E The Recall Experiment Results for RQ1

We summarized the recall results in Section 8.1; here, Table 8 presents the detailed experimental data. All results can be reproduced using our accompanying artifact.

As mentioned in Section 8.1, we conducted a recall experiment to validate the soundness of Cut-Shortcut. For each benchmark, we recorded the reachable methods and call-graph edges observed during dynamic execution, and then measured how many of these were recalled by Cut-Shortcut and by other analyses. As shown in Table 8, Cut-Shortcut 's recall rate is comparable to, and in most cases no lower than, that of `2obj`, `2type`, Zipper^e-`2obj`, and Zipper^e-`2type`. The only exception is the call-edge metric for `jython`, where Cut-Shortcut 's recall rate is slightly lower than all baselines. Note that a static analysis cannot achieve 100% recall because, in general, dynamic features—like reflection–cannot be soundly analyzed by a static analysis.

We further examined the reachable methods and call-graph edges that Cut-Shortcut missed (relative to the dynamic results) but that were not missed by any of the compared analyses. Items missed by both Cut-Shortcut and at least one baseline are excluded from consideration, because such misses cannot be attributed to Cut-Shortcut. Across all benchmarks, Cut-Shortcut missed only 1 reachable method and 8 call-graph edges. Manual inspection revealed that all of these were incorrectly recorded by the instrumentation tool rather than being true misses. Therefore, we conclude that the set of dynamically recorded reachable methods and call-graph edges recalled by existing analyses can also be recalled by Cut-Shortcut. This result indicates that our implementation of Cut-Shortcut does not introduce additional unsoundness compared to the analyses we compares.

Table 8. Recall results against dynamic ground truth on 10 benchmarks.

Program	Dynamic		Analysis	#reach-mtd		#call-edge	
	#reach-mtd	#call-edge		Recall	Rate	Recall	Rate
eclipse	8093	20916	CI	7083	87.52%	18068	86.38%
			2obj	7078	87.46%	18053	86.31%
			2type	7078	87.46%	18054	86.32%
			Zipper^e-2obj	7078	87.46%	18056	86.33%
			Zipper^e-2type	7078	87.46%	18059	86.34%
			Cut-Shortcut	7080	87.48%	18057	86.33%
freecol	19387	57455	CI	19126	98.65%	56612	98.53%
			2obj	—	—	—	—
			2type	—	—	—	—
			Zipper^e-2obj	19124	98.64%	56605	98.52%
			Zipper^e-2type	19126	98.65%	56606	98.52%
			Cut-Shortcut	19126	98.65%	56606	98.52%
briss	14244	39575	CI	14009	98.35%	38746	97.91%
			2obj	—	—	—	—
			2type	—	—	—	—
			Zipper^e-2obj	14001	98.29%	38702	97.79%
			Zipper^e-2type	14001	98.29%	38706	97.80%
			Cut-Shortcut	14001	98.29%	38705	97.80%
hsqldb	2733	6295	CI	2608	95.43%	5856	93.03%
			2obj	—	—	—	—
			2type	2607	95.39%	5846	92.87%
			Zipper^e-2obj	2607	95.39%	5846	92.88%
			Zipper^e-2type	2607	95.39%	5850	92.93%
			Cut-Shortcut	2607	95.39%	5850	92.93%
jedit	6028	13203	CI	5859	97.20%	12714	96.30%
			2obj	5853	97.10%	12695	96.15%
			2type	5853	97.10%	12695	96.15%
			Zipper^e-2obj	5853	97.10%	12695	96.15%
			Zipper^e-2type	5853	97.10%	12698	96.18%
			Cut-Shortcut	5853	97.10%	12695	96.15%
gruntspud	14543	42099	CI	14359	98.73%	41441	98.44%
			2obj	—	—	—	—
			2type	—	—	—	—
			Zipper^e-2obj	14358	98.73%	41432	98.42%
			Zipper^e-2type	14358	98.73%	41435	98.42%
			Cut-Shortcut	14358	98.73%	41433	98.42%
soot	4372	16452	CI	4288	98.08%	16232	98.66%
			2obj	—	—	—	—
			2type	—	—	—	—
			Zipper^e-2obj	4287	98.06%	16229	98.64%
			Zipper^e-2type	4288	98.08%	16229	98.64%
			Cut-Shortcut	4287	98.06%	16229	98.64%
columba	6757	14689	CI	6674	98.77%	14369	97.82%
			2obj	—	—	—	—
			2type	—	—	—	—
			Zipper^e-2obj	6674	98.77%	14369	97.82%
			Zipper^e-2type	6674	98.77%	14369	97.82%
			Cut-Shortcut	6674	98.77%	14369	97.82%
jython	4835	35970	CI	4243	87.76%	11642	32.37%
			2obj	—	—	—	—
			2type	—	—	—	—
			Zipper^e-2obj	4241	87.71%	11636	32.35%
			Zipper^e-2type	4241	87.71%	11636	32.35%
			Cut-Shortcut	4241	87.71%	11629	32.33%
findbugs	5857	14075	CI	5808	99.16%	13901	98.76%
			2obj	5808	99.16%	13900	98.76%
			2type	5808	99.16%	13900	98.76%
			Zipper^e-2obj	5808	99.16%	13901	98.76%
			Zipper^e-2type	5808	99.16%	13901	98.76%
			Cut-Shortcut	5808	99.16%	13900	98.76%