A CFL-Reachability Formulation of Callsite-Sensitive Pointer Analysis with Built-In On-The-Fly Call Graph Construction

Dongjie He[1]
University of New South Wales, Sydney, Australia
Chongqing University, China

Jingbo Lu[1]
University of New South Wales, Sydney, Australia
Shanghai Sectrend Information Technology Co., Ltd, China

Jingling Xue
University of New South Wales, Sydney, Australia

Abstract

In object-oriented languages, the traditional CFL-reachability formulation for k-callsite-sensitive pointer analysis (kCFA) focuses on modeling field accesses and calling contexts, but it relies on a separate algorithm for call graph construction. This division can result in a loss of precision in kCFA, a problem that persists even when using the most precise call graphs, whether pre-constructed or generated on the fly. Moreover, pre-analyses based on this framework aiming to improve the efficiency of kCFA may inadvertently reduce its precision, due to the framework's lack of native call graph construction, essential for precise analysis.

Addressing this gap, this paper introduces a novel CFL-reachability formulation of kCFA for Java, uniquely integrating on-the-fly call graph construction. This advancement not only addresses the precision loss inherent in the traditional CFL-reachability-based approach but also enhances its overall applicability. In a significant secondary contribution, we present the first precision-preserving pre-analysis to accelerate kCFA. This pre-analysis leverages selective context sensitivity to improve the efficiency of kCFA without sacrificing its precision. Collectively, these contributions represent a substantial step forward in pointer analysis, offering both theoretical and practical advancements that could benefit future developments in the field.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases Pointer Analysis, CFL Reachability, Call Graph Construction

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.18

Supplementary Material *Software (Artifact)*: `https://doi.org/10.5281/zenodo.11061892` [15]

Funding ARC Grants DP230102871, DP240103194, and the Fundamental Research Funds for the Central Universities of Ministry of Education of China (No. 2024CDJXY015).

Acknowledgements We thank the anonymous reviewers for their constructive comments.

1 Introduction

Pointer analysis is fundamental to numerous static analyses, including program understanding, program verification, security analysis, compiler optimization, and symbolic execution. Over the past two decades, *k-callsite-sensitivity* [49], which distinguishes method contexts on their k-most-recent callsites, has emerged as a prevalent context abstraction in both whole-program [5, 60, 40] and demand-driven [53, 48, 62] pointer analyses for Java programs.

[1] The first two authors contributed equally to this work.

38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 18; pp. 18:1–18:29
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Traditionally, k-callsite-sensitive pointer analysis, abbreviated to kCFA (Control-Flow Analysis) [49], is either inclusion-based [1] or founded on context-free language (CFL) reachability [44]. The inclusion-based formulation for kCFA [22, 57] has been incorporated into several pointer analysis frameworks for Java [40, 59, 60, 5, 17]. In this approach, a program's statements are represented as points-to set constraints. The methods' calling contexts are delineated by parameterizing these constraints with context abstractions. Often, the call graph for the program is constructed dynamically, i.e., on the fly to maximize precision and efficiency [11, 47, 26, 27, 50]. Conversely, the CFL-reachability formulation for kCFA [53] plays a pivotal role in the development of a diverse array of pointer analysis algorithms. These include demand-driven pointer/alias analysis [53, 64, 62, 48], context transformations [57], library-code summarization [48], and selective context-sensitivity [33]. In this approach, a program's points-to information is determined by resolving a graph reachability problem within a specifically constructed *pointer assignment graph* (PAG) [26]. This CFL-reachability formulation involves analyzing the intersection of two context-free languages (CFLs), denoted as $L_{FC} = L_F \cap L_C$, where L_F describes field accesses as balanced parentheses and L_C enforces callsite-sensitivity by matching method calls and returns, also represented through balanced parentheses [53]. However, this formulation employs a distinct, external algorithm for call graph construction, further elaborated in Section 2.

In comparison to the inclusion-based approach, the L_{FC}-based CFL-reachability formulation for kCFA suffers from two major limitations, primarily due to its reliance on a separate algorithm for call graph construction. Firstly, this segregation can lead to a decrease in precision within kCFA, a problem that persists regardless of whether the call graphs are pre-constructed or generated on the fly. Secondly, certain pre-analyses, such as SELECTX [33], aim to enhance kCFA's efficiency through the L_{FC}-based CFL-reachability formulation. However, these pre-analyses might unintentionally compromise its precision, undermining the overall effectiveness of the pointer analysis.

The primary contribution of this research lies in addressing the aforementioned limitations by introducing a new CFL-reachability formulation of kCFA. This novel formulation, for the first time, demonstrates the feasibility of specifying kCFA entirely through CFL-reachability, eliminating the need for a separate call graph algorithm. Our approach utilizes three CFLs, $L_{DCR} = L_D \cap L_C \cap L_R$, within a new PAG framework. Here, L_D extends beyond field accesses (as in L_F) to include dynamic dispatch, L_C maintains callsite-sensitivity as per previous formulation [53], and L_R introduces support for parameter passing required by its built-in on-the-fly call graph construction. Theoretically, we demonstrate for the first time that kCFA can be characterized as a specific type of context-sensitive language – the intersection of multiple CFLs. This is a notable distinction, as not all context-sensitive languages can be expressed in this manner [31, 25], underscoring the uniqueness of our approach. The subsequent sections will delve into the challenges of designing L_{DCR} and provide insights into our formulation's underpinnings.

As a secondary contribution of this research, we demonstrate the practical utility of L_{DCR} by introducing P3CTX, the first precision-preserving pre-analysis designed to accelerate kCFA in Java programs. Given the critical importance of precision in tasks such as software security analysis, our approach distinguishes itself as the preferable option. It provides a speed advantage without sacrificing precision. P3CTX employs an L_{DCR}-enabled selective context-sensitivity technique, further substantiating the correctness of L_{DCR}. In contrast, SELECTX [33], developed based on L_{FC} [53], invariably encounters precision loss, thus underscoring the superiority of our approach.

$$\frac{\texttt{x} = \textbf{new}\ \texttt{T}\ //\ O \quad ctx \in \texttt{MethodCtx(M)}}{\langle O, \lceil ctx \rceil_{hk} \rangle \in \texttt{PTS}(\texttt{x}, ctx)} \text{ [I-NEW]} \qquad \frac{\texttt{x} = \texttt{y} \quad ctx \in \texttt{MethodCtx(M)}}{\texttt{PTS}(\texttt{y}, ctx) \subseteq \texttt{PTS}(\texttt{x}, ctx)} \text{ [I-ASSIGN]}$$

$$\frac{\texttt{x} = \texttt{y.f} \quad ctx \in \texttt{MethodCtx(M)} \quad \langle O, htx \rangle \in \texttt{PTS}(\texttt{y}, ctx)}{\texttt{PTS}(O.\texttt{f}, htx) \subseteq \texttt{PTS}(\texttt{x}, ctx)} \text{ [I-LOAD]} \qquad \frac{\texttt{x.f} = \texttt{y} \quad ctx \in \texttt{MethodCtx(M)} \quad \langle O, htx \rangle \in \texttt{PTS}(\texttt{x}, ctx)}{\texttt{PTS}(\texttt{y}, ctx) \subseteq \texttt{PTS}(O.\texttt{f}, htx)} \text{ [I-STORE]}$$

$$\frac{\texttt{x} = \texttt{m}(a_1, \ldots, a_n)\ //\ \texttt{c} \quad ctx \in \texttt{MethodCtx(M)} \quad ctx' = \lceil \texttt{c} :: ctx \rceil_k}{ctx' \in \texttt{MethodCtx(m)} \quad \texttt{PTS}(\texttt{ret}^{\texttt{m}}, ctx') \subseteq \texttt{PTS}(\texttt{x}, ctx) \quad \forall i \in [1, n] : \texttt{PTS}(a_i, ctx) \subseteq \texttt{PTS}(p_i^{\texttt{m}}, ctx')} \text{ [I-SCALL]}$$

$$\frac{\begin{array}{c}\texttt{x} = \texttt{r.m}(a_1, \ldots, a_n)\ //\ \texttt{c} \quad ctx \in \texttt{MethodCtx(M)} \quad \langle O, htx \rangle \in \texttt{PTS}(\texttt{r}, ctx) \\ \texttt{t} = \texttt{DynTypeOf}(O) \quad \texttt{m}' = \texttt{dispatch}(\texttt{c}, \texttt{t}) \quad ctx' = \lceil \texttt{c} :: ctx \rceil_k\end{array}}{\begin{array}{c}ctx' \in \texttt{MethodCtx}(\texttt{m}') \quad \texttt{PTS}(\texttt{ret}^{\texttt{m}'}, ctx') \subseteq \texttt{PTS}(\texttt{x}, ctx) \\ \langle O, htx \rangle \in \texttt{PTS}(\texttt{this}^{\texttt{m}'}, ctx') \quad \forall i \in [1, n] : \texttt{PTS}(a_i, ctx) \subseteq \texttt{PTS}(p_i^{\texttt{m}'}, ctx')\end{array}} \text{ [I-VCALL]}$$

Figure 1 Inclusion-based formulation (M is the containing method of the statement being analyzed).

In summary, this paper makes the following two major contributions:

- A new CFL-reachability formulation of kCFA with built-in call graph construction.
- An L_{DCR}-enabled precision-preserving pre-analysis for accelerating kCFA with selective context-sensitivity. Compared with two state-of-the-art pre-analyses [33, 29], our pre-analysis enables better efficiency-precision trade-offs in several application scenarios.

The rest of this paper is organized as follows. Section 2 provides background knowledge and motivates the development of L_{DCR} by highlighting several design challenges. Section 3 introduces L_{DCR}, explaining how these challenges are addressed and offering insights into its design. Section 4 presents and evaluates, P3CTX, our L_{DCR}-enabled pre-analysis for accelerating kCFA. Section 5 discusses related work and Section 6 concludes the paper.

2 Background and Motivation

We start by reviewing the inclusion-based and traditional CFL-reachability L_{FC} formulations of kCFA (Section 2.1). Next, we use an example to illustrate their approaches to call graph construction, discuss L_{FC}'s limitations, and highlight the necessity of and challenges faced in designing L_{DCR}, a new CFL-reachability formulation with an integrated on-the-fly call graph construction (Section 2.2).

In our formalization, we consider a simplified Java language with six types of statements: New for object creation ("`x = new T // O`"); Assign for variable assignments ("`x = y`"); Load for retrieving field values ("`x = y.f`"); Store for assigning values to fields ("`x.f = y`"); Virtual Calls for instance method calls ("$\texttt{x} = \texttt{r.m}(a_1, \ldots, a_n)$ `// c`"); and Static Calls for static method calls ("$\texttt{x} = \texttt{m}(a_1, \ldots, a_n)$ `// c`"). Here, `O` identifies the unique abstract object created by a particular New statement, `x` and `y` are local variables, and `c` identifies a callsite. For a virtual call $\texttt{r.m}(a_1, \ldots, a_n)$, we write $\texttt{this}^{\texttt{m}'}$, $p_i^{\texttt{m}'}$ and $\texttt{ret}^{\texttt{m}'}$ as its "this" variable, i-th parameter and return variable for a virtual method $\texttt{m}'$ invoked at this callsite, respectively. For a static call $\texttt{m}(a_1, \ldots, a_n)$, only $p_i^{\texttt{m}}$ and $\texttt{ret}^{\texttt{m}}$ are relevant. In scenarios where method calls do not return a value, the flow from $\texttt{ret}^{\texttt{m}}$ to `x` is disregarded.

$$\frac{\mathtt{x} = \mathbf{new}\ \mathtt{T}\ //\ O}{O \xrightarrow{\mathsf{new}} \mathtt{x}}\ \text{[P-New]} \qquad \frac{\mathtt{x} = \mathtt{y}}{\mathtt{y} \xrightarrow{\mathsf{assign}} \mathtt{x}}\ \text{[P-Assign]} \qquad \frac{\mathtt{x} = \mathtt{y.f}}{\mathtt{y} \xrightarrow{\mathsf{load[f]}} \mathtt{x}}\ \text{[P-Load]}$$

$$\frac{\mathtt{x.f} = \mathtt{y}}{\mathtt{y} \xrightarrow{\mathsf{store[f]}} \mathtt{x}}\ \text{[P-Store]} \qquad \frac{\mathtt{x} = \mathtt{m}(a_1, \ldots, a_n)\ //\ \mathtt{c}}{\forall\, i \in [1, n] : a_i \xrightarrow[\hat{c}]{\mathsf{assign}} p_i^{\mathtt{m}} \quad \mathtt{ret}^{\mathtt{m}} \xrightarrow[\check{c}]{\mathsf{assign}} \mathtt{x}}\ \text{[P-SCall]}$$

$$\frac{\mathtt{x} = \mathtt{r.m}(a_1, \ldots, a_n)\ //\ \mathtt{c} \quad \mathtt{m}' \text{ is a target of this callsite}}{\mathtt{r} \xrightarrow[\hat{c}]{\mathsf{assign}} \mathtt{this}^{\mathtt{m}'} \quad \mathtt{ret}^{\mathtt{m}'} \xrightarrow[\check{c}]{\mathsf{assign}} \mathtt{x} \quad \forall\, i \in [1, n] : a_i \xrightarrow[\hat{c}]{\mathsf{assign}} p_i^{\mathtt{m}'}}\ \text{[P-VCall]}$$

Figure 2 Rules for building the PAG required by L_{FC}.

2.1 Background

2.1.1 Inclusion-based Formulation

Figure 1 gives the rules for such a formulation [22, 51, 57], where several auxiliary functions are used: (1) `MethodCtx` maintains the contexts used for analyzing a method, (2) `dispatch` resolves a virtual call to a target method, and (3) `PTS` records the points-to information found context-sensitively for a variable or an object's field. In kCFA, context sensitivity is achieved by parameterizing variables and objects with contexts as modifiers. A calling context of a method is abstracted by its last k callsites. Given a context $ctx = [c_1, \ldots, c_n]$ and a context element c, $c :: ctx$ stands for $[c, c_1, \ldots, c_n]$ and $\lceil ctx \rceil_k$ stands for $[c_1, \ldots, c_k]$.

Let us examine the six rules in Figure 1. In [I-New], hk represents the (heap) context length for a heap object, typically set as $hk = k - 1$ [51, 58, 20, 30]. [I-Assign], [I-Load], and [I-Store] address standard assignments and field accesses. [I-SCall] and [I-VCall] handle static and virtual calls, respectively. Let us explain [I-VCall] only. In this rule, $\mathtt{m}'$ is a target method dynamically resolved for a receiver object O (based on its dynamic type $\mathtt{t} = \mathtt{DynTypeOf}(O)$) at callsite $\mathtt{c}$. Thus, this rule is also responsible for performing on-the-fly call graph construction during the pointer analysis. In its conclusion, $ctx' \in \mathtt{MethodCtx}(\mathtt{m}')$ reveals how the contexts of a method are introduced. Initially, for the program being analyzed, its entry methods have only the empty context, e.g., $\mathtt{MethodCtx}(\text{“}\mathtt{main}\text{”}) = \{[\,]\}$. Importantly, the receiver variable $\mathtt{r}$ and the other arguments $a_1, \ldots, a_n$ are handled differently: a receiver object flows only to the method it dispatches, while the objects pointed to by $a_i (i \in [1, n])$ flow to all methods dispatched at this callsite.

2.1.2 L_{FC}-based CFL-Reachability Formulation

In L_{FC} [53], kCFA is solved by reasoning about CFL-reachability on a PAG representation [26]. Figure 2 gives six rules for building the PAG. For a PAG edge, its label above indicates whether it is an assignment or field access. There are two types of assign edges: *intra-procedural* (for modeling regular assignments without a below-edge label) and *inter-procedural* (for modeling parameter passing with a below-edge label representing a callsite).

In L_{FC}, passing arguments to parameters at both static and virtual callsites is modeled identically by using inter-procedural assign edges ([P-SCall] and [P-VCall]). For example, in [P-VCall], $\hat{c}$ ($\check{c}$) signifies an inter-procedural value-flow entering into (exiting from) $\mathtt{m}'$ at callsite c, where $\mathtt{m}'$ represents a virtual method discovered by a separate call graph construction algorithm (either in advance [9, 2, 55] or on the fly [54, 53]). Therefore, $\hat{c}$ ($\check{c}$) is also known as an *entry* (*exit*) *context*.

```
class A {
  void foo(D p) {
    Object v = p.f;
  }
}
class B extends A {
  void foo(D q) { }
}
class C extends A {
  void foo(D r) {}
}
class D { Object f; }
class O { }
static void bar(A x, O o) {
  D d = new D(); // D1
  d.f = o;
  x.foo(d); // c3
}
static void main() {
  O o1 = new O(); // O1
  O o2 = new O(); // O2
  A a = new A(); // A1
  A b = new B(); // B1
  bar(a, o1); // c1
  bar(b, o2); // c2
}
```

Figure 3 A motivating example.

For a PAG edge $x \xrightarrow[c]{\ell} y$, its *inverse edge*, which is omitted in Figure 2 but required by L_{FC}, is defined as $y \xrightarrow[\bar{c}]{\bar{\ell}} x$. For a below-edge label $\hat{c}$ or $\check{c}$, $\bar{\hat{c}} = \check{c}$ and $\bar{\check{c}} = \hat{c}$, implying that the concepts of entry and exit contexts for inter-procedural assign edges are swapped if they are traversed inversely.

L_{FC} is defined as the intersection of two distinct CFLs, $L_{FC} = L_F \cap L_C$, with L_F pertaining to the PAG's above-edge labels and L_C to its below-edge labels. L_F, a CFL over Σ_{L_F}, is created from above-edge labels. For each path p in the PAG, $L_F(p)$ is a string in $\Sigma^*_{L_F}$, made by sequentially concatenating p's above-edge labels. A node v is L_F*-reachable* from node u if a path p, termed L_F*-path*, exists from u to v such that $L_F(p) \in L_F$. L_C follows a similar definition, but with Σ_{L_C} comprising below-edge labels.

We give L_F and L_C below and illustrate both with an example in Section 2.2. L_F enforces field-sensitivity for field accesses by matching stores and loads as balanced parentheses:

$$\begin{array}{rcl} \text{flowsto} & \longrightarrow & \text{new flows}^* \\ \text{flows} & \longrightarrow & \text{assign} \mid \text{store[f] alias load[f]} \\ \text{alias} & \longrightarrow & \overline{\text{flowsto}}\ \text{flowsto} \\ \overline{\text{flowsto}} & \longrightarrow & \overline{\text{flows}}^*\ \overline{\text{new}} \\ \overline{\text{flows}} & \longrightarrow & \overline{\text{assign}} \mid \overline{\text{load[f]}}\ \text{alias}\ \overline{\text{store[f]}} \end{array} \tag{1}$$

Note that u alias v iff u $\overline{\text{flowsto}}$ O flowsto v for some object O. In addition, O flowsto v iff v $\overline{\text{flowsto}}$ O, meaning that $\overline{\text{flowsto}}$ actually represents the standard points-to relation.

L_C enforces callsite-sensitivity by matching "calls" and "returns" as balanced parentheses:

$$\begin{array}{rcl} \text{realizable} & \longrightarrow & \text{exit entry} \\ \text{exit} & \longrightarrow & \text{exit balanced} \mid \text{exit } \check{c} \mid \epsilon \\ \text{entry} & \longrightarrow & \text{entry balanced} \mid \text{entry } \hat{c} \mid \epsilon \\ \text{balanced} & \longrightarrow & \text{balanced balanced} \mid \hat{c} \text{ balanced } \check{c} \mid \epsilon \end{array} \tag{2}$$

A path p in the PAG of the program is *realizable* iff p is an L_C-path.

Finally, a variable v points to an object O iff there exists an L_{FC}*-path* p from O to v, such that $L_F(p) \in L_F$ (p is a flowsto-path) and $L_C(p) \in L_C$ (p is a realizable-path). Ignoring all balanced contexts, the contexts for v and O can be directly read off from p (Section 3.2.2).

Table 1 Points-to results for the program in Figure 3 computed by 2CFA according to Figure 1.

Method	Pointers	PTS	Method	Pointers	PTS
main()	$\langle \texttt{o1}, [\,]\rangle$	$\{\langle \texttt{O1}, [\,]\rangle\}$	bar()	$\langle \texttt{x}, [\texttt{c1}]\rangle$	$\{\langle \texttt{A1}, [\,]\rangle\}$
	$\langle \texttt{o2}, [\,]\rangle$	$\{\langle \texttt{O2}, [\,]\rangle\}$		$\langle \texttt{o}, [\texttt{c1}]\rangle$	$\{\langle \texttt{O1}, [\,]\rangle\}$
	$\langle \texttt{a}, [\,]\rangle$	$\{\langle \texttt{A1}, [\,]\rangle\}$		$\langle \texttt{d}, [\texttt{c1}]\rangle$	$\{\langle \texttt{D1}, [\texttt{c1}]\rangle\}$
	$\langle \texttt{b}, [\,]\rangle$	$\{\langle \texttt{B1}, [\,]\rangle\}$		$\langle \texttt{x}, [\texttt{c2}]\rangle$	$\{\langle \texttt{B1}, [\,]\rangle\}$
B:foo()	$\langle \texttt{this}, [\texttt{c3}, \texttt{c2}]\rangle$	$\{\langle \texttt{B1}, [\,]\rangle\}$		$\langle \texttt{o}, [\texttt{c2}]\rangle$	$\{\langle \texttt{O2}, [\,]\rangle\}$
	$\langle \texttt{q}, [\texttt{c3}, \texttt{c2}]\rangle$	$\{\langle \texttt{D1}, [\texttt{c2}]\rangle\}$		$\langle \texttt{d}, [\texttt{c2}]\rangle$	$\{\langle \texttt{D1}, [\texttt{c2}]\rangle\}$
A:foo()	$\langle \texttt{this}, [\texttt{c3}, \texttt{c1}]\rangle$	$\{\langle \texttt{A1}, [\,]\rangle\}$	Field	Pointers	PTS
	$\langle \texttt{p}, [\texttt{c3}, \texttt{c1}]\rangle$	$\{\langle \texttt{D1}, [\texttt{c1}]\rangle\}$	f	$\langle \texttt{D1.f}, [\texttt{c1}]\rangle$	$\{\langle \texttt{O1}, [\,]\rangle\}$
	$\langle \texttt{v}, [\texttt{c3}, \texttt{c1}]\rangle$	$\{\langle \texttt{O1}, [\,]\rangle\}$		$\langle \texttt{D1.f}, [\texttt{c2}]\rangle$	$\{\langle \texttt{O2}, [\,]\rangle\}$

2.2 Motivation

We begin with a motivating example (Section 2.2.1) and an inclusion-based framework featuring on-the-fly call graph construction (Section 2.2.2). We explore the limitations of L_{FC} without this feature (Section 2.2.3) and the challenges of developing L_{DCR} with it (Section 2.2.4). Transitioning from L_{FC} to L_{DCR} requires a new PAG representation specific to L_{DCR}.

2.2.1 Example

In Figure 3, classes `A`, `B`, `C`, `D`, and `O` are defined. `B` and `C`, subclasses of `A`, override the `foo()` method from `A`. The notation `T:m()` represents method `m()` in class `T`. The method `bar()` is a wrapper, storing the object pointed to by `o` in `D1.f`, and then invoking `A:foo()`, `B:foo()`, or `C:foo()` based on the dynamic type of object `x` points to. In `main()`, `O1`, `O2`, `A1`, and `B1` are created, in which `A1` and `O1` (`B1` and `O2`) are passed into `bar()` as its two arguments at callsite `c1` (`c2`).

2.2.2 Inclusion-based Formulation

Table 1 lists the points-to results computed for the program in Figure 3 by 2CFA following the rules in Figure 1. For `main()`, analyzed under $[\,]$, its points-to relations are obtained trivially. As for `bar()`, there are two calling contexts, $[\texttt{c1}]$ and $[\texttt{c2}]$. Under $[\texttt{c1}]$, we have $\texttt{PTS}(\texttt{x}, [\texttt{c1}]) = \{\langle \texttt{A1}, [\,]\rangle\}$, $\texttt{PTS}(\texttt{d}, [\texttt{c1}]) = \{\langle \texttt{D1}, [\texttt{c1}]\rangle\}$, and $\texttt{PTS}(\texttt{D1.f}, [\texttt{c1}]) = \texttt{PTS}(\texttt{o}, [\texttt{c1}]) = \{\langle \texttt{O1}, [\,]\rangle\}$. Then `A:foo()` is found to be the target invoked by `x.foo()` at callsite `c3` in line 17 ([I-VCall]). Thus, $\texttt{PTS}(\texttt{p}, [\texttt{c3}, \texttt{c1}]) = \{\langle \texttt{D1}, [\texttt{c1}]\rangle\}$ and $\texttt{PTS}(\texttt{v}, [\texttt{c3}, \texttt{c1}]) = \{\langle \texttt{O1}, [\,]\rangle\}$. Similarly, when `bar()` is analyzed under $[\texttt{c2}]$, we have $\texttt{PTS}(\texttt{x}, [\texttt{c2}]) = \{\langle \texttt{B1}, [\,]\rangle\}$. Thus, `x.foo()` at callsite `c3` is now resolved to `B:foo()`. Note that [I-VCall] supports on-the-fly call graph construction during the analysis and 2CFA is precise enough by not resolving `C:foo()` as a spurious target at `c3`.

2.2.3 L_{FC}-based Formulation

L_{FC} addresses kCFA using a separate call graph construction algorithm. This approach separates, both conceptually and algorithmically, the parameter passing at a virtual callsite from the dynamic dispatch process. The limitations arising from this separation are explored below, considering whether the call graph is pre-constructed or constructed on the fly.

In Figure 3, L_{FC} uses a PAG as shown in Figure 4, constructed with CHA [9], an imprecise yet fast and sound call graph algorithm. In this scenario, `C:foo()` is conservatively marked as a target method at callsite `c3` (line 17). However, as explained later, L_{FC} would exclude such spurious targets when employing a more precise call graph in its analysis.

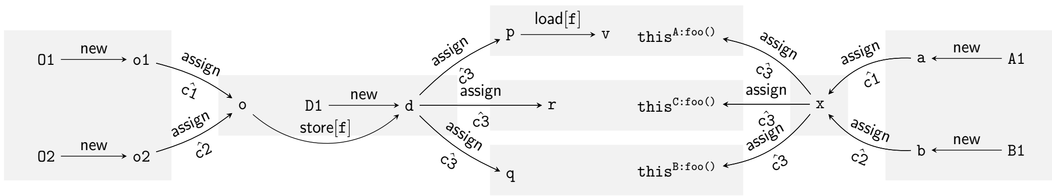

Figure 4 The PAG operated on by L_{FC} for the program given in Figure 3.

We analyze a specific traversal path leading to `d`, an argument in the call to `foo()` at callsite `c3` (line 17), originating from `O1` in `bar(a,o1)` under [`c1`] or `O2` in `bar(b,o2)` under [`c2`]. The subsequent task is to assign `d` to the appropriate parameter, based on the target method identified at this callsite: `p` for `A:foo()`, `q` for `B:foo()`, or `r` for `C:foo()`.

2.2.3.1 Using a Call Graph Constructed in Advance

Even if L_{FC} uses the most precise pre-built call graph, kCFA can still lose precision. For instance, at callsite `c3` (line 17) in Figure 3, both `A:foo()` and `B:foo()` are identified as possible target methods. This means `A:foo()` is always considered a target method, whether the call is from `bar(a,o1)` under [`c1`] or `bar(b,o2)` under [`c2`]. As a result, this scenario leads to the identification of two L_{FC}-paths:

$$\mathtt{O1} \xrightarrow{\text{new}} \mathtt{o1} \xrightarrow[\hat{\mathtt{c1}}]{\text{assign}} \mathtt{o} \xrightarrow{\text{store}[\mathtt{f}]} \mathtt{d} \xrightarrow{\overline{\text{new}}} \mathtt{D1} \xrightarrow{\text{new}} \mathtt{d} \xrightarrow[\hat{\mathtt{c3}}]{\text{assign}} \mathtt{p} \xrightarrow{\text{load}[\mathtt{f}]} \mathtt{v} \tag{3}$$

$$\mathtt{O2} \xrightarrow{\text{new}} \mathtt{o2} \xrightarrow[\hat{\mathtt{c2}}]{\text{assign}} \mathtt{o} \xrightarrow{\text{store}[\mathtt{f}]} \mathtt{d} \xrightarrow{\overline{\text{new}}} \mathtt{D1} \xrightarrow{\text{new}} \mathtt{d} \xrightarrow[\hat{\mathtt{c3}}]{\text{assign}} \mathtt{p} \xrightarrow{\text{load}[\mathtt{f}]} \mathtt{v} \tag{4}$$

Thus, in this L_{FC}-based pointer analysis, `v` is concluded to point to both `O1` and `O2`, despite `v` actually pointing only to `O1` as per 2CFA (Table 1), meaning that `O2` is spurious.

L_{FC}'s precision loss stems from its approach to parameter passing at virtual callsites ([P-VCall]), treating them similarly to static callsites ([P-SCall]) using inter-procedural assign edges, without accounting for CFL-reachability for specific receiver objects. As a result, this causes L_{FC} to overlook that the L_{FC}-path in Equation (3) and the L_{FC}-path in Equation (4) are relevant only when `x` points to `A1` at [`c1`] and `B1` at [`c2`], respectively.

If L_{FC} uses a less precise call graph, which is pre-built by, say, CHA [9], then `C:foo()` will also be identified as a target method at callsite `c3` (line 17), leading to `r` pointing to `D1` spuriously due to $\mathtt{D1} \xrightarrow{\text{new}} \mathtt{d} \xrightarrow[\hat{\mathtt{c3}}]{\text{assign}} \mathtt{r}$. However, `r`'s points-to set is actually empty as per 2CFA (not listed in Table 1).

2.2.3.2 Using a Call Graph Constructed On the Fly

When `d` is reached at callsite `c3` in line 17 of Figure 3, using a call graph constructed on the fly as in demand-driven analyses [53, 62, 48], where methods invoked at a virtual callsite are context-specific, enables us to discern that the path in Equation (3) is an L_{FC}-path, while that in Equation (4) is not. This precision ensures that `v` points only to `O1`. In the first path, `x` points to `A1` under context [`c1`], identifying `A:foo()` as the target at `c3`. The path $\xrightarrow[\hat{\mathtt{c3}}]{\text{assign}} \mathtt{p} \xrightarrow{\text{load}[\mathtt{f}]} \mathtt{v}$ confirms that `v` points to `O1`. In the second path, reaching `d` under [`c2`] leads to `B:foo()` at `c3` (with `x` pointing to `B1`), blocking the same path.

While L_{FC} can address kCFA on-demand more accurately than a pre-built call graph, precision loss may still occur in scenarios where a callsite has multiple dispatch targets under a common context. For example, in Figure 5 (where classes `E`, `F`, and `G` are renamed

```
class E {
  void foo(G p) {
    Object v = p.g;
}}
class F extends E {
  void foo(G q) { }
}
class G { Object g; }
G w = new G(); // G1
if (...) {
  E e1 = new E(); // E1
  w.g = e1;
} else {
  F f1 = new F(); // F1
  w.g = f1;
}
E x = w.g;
x.foo(null); // c
```

Figure 5 A small example.

from classes `A`, `B`, and `D` in Figure 3 to prevent name collisions), using a separate call graph construction algorithm to identify all potential target methods at "`x.foo(null)`" under any context results in the discovery of both `E:foo()` and `F:foo()`. Subsequent analysis of CFL-reachability with L_{FC} yields:

$$\mathtt{E1} \xrightarrow{\mathsf{new}} \mathtt{e1} \xrightarrow{\mathsf{store}[g]} \mathtt{w} \xrightarrow{\overline{\mathsf{new}}} \mathtt{G1} \xrightarrow{\mathsf{new}} \mathtt{w} \xrightarrow{\mathsf{load}[g]} \mathtt{x} \xrightarrow[\hat{c}]{\mathsf{assign}} \mathtt{this}^{\mathtt{E:foo()}} \tag{5}$$

$$\mathtt{F1} \xrightarrow{\mathsf{new}} \mathtt{f1} \xrightarrow{\mathsf{store}[g]} \mathtt{w} \xrightarrow{\overline{\mathsf{new}}} \mathtt{G1} \xrightarrow{\mathsf{new}} \mathtt{w} \xrightarrow{\mathsf{load}[g]} \mathtt{x} \xrightarrow[\hat{c}]{\mathsf{assign}} \mathtt{this}^{\mathtt{E:foo()}} \tag{6}$$

Therefore, both `E1` and `F1` will flow to $\mathtt{this}^{\mathtt{E:foo}}$ although `F1` is spurious by [I-VCall]. Similarly, both `E1` and `F1` will flow to $\mathtt{this}^{\mathtt{F:foo}}$ with `E1` being spurious.

L_{FC}'s precision loss stems from treating the receiver variable the same as other arguments ([P-VCall] in Figure 2), in contrast to the inclusion-based approach ([I-VCall] in Figure 1). Attempting to eliminate spurious receiver objects like `F1` for `E:foo()` informally, outside the specifications of L_{FC} or any call graph construction algorithm, is an ad hoc solution. This problem has persisted in the L_{FC} on-demand algorithm for kCFA [53], released as part of the Soot compiler [59] and used by many other researchers [61, 48], in the last 15 years.

2.2.3.3 Discussion

In addressing kCFA, L_{FC} depends on an external algorithm for call graph construction. This approach not only leads to the precision loss in kCFA as previously mentioned, but also presents another limitation: L_{FC} is unable to track all value-flow paths involved in method dispatch, whether the call graph is constructed beforehand or generated on-the-fly.

In analyzing "`x.foo(d)`" in line 17 of Figure 3, for parameter passing of `d` at the callsite as per [I-VCall], it is necessary to first identify methods dispatched on the receiver objects that `x` points to, then proceed with parameter passing (from `d` to `p` for `A:foo()`, and `d` to `q` for `B:foo()`). However, in L_{FC}, parameter passing, achieved through inter-procedural assign edges ([P-VCall]), is conceptually and algorithmically detached from dynamic dispatch at the callsite. It does not relate this process via CFL-reachability to its receiver objects, a limitation also evident in the PAG shown in Figure 4.

The limitations of L_{FC} indicate that its pre-analyses, designed to boost kCFA efficiency, can unintentionally compromise its precision. For example, Selectx [33] aims to accelerate kCFA through selective context-sensitivity with L_{FC}, often leading to reduced precision.

2.2.4 L_{DCR} : Challenges and Our Solution

In developing L_{DCR}, it is crucial to facilitate CFL-reachability for parameter passing in line with kCFA. For a virtual call $\texttt{r.m}(a_1, \ldots, a_n)$ at callsite c, passing an argument, denoted a, to its corresponding parameter p in a yet-to-be-discovered target method m' under context $\mathbf{C}$ involves establishing a CFL-reachability path in a PAG representation, starting from a, through receiver variable r for dynamic dispatch (based on the dynamic type of the object pointed to by r under $\mathbf{C}$), and ending at p. Linking a to r, especially when $a \neq \texttt{r}$, is complex. Additionally, in CFL-reachability, some context elements in $\mathbf{C}$ are consumed, i.e., matched during dynamic dispatch and must be restored for passing a to p under the same context $\mathbf{C}$. We identify three key challenges in handling this complex parameter-passing task:

- **CHL1**. How do we precisely pass r to the “this” variable of a target method m' invoked at callsite c, avoiding the precision loss as illustrated in Figure 5?
- **CHL2**. How do we establish a CFL-reachability path in a PAG representation of the program from a_i to p_i, passing through r to trigger dynamic dispatch during parameter passing, where p_i is the i-th parameter of a target method m' discovered at callsite c under $\mathbf{C}$?
- **CHL3**. How do we ensure the passage of a_i to p_i for the target method m' invoked at callsite c with a context abstraction that accurately characterizes parameter passing for callsite c under $\mathbf{C}$?

In our approach, illustrated using our motivating example (Figure 3), L_{DCR} is applied to a novel PAG representation depicted in Figure 7, distinct from the PAG used by L_{FC} (Figure 4). In this new formulation, we demonstrate that v points exclusively to O1, attributable to a unique path from O1 to v:

$$\texttt{O1} \xrightarrow{\textsf{new[O]}} \texttt{o1} \xrightarrow[\hat{\texttt{c1}}]{\textsf{assign}} \texttt{o} \xrightarrow{\textsf{store[f]}} \texttt{d} \xrightarrow{\overline{\textsf{new[D]}}} \texttt{D1} \xrightarrow{\textsf{new[D]}} \texttt{d} \xrightarrow[\boxed{\hat{\texttt{c3}}}]{\textsf{store[1]}} \texttt{x} \xrightarrow[\check{\texttt{c1}}]{\overline{\textsf{assign}}} \texttt{a} \xrightarrow{\overline{\textsf{new[A]}}} \texttt{A1}$$
$$\xrightarrow{\textsf{new[A]}} \texttt{a} \xrightarrow[\hat{\texttt{c1}}]{\textsf{assign}} \texttt{x} \xrightarrow[\boxed{\check{\texttt{c3}}}]{\textsf{assign}} \texttt{x\#c3} \xrightarrow[\hat{\texttt{c3}}]{\textsf{dispatch[A]}} \texttt{this}^{\texttt{A:foo()}} \xrightarrow{\textsf{load[1]}} \texttt{p} \xrightarrow{\textsf{load[f]}} \texttt{v} \tag{7}$$

The technical specifics of this path will be further elaborated in Section 3.

This path represents the flow of O1 to v through two calls, c1 (line 24) and c3 (line 17). Focusing on parameter passing of d at c3 under context $\mathbf{C} = [\texttt{c1}]$, where A:foo() is the sole target, L_{DCR} employs a more indirect approach than L_{FC}'s direct inter-procedural assign edge $d \xrightarrow[\hat{\texttt{c3}}]{\textsf{assign}} \texttt{p}$. L_{DCR} dynamically identifies dispatch targets in the path from d to p using a sequence of PAG edges. To address **CHL1**, we match new[A] with dispatch[A]. For **CHL2**, d is stored in a special field of x to initiate dynamic dispatch, then loaded from the same field of $\texttt{this}^{\texttt{A:foo()}}$ into p (highlighted in ■). Afterwards, dynamic dispatch under $\mathbf{C} = [\texttt{c1}]$ is performed similarly to L_{FC} (highlighted in ■). To tackle **CHL3**, d is passed to p under context [c3,c1], where c3 denotes the callsite and c1 the context for A1 flowing into x (highlighted in ■). The importance of the two boxed below-edge labels, $\boxed{\hat{\texttt{c3}}}$ and $\boxed{\check{\texttt{c3}}}$, in meeting **CHL3** will be elaborated upon in Section 3.

3 L_{DCR} : Design and Insights

When tackling a CFL-reachability problem, the selection of CFLs and their corresponding graph representations are closely interconnected and thoughtfully designed. To separate this interdependency, we first introduce a new PAG representation for a program, which supports

$$\frac{\mathtt{x} = \mathbf{new}\ \mathtt{T}\ //\ O}{O \xrightarrow{\mathsf{new[T]}} \mathtt{x}}\ \text{[C-New]} \qquad \frac{\mathtt{M} \text{ is an instance method}}{\mathtt{this}^{\mathtt{M}} \xrightarrow{\mathsf{load}[i]} p_i^{\mathtt{M}}}\ \text{[C-Param]} \qquad \frac{\mathtt{M} \text{ is an instance method}}{\mathtt{ret}^{\mathtt{M}} \xrightarrow{\mathsf{store[0]}} \mathtt{this}^{\mathtt{M}}}\ \text{[C-Ret]}$$

$$\frac{\mathtt{x} = \mathtt{r.m}(a_1, \ldots, a_n)\ //\ \mathtt{c} \quad \mathtt{t} <: \mathtt{DeclTypeOf(r)} \quad \mathtt{m}' = \mathtt{dispatch(c,t)}}{\forall\, i \in [1, n] : a_i \xrightarrow[\boxed{\hat{c}}]{\mathsf{store}[i]} \mathtt{r} \quad \mathtt{r} \xrightarrow[\boxed{\check{c}}]{\mathsf{load[0]}} \mathtt{x} \quad \mathtt{r} \xrightarrow{\mathsf{assign}} \mathtt{r\#c} \quad \mathtt{r} \xrightarrow[\boxed{\check{c}}]{\mathsf{assign}} \mathtt{r\#c} \quad \mathtt{r\#c} \xrightarrow[\hat{c}]{\mathsf{dispatch[t]}} \mathtt{this}^{\mathtt{m}'}}\ \text{[C-VCall]}$$

Figure 6 Rules for building the PAG required by L_{DCR}. [C-Assign], [C-Load], [C-Store] and [C-SCall] mirror those in Figure 2 and are excluded here to conserve space.

on-the-fly call graph construction (Section 3.1). Following this, we elaborate on L_{DCR} by detailing our solutions to the three challenges (**CHL1** – **CHL3**) and providing insights into its design (Section 3.2).

3.1 Pointer Assignment Graph

For representing a program in L_{DCR}, we employ the rules specified in Figure 6 to construct a PAG. The inverse of a PAG edge $x \xrightarrow[c]{\ell} y$, implicitly defined, is $y \xrightarrow[\overline{c}]{\overline{\ell}} x$, mirroring the approach in L_{FC} (Section 2.1.2). However, our approach uniquely allows below-edge labels to be also either $\boxed{\hat{c}}$ or $\boxed{\check{c}}$, where $\overline{\boxed{\hat{c}}} = \boxed{\check{c}}$ and $\overline{\boxed{\check{c}}} = \boxed{\hat{c}}$, with c denoting a callsite. To initiate dynamic dispatch at a callsite c, edges with boxed below-edge labels symbolize a novel type of inter-procedural value-flow entering (indicated by $\boxed{\hat{c}}$) or exiting (marked by $\boxed{\check{c}}$) a method at c. These specific boxed below-edge labels are introduced solely for addressing **CHL3**, and their significance will be explained in Section 3.2.2.

Our PAG, designed for L_{DCR}, primarily differs from the one for L_{FC} (Figure 2) in handling virtual callsites. Consequently, [C-Assign], [C-Load], [C-Store], and [C-SCall] are the same as [P-Assign], [P-Load], [P-Store], and [P-SCall], respectively. The additional rules in Figure 6 construct PAG edges that facilitate on-the-fly call graph construction at virtual callsites, addressing **CHL1** and **CHL2**.

In [C-New], $O \xrightarrow{\mathsf{new[T]}} \mathtt{x}$ specifically encodes $\mathtt{T}$, the dynamic type of O, to facilitate dynamic dispatch on O and enable its use as a receiver object, avoiding precision loss as depicted in Figure 5.

For [C-Param] and [C-Ret], we treat the i-th (non-`this`) parameter of an instance method $\mathtt{M}$ (denoted as $p_i^{\mathtt{M}}$, with i starting from 1) and its return variable $\mathtt{ret}^{\mathtt{M}}$ as special fields of $\mathtt{this}^{\mathtt{M}}$, identified by offset i and 0, respectively. This allows the initialization of $\mathtt{this}^{\mathtt{M}}.0$ with a store $\mathtt{ret}^{\mathtt{M}} \xrightarrow{\mathsf{store[0]}} \mathtt{this}^{\mathtt{M}}$ and a non-this parameter $p_i^{\mathtt{M}}$ with a load $\mathtt{this}^{\mathtt{M}} \xrightarrow{\mathsf{load}[i]} p_i^{\mathtt{M}}$.

In [C-VCall], we uniquely handle virtual calls like "$\mathtt{x} = \mathtt{r.m}(a_1, \ldots, a_n)\ //\mathtt{c}$" differently from [P-VCall] (Figure 2), using $\mathtt{r\#c}$ to uniquely identify $\mathtt{r}$ at callsite $\mathtt{c}$. There are two edges between $\mathtt{r}$ and $\mathtt{r\#c}$: the edge $\mathtt{r} \xrightarrow{\mathsf{assign}} \mathtt{r}\#c$, which is essential for passing the receiver variable, and the edge $\mathtt{r} \xrightarrow[\boxed{\check{c}}]{\mathsf{assign}} \mathtt{r\#c}$, which is crucial for passing other arguments during parameter passing, as will be explained shortly. We initially over-approximate target methods at $\mathtt{c}$ using CHA ([9]), similar to L_{FC}, for later refinement by L_{DCR}. For each target method $\mathtt{m}'$, the argument a_i is passed to the corresponding parameter $p_i^{\mathtt{m}'}$ $(1 \leqslant i \leqslant n)$ via a store $a_i \xrightarrow[\boxed{\hat{c}}]{\mathsf{store}[i]} \mathtt{r}$ and a matching load $\mathtt{this}^{\mathtt{m}'} \xrightarrow{\mathsf{load}[i]} p_i^{\mathtt{m}'}$ ([C-Param]). CFL-reachability under L_{DCR} involves traversing this store edge to find the dynamic type of each receiver object pointed by $\mathtt{r}$ (marked by $\boxed{\hat{c}}$). The sequence $\mathtt{r} \xrightarrow[\boxed{\check{c}}]{\mathsf{assign}} \mathtt{r\#c} \xrightarrow[\hat{c}]{\mathsf{dispatch[t]}} \mathtt{this}^{\mathtt{m}'}$ indicates finding the dynamic type $\mathtt{t}$ (marked by $\boxed{\check{c}}$), enabling dispatch of $\mathtt{m}'$ with $\hat{c}$ as its entry context (i.e.,

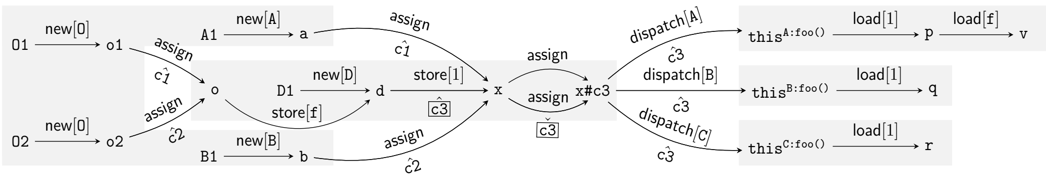

Figure 7 The PAG for L_{DCR} constructed for the program given in Figure 3.

$\mathtt{m}' = \mathtt{dispatch}(\mathtt{c}, \mathtt{t})$ as desired). A dispatch edge also functions as an assign edge. For the receiver variable r, we simply use $\mathtt{r} \xrightarrow{\text{assign}} \mathtt{r}\#c$ (without the need for relating r to itself). Finally, x is assigned $\mathtt{ret}^{\mathtt{m}'}$ (stored in $\mathtt{this}^{\mathtt{m}'}.0$ ([C-Ret])) via a load $r \xrightarrow[\check{\boxed{c}}]{\text{load}[0]} \mathtt{x}$, with $\check{\boxed{c}}$ marking the conclusion of the dynamic dispatch at callsite c.

Figure 7 illustrates the PAG leveraged by L_{DCR} for our motivating example, as presented in Figure 3. This PAG, uniquely tailored to support L_{DCR}'s integrated call graph construction, shows notable differences from the PAG employed by L_{FC}, as depicted in Figure 4.

3.2 L_{DCR} : A New CFL-Reachability Formulation for kCFA

L_{DCR} combines three CFLs ($L_{DCR} = L_D \cap L_C \cap L_R$) for addressing **CHL1** – **CHL3**. L_D, detailed in Section 3.2.1, deals with field accesses and dynamic dispatch, catering to **CHL1** and **CHL2**. L_C, defined in Equation (2), ensures callsite-sensitivity using below-edge labels Σ_{L_C}, which include $\hat{c}$ and $\check{c}$, and treats L_{DCR}'s unique boxed labels $\hat{\boxed{c}}$ and $\check{\boxed{c}}$ as ϵ. L_R, presented in Section 3.2.2, facilitates parameter passing in on-the-fly call graph construction, addressing **CHL3**. The focus will predominantly be on L_D and L_R, concentrating on parameter passing, with method returns being similarly handled.

Basic Idea. L_{DCR}, a CFL-reachability formulation, differs from L_{FC} mainly in managing parameter passing at virtual callsites, enabling L_{DCR}'s built-in call graph construction compared to L_{FC}'s reliance on a separate algorithm (Sec. 2.2.3.2). At a virtual callsite "$\mathtt{r.m}(a_1, \ldots, a_n)$; //c" under context $\mathbf{C}$, handling the receiver variable r (pointing to receiver objects) involves addressing **CHL1**: passing a receiver object to $\mathtt{this}^{m'}$ for dispatch on m'. In addition, for an argument a_i, **CHL2** and **CHL3** are met by storing a_i in $\mathtt{r}.i$, verifying if any object pointed by r under $\mathbf{C}$ matches dynamic type t, dynamically dispatching to $\mathtt{m}'$ ($\mathtt{m}' = \mathtt{dispatch}(\mathtt{c}, \mathtt{t})$), and assigning $\mathtt{this}^{\mathtt{m}'}.i$ to $p_i^{\mathtt{m}'}$ at callsite c under context $\mathbf{C}$. Method returns are handled in a similar fashion.

▶ **Example 1.** Revisiting our motivating example (Figure 3) and its PAG (Figure 7), L_{DCR} ensures a unique path from O1 to v, as shown in Equation (7), so that v points only to O1 when bar() is invoked at c1. The sub-path from O1 to d shows that O1 is stored into d.f, with d pointing to D1. The sub-path from d to p indicates parameter passing at callsite c3 to p for A:foo(), dynamically identified by L_{DCR} under $\mathbf{C} = [\mathtt{c1}]$. We have discussed addressing **CHL1** – **CHL3** at this callsite in Section 2.2.4. We wish to emphasize that $\hat{\boxed{\mathtt{c3}}}$ and $\check{\boxed{\mathtt{c3}}}$ signify dynamic dispatch's start and end at callsite c3 for d. CFL-reachability traversal between these markers confirms that x points to A1 under [c1], necessitating a return to x under [c1]. With receiver object A1, A:foo() is dispatched via $\mathtt{x\#c3} \xrightarrow[\hat{\mathtt{c3}}]{\text{dispatch[A]}} \mathtt{this}^{\mathtt{A:foo()}}$, allowing d to pass to p under [c3, c1]. Unlike L_{FC} [53] that uses [c3], L_{DCR} specifies [c3, c1] to indicate this occurs only when x points to A1 under [c1]. C:foo(), present in the PAG due to CHA [9], is filtered out by L_{DCR}'s on-the-fly call graph construction.

Let L^{dd}_{FC} be a demand-driven formulation of L_{FC} that is identical in all aspects except for one modification. This version continues to utilize a separate algorithm for on-the-fly call graph construction, but it has been specifically enhanced to accurately handle parameter passing for receiver variables, effectively avoiding the precision loss discussed in Section 2.2.3.2.

When developing L_{DCR}, we treat soundness fundamentally as an issue of precision.

▶ **Definition 2** (Soundness and Precision of On-the-Fly Call Graph Construction)**.** *For any given callsite and context $\boldsymbol{C}$, let T be the set of target methods identified under $\boldsymbol{C}$ through L^{dd}_{FC}. Suppose L is a language differing from L^{dd}_{FC} solely in managing parameter passing at virtual callsites. We regard L as* sound *if it facilitates parameter passing under $\boldsymbol{C}$ for at least the methods in T, and as* precise *(besides being sound) if it enables parameter passing under $\boldsymbol{C}$ for precisely the target methods in T.*

We write $L_{DC} = L_D \cap L_C$ as the intersection of L_D and L_C. A path p qualifies as an L_{DCR}*-path* if $L_D(p) \in L_D$, $L_C(p) \in L_C$, and $L_R(p) \in L_R$. An L_{DC}-path is defined similarly. As we will discuss further, L_{DC} is sound yet imprecise, whereas L_{DCR} is precise.

3.2.1 The L_D Language

This CFL captures both field-sensitive accesses, similar to L_F in Equation (1), and dynamic dispatch within its language framework:

$$\begin{aligned}
\mathsf{flowsto} &\longrightarrow \mathsf{new}[\mathtt{t}]\ (\mathsf{flows} \mid \mathsf{dispatch}[\mathtt{t}])^* \\
\mathsf{flows} &\longrightarrow \mathsf{assign} \mid \mathsf{store}[\mathtt{f}]\ \mathsf{alias}\ \mathsf{load}[\mathtt{f}] \\
\mathsf{alias} &\longrightarrow \overline{\mathsf{flowsto}}\ \mathsf{flowsto} \\
\overline{\mathsf{flowsto}} &\longrightarrow (\overline{\mathsf{dispatch}[\mathtt{t}]} \mid \overline{\mathsf{flows}})^*\ \overline{\mathsf{new}[\mathtt{t}]} \\
\overline{\mathsf{flows}} &\longrightarrow \overline{\mathsf{assign}} \mid \overline{\mathsf{load}[\mathtt{f}]}\ \mathsf{alias}\ \overline{\mathsf{store}[\mathtt{f}]}
\end{aligned} \tag{8}$$

Here, Σ_{L_D} includes all above-edge labels in the program's PAG. L_D extends L_F from Equation (1) [54, 53] by retaining its balanced parentheses approach for field accesses and adding support for dynamic dispatch, which facilitates on-the-fly call graph construction. Next, we describe how L_D is specifically designed to address **CHL1** and **CHL2**.

3.2.1.1 CHL1

To address **CHL1** regarding parameter passing at a virtual callsite, it is crucial that a receiver object O, pointed to by its receiver variable, is only passed to the `this` variable of a method dispatchable on O. For instance, in `x.foo(`**null**`)` from Figure 5, where `x` might point to both `E1` and `F1`, L_{FC} might incorrectly pass both `E1` and `F1` to $\mathtt{this}^{\mathtt{E:foo()}}$, as shown in Equations (5) and (6), despite `F1` being spurious.Note that L^{dd}_{FC}, introduced just before Definition 2, was specifically conceptualized to mitigate such precision loss.

In L_D, we explicitly specify the dynamic types of objects in four terminals: $\mathsf{new}[\mathtt{t}]$, $\overline{\mathsf{new}[\mathtt{t}]}$, $\mathsf{dispatch}[\mathtt{t}]$, and $\overline{\mathsf{dispatch}[\mathtt{t}]}$. This modification alters the two L_{FC}-paths discussed in Equations (5) and (6) for Figure 5 as follows:

$$\mathtt{E1} \xrightarrow{\mathsf{new}[\mathtt{E}]} \mathtt{e1} \xrightarrow{\mathsf{store}[\mathtt{g}]} \mathtt{w} \xrightarrow{\overline{\mathsf{new}[\mathtt{G}]}} \mathtt{G1} \xrightarrow{\mathsf{new}[\mathtt{G}]} \mathtt{w} \xrightarrow{\mathsf{load}[\mathtt{g}]} \mathtt{x} \xrightarrow{\mathsf{assign}} \mathtt{x\#c} \xrightarrow[\hat{\mathsf{c}}]{\mathsf{dispatch}[\mathtt{E}]} \mathtt{this}^{\mathtt{E:foo()}} \tag{9}$$

$$\mathtt{F1} \xrightarrow{\mathsf{new}[\mathtt{F}]} \mathtt{f1} \xrightarrow{\mathsf{store}[\mathtt{g}]} \mathtt{w} \xrightarrow{\overline{\mathsf{new}[\mathtt{G}]}} \mathtt{G1} \xrightarrow{\mathsf{new}[\mathtt{G}]} \mathtt{w} \xrightarrow{\mathsf{load}[\mathtt{g}]} \mathtt{x} \xrightarrow{\mathsf{assign}} \mathtt{x\#c} \xrightarrow[\hat{\mathsf{c}}]{\mathsf{dispatch}[\mathtt{E}]} \mathtt{this}^{\mathtt{E:foo()}} \tag{10}$$

During a flowsto ($\overline{\text{flowsto}}$) traversal, the type in dispatch[t] ($\overline{\text{dispatch[t]}}$) must align with its corresponding new[t] ($\overline{\text{new[t]}}$). Thus, the path in Equation (9) qualifies as an L_D-path, as new[E] flows* dispatch[E] $\in L_D$, but the path in Equation (10) does not as new[F] flows* dispatch[E] $\notin L_D$. Hence, in L_D, `F1` cannot spuriously flow to $\texttt{this}^{\texttt{E:foo()}}$. Similarly, in Equation (7), only `A1` can be passed to $\texttt{this}^{\texttt{A:foo()}}$, as `A:foo()` is dispatchable on `A1`.

▶ **Lemma 3.** *Consider a virtual callsite* $\texttt{x} = \texttt{r.m}(a_1, \ldots, a_n)$. *In* L_D, *every receiver object pointed to by* `r` *flows only to the* `this` *variable of a method that can be dispatched on the receiver object.*

Proof Sketch. Follows from the definition of L_D. ◀

3.2.1.2 CHL2

To meet **CHL2** and trigger dynamic dispatch at virtual callsites during parameter passing, we use $L_{DC} = L_D \cap L_C$. Re-examining the L_{DCR}-path in Equation (7) without $\boxed{\hat{\texttt{c3}}}$ and $\boxed{\check{\texttt{c3}}}$, we assess if `O1` flows into `v` starting from `c1`. Parameter passing for `d` at "`x.foo(d); // c3`" under $\mathbf{C} = [\texttt{c1}]$ involves traversing the sub-path from `d` to `p` of `A:foo()`. Starting with $\texttt{d} \xrightarrow{\text{store[1]}} \texttt{x}$, a $\overline{\text{flowsto}}$ traversal is initiated via $\texttt{x} \xrightarrow[\check{\texttt{c1}}]{\overline{\text{assign}}} \texttt{a} \xrightarrow{\overline{\text{new[A]}}} \texttt{A1}$ under $\mathbf{C} = [\texttt{c1}]$, returning to `x` via $\texttt{A1} \xrightarrow{\text{new[A]}} \texttt{a} \xrightarrow[\hat{\texttt{c1}}]{\text{assign}} \texttt{x}$, dispatching at `c3` through $\texttt{x} \xrightarrow{\text{assign}} \texttt{x\#c3} \xrightarrow[\hat{\texttt{c3}}]{\text{dispatch[A]}} \texttt{this}^{\texttt{A:foo()}}$, and finally passing `d` to `p` via $\texttt{this}^{\texttt{A:foo()}} \xrightarrow{\text{load[1]}} \texttt{p}$. Unlike L_{FC}'s direct passage of `d` to `p` in Equation (3), L_{DCR} uses a series of edges under [c3,c1], indicating dispatch occurs only when `x` points to `A1` under [c1].

▶ **Lemma 4.** L_{DC} *is sound in handling parameter passing at virtual callsites.*

Proof Sketch. Consider a virtual callsite $\texttt{r.m}(a_1, \ldots, a_n)$; // c", where parameter passing for an argument occurs under context $\mathbf{C}$. Let T represent the set of target methods identified on the fly for this callsite under $\mathbf{C}$ by applying a separate call graph algorithm as in L_{FC}^{dd}. As `r` is handled similarly as in L_{FC}^{dd}, it suffices to consider parameter passing for a non-this argument a_i. Focusing on a_i, L_{DC} initiates dynamic dispatch by locating receiver objects pointed to by `r` under also $\mathbf{C}$. Since L_{DC} differs from L_{FC}^{dd} only in handling parameter passing at virtual callsites, the set of target methods found by L_{DC} must include T. In addition, for each target $m' \in T$, there always exists a PAG path q:

$$a_i \xrightarrow{\text{store}[i]} \texttt{r}\ \overline{\text{flowsto}}\ O\ \text{flowsto}\ \texttt{r} \xrightarrow{\text{assign}} \texttt{r}\#c \xrightarrow[\hat{c}]{\text{dispatch[\_]}} \texttt{this}^{m'} \xrightarrow{\text{load}[i]} p_i^{m'} \tag{11}$$

Here, if u represents "r $\overline{\text{flowsto}}$ O", then "O flowsto r" is its inverse $\overline{u}$. This ensures a_i flows p_i by L_D and $L_C(q) \in L_C$ by L_C. Moreover, $L_C(q)$ forms a sequence of contexts feasible under $\mathbf{C}$, as u is traversed under $\mathbf{C}$. Therefore, by Definition 2, L_{DC} is sound. ◀

3.2.2 The L_R Language

L_{DC}, though sound, is not precise. This is illustrated in examples from Figures 8 and 9, highlighting L_{DC}'s limitations and underscoring the importance of L_R in L_{DCR}.

```java
static void main() {
  H h = new H(); // H1
  I i1 = new I(); // I1
  I i2 = new I(); // I2
  h.m(i1); // c4
  h.n(i2); // c5
}
class I {}
class H {
  void m(Object p) { ... }
  void n(Object q) { ... }
}
```

Figure 8 An example for illustrating the imprecision of L_{DC} caused by an incorrect dispatch site.

```java
static void main() {
  J j1 = new J(); // J1
  K k1 = new K(); // K1
  K k2 = new K(); // K2
  K v1 = wid(j1, k1); // c6
  K v2 = wid(j1, k2); // c7
}
class K { }
class J {
  K id(K p) {
    return p;
}}
static K wid(J j, K k) {
  K v = j.id(k); // c8
  return v;
}
```

Figure 9 An example for showing the imprecision of L_{DC} caused by an incorrect dispatch context.

L_{DC}'s precision loss can occur from a spurious dispatch callsite, shown by the following two L_{DC}-paths for Figure 8, temporarily ignoring the boxed labels $\boxed{\hat{\mathsf{c4}}}$, $\boxed{\check{\mathsf{c4}}}$, and $\boxed{\check{\mathsf{c5}}}$:

$$\texttt{I1} \xrightarrow{\mathsf{new[I]}} \texttt{i1} \xrightarrow[\boxed{\hat{\mathsf{c4}}}]{\mathsf{store[1]}} \texttt{h} \xrightarrow{\overline{\mathsf{new[H]}}} \texttt{H1} \xrightarrow{\mathsf{new[H]}} \texttt{h} \xrightarrow[\boxed{\check{\mathsf{c4}}}]{\mathsf{assign}} \texttt{h\#c4} \xrightarrow[\hat{\mathsf{c4}}]{\mathsf{dispatch[H]}} \texttt{this}^{\texttt{m}} \xrightarrow{\mathsf{load[1]}} \texttt{p} \tag{12}$$

$$\texttt{I1} \xrightarrow{\mathsf{new[I]}} \texttt{i1} \xrightarrow[\boxed{\hat{\mathsf{c4}}}]{\mathsf{store[1]}} \texttt{h} \xrightarrow{\overline{\mathsf{new[H]}}} \texttt{H1} \xrightarrow{\mathsf{new[H]}} \texttt{h} \xrightarrow[\boxed{\check{\mathsf{c5}}}]{\mathsf{assign}} \texttt{h\#c5} \xrightarrow[\hat{\mathsf{c5}}]{\mathsf{dispatch[H]}} \texttt{this}^{\texttt{n}} \xrightarrow{\mathsf{load[1]}} \texttt{q} \tag{13}$$

Both L_{DC}-paths track `I1`'s flow in the program's PAG. The first path correctly leads `I1` to `p`. However, the second path spuriously directs `I1` to `q`, as the $\overline{\mathsf{flowsto}}$ traversal to identify `a`'s receiver object starts at `c4` but concludes at `c5` spuriously. L_R addresses this precision issue by requiring matched boxed edge labels. As a result, the first path in Equation (12) is a valid L_{DCR}-path (with $\boxed{\hat{\mathsf{c4}}}$ matched by $\boxed{\check{\mathsf{c4}}}$), while the second path in Equation (13) is invalidated (due to the mismatch of $\boxed{\hat{\mathsf{c4}}}$ and $\boxed{\check{\mathsf{c5}}}$).

L_{DC} may also experience precision loss due to a spurious dispatch context. Consider the following two L_{DC}-paths in the PAG of Figure 9 (by ignoring the boxed labels $\boxed{\hat{\mathsf{c8}}}$ and $\boxed{\check{\mathsf{c8}}}$ for now):

$$\begin{aligned}&\texttt{K1} \xrightarrow{\mathsf{new[K]}} \texttt{k1} \xrightarrow[\hat{\mathsf{c6}}]{\mathsf{assign}} \texttt{k} \xrightarrow[\boxed{\hat{\mathsf{c8}}}]{\mathsf{store[1]}} \texttt{j} \xrightarrow[\check{\mathsf{c6}}]{\overline{\mathsf{assign}}} \texttt{j1} \xrightarrow{\overline{\mathsf{new[J]}}} \texttt{J1} \xrightarrow{\mathsf{new[J]}} \texttt{j1} \xrightarrow[\hat{\mathsf{c6}}]{\mathsf{assign}} \texttt{j} \xrightarrow[\boxed{\check{\mathsf{c8}}}]{\mathsf{assign}} \texttt{j\#c8} \xrightarrow[\hat{\mathsf{c8}}]{\mathsf{dispatch[J]}} \texttt{this}^{\texttt{id}} \xrightarrow{\mathsf{load[1]}}\\ &\texttt{p} \xrightarrow{\mathsf{store[0]}} \texttt{this}^{\texttt{id}} \xrightarrow[\check{\mathsf{c8}}]{\overline{\mathsf{dispatch[J]}}} \texttt{j\#c8} \xrightarrow[\boxed{\hat{\mathsf{c8}}}]{\overline{\mathsf{assign}}} \texttt{j} \xrightarrow[\check{\mathsf{c6}}]{\overline{\mathsf{assign}}} \texttt{j1} \xrightarrow{\overline{\mathsf{new[J]}}} \texttt{J1} \xrightarrow{\mathsf{new[J]}} \texttt{j1} \xrightarrow[\hat{\mathsf{c6}}]{\mathsf{assign}} \texttt{j} \xrightarrow[\boxed{\check{\mathsf{c8}}}]{\mathsf{load[0]}} \texttt{v} \xrightarrow[\check{\mathsf{c6}}]{\mathsf{assign}} \texttt{v1}\end{aligned} \tag{14}$$

$$\begin{aligned}&\texttt{K1} \xrightarrow{\mathsf{new[K]}} \texttt{k1} \xrightarrow[\hat{\mathsf{c6}}]{\mathsf{assign}} \texttt{k} \xrightarrow[\boxed{\hat{\mathsf{c8}}}]{\mathsf{store[1]}} \texttt{j} \xrightarrow[\check{\mathsf{c6}}]{\overline{\mathsf{assign}}} \texttt{j1} \xrightarrow{\overline{\mathsf{new[J]}}} \texttt{J1} \xrightarrow{\mathsf{new[J]}} \texttt{j1} \xrightarrow[\hat{\mathsf{c7}}]{\mathsf{assign}} \texttt{j} \xrightarrow[\boxed{\check{\mathsf{c8}}}]{\mathsf{assign}} \texttt{j\#c8} \xrightarrow[\hat{\mathsf{c8}}]{\mathsf{dispatch[J]}} \texttt{this}^{\texttt{id}} \xrightarrow{\mathsf{load[1]}}\\ &\texttt{p} \xrightarrow{\mathsf{store[0]}} \texttt{this}^{\texttt{id}} \xrightarrow[\check{\mathsf{c8}}]{\overline{\mathsf{dispatch[J]}}} \texttt{j\#c8} \xrightarrow[\boxed{\hat{\mathsf{c8}}}]{\overline{\mathsf{assign}}} \texttt{j} \xrightarrow[\check{\mathsf{c7}}]{\overline{\mathsf{assign}}} \texttt{j1} \xrightarrow{\overline{\mathsf{new[J]}}} \texttt{J1} \xrightarrow{\mathsf{new[J]}} \texttt{j1} \xrightarrow[\hat{\mathsf{c7}}]{\mathsf{assign}} \texttt{j} \xrightarrow[\boxed{\check{\mathsf{c8}}}]{\mathsf{load[0]}} \texttt{v} \xrightarrow[\check{\mathsf{c7}}]{\mathsf{assign}} \texttt{v2}\end{aligned} \tag{15}$$

These two L_{DC}-paths in Figure 9 vary only by context: Equation (15) is similar to Equation (14), but replaces `c7` with `c6` and `v2` with `v1`. Both track where `K1` flows, starting from "`wid(j1,k1); // c6`". According to Equation (14), `v1` points to `K1` as expected. However, Equation (15) inaccurately allows `K1`, passed at `c6`, to flow into `v2` at `c7`, spuriously indicating v2 points to K1. Focusing on dynamic dispatch at callsite c8 in line 14 due to the call at c6 in line 5 (Figure 9), Equation (14) shows that j initially pointing to J1 under [c6] and maintains this during both $\overline{\mathsf{flowsto}}$ and $\mathsf{flowsto}$ traversals from c6. However, Equation (15) starts similarly but ends with j pointing to J1 under [c7], which is inconsistent with the call at c6.

In general, L_{DC} may lack precision as it sometimes includes spurious sub-paths for dynamic dispatch. Consider a generic virtual callsite $\mathtt{r.m}(a_1, \ldots, a_n)$ // c, L_{DC} initiates dynamic dispatch by executing the following alias-related traversal on its receiver variable $\mathtt{r}$:

$$\cdots \xrightarrow[\hat{\boxed{c}}]{\mathsf{store[i]}} \mathtt{r}\ \overline{\mathsf{flowsto}}\ O\ \mathsf{flowsto}\ \mathtt{r}' \xrightarrow[\check{\boxed{c'}}]{\mathsf{assign}} \mathtt{r}'\#c' \xrightarrow[\hat{c}']{\mathsf{dispatch[\_]}} \cdots \tag{16}$$

Such a *dispatch path*, which starts from $\hat{\boxed{c}}$ and ends at $\check{\boxed{c'}}$, is *valid* if two conditions are met:

- DP-C1: $c = c'$ (implying that $\mathtt{r} = \mathtt{r}'$), and
- DP-C2: O is pointed by both $\mathtt{r}$ and $\mathtt{r}'$ under exactly the same context.

However, L_{DC} can ensure that $\mathtt{r}$ and $\mathtt{r}'$ are aliases but cannot guarantee the validity of this dispatch path. For example, Equation (13) contains a dispatch path violating DP-C1, and Equation (15) violates DP-C2. To exclude such invalid dispatch paths in L_{DC}-paths, L_R is designed to utilize all below-edge labels in the PAG (i.e., $\hat{c}$, $\check{c}$, $\hat{\boxed{c}}$, and $\check{\boxed{c}}$) as terminals:

$$\begin{aligned}
\mathsf{recoveredCtx} &\longrightarrow \mathsf{recoveredCtx}\ \hat{c} \mid \mathsf{recoveredCtx}\ \check{c} \mid \mathsf{recoveredCtx}\ \mathsf{siteRecovered} \mid \epsilon \\
\mathsf{siteRecovered} &\longrightarrow \hat{\boxed{c}}\ \mathsf{ctxRecovered}\ \check{\boxed{c}} \\
\mathsf{ctxRecovered} &\longrightarrow \mathsf{matched}\ \mathsf{ctxRecovered} \mid \mathsf{ctxRecovered}\ \mathsf{matched} \mid \check{c}\ \mathsf{ctxRecovered}\ \hat{c} \mid \epsilon \\
\mathsf{matched} &\longrightarrow \mathsf{matched}\ \mathsf{matched} \mid \hat{c}\ \mathsf{matched}\ \check{c} \mid \mathsf{siteRecovered} \mid \epsilon
\end{aligned} \tag{17}$$

Here, Σ_{L_R} includes all below-edge labels in the program's PAG. The start symbol recoveredCtx would define a language that contains L_C if its third alternative "recoveredCtx siteRecovered" were changed to "recoveredCtx". Thus, L_R is engaged during a dispatch path traversal. The siteRecovered production enforces DP-C1, and the ctxRecovered and matched productions collectively enforce DP-C2. This design enables L_R to address **CHL3** by reinstating the context of $\mathtt{r}$.

By incorporating L_R into L_{DC}, the composite language $L_{DCR} = L_D \cap L_C \cap L_R$ achieves precision in managing parameter passing at virtual callsites. Reexamining the paths in Equations (14) and (15), with the inclusion of $\hat{\boxed{\mathtt{c8}}}$ and $\check{\boxed{\mathtt{c8}}}$, it is clear that the first path qualifies as an L_{DCR}-path, while the second does not. In the first path, the dynamic dispatch starts at callsite c8 under context [c6] and returns to the same callsite under the same context, signified by $\hat{\boxed{\mathtt{c8}}}$ and $\check{\boxed{\mathtt{c8}}}$. Conversely, the second path, while also starting dispatch at callsite c8 under context [c6], mistakenly returns under a different context, [c7], making it invalid for L_{DCR}. As a result, L_{DCR} correctly determines that K1 is pointed to by v1, but not by v2, effectively preventing v2 from pointing to K1 spuriously.

Below, we give a formal development of L_R, followed by a proof of L_{DCR}'s precision.

To determine the points-to set of a variable v, $\mathtt{PTS}(v, c_v)$, using L_{DC}, consider an L_C-path p with label $L_C(p) = \ell_1, \ldots, \ell_n$, where each ℓ_i is a context label on an inter-procedural assign edge. The inverse of p, $\overline{p}$, has a label $L_C(\overline{p}) = \overline{\ell_n}, \ldots, \overline{\ell_1}$. Splitting p into sub-paths p^{ex} and p^{en}, we define ${L_C}^{\mathsf{ex}}(p) = L_C(p^{\mathsf{ex}})$ and ${L_C}^{\mathsf{en}}(p) = L_C(p^{\mathsf{en}})$, with $L_C(p) = {L_C}^{\mathsf{ex}}(p){L_C}^{\mathsf{en}}(p)$. Here, ${L_C}^{\mathsf{ex}}(p)$ and ${L_C}^{\mathsf{en}}(p)$ are derived from exit and entry in L_C's grammar (Equation (2)). For $s \in L_C$, $\mathscr{B}(s)$ returns s's canonical form with balanced contexts removed. If c is a string of exit contexts like $\check{c_1} \ldots \check{c_n}$, $\mathscr{E}(c) = [c_1, \ldots, c_n]$ converts it into a context representation, noting $\mathscr{E}(\epsilon) = [\,]$.

For an L_{DC}-path p from an object O to a variable v, we can clearly deduce the following points-to relationship, including the specific contexts of O and v:

$$\langle O, \mathscr{E}(\mathscr{B}(L_C{}^{\mathsf{ex}}(p)))\rangle \quad \in \quad \texttt{PTS}(v, \mathscr{E}(\overline{\mathscr{B}(L_C{}^{\mathsf{en}}(p))})) \tag{18}$$

▶ **Example 5.** Let us take $p_{\texttt{O1},\texttt{v}}$, the L_{DC}-path from Equation (7), by ignoring $\hat{\boxed{\texttt{c3}}}$ and $\check{\boxed{\texttt{c3}}}$. By definition, $L_C(p_{\texttt{O1},\texttt{v}}) = \hat{\texttt{c}}\texttt{1}\check{\texttt{c}}\texttt{1}\hat{\texttt{c}}\texttt{1}\hat{\texttt{c}}\texttt{3}$, where $p^{\mathsf{ex}}_{\texttt{O1},\texttt{v}}$ denotes the sub-path from $\texttt{O1}$ to $\texttt{A1}$ and $p^{\mathsf{en}}_{\texttt{O1},\texttt{v}}$ denotes the sub-path from $\texttt{A1}$ to $\texttt{v}$. Thus, $L_C{}^{\mathsf{ex}}(p_{\texttt{O1},\texttt{v}}) = \hat{\texttt{c}}\texttt{1}\check{\texttt{c}}\texttt{1}$ and $L_C{}^{\mathsf{en}}(p_{\texttt{O1},\texttt{v}}) = \hat{\texttt{c}}\texttt{1}\hat{\texttt{c}}\texttt{3}$. Since $\mathscr{E}(\mathscr{B}(\hat{\texttt{c}}\texttt{1}\check{\texttt{c}}\texttt{1})) = [\,]$ and $\mathscr{E}(\overline{\mathscr{B}(\hat{\texttt{c}}\texttt{1}\hat{\texttt{c}}\texttt{3})}) = \mathscr{E}(\overline{\hat{\texttt{c}}\texttt{1}\hat{\texttt{c}}\texttt{3}}) = [\texttt{c3}, \texttt{c1}]$, we have: $\langle \texttt{O1}, [\,]\rangle \in \texttt{PTS}(\texttt{v}, [\texttt{c3}, \texttt{c1}])$.

To enforce **DP-C1**, the production $\mathsf{siteRecovered} \longrightarrow \hat{\boxed{c}}\ \mathsf{ctxRecovered}\ \check{\boxed{c}}$ ensures that a dispatch process starting at a callsite (indicated by $\hat{\boxed{c}}$) concludes at the same callsite (marked by $\check{\boxed{c}}$). In the dispatch path from Equation (16), this guarantees $c = c'$ and $\texttt{r} = \texttt{r}'$. Thus, matching $\hat{\boxed{c}}$ with $\check{\boxed{c}}$ allows c to be reinstated at the next $\mathsf{dispatch}$ edge, ensuring dynamic dispatch occurs specifically at callsite c.

To enforce **DP-C2**, the $\mathsf{ctxRecovered}$- and $\mathsf{matched}$-productions are crucial, with $\mathsf{ctxRecovered}$
$\longrightarrow \check{c}\ \mathsf{ctxRecovered}\ \hat{c}$ being central. This is best understood through a generic dispatch path in Equation (16). **DP-C2** can be rephrased as follows. Let $p_{\texttt{r},O}$ be the $\overline{\mathsf{flowsto}}$ path from $\texttt{r}$ to O, and its inverse $\overline{p_{\texttt{r},O}}$ a $\mathsf{flowsto}$ path. Consider $p_{O,\texttt{r}'}$ as the $\mathsf{flowsto}$ path from O to $\texttt{r}'$. The path from $\texttt{r}$ to $\texttt{r}'$ is composed of $p_{\texttt{r},O}\ p_{O,\texttt{r}'}$ or equivalently $p^{\mathsf{ex}}_{\texttt{r},O}\ p^{\mathsf{en}}_{\texttt{r},O}\ p^{\mathsf{ex}}_{O,\texttt{r}'}\ p^{\mathsf{en}}_{O,\texttt{r}'}$. Applying Equation (18), we deduce:

$$\begin{array}{rcl} \langle O, \mathscr{E}(\mathscr{B}(L_C{}^{\mathsf{ex}}(\overline{p_{\texttt{r},O}})))\rangle & \in & \texttt{PTS}(\texttt{r}, \mathscr{E}(\overline{\mathscr{B}(L_C{}^{\mathsf{en}}(\overline{p_{\texttt{r},O}}))})) \\ \langle O, \mathscr{E}(\mathscr{B}(L_C{}^{\mathsf{ex}}(p_{O,\texttt{r}'})))\rangle & \in & \texttt{PTS}(\texttt{r}', \mathscr{E}(\overline{\mathscr{B}(L_C{}^{\mathsf{en}}(p_{O,\texttt{r}'}))})) \end{array} \tag{19}$$

As $\texttt{r}$ and $\texttt{r}'$ are aliases, they must always point to O with exactly the same heap context, i.e., $\mathscr{E}(\mathscr{B}(L_C{}^{\mathsf{ex}}(\overline{p_{\texttt{r},O}}))) = \mathscr{E}(\mathscr{B}(L_C{}^{\mathsf{ex}}(p_{O,\texttt{r}'})))$. Thus, $\mathscr{B}(\overline{\mathscr{B}(L_C{}^{\mathsf{ex}}(\overline{p_{\texttt{r},O}}))}\mathscr{B}(L_C{}^{\mathsf{ex}}(p_{O,\texttt{r}'}))) = \epsilon$ holds, implying the edge labels on path $p^{\mathsf{en}}_{\texttt{r},O}\ p^{\mathsf{ex}}_{O,\texttt{r}'}$ must be balanced. Besides, $\texttt{r}$ and $\texttt{r}'$ are required to have the same context, i.e., $\mathscr{E}(\overline{\mathscr{B}(L_C{}^{\mathsf{en}}(\overline{p_{\texttt{r},O}}))}) = \mathscr{E}(\overline{\mathscr{B}(L_C{}^{\mathsf{en}}(p_{O,\texttt{r}'}))})$. Thus, the following must be true:

$$\mathscr{B}\Big(\mathscr{B}(L_C{}^{\mathsf{en}}(p_{O,\texttt{r}'})\overline{\mathscr{B}(L_C{}^{\mathsf{en}}(\overline{p_{\texttt{r},O}}))}\Big) \quad = \quad \epsilon \tag{20}$$

implying that the edge labels in path $p^{\mathsf{en}}_{O,\texttt{r}'}\ p^{\mathsf{ex}}_{\texttt{r},O}$ must be balanced out.

Both the $\mathsf{ctxRecovered}$- and $\mathsf{matched}$- productions in L_R play key roles during dispatch path traversal, as illustrated in Equation (16). The production $\mathsf{ctxRecovered} \longrightarrow \check{c}\ \mathsf{ctxRecovered}\ \hat{c}$ enforces **DP-C2** (see Equation (20)), while $\mathsf{matched} \rightarrow \mathsf{siteRecovered}$ initiates traversal of another dispatch path. The other productions help bypass matched contexts and callsites. In simple terms, for a traversal from $\texttt{r}$ to O ($\texttt{r}\ \overline{\mathsf{flowsto}}\ O$), writing down all unmatched exit contexts as $\check{c_1}, \ldots, \check{c_n}$ implies that the unmatched entry contexts seen on the return from O to $\texttt{r}'$ ($O\ \mathsf{flowsto}\ \texttt{r}'$) should be $\hat{c_n}, \ldots, \hat{c_1}$.

Revisiting the two L_{DC}-paths from Equations (14) and (15), as introduced in Section 3.2.2, the L_{DC}-path in Equation (14) qualifies as an L_{DCR}-path due to its valid dispatch paths. However, the L_{DC}-path in Equation (15) does not, as its initial dispatch path at callsite $\texttt{c8}$ from $\texttt{j}$ to $\texttt{j\#c8}$ is invalid. With $\overline{\mathscr{B}(L_C{}^{\mathsf{en}}(\overline{p_{\texttt{j},\texttt{J1}}}))} = \check{\texttt{c}}\texttt{6}$ and $\mathscr{B}(L_C{}^{\mathsf{en}}(p_{\texttt{J1},\texttt{j}})) = \hat{\texttt{c}}\texttt{7}$, we find $\mathscr{B}(\mathscr{B}(L_C{}^{\mathsf{en}}(p_{\texttt{J1},\texttt{j}}))\overline{\mathscr{B}(L_C{}^{\mathsf{en}}(\overline{p_{\texttt{j},\texttt{J1}}}))}) = \hat{\texttt{c}}\texttt{7}\check{\texttt{c}}\texttt{6} \neq \epsilon$, indicating the path is invalid as $\check{\texttt{c}}\texttt{6}\hat{\texttt{c}}\texttt{7}$ does not balance out according to $\mathsf{ctxRecovered} \longrightarrow \check{c}\ \mathsf{ctxRecovered}\ \hat{c}$.

▶ **Theorem 1.** *L_{DCR} is precise in handling parameter passing for virtual callsites.*

Proof. Drawing from Lemmas 3 and 4, it suffices to show that for every virtual callsite "`r.m`$(a_1, \ldots, a_n)$; // `c`" under context $\mathbf{C}$, L_{DCR} precisely handles parameter passing for the same target method set T identified at this callsite under $\mathbf{C}$ by L_{FC}^{dd}'s call graph algorithm. This holds as L_R filters out only those L_{DC}-paths with invalid dispatch paths. ◀

L_{DCR} achieves the same level of precision as L_{FC}^{dd}, thereby ensuring both soundness and precision in computing points-to information. We now employ L_{DCR} to determine points-to information in our motivating example (Figure 3), with Equation (18) being relevant but focusing solely on L_{DCR}-paths in the program's PAG. Although CHA [9] in the PAG (Figure 7) broadly predicts target methods at virtual callsites, L_{DCR}'s on-the-fly call graph construction process efficiently filters out spurious target methods like `C:foo()`.

Finally, let us compare L_{DCR}, a CFL-reachability-based pointer analysis, with kCFA (Figure 1). While L_{DCR}, like L_{FC} [53], is suited for demand-driven analysis, kCFA is for whole-program analysis. Their key difference is the starting point: kCFA begins with entry methods M, including `main()`, and L_{DCR} with query variables V. Thus, kCFA may not compute points-to information for variables in V not reachable from M. In terms of precision, if kCFA determines $\mathtt{PTS}(v, c)$ for variable v from M under context c, L_{DCR} can obtain exactly the same points-to set for v under c according to Equation (18). However, kCFA may overlook points-to information in the code unreachable from M.

3.3 Time Complexities

The PAG construction shown in Figures 2 and 6 scales linearly with the number of program statements. Yet, the L_{DCR}-reachability problem, like the L_{FC}-reachability problem [53], is undecidable due to being an intersection of three interwoven CFLs (L_D, L_C, L_R), making the combinations of $L_D \cap L_C$, $L_D \cap L_R$, and $L_C \cap L_R$ also undecidable [45]. For any individual CFL language $L \in \{L_D, L_C, L_R\}$, the reachability problem's time complexity can reach up to $O(m^3n^3)$, where m is the grammar size and n is the number of nodes in the PAG.

Similar to kCFA (Figure 1), which introduces k-limiting to L_C in L_{FC}, resulting in a complexity of $O(n^3)$, we can also render the L_{DCR}-reachability problem computable within polynomial time for practical applications by applying k-limiting to both L_C and L_R.

4 P3Ctx : An Application of L_{DCR}

In our secondary contribution, we demonstrate the effectiveness of L_{DCR} through P3Ctx, the first pre-analysis tool powered by L_{DCR} for accelerating kCFA with selective context-sensitivity, always maintaining its precision. This also confirms L_{DCR}'s correctness. Conversely, Selectx [33], an L_{FC}-enabled pre-analysis does not guarantee precision preservation.

4.1 Selective Context-Sensitivity

Selective context sensitivity enhances the efficiency of context-sensitive analyses, maintaining much of their precision. It applies context-sensitivity selectively to crucial program variables and objects, treating the rest context-insensitively. Selectx [33], a recent method for selective context-sensitive pointer analysis in kCFA, is built on L_{FC}, an incomplete formulation dependent on an external call graph construction algorithm. As a result, Selectx inaccurately categorizes some vital variables and objects, causing precision loss. To remedy this, we introduce P3Ctx, a new L_{DCR}-based pre-analysis technique for selective context-sensitivity in kCFA, ensuring precision. P3Ctx is developed following the fundamental approach used in [33] for creating Selectx.

4.1.1 CFL-Reachability-Guided Selections

Applying L_{FC} to develop SELECTX [33] is straightforward. For a flowsto path $p_{O,n,v}$ in L_{FC}, starting from an object O to a variable v via n (a variable or object in method `M`), consider $p_{O,n}$ as the segment from O to n, and $p_{n,v}$ from n to v. Then n requires context-sensitivity in kCFA to avoid potential precision loss *only if* three conditions are met:

$$\begin{aligned}&\texttt{CS-C1} : L_F(p_{O,n,v}) \in L_F\\ &\texttt{CS-C2} : L_C(p_{O,n}) \in L_C \wedge L_C(p_{n,v}) \in L_C\\ &\texttt{CS-C3} : L_C^{\mathsf{en}}(p_{O,n}) \neq \epsilon \wedge L_C^{\mathsf{ex}}(p_{n,v}) \neq \epsilon\end{aligned} \tag{21}$$

where L_C^{en} and L_C^{ex} are from Section 3.2.2. O from outside `M` flows into n along $p_{O,n}$ context-sensitively and n flows out of `M` into v along $p_{n,v}$ context-sensitively, via `M`'s parameters (or return variable) along each path. Note that $p_{O,n,v}$ itself is not required to be an L_{FC}-path.

By replacing L_F with L_D in Equation (21), P3CTX also determines n to be context-sensitive *if* `CS-C1`– `CS-C3` are met. Viewing these conditions as sufficient (rather than merely necessary) makes both SELECTX and P3CTX conservative, potentially marking some n as context-sensitive even when kCFA would not lose precision with context-insensitive analysis. While SELECTX could lead to precision loss due to L_{FC}'s incompleteness, P3CTX, in contrast, always preserves precision. This is because L_{DCR} works with a PAG that clearly includes dispatch paths for all virtual callsites in the program.

▶ **Example 6.** In our motivating example (Figure 3), whether `v` spuriously points to `O2` hinges on the context sensitivity of `d`, `o`, `x`, and `D1` in `bar()`. Using L_{FC} and analyzing the PAG in Figure 4, SELECTX deems all four as context-insensitive, causing `v` to erroneously point to `O2` because they cannot flow out of `bar()` via its parameter `x`, failing to meet `CS-C3`. In L_{FC}'s PAG, which relies on an external call graph construction algorithm, there are no dispatch paths for these variables/objects to flow out of `bar()` through `x`.

In L_{DCR}, the parameter passing of `d` at `x.foo(d)` (line 17) directly relates to `x` via CFL-reachability (Figure 7). Consider $p_{\texttt{O1},n,\texttt{v}}$ in Equation (7), which is an L_{DCR}-path. For $n \in \{\texttt{d}, \texttt{o}, \texttt{x}, \texttt{D1}\}$, P3CTX designates each n as context-sensitive. This decision is because $p_{\texttt{O1},n,\texttt{v}}$ qualifies as an L_D-path (`CS-C1`), with both $p_{\texttt{O1},n}$ and $p_{n,v}$ being L_C-paths (`CS-C2`). Furthermore, $L_C{}^{\mathsf{en}}(p_{\texttt{O1},n}) = \hat{c1} \neq \epsilon$ and $L_C{}^{\mathsf{ex}}(p_{n,v}) = \check{c1} \neq \epsilon$, satisfying `CS-C3`.

4.1.2 Regularization

To make P3CTX as lightweight as possible so that we can efficiently make context-sensitivity selections without losing the performance benefits obtained from a subsequent main pointer analysis, we have decided to keep L_C unchanged as done in several earlier pre-analyses [35, 32, 33] but regularize L_D and L_R. We first regularize L_R to L_R^r as follows:

$$\mathsf{recoveredCtx} \longrightarrow \mathsf{recoveredCtx}\ \hat{c} \mid \mathsf{recoveredCtx}\ \check{c} \mid \mathsf{recoveredCtx}\ \boxed{\hat{c}} \mid \mathsf{recoveredCtx}\ \boxed{\check{c}} \mid \epsilon \tag{22}$$

Thus, $L_D \cap L_C \cap L_R^r = L_D \cap L_C = L_{DC}$. By noting further that the boxed edge labels in $L_R{}^r$ (i.e., $\boxed{\hat{c}}$ and $\boxed{\check{c}}$) are irrelevant to context-sensitivity selections and the regular entry/exit context labels in $L_R{}^r$ (i.e., $\hat{c}$ and $\check{c}$) have already been included in L_C, we conclude that $L_R{}^r$ (i.e., L_R) can be ignored safely (or conservatively). As $L_{DC} \supseteq L_{DCR}$ (i.e., L_{DC} captures all the possible value-flows that are captured by L_{DCR} for a given program) according to Lemma 4, it suffices to use L_{DC} in place of L_{FC} in Equation (21) in developing our precision-preserving pre-analysis. Like the L_{FC}-reachability problem, the L_{DC}-reachability problem is also undecidable [45]. Following [33], we regularize L_D into L_{D^r} and subsequently over-approximate L_{DC} to obtain $L_{D^rC} = L_{D^r} \cap L_C$. In Section 4.1.3, we present an algorithm to verify `CS-C1`– `CS-C3` using L_{D^rC} efficiently.

We start with $L_0 = L_D$. We first over-approximate L_0 by disregarding its field-sensitivity requirement and thus obtain L_1 given below:

$$
\begin{array}{rcl}
\mathsf{flowsto} & \longrightarrow & \mathsf{new}\ (\mathsf{flows} \mid \mathsf{dispatch})^* \\
\mathsf{flows} & \longrightarrow & \mathsf{assign} \mid \mathsf{store}\ \overline{\mathsf{flowsto}}\ \mathsf{flowsto}\ \mathsf{load} \\
\overline{\mathsf{flowsto}} & \longrightarrow & (\overline{\mathsf{dispatch}} \mid \overline{\mathsf{flows}})^*\ \overline{\mathsf{new}} \\
\overline{\mathsf{flows}} & \longrightarrow & \overline{\mathsf{assign}} \mid \overline{\mathsf{load}}\ \overline{\mathsf{flowsto}}\ \mathsf{flowsto}\ \overline{\mathsf{store}}
\end{array}
\tag{23}
$$

In the absence of field-sensitivity, a $\mathsf{dispatch}$ ($\overline{\mathsf{dispatch}}$) edge behaves just like an assign ($\overline{\mathsf{assign}}$) edge and can thus be interpreted this way. As a result, we obtain L_2 below:

$$
\begin{array}{rcl}
\mathsf{flowsto} & \longrightarrow & \mathsf{new}\ \mathsf{flows}^* \\
\overline{\mathsf{flowsto}} & \longrightarrow & \overline{\mathsf{flows}}^*\ \overline{\mathsf{new}} \\
\mathsf{flows} & \longrightarrow & \mathsf{assign} \mid \mathsf{store}\ \overline{\mathsf{flowsto}}\ \mathsf{flowsto}\ \mathsf{load} \\
\overline{\mathsf{flows}} & \longrightarrow & \overline{\mathsf{assign}} \mid \overline{\mathsf{load}}\ \overline{\mathsf{flowsto}}\ \mathsf{flowsto}\ \overline{\mathsf{store}}
\end{array}
\tag{24}
$$

Our approximation goes further by treating a load ($\overline{\mathsf{load}}$) edge as also an assign ($\overline{\mathsf{assign}}$). As a result, we will no longer require a store ($\overline{\mathsf{load}}$) edge to be matched by a load ($\overline{\mathsf{store}}$) edge. This will give rise to L_3 below:

$$
\begin{array}{rcl}
\mathsf{flowsto} & \longrightarrow & \mathsf{new}\ \mathsf{flows}^* \\
\overline{\mathsf{flowsto}} & \longrightarrow & \overline{\mathsf{flows}}^*\ \overline{\mathsf{new}} \\
\mathsf{flows} & \longrightarrow & \mathsf{assign} \mid \mathsf{store}\ \overline{\mathsf{flowsto}}\ \mathsf{flowsto} \\
\overline{\mathsf{flows}} & \longrightarrow & \overline{\mathsf{assign}} \mid \overline{\mathsf{flowsto}}\ \mathsf{flowsto}\ \overline{\mathsf{store}}
\end{array}
\tag{25}
$$

Finally, we obtain $L_{D^r} = L_4$ given below by no longer distinguishing a store edge from its inverse, $\overline{\mathsf{store}}$ edge, so that we can represent both types of edges as a store edge:

$$
\begin{array}{rcl}
\mathsf{flowsto} & \longrightarrow & \mathsf{new}\ \mathsf{flows}^* \\
\overline{\mathsf{flowsto}} & \longrightarrow & \overline{\mathsf{flows}}^*\ \overline{\mathsf{new}} \\
\mathsf{flows} & \longrightarrow & \mathsf{assign} \mid \mathsf{store}\ \overline{\mathsf{assign}}^*\ \overline{\mathsf{new}}\ \mathsf{new} \\
\overline{\mathsf{flows}} & \longrightarrow & \overline{\mathsf{assign}} \mid \overline{\mathsf{new}}\ \mathsf{new}\ \mathsf{assign}^*\ \mathsf{store}
\end{array}
\tag{26}
$$

▶ **Lemma 1.** $L_D \subseteq L_{D^r}$.

Proof. Follows from the fact that $L_i \subseteq L_{i+1}$. ◀

While L_{D^r} is identical to L_R regularized from L_F in SELECTX [33], our PAG (Figure 6), which makes dynamic dispatch paths explicitly, differs fundamentally from the one operated by L_{FC} (Figure 2). This distinction ensures that P3CTX preserves precision, unlike SELECTX.

Let $G = (N, E)$ be the PAG of a program. We use Andersen's algorithm [1] instead of CHA [9] to build its call graph in order to sharpen the precision of P3CTX.

We use a simple DFA shown in Figure 10 to accept L_{D^r} exactly. P3CTX runs inter-procedurally in linear time of the number of the PAG edges in G. To deal with L_C, we use summary edges added into the PAG (facilitated by the dotted transition labeled as $\mathsf{balanced}$).

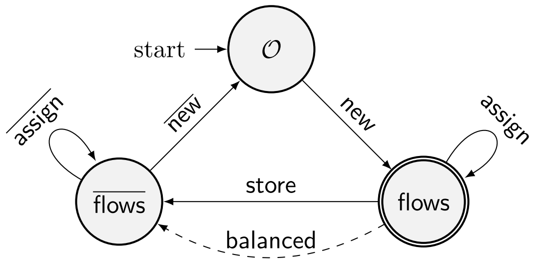

Figure 10 A DFA for accepting L_{D^r}.

4.1.3 P3Ctx

We follow [14] to develop a simple algorithm to verify **CS-C1**– **CS-C3** efficiently based on two properties that can be easily deduced from the DFA given in Figure 10 as stated below.

Define $Q = \{\mathcal{O}, \mathsf{flows}, \overline{\mathsf{flows}}\}$ as the state set and $\delta : Q \times \Sigma \to Q$ as the transition function. For each PAG edge $n_1 \xrightarrow{\ell} n_2$ in G, the transition $\delta(q_1, \ell) = q_2$ leads to a one-step transition $(n_1, q_1) \rightarrowtail (n_2, q_2)$. The multiple-step transition $\rightarrowtail^+$ is the transitive closure of $\rightarrowtail$. The symmetry of $\mathsf{flowsto}$ and $\overline{\mathsf{flowsto}}$ in L_{D^r} yields two straightforward properties of this DFA:

- **PROP-O**. Let O be an object created in a method `M`. Then $\langle \texttt{this}^{\texttt{M}}, \mathsf{flows}\rangle \rightarrowtail^+ \langle O, \mathcal{O}\rangle \iff \langle O, \mathcal{O}\rangle \rightarrowtail^+ \langle \texttt{this}^{\texttt{M}}, \overline{\mathsf{flows}}\rangle$ always holds.
- **PROP-V**. Let v be a variable defined in a method `M`. Then $\langle \texttt{this}^{\texttt{M}}, \mathsf{flows}\rangle \rightarrowtail^+ \langle v, q\rangle \iff \langle v, \overline{q}\rangle \rightarrowtail^+ \langle \texttt{this}^{\texttt{M}}, \overline{\mathsf{flows}}\rangle$ always holds, where $q \in \{\mathsf{flows}, \overline{\mathsf{flows}}\}$ (since v is a variable).

To handle static callsites uniformly as virtual callsites, we assume that a static callsite is invoked on a dummy receiver object. Thus, in our PAG representation (Figure 6), passing arguments and receiving return values for a method must all flow through its "**this**" variable.

P3Ctx efficiently verify **CS-C1**– **CS-C3** as follows: For **CS-C1** (Equation (21)), where L_F is substituted with L_{D^r}, it is unnecessary to trace from an object along its $\mathsf{flowsto}$ paths. Instead, for each method, we start from its "**this**" variable, over-approximating that some object O can flow into it. For **CS-C2**, summary edges are utilized to confirm the balanced-parentheses property in L_C-paths. Finally, to ascertain **CS-C3**, we check for the existence of any $q \in Q$ such that:

$$\langle \texttt{this}^{\texttt{M}}, \mathsf{flows}\rangle \rightarrowtail^+ \langle n, q\rangle \rightarrowtail^+ \langle \texttt{this}^{\texttt{M}}, \overline{\mathsf{flows}}\rangle \tag{27}$$

where `M` is the containing method of n. This implies that n lies on an L_{D^r}-path collecting some values coming from outside `M` via $\texttt{this}^{\texttt{M}}$ and pumping them out of `M` via $\texttt{this}^{\texttt{M}}$.

Let $R : Q \mapsto \wp(N)$ return the set of nodes in G reached at a state $q \in Q$. Then verifying **CS-C3**, i.e., checking Equation (27) involves determining if the following condition holds:

$$n \in R(\mathcal{O}) \quad \vee \quad n \in R(\mathsf{flows}) \cap R(\overline{\mathsf{flows}}) \tag{28}$$

Equation (27) is satisfied either when the first disjunct applies (due to **PROP-O**) or when the second disjunct applies (due to **PROP-V**).

Figure 11 outlines P3Ctx's pre-analysis algorithm using three rules that streamline inter-procedural reachability in G. Here, $R^{-1} : N \mapsto \wp(Q)$ inversely maps nodes to their reachable states. The rules are: [F-Init] for initializations, [F-Propa] for iterative state reachability determination, and [F-Sum] for applying standard context-sensitive summaries [46] at callsites. This involves adding summary edges $n_1 \xrightarrow{\mathsf{balanced}} n_2$ to encapsulate inter-procedural reachability, thereby streamlining reachability computations for method `M`.

$$\frac{n_1 \underset{\hat{c}}{\Rightarrow} \texttt{this}^{\texttt{M}} \in E}{\texttt{this}^{\texttt{M}} \in R(\mathsf{flows}) \quad \mathsf{flows} \in R^{-1}(\texttt{this}^{\texttt{M}})} \quad \text{[F-INIT]}$$

$$\frac{n_1 \xrightarrow{\ell} n_2 \in E \quad q_1 \in R^{-1}(n_1) \quad \delta(q_1, \ell) = q_2}{n_2 \in R(q_2) \quad q_2 \in R^{-1}(n_2)} \quad \text{[F-PROPA]}$$

$$\frac{n_1 \underset{\hat{c}}{\Rightarrow} \texttt{this}^{\texttt{M}} \in E \quad \texttt{this}^{\texttt{M}} \underset{\check{c}}{\Rightarrow} n_2 \in E \quad \overline{\mathsf{flows}} \in R^{-1}(\texttt{this}^{\texttt{M}})}{n_1 \xrightarrow{\mathsf{balanced}} n_2 \in E} \quad \text{[F-SUM]}$$

Figure 11 Rules for conducting P3CTX over $G = (N, E)$.

▶ **Theorem 7.** *kCFA (performed in terms of the rules in Figure 1) produces exactly the same points-to information when performed with selective context-sensitivity under P3CTX.*

Proof. Follows from the facts that (1) Equation (21) provides necessary conditions for supporting selective context-sensitivity, (2) L_{DCR} provides a specification of kCFA with CFL-reachability for callgraph construction, (3) $L_{D^rC} \supseteq L_{DCR}$, and (4) [F-INIT] has weakened `CS-C1` by starting from the `this` variable of every method instead of every object O. ◀

The worst-case time complexity of P3CTX in analyzing a program on $G = (N, E)$ is $O(|E| \times |Q|)$, which is linear to $|E|$ as $|Q|$ (the number of states in our DFA) is a constant.

4.2 Evaluation

We demonstrate that P3CTX significantly speeds up kCFA while maintaining precision. Compared to non-precision-preserving pre-analyses, SELECTX [33] and ZIPPER [29], P3CTX excels in achieving more efficient precision trade-offs in certain application scenarios.

4.2.1 Experimental Setup

We implemented kCFA (Figure 1) and P3CTX (Figure 11) in SOOT [59], using its context-insensitive pointer analysis, SPARK [26], for PAG construction. To compare P3CTX with SELECTX and ZIPPER, we used their existing implements from the SELECTX artifact [34]. Our evaluation follows pointer analysis standards [35, 33, 32, 42, 58, 14, 16], including using TAMIFLEX [4] for Java reflection, SOOT's native code summaries, and context-insensitive analysis for special objects like strings and exceptions, distinguished per dynamic type.

We selected a set of 13 benchmarks from the DaCapo benchmark suite (latest version `6cf0380`) [3] along with a large Java library (`JRE1.8.0_31`). We excluded `jython` because both kCFA and P-kCFA could not scale this benchmark due to its overly conservative reflection log [57]. Our artifact is publicly available at [19].

Our experiments were conducted on an Intel(R) Xeon(R) W-2245 3.90GHz machine with 512GB of RAM, operating under Ubuntu 20.04.3 LTS (Focal Fossa).

4.2.2 Results

Table 2 presents the results for kCFA and its three accelerated variants: P-kCFA (by P3CTX), S-kCFA (by SELECTX), and Z-kCFA (by ZIPPER), along with SPARK for comparison purposes, focusing on $k \in \{1, 2\}$. For $k \geqslant 3$, kCFA is unscalable for all 13 programs under a 12-hour budget and thus has never been considered in the literature [33, 42, 29, 30, 58, 50, 20, 57].

Table 2 Main analysis results. The analysis times for P-kCFA, S-kCFA, and Z-kCFA are given as $x(y)$, where x is the pointer analysis time and y is the pre-analysis time (in seconds). For all metrics, smaller is better.

Program	Metrics	SPARK	1CFA	P-1CFA	S-1CFA	Z-1CFA	2CFA	P-2CFA	S-2CFA	Z-2CFA
avrora	Time(secs)	6.6	18.0	4.7 (1.2)	3.1 (21.5)	2.8 (4)	577.1	142.5 (1.2)	16.8 (21.6)	11.2 (4)
	#Call Edges	57509	55267	55267	55267	55403	54505	54505	54506	54662
	#Fail Casts	1197	931	931	931	965	890	890	895	942
	#Alias Pairs	22327	13700	13700	13700	13703	13268	13268	13280	13547
	Avg PTS	36.19	25.87	25.87	25.87	26.48	24.78	24.78	24.80	25.47
batik	Time(secs)	30.9	81.0	28.0 (4.7)	25.3 (169.5)	23.1 (243)	1473.9	466.5 (4.8)	271.1 (174.4)	276.5 (234)
	#Call Edges	171409	151995	151995	151997	152025	147428	147428	147430	150549
	#Fail Casts	4573	3709	3709	3709	3713	3485	3485	3490	3620
	#Alias Pairs	68130	38005	38005	38005	38012	32288	32288	32300	33295
	Avg PTS	114.43	71.67	71.67	71.67	71.71	66.65	66.65	66.65	68.21
eclipse	Time(secs)	14.8	48.7	23.3 (2.0)	20.1 (54.6)	19.7 (14)	1221.1	331.0 (2.0)	171.8 (56.8)	143.9 (14)
	#Call Edges	110089	97960	97960	98000	98052	93662	93662	93703	93746
	#Fail Casts	2896	2470	2470	2471	2474	2322	2322	2328	2337
	#Alias Pairs	107389	58489	58489	58500	58504	51404	51404	51427	51716
	Avg PTS	101.12	63.49	63.49	63.47	63.80	59.28	59.28	59.26	59.64
fop	Time(secs)	76.0	318.8	123.1 (10.6)	113.1 (603.8)	104.0 (355)	6019.6	2399.6 (10.8)	1901.7 (604.5)	1405.1 (354)
	#Call Edges	358738	325547	325547	325551	325591	313954	313954	313958	321008
	#Fail Casts	9057	8226	8226	8228	8239	7931	7931	7938	8084
	#Alias Pairs	323628	277047	277047	277047	277065	267389	267389	267401	268943
	Avg PTS	233.48	141.19	141.19	141.19	141.25	132.98	132.98	132.98	135.43
h2	Time(secs)	16.1	75.7	18.5 (2.9)	15.8 (74.1)	14.3 (40)	6406.8	4164.6 (2.8)	3807.8 (74.4)	3127.4 (39)
	#Call Edges	144711	135775	135775	135782	135806	134234	134234	134241	134274
	#Fail Casts	2880	2477	2477	2477	2482	2398	2398	2404	2433
	#Alias Pairs	77978	39209	39209	39209	39236	33331	33331	33351	33632
	Avg PTS	72.61	34.61	34.61	34.61	34.68	32.63	32.63	32.64	33.20
luindex	Time(secs)	18.5	41.0	24.0 (1.9)	22.6 (48.1)	20.1 (8)	829.1	232.3 (1.9)	109.0 (48.2)	82.3 (8)
	#Call Edges	85850	79431	79431	79431	79602	78190	78190	78190	78404
	#Fail Casts	1726	1359	1359	1360	1376	1286	1286	1292	1314
	#Alias Pairs	50530	32905	32905	32905	32908	31795	31795	31807	32083
	Avg PTS	53.10	24.75	24.75	24.75	24.87	23.04	23.04	23.04	23.15
lusearch	Time(secs)	5.3	12.6	3.5 (1.0)	2.3 (13.9)	1.9 (3)	414.0	129.3 (1.0)	9.6 (13.9)	7.1 (3)
	#Call Edges	45285	43117	43117	43117	43198	42412	42412	42412	42516
	#Fail Casts	955	702	702	702	719	660	660	665	696
	#Alias Pairs	20382	11693	11693	11693	11696	11263	11263	11275	11542
	Avg PTS	31.38	20.73	20.73	20.74	20.85	19.73	19.73	19.75	19.94
pmd	Time(secs)	20.3	109.5	42.6 (3.0)	37.2 (139.1)	35.9 (25)	16006.8	13715.8 (3.0)	13671.4 (139.1)	9356.3 (25)
	#Call Edges	159395	153150	153150	153150	153387	152090	152090	152090	152242
	#Fail Casts	4702	4321	4321	4321	4325	4233	4233	4238	4263
	#Alias Pairs	114914	95977	95977	95977	95979	93083	93083	93095	93353
	Avg PTS	90.97	68.76	68.76	68.76	68.79	67.48	67.48	67.49	67.58
sunflow	Time(secs)	9.9	25.9	7.4 (1.8)	5.5 (46.4)	5.3 (9)	643.1	165.1 (1.7)	33.0 (45.9)	27.7 (9)
	#Call Edges	77346	74198	74198	74200	74241	73392	73392	73394	73685
	#Fail Casts	2192	1771	1771	1773	1776	1649	1649	1656	1684
	#Alias Pairs	36952	21670	21670	21670	21678	20703	20703	20715	21041
	Avg PTS	51.31	33.62	33.62	33.62	33.69	31.34	31.34	31.36	31.79
tomcat	Time(secs)	7.4	18.9	5.8 (1.3)	4.0 (20.8)	3.7 (4)	632.9	148.7 (1.3)	16.1 (20.8)	11.7 (4)
	#Call Edges	60649	57933	57933	57933	58024	57073	57073	57073	57369
	#Fail Casts	1264	959	959	960	963	874	874	880	910
	#Alias Pairs	30775	24504	24504	24504	24507	22202	22202	22214	22482
	Avg PTS	39.88	25.37	25.37	25.37	25.51	24.03	24.03	24.04	24.62
tradebeans	Time(secs)	8.7	25.9	7.6 (1.5)	5.6 (41.7)	5.2 (9)	737.4	166.5 (1.5)	30.2 (43.4)	18.2 (9)
	#Call Edges	70911	67742	67742	67742	67858	66814	66814	67018	67207
	#Fail Casts	1523	1132	1132	1132	1135	1054	1054	1059	1068
	#Alias Pairs	36256	27175	27175	27175	27178	25683	25683	25695	25950
	Avg PTS	47.67	31.80	31.80	31.80	31.87	29.95	29.95	29.98	30.18
tradesoap	Time(secs)	8.4	24.8	7.7 (1.6)	5.8 (46.8)	5.2 (9)	703.0	162.8 (1.5)	29.9 (49.4)	17.9 (9)
	#Call Edges	70911	67742	67742	67742	67858	66814	66814	67018	67207
	#Fail Casts	1523	1132	1132	1132	1135	1054	1054	1059	1068
	#Alias Pairs	36256	27175	27175	27175	27178	25683	25683	25695	25950
	Avg PTS	47.67	31.80	31.80	31.80	31.87	29.95	29.95	29.98	30.18
xalan	Time(secs)	8.5	27.3	7.4 (1.4)	5.5 (42.6)	5.0 (16)	702.8	162.3 (1.6)	34.2 (42.3)	26.0 (16)
	#Call Edges	69608	67132	67132	67132	67210	66360	66360	66360	66448
	#Fail Casts	1807	1473	1473	1473	1477	1419	1419	1424	1441
	#Alias Pairs	42119	28280	28280	28280	28283	27259	27259	27271	27539
	Avg PTS	45.29	29.41	29.41	29.41	29.47	28.29	28.29	28.30	28.41

4.2.2.1 Precision

Pointer analysis precision is gauged using four key metrics: (1) "#Call Edges", indicating discovered call graph edges; (2) "#Fail Casts", representing potential type cast failures; (3) "#Alias Pairs", counting base variable pairs in stores and loads that may alias, excluding trivial must-aliases like direct assignments [10]; and (4) "Avg PTS", the average number of objects pointed to by reachable local variables. Lower metric values signify higher precision.

For each metric M, M_{PTA} denotes the result obtained by PTA, where PTA denotes any pointer analysis in {Spark, kCFA, P-kCFA, S-kCFA, Z-kCFA }. Let A-kCFA $\in$ $\{P\text{-}k\mathsf{CFA}, S\text{-}k\mathsf{CFA}, Z\text{-}k\mathsf{CFA}\}$ be one of the three variants of kCFA such that A-kCFA is no less precise than Spark but no more precise than kCFA. We define the precision loss of A-kCFA with respect to kCFA on metric M as:

$$\Delta^M_{A\text{-}k\mathsf{CFA}} = \frac{(M_{\textsc{Spark}} - M_{k\mathsf{CFA}}) - (M_{\textsc{Spark}} - M_{A\text{-}k\mathsf{CFA}})}{M_{\textsc{Spark}} - M_{k\mathsf{CFA}}} = \frac{M_{A\text{-}k\mathsf{CFA}} - M_{k\mathsf{CFA}}}{M_{\textsc{Spark}} - M_{k\mathsf{CFA}}} \tag{29}$$

The precision gain from Spark to kCFA is 100%. If A-kCFA matches kCFA in precision ($M_{A\text{-}k\mathsf{CFA}} = M_{k\mathsf{CFA}}$), then $\Delta^M_{A\text{-}k\mathsf{CFA}} = 0\%$, indicating no precision loss in A-kCFA. Conversely, if A-kCFA reverts to Spark's precision ($M_{A\text{-}k\mathsf{CFA}} = M_{\textsc{Spark}}$), $\Delta^M_{A\text{-}k\mathsf{CFA}} = 100\%$, reflecting a complete loss of kCFA's precision advantage.

P-kCFA retains precision, matching kCFA across all metrics in 13 benchmarks, supported by Theorem 7 and Table 2. S-kCFA, leveraging L_{FC} for context-sensitivity, has small average precision losses of 0.8%, 1.2%, 0.1%, and 0.1% in "#Call Edges", "#Fail Casts", "#Alias Pairs", and "Avg PTS", respectively, at $k = 2$. However, for "#Call Edges", S-2CFA incurs a 5% precision loss in both `tradebeans` and `tradesoap`. Conversely, Z-kCFA experiences higher average precision losses of 6.2%, 8.1%, 2.2%, and 2.0% for the same metrics at $k = 2$, attributed to Zipper's use of pattern-based heuristics for context-sensitivity decisions.

To explore S-2CFA's precision loss in `tradebeans` (Figure 12), it is noted that S-2CFA fails to identify the call in line 15 as monomorphic, unlike P-2CFA. When `put()` is invoked on a `TreeMap` object, a virtual call `compare()` occurs on the `comparator` object stored in the `TreeMap` object. With 2CFA, `put()` is analyzed under contexts [`L1`] and [`L2`]. Under [`L1`], `cmp` links to `CMP1` and `k` to `I`, leading to `compare()` from line 10 to be invoked under [`L3`, `L1`]. Under [`L2`], `cmp` points to `CMP2` and `k` to `S1`, calling `compare()` from line 14 under [`L3`, `L2`], making `o1` point uniquely to `S1`. Thus, the virtual call in line 15 invokes only the `toString()` method defined in `java.lang.String`.

SelectX, using L_{FC}, treats `cmp` and `k` in `put()` as context-insensitive, violating `CS-C3` in Equation (21). With S-2CFA, `o1` erroneously points to both `I` and `S1` under [`L3`, `L2`], leading to a polymorphic call in line 15. In contrast, P3Ctx with L_{DCR} treats these as context-sensitive, adhering to `CS-C3`, resulting in `o1` pointing only to `S1` and ensuring a monomorphic call in line 15. This change prevents a 5% precision loss in "#Call Edges", potentially enhancing critical software security analyses.

4.2.2.2 Efficiency

In Table 2, the efficiency of a pointer analysis is gauged by the time required in analyzing a program. This includes time for both the pointer analysis and the corresponding pre-analysis in each kCFA variant, denoted as $A\mathrm{-}k\mathsf{CFA}$ ($A \in \{P, S, Z\}$). For $k = 1$ and $k = 2$, pre-analysis is done separately, causing slight differences in pre-analysis times for the same program. Spark's time is not included, as its results are shared by all three pre-analyses.

```
class TreeMap {
  Comparator comparator;
  TreeMap(Comparator cmp1) { this.comparator = cmp1; }
  void put(Object k, Object v) {
    Comparator cmp = this.comparator;
    int i = cmp.compare(k, ...); // L3
}}
// in java.lang.String
class CaseInsensitiveComparator implements Comparator {
  int compare(String p1, String p2) { return 0; }
}
// in org.apache.geronimo.main
class StringComparator implements Comparator {
  int compare(Object o1, Object o2) {
    String s1 = o1.toString(); // #Call Edges?
    return s1.compareTo(o2.toString());
}}
void main() {
  Comparator cmp1 = new CaseInsensitiveComparator(); // CMP1
  Comparator cmp2 = new StringComparator(); // CMP2
  TreeMap map1 = new TreeMap(cmp1); // M1
  TreeMap map2 = new TreeMap(cmp2); // M2
  Integer x = new Integer(1); // I
  String y = new String(); // S1
  z = new String(); // S2
  map1.put(x, z); // L1
  map2.put(y, z); // L2
}
```

Figure 12 An example abstracted from `tradebeans` and `JDK8` to illustrate why SELECTX is not precision-preserving (by applying L_{FC} to determine precision-critical variables/objects in a program).

Table 2 reveals that P3CTX, SELECTX, and ZIPPER significantly boost kCFA for $k = 2$. Z-2CFA leads with 1.7× to 41.0× speedups, averaging 10.9×. S-2CFA ranges from 1.2× to 17.6×, averaging 6.0×. P3CTX increases speeds from 1.2× to 4.4×, averaging 3.2×. At $k = 1$, P3CTX performs best due to lower pre-analysis overhead and faster 1CFA. ZIPPER moderately improves 1CFA for most programs, but less effectively than P3CTX. SELECTX slows down 1CFA when including pre-analysis time. For P-1CFA, speedups range from 1.6× to 3.5×, averaging 2.6×. Z-1CFA sees 0.3× to 2.6× speedups, averaging 1.5×. S-1CFA shows no gains, with 0.4× to 0.8× speedups, averaging 0.6×.

When assessing the precision and efficiency of P-kCFA, S-kCFA, and Z-kCFA, several key insights emerge. For tasks where precision is paramount, such as in software security analysis, P-kCFA emerges as the superior choice. It offers a speed advantage without compromising the precision inherent to kCFA. In contexts where the precision of 1CFA is needed, but with greater efficiency, P-1CFA is the standout option. It surpasses both S-1CFA and Z-1CFA in terms of speed while retaining the precision level of 1CFA. Finally, for applications requiring pointer analysis at the precision level of 2CFA, the recommendation depends on the user's priorities: Z-2CFA for those valuing efficiency above precision, S-2CFA for those who prioritize efficiency but can accept minor precision loss, and P-2CFA for those who deem precision crucial but also desire increased speed.

5 Related Work

In this section, we focus exclusively on prior work that is directly relevant to our study.

CFL-Reachability. CFL-reachability, introduced in program analysis for inter-procedural dataflow analysis [46, 44], has been applied in tackling various problems such as pointer analysis [54, 53, 64, 61, 62, 48, 63, 35, 32], information flow [37, 28, 36], and type inference [43, 41]. Traditionally, kCFA's CFL-reachability formulation [53, 62, 48] relies on a separate call graph construction algorithm, either pre-applied or on-the-fly. This paper introduces L_{DCR}, a new CFL-reachability formulation for kCFA, integrating built-in call graph construction. An earlier attempt to address the same problem by Sridharan [52] is sound but less precise than L_{DCR} due to the lack of L_R. Without L_R, a context used for parameter passing at a virtual callsite can be incorrectly restored as a different context after finding the dispatched method and returning to the same callsite (as in Figure 9).

Another line of research on CFL-reachability focuses on its computational complexity. Generally, the all-pairs CFL-reachability problem can be resolved in $O(m^3n^3)$ time, where m is the CFL grammar size and n is the graph node count. Kodumal et al. [23] efficiently solved Dyck-CFL-reachability in $O(mn^3)$. Chaudhuri [7] later optimized the general CFL-reachability algorithm to subcubic time using the Four Russians' Trick [24]. Zhang et al. [63] demonstrated that bidirected Dyck-CFL reachability could be solved in $O(n + p \log p)$ (with p being the graph edge count), noting that reachability in a bidirected graph forms an equivalence relation. This complexity was further reduced to $O(p + n \cdot \alpha(n))$ in [6], where $\alpha(n)$ is the inverse Ackermann function. This paper introduces P3CTX, an L_{DCR}-enabled pre-analysis for accelerating kCFA, linear in terms of the number of PAG edges in the program's PAG and preserving precision.

A CFL-reachability-based formulation recently proposed for object-sensitive pointer analysis [35, 38, 39] naturally includes call graph construction, as it uses receiver objects as context elements. However, integrating call graph construction into callsite-sensitive analyses using the traditional CFL-reachability framework [53, 62, 48] is challenging, as detailed in Section 2. An earlier attempt [52] was sound but lacked precision, particularly in restoring contexts correctly after method dispatch and return at virtual callsites, as shown in Figure 9. L_{DCR} is the first known solution to effectively integrate call graph construction into CFL-reachability for callsite-sensitive analyses.

Selective Context-sensitivity. In the realm of pointer analysis acceleration, three primary approaches exist: pattern-based [51, 12, 29, 30], data-driven [21, 20], and CFL-reachability-guided [35, 33, 14, 13]. By exploiting CFL-reachability, EAGLE [35, 32], TURNER [14], CONCH [16, 18], and DEBLOATERX [13] represent recent efforts in accelerating object-sensitive pointer analysis [39]. SELECTX [33] marks the initial CFL-reachability-based effort to accelerate kCFA, but it lacks precision preservation due to its reliance on L_{FC} [53]. This paper introduces P3CTX, the first precision-preserving pre-analysis for kCFA, grounded in L_{DCR}.

6 Conclusion

We have introduced L_{DCR}, a new CFL-reachability formulation for supporting k-callsite-based context-sensitive pointer analysis (kCFA), featuring a unique built-in call graph construction to effectively handle dynamic dispatch. To demonstrate its utility, we have also introduced P3CTX, which is developed based on L_{DCR}, to enhance the performance of kCFA while preserving its precision. We hope that L_{DCR} can provide some new insights on understanding kCFA and its demand-driven forms [54, 53, 62], potentially inspiring novel algorithmic advancements. Future explorations include applying L_{DCR} to selective context sensitivity and extending its application to areas such as library-code summarization [48, 56, 8] and information flow analysis [28, 36].

References

1 Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

2 David F Bacon and Peter F Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, New York, NY, USA, 1996. Association for Computing Machinery.

3 Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, New York, NY, USA, 2006. Association for Computing Machinery.

4 Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 241–250, Honolulu, HI, USA, 2011. IEEE.

5 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 243–262, New York, NY, USA, 2009. Association for Computing Machinery.

6 Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. Optimal Dyck reachability for data-dependence and alias analysis. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2017.

7 Swarat Chaudhuri. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 159–169, New York, NY, USA, 2008. Association for Computing Machinery.

8 Yifan Chen, Chenyang Yang, Xin Zhang, Yingfei Xiong, Hao Tang, Xiaoyin Wang, and Lu Zhang. Accelerating program analyses in datalog by merging library facts. In *International Static Analysis Symposium*, pages 77–101, Cham, 2021. Springer, Springer International Publishing.

9 Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101, Berlin, Heidelberg, 1995. Springer, Springer Berlin Heidelberg.

10 Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. Bottom-up context-sensitive pointer analysis for Java. In *Programming Languages and Systems: 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30-December 2, 2015, Proceedings*, pages 465–484, Cham, 2015. Springer International Publishing.

11 David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):685–746, 2001.

12 Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. 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*, pages 13–18, New York, NY, USA, 2017. Association for Computing Machinery.

13 Dongjie He, Yujiang Gui, Wei Li, Yonggang Tao, Changwei Zou, Yulei Sui, and Jingling Xue. A container-usage-pattern-based context debloating approach for object-sensitive pointer analysis. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):971–1000, 2023.

14 Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. Accelerating object-sensitive pointer analysis by exploiting object containment and reachability. In *Proceedings of the 35th European Conference on Object-Oriented Programming (ECOOP 2021)*, pages 18:1–18:31, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

15 Dongjie He, Jingbo Lu, and Jingling Xue. A CFL-Reachability Formulation of Callsite-Sensitive Pointer Analysis with Built-in On-the-Fly Call Graph Construction (Artifact). Software, version 1.0. (visited on 2024-08-27). URL: `https://doi.org/10.5281/zenodo.11061892`.

16 Dongjie He, Jingbo Lu, and Jingling Xue. Context debloating for object-sensitive pointer analysis. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 79–91, New York, NY, USA, 2021. IEEE. `doi:10.1109/ASE51524.2021.9678880`.

17 Dongjie He, Jingbo Lu, and Jingling Xue. Qilin: A new framework for supporting fine-grained context-sensitivity in Java pointer analysis. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:29, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECOOP.2022.30`.

18 Dongjie He, Jingbo Lu, and Jingling Xue. IFDS-based context debloating for object-sensitive pointer analysis. *ACM Transactions on Software Engineering and Methodology*, 2023.

19 Dongjie He, Jingbo Lu, and Jingling Xue. A CFL-reachability formulation of callsite- sensitive pointer analysis with built-in on-the- fly call graph construction (artifact), July 2024. `doi:10.5281/zenodo.11061892`.

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

21 Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. Data-driven context-sensitivity for points-to analysis. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):100, 2017.

22 George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 423–434, New York, NY, USA, 2013. Association for Computing Machinery.

23 John Kodumal and Alex Aiken. The set constraint/CFL reachability connection in practice. *ACM Sigplan Notices*, 39(6):207–218, 2004.

24 VL Arlazarov EA Dinic MA Kronrod and IA Faradzev. On economic construction of the transitive closure of a directred graph. In *Dokl. Acad. Nauk SSSR*, pages 487–88, 1970.

25 Michael John Latta. *The intersection of context-free languages*. PhD thesis, University of Texas at Austin, USA, 1993. URL: `https://www.proquest.com/docview/304086568?pq-origsite=gscholar&fromopenview=true`.

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

27 Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(1):1–53, 2008.

28 Yuanbo Li, Qirun Zhang, and Thomas Reps. Fast graph simplification for interleaved Dyck-reachability. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 780–793, New York, NY, USA, 2020. Association for Computing Machinery.

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

30 Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. A principled approach to selective context sensitivity for pointer analysis. *ACM Transactions on Programming Languages and Systems*, 42(TOPLAS):1–40, 2020.

31 Leonard Y Liu and Peter Weiner. An infinite hierarchy of intersections of context-free languages. *Mathematical systems theory*, 7:185–192, 1973. `doi:10.1007/BF01762237`.

32 Jingbo Lu, Dongjie He, and Jingling Xue. Eagle: CFL-reachability-based precision-preserving acceleration of object-sensitive pointer analysis with partial context sensitivity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(4):1–46, 2021.

33 Jingbo Lu, Dongjie He, and Jingling Xue. Selective context-sensitivity for k-CFA with CFL-reachability. In *International Static Analysis Symposium*, pages 261–285, Cham, 2021. Springer, Springer International Publishing.

34 Jingbo Lu, Dongjie He, and Jingling Xue. Selective context-sensitivity for k-CFA with CFL-reachability (artifact), July 2021. `doi:10.5281/zenodo.4732680`.

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

36 Ana Milanova. FlowCFL: generalized type-based reachability analysis: graph reduction and equivalence of CFL-based and type-based reachability. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

37 Ana Milanova, Wei Huang, and Yao Dong. CFL-reachability and context-sensitive integrity types. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 99–109, New York, NY, USA, 2014. Association for Computing Machinery.

38 Ana Milanova, Atanas Rountev, and Barbara G Ryder. 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*, pages 1–11, New York, NY, USA, 2002. Association for Computing Machinery.

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

40 Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, New York, NY, USA, 2006. Association for Computing Machinery.

41 Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. Existential label flow inference via CFL reachability. In *International Static Analysis Symposium*, pages 88–106, Berlin, Heidelberg, 2006. Springer, Springer Berlin Heidelberg.

42 Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. User-guided program reasoning using bayesian inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 722–735, New York, NY, USA, 2018. Association for Computing Machinery.

43 Jakob Rehof and Manuel Fähndrich. Type-based flow analysis: from polymorphic subtyping to CFL-reachability. *ACM SIGPLAN Notices*, 36(3):54–66, 2001.

44 Thomas Reps. Program analysis via graph reachability. *Information and software technology*, 40(11-12):701–726, 1998.

45 Thomas Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems*, 22(1):162–186, 2000.

46 Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, New York, NY, USA, 1995. Association for Computing Machinery.

47 Barbara G Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *International Conference on Compiler Construction*, pages 126–137, Berlin, Heidelberg, 2003. Springer, Springer Berlin Heidelberg.

48 Lei Shang, Xinwei Xie, and Jingling Xue. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 264–274, New York, NY, USA, 2012. Association for Computing Machinery.

49 Olin Grigsby Shivers. *Control-flow analysis of higher-order languages or taming lambda.* PhD thesis, Carnegie Mellon University, 1991. CMU-CS-91-145.

50 Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–30, New York, NY, USA, 2011. Association for Computing Machinery.

51 Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 485–495, New York, NY, USA, 2014. Association for Computing Machinery.

52 Manu Sridharan. *Refinement-based program analysis tools.* University of California, Berkeley, 2007.

53 Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 387–400, New York, NY, USA, 2006. Association for Computing Machinery.

54 Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 59–76, New York, NY, USA, 2005. Association for Computing Machinery.

55 Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. *ACM SIGPLAN Notices*, 35(10):264–280, 2000.

56 Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–95, New York, NY, USA, 2015. Association for Computing Machinery.

57 Rei Thiessen and Ondřej Lhoták. Context transformations for pointer analysis. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 263–277, New York, NY, USA, 2017. Association for Computing Machinery.

58 Tian Tan, Yue Li and Jingling Xue. 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*, pages 278–291, New York, NY, USA, 2017. Association for Computing Machinery.

59 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., USA, 2010.

60 WALA. WALA: T.J. Watson Libraries for Analysis, 2024. URL: `https://github.com/wala/WALA`.

61 Guoqing Xu, Atanas Rountev, and Manu Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *European Conference on Object-Oriented Programming*, pages 98–122, Berlin, Heidelberg, 2009. Springer, Springer Berlin Heidelberg.

62 Dacong Yan, Guoqing Xu, and Atanas Rountev. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 155–165, New York, NY, USA, 2011. Association for Computing Machinery.

63 Qirun Zhang, Michael R Lyu, Hao Yuan, and Zhendong Su. 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*, pages 435–446, New York, NY, USA, 2013. Association for Computing Machinery.

64 Xin Zheng and Radu Rugina. Demand-driven alias analysis for c. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 197–208, New York, NY, USA, 2008. Association for Computing Machinery.