Efficient and Precise Pointer Analysis with Fine-Grained Context Sensitivity

Author:
He, Dongjie

Publication Date:
2022

License:
https://creativecommons.org/licenses/by/4.0/
Link to license to see what you are allowed to do with this resource.

Downloaded from http://hdl.handle.net/1959.4/100280 in https://unsworks.unsw.edu.au on 2022-05-02

Efficient and Precise Pointer Analysis with Fine-Grained Context Sensitivity

by

Dongjie He

A THESIS IN FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

April 30, 2022

School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

All rights reserved. This work may not be reproduced in any form without the permission of the author. © Dongjie He 2022

Thesis submission for the degree of Doctor of Philosophy

Thesis Title and Abstract | Declarations | Inclusion of Publications Statement | Corrected Thesis and Responses

ORIGINALITY STATEMENT

☑ I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

COPYRIGHT STATEMENT

☑ I hereby grant the University of New South Wales or its agents a non-exclusive licence to archive and to make available (including to members of the public) my thesis or dissertation in whole or part in the University libraries in all forms of media, now or here after known. I acknowledge that I retain all intellectual property rights which subsist in my thesis or dissertation, such as copyright and patent rights, subject to applicable law. I also retain the right to use all or part of my thesis or dissertation in future works (such as articles or books).

For any substantial portions of copyright material used in this thesis, written permission for use has been obtained, or the copyright material is removed from the final public version of the thesis.

AUTHENTICITY STATEMENT

☑ I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis.

Thesis submission for the degree of Doctor of Philosophy

Thesis Title and Abstract | Declarations | **Inclusion of Publications Statement** | Corrected Thesis and Responses

UNSW is supportive of candidates publishing their research results during their candidature as detailed in the UNSW Thesis Examination Procedure.

Publications can be used in the candidate's thesis in lieu of a Chapter provided:

- The candidate contributed **greater than 50%** of the content in the publication and are the "primary author", i.e. they were responsible primarily for the planning, execution and preparation of the work for publication.
- The candidate has obtained approval to include the publication in their thesis in lieu of a Chapter from their Supervisor and Postgraduate Coordinator.
- The publication is not subject to any obligations or contractual agreements with a third party that would constrain its inclusion in the thesis.

☑ The candidate has declared that **their thesis has publications - either published or submitted for publication - incorporated into it in lieu of a Chapter/s. Details of these publications are provided below.**.

Publication Details #1

Full Title:	Accelerating Object-Sensitive Pointer Analysis by Exploiting Object Containment and Reachability
Authors:	Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue
Journal or Book Name:	35th European Conference on Object-Oriented Programming (ECOOP'21)
Volume/Page Numbers:	
Date Accepted/Published:	
Status:	published
The Candidate's Contribution to the Work:	I have designed and implemented the whole work.
Location of the work in the thesis and/or how the work is incorporated in the thesis:	I have included this publication in Chapter 3.

Publication Details #2

Full Title:	Context Debloating for Object- Sensitive Pointer Analysis
Authors:	Dongjie He, Jingbo Lu, and Jingling Xue
Journal or Book Name:	36th IEEE/ACM International Conference on Automated Software Engineering (ASE'21)
Volume/Page Numbers:	
Date Accepted/Published:	
Status:	published
The Candidate's Contribution to the Work:	I have designed and implemented the whole work.
Location of the work in the thesis and/or how the work is incorporated in the thesis:	I have included this publication in Chapter 4.

Publication Details #3

Full Title:	Qilin: A New Framework for Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis
Authors:	Dongjie He, Jingbo Lu, and Jingling Xue
Journal or Book Name:	36th European Conference on Object-Oriented Programming (ECOOP'22)
Volume/Page Numbers:	
Date Accepted/Published:	
Status:	accepted
The Candidate's Contribution to the Work:	I have designed and implemented the whole work.
Location of the work in the thesis and/or how the work is incorporated in the thesis:	I have included part of this publication in Section 2.3.4 of Chapter 2.

Publication Details #4

Full Title:	CFL-Reachability Formulation of Callsite-Sensitive Pointer Analysis with Built-in On-the-Fly Call Graph Construction.
Authors:	Dongjie He, Jingbo Lu, and Jingling Xue
Journal or Book Name:	the 2022 volume of PACMPL(OOPSLA)
Volume/Page Numbers:	
Date Accepted/Published:	
Status:	submitted
The Candidate's Contribution to the Work:	I have designed and implemented most of this work.
Location of the work in the thesis and/or how the work is incorporated in the thesis:	I have included this publication in Chapter 5.

Candidate's Declaration

I confirm that where I have used a publication in lieu of a chapter, the listed publication(s) above meet(s) the requirements to be included in the thesis. I also declare that I have complied with the Thesis Examination Procedure.

Abstract

Pointer analysis addresses a fundamental problem in program analysis: determining statically whether or not a given pointer may reference an object in the program. It underpins almost all forms of other static analysis, including program understanding, program verification, bug detection, security analysis, compiler optimization, and symbolic execution. However, existing pointer analysis techniques suffer from efficiency and scalability issues for large programs. Improving their efficiency while still maintaining their precision is a long-standing hard problem.

This thesis aims to improve the efficiency and scalability of pointer analysis for object-oriented programming languages such as Java by exploring *fine-grained context sensitivity*. Unlike traditional approaches, which apply context-sensitivity either uniformly to all methods or selectively to a subset of methods in a program, we go one step further by applying context-sensitivity only to a subset of precision-critical variables and objects so that we can reduce significantly the scale of *Pointer Assignment Graph* (PAG). Conducting pointer analysis on a smaller PAG enables the pointer analysis to run significantly faster while preserving most of its precision.

This thesis makes its contributions by introducing three different fine-grained pointer analysis approaches for Java programs. The first approach, called TURNER, can accelerate k-object-sensitive pointer analysis (i.e., kOBJ) for Java significantly with negligible precision loss by exploiting object containment and reachability.

The second approach, called *context debloating*, can accelerate all existing object-sensitive pointer analysis algorithms for Java by eliminating the context explosion problem completely for context-independent objects. In addition, we have also developed the first supporting tool, named CONCH, for identifying context-independent objects. The last approach, called P3CTX, represents the first precision-preserving technique for accelerating k-callsite-sensitive pointer analysis (k-CFA) for Java based on a complete CFL-reachability formulation of k-CFA for Java with built-in on-the-fly call graph construction (for the first time).

Publications

- **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. In *submission*.

- **Dongjie He**, Jingbo Lu, Yaoqing Gao and Jingling Xue. Selecting Context-Sensitivity Modularly for Accelerating Object-Sensitive Pointer Analysis. In *IEEE Transactions on Software Engineering (TSE'22)*.

- **Dongjie He**, Jingbo Lu, and Jingling Xue. QILIN: A New Framework for Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis. In *36th European Conference on Object-Oriented Programming (ECOOP'22)*.

- **Dongjie He**, Jingbo Lu, and Jingling Xue. Context Debloating for Object-Sensitive Pointer Analysis. In *36th IEEE/ACM International Conference on Automated Software Engineering (ASE'21)*.

- Jingbo Lu, **Dongjie He**, and Jingling Xue. Selective Context-Sensitivity for k-CFA with CFL-Reachability. In *28th Static Analysis Symposium (SAS'21)*.

- **Dongjie He**, Jingbo Lu, Yaoqing Gao and Jingling Xue. Accelerating Object-Sensitive Pointer Analysis by Exploiting Object Containment and Reachability. In *35th European Conference on Object-Oriented Programming (ECOOP'21)*.

- Jingbo Lu, **Dongjie He**, and Jingling Xue. CFL-Reachability-based Precision-Preserving Acceleration of Object-Sensitive Pointer Analysis. In *ACM Transactions on Software Engineering and Methodology (TOSEM'21).*

- Diyu Wu, **Dongjie He**, Shiping Chen and Jingling Xue. Exposing Android Event-Based Races by Selective Branch Instrumentation. In *31st IEEE International Symposium on Software Reliability Engineering (ISSRE'20).*

- Jie Liu, **Dongjie He**, Diyu Wu and Jingling Xue. Correlating UI Contexts with Sensitive API Calls: Dynamic Semantic Extraction and Analysis. In *31st IEEE International Symposium on Software Reliability Engineering (ISSRE'20).*

- **Dongjie He**, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li and Jingling Xue. Performance-Boosting Sparsification of the IFDS Algorithm with Applications to Taint Analysis. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)* . **Distinguished Paper Award**.

Acknowledgments

I am incredibly fortunate to have many people to thank.

First and foremost, I would like to thank my supervisor Jingling Xue for his support, patience, and guidance throughout my Ph.D. study. He always discusses academic problems with me and teaches me academic writing. It was his mentorship and encouragement which made this thesis possible. His diligence, enthusiasm, and meticulous attitude to the research influence me and will continue to benefit me all my life.

I also thank my master advisor Lian Li, who guided me into the land of program analysis. He taught me writing data flow analysis in LLVM hand by hand and gave me lots of free time to explore static analysis frameworks like Doop, Soot and Wala. All these have laid a solid foundation for my doctoral research.

My special thanks go to Jingbo Lu, a great research partner during my Ph.D. study. His implementation of Eagle (together with the underlying in-house implementation of Java pointer analysis) brings me quickly to the state of the art. His work also inspired me to develop Turner, Conch, P3Ctx, and Qilin.

Great thanks to Lei Wang, who taught me the IFDS algorithm when I was designing `IctApiFinder`. In addition, the original idea of SparseDroid also came from him. I also thank Jie Lu and Yue Li for encouragement during some difficult times of my Ph.D. studies.

Also, many thanks to all the other members of the CORG group, past and present, during my Ph.D. study: Xinwei Xie, Ding Ye, Hua Yan, Jieyuan Zhang, Jie Liu, Diyu Wu, Xuezheng Xu, Changwei Zou, Xudong Wang, Yong Joo Park, Yujiang Gui, Wei Li, Yonggang Tao, and Runze You. They have made my time at UNSW memorable.

I want to thank my annual progress review panel members: Dr. Hui Wu, Dr. Annie Guo, and Prof. Sri Parameswaran, for their encouragement and invaluable advice. I also thank A/Prof. Ana Milanova, and Asst/Prof. Qirun Zhang for serving on my Examining Committee and providing valuable feedback and guidance.

Last but not least, I would like to thank all my family members: my wife, my parents, and my younger brother for their unconditional support and endless love, and my son for the joy that he brings to me. I dedicate my thesis and dear to them.

Contents

Abstract **i**

Publications **iii**

Acknowledgments **v**

List of Figures **xiii**

List of Tables **xvi**

List of Algorithms **xvii**

1 Introduction **1**

1.1 Preliminaries . . . 3

1.1.1 Pointer Analysis in a Nutshell . . . 3

1.1.2 Context Sensitivity . . . 4

1.1.3 CFL-Reachability . . . 6

1.2 Contribution Overview . . . 8

1.2.1 Accelerating kOBJ by Exploiting Object Containment and Reachability . . . 9

1.2.2 Context Debloating for Object-Sensitive Pointer Analysis . . 10

1.2.3 Precision-Preserving Acceleration for kCFA . . . 10

1.3 Publications and Organization 11

2 Background 13

2.1 A Simplified Java Language 13

2.2 Pointer Analysis: Concepts and Measurements 15

2.3 Inclusion-based Formulation 16

2.3.1 Notations 17

2.3.2 Andersen-Style Inclusion-based Formulation 17

2.3.3 Inclusion-based Formulation with Context Sensitivity 19

2.3.4 Fine-Grained Context-sensitive Pointer Analysis 22

2.4 CFL-Reachability Formulation 23

2.4.1 Callsite-based CFL-Reachability Formulation 23

2.4.2 Object-based CFL-Reachability Formulation 27

3 Accelerating *k*OBJ by Exploiting Object Containment and Reachability 31

3.1 Overview 32

3.2 Motivation 35

3.2.1 Challenges 36

3.2.2 Example 39

3.2.3 Turner: Our Approach 43

3.3 Turner 46

3.3.1 Object Containment 47

3.3.2 Object Reachability 47

3.3.3 Time Complexity 60

3.4 Evaluation 61

3.4.1 RQ1: Precision 63

3.4.2 RQ2: Efficiency 65

3.4.3 RQ3: Effectiveness 68

3.5 Conclusion 72

4 Context Debloating for Object-sensitive Pointer Analysis 75

4.1 Overview 76

4.2 Motivation 78

4.2.1 Object Sensitivity 78

4.2.2 Limitations of Existing Algorithms 80

4.2.3 Conch: Our Context Debloating Approach 82

4.3 Context Debloating 88

4.4 Conch 88

4.4.1 Verifying Observation 4.1 89

4.4.2 Verifying Observation 4.2 89

4.4.3 Verifying Observation 4.3 94

4.4.4 Soundness and Time Complexity 97

4.5 Evaluation 98

4.5.1 RQ1: Is Conch Precise and Efficient? 99

4.5.2 RQ2: Can Conch Speed Up Baseline Analyses? 100

4.6 Conclusion 105

5 Precision-Preserving Acceleration for k-CFA 107

5.1 Overview 108

5.2 Motivation 110

5.2.1 Example 110

5.2.2 Andersen-Style Inclusion-based Formulation 111

5.2.3 L_{FC}-based CFL-Reachability Formulation 112

5.2.4 $\mathcal{L}_{FCR}$: Necessity and Challenges 116
5.3 $\mathcal{L}_{FCR}$: Design and Insights 117
5.3.1 Pointer Assignment Graph 117
5.3.2 $\mathcal{L}_{FCR}$ 120
5.3.3 Time Complexities 131
5.4 P3Ctx: An $\mathcal{L}_{FCR}$ Application 132
5.4.1 CFL-Reachability-Guided Selections 132
5.4.2 Regularization of $\mathcal{L}_F$ into $\mathcal{L}_{F^r}$ 133
5.4.3 P3Ctx 135
5.5 Evaluation 138
5.5.1 Experimental Setup 138
5.5.2 Results 139
5.6 Conclusion 141

6 Related Work **143**
6.1 Selective Context-Sensitivity 143
6.2 Other Efficient Pointer Analysis Techniques 144
6.3 CFL-Reachability 145
6.4 IFDS Analysis 147

7 Summary and Future Directions **149**
7.1 Fine-Grained Pointer Analysis with Variable-Level Context Lengths 151
7.2 Design-Pattern-based Acceleration technique for Pointer Analysis . 152
7.3 Client-Oriented Pointer Analysis 152
7.4 Context Debloating for Other Context-Sensitive Program Analysis . 153
7.5 Other Potential Directions 153

Bibliography **154**

List of Figures

1.1 A graphical illustration of value flow across `id()`. 6

1.2 The labeled PAG for the `id()` example in Section 1.1.2. 7

2.1 A simplified Java language. 14

2.2 Anderson-style Inclusion-based Formulation. 18

2.3 Inclusion-based formulation with context-sensitivity (`M` is the containing method of a statement being analyzed). 20

2.4 Fine-grained Context-sensitive pointer analysis (`M` is the containing method of a statement being analyzed). 22

2.5 Inference Rules for building the PAG required by the traditional callsite-based CFL-reachability formulation [76]. 24

2.6 The PAG for a code snippet. 26

2.7 Rules for building the PAG required by L^{o}_{FC}. 27

3.1 A total of four possible value-flow patterns for determinng whether a variable `x` should be precision-critical or not. 37

3.2 A Java program abstracted from real code using the standard JDK library. 40

3.3 Computing $\overline{\texttt{PTS}}(\texttt{w1}) = \{\texttt{O1}\}$ for Figure 3.2 by 2OBJ, E-2OBJ, Z-2OBJ and T-2OBJ. 42

3.4 Rule for treating all the objects in $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}}$ as context-insensitive. . . 48

3.5 Rule for analyzing a method call. 52
3.6 Rule for adding the PAG edges for parameters. 55
3.7 The DFA as an equivalent representation of the grammar for defining L_5. 59
3.8 Rules for computing R_M and R_M^{-1} for a method M with $G_M = (N_M, E_M)$. 60
3.9 Percentage contributions made by TURNER's two analysis stages for the speedups of T-2OBJ over 2OBJ. 69
3.10 The Venn diagram of the objects in a program. 70
3.11 Imprecise points-to information computed by T-2OBJ for a top container `P`. 71
3.12 Imprecise points-to information computed by T-2OBJ for a bottom container `D`. 71

4.1 An example for illustrating object sensitivity. 79
4.2 Computing the points-to information for `v1` and `v2` in Figure 4.1 by applying Andersen's analysis and 2OBJ. 80
4.3 An example for motivating CONCH ($1 \leqslant i \leqslant n$ and $0 \leqslant j < 2^i$), reusing class `B` defined in lines 12-28 in Figure 4.1. 81
4.4 The object allocation graph (OAG) for Figure 4.3, where only the two edges in red will remain after context debloating. 82
4.5 Illustrating the conditions for an object to be context-dependent. . 83
4.6 A context-dependent object `B` violating Obs 4.1. 85
4.7 Three common cases abstracted from JDK for Obs 4.2. 85
4.8 Three common cases abstracted from JDK for Obs 4.3. 86
4.9 PAG edges for a parameterless method with no calls. 91
4.10 Two intermediate DFAs for the DFA in Figure 4.12. 91

4.11 PAG edges for parameters and return variables. 92
4.12 The DFA for verifying Obs 4.2. 93
4.13 Rules for computing leakObjects, i.e., the set of objects that can flow out of their containing methods for verifying Obs 4.2. $S_i \in \{H, F, B\}$, where $i \in \{1, 2, 3\}$ and Sym_l is a symbolic object abstracting all objects returned from call site l. 95
4.14 The DFA used for computing depOnParam. 96
4.15 Percentage distribution of the two types of objects. 103

5.1 A motivating example. 111
5.2 Disconnection in the value flows between parameter passing in L_{FC} and dynamic dispatch at a virtual callsite. 115
5.3 Rules for building the PAG required by $\mathcal{L}_{FCR}$. 118
5.4 The PAG constructed for the program given in Figure 5.1. 119
5.5 Three different approaches for performing dynamic dispatch at a virtual callsite during parameter passing. 123
5.6 A DFA for accepting $\mathcal{L}_{F^r}$. 135
5.7 Rules for conducting P3CTX over $G = (N, E)$. 137

List of Tables

3.1 The contexts used for analyzing the variables/objects in Figure 3.2 by 2OBJ, E-2OBJ, Z-2OBJ, and T-2OBJ (where i in each context containing $\mathtt{A}_i/\mathtt{a}_i$ ranges over $[1, n]$). 41

3.2 Main results. For a given $k \in \{2, 3\}$, the speedups of E-kOBJ, Z-kOBJ, and T-kOBJ are normalized with kOBJ as the baseline. For all the metrics except "Speedup", smaller is better. 64

3.3 Context-sensitive facts (in millions). For all the metrics, smaller is better. 67

3.4 Times spent by SPARK and the three pre-analyses in seconds. . . . 68

4.1 Main results. In all metrics (except for speedups), smaller is better. Given an analysis Base, Base+D is its debloated version by CONCH. OoM stands for "Out of Memory". 101

4.2 Times spent by pre-analyses in seconds. 102

4.3 Average number of contexts analyzed for a method by kOBJ, kOBJ+D, ZkOBJ and ZkOBJ+D, where $k \in \{2, 3\}$. 103

4.4 Context-sensitive facts. For all the metrics, smaller is better. . . . 104

5.1 The points-to results for the program in Figure 5.1 computed by 2-CFA according to the rules in Figure 2.3. 112

5.2 The precision and efficiency of k-CFA, and Pk-CFA. For all the metrics except speedup, smaller is better. 140

5.3 The analysis times of SPARK and P3CTX in seconds. 141

List of Algorithms

1 Conch: context debloating. 90

Chapter 1

Introduction

Modern programming languages such as C/C++, Rust, and Java come with rich program analysis techniques for studying various kinds of program properties. *Pointer Analysis* is one of the fundamental program analysis techniques that statically analyze which memory locations a pointer in a program can point to at runtime without executing the program. For object-oriented languages such as Java, a memory location is usually abstracted as a heap allocation site, and a pointer is often known as a variable (reference) or an object's field (e.g., `O.f`).

Pointer analysis underpins almost all forms of other static analysis including call graph construction [1, 36, 61], program understanding [57, 77], bug detection [17, 43, 56, 93], security analysis [4, 18, 20], compiler optimization [15, 79], and symbolic execution [30, 87, 88]. Hence, effective and precise pointer analyses are highly called for as they can benefit many client applications and other program analysis techniques.

Every day, software becomes more complex. Consequently, existing pointer analysis techniques require more time and memory resources to obtain precise points-to information. Even worse, they often suffer from severe scalability issues.

For instance, for a reasonably large object-oriented program, k-object-sensitive pointer analysis, denoted kOBJ, often requires tens of minutes or hours' analysis time to complete and usually fails to scale when $k \geq 3$.

As a result, production compilers and program analysis frameworks must often trade precision for efficiency by adopting some imprecise pointer analysis techniques, by default. For example, LLVM [35] relies on three simple (ad hoc) pointer analyses (which are either field-based, flow-insensitive or context-insensitive analysis) to perform IR optimization [46]. SOOT [89] relies on SPARK [36] (a context-insensitive, flow-insensitive inclusion-style pointer analysis) for callgraph construction. As a result, many potential optimization opportunities may be missed or many false alarms may be reported.

Theoretically, precise pointer analysis techniques (with flow-sensitivity, field-sensitivity, and context-sensitivity all considered) are undecidable (by Rice's theorem and [64]). Thus, how to make a good trade-off between precision and efficiency remains a long-standing hard problem.

In the last two decades, context sensitivity has been widely adopted in developing highly precise pointer analysis algorithms for object-oriented languages. Context-sensitive approaches analyze a method separately under different calling contexts that abstract its different run-time invocations. There are several kinds of context-sensitivity: callsite-sensitivity [70], object-sensitivity [53,54], type-sensitivity [73] and hybrid sensitivity [31].

Traditionally, context-sensitivity is enforced for all the analyzed methods indiscriminately [31, 37, 53, 54, 73]. A few recent approaches on selective context-sensitivity [29,40,74], which enforce context-sensitivity only to a subset of so-called precision-critical methods, have been proposed to improve the efficiency of pointer analysis while sometimes incurring substantial precision loss. Is it possible to im-

prove the efficiency of these existing pointer analyses further while still maintaining their precision? What are the speedups can we achieve potentially? The central theme of this thesis will unfold around these two questions.

This thesis aims to accelerate existing pointer analysis algorithms and ease their scalability issues. In particular, we are interested in exploring *fine-grained context-sensitivity* techniques on Java *pointer analysis* by applying context-sensitivity only to a subset of precision-critical variables and objects. Before presenting our contributions, we first provide some preliminaries.

1.1 Preliminaries

1.1.1 Pointer Analysis in a Nutshell

Many programming languages, such as C, C++, Rust, and Java, contain reference variables (also called pointers). In such languages, reference variables usually refer to some (heap or stack) values according to the addresses they keep. Take Java for an example by considering the following two statements:

```
A a1 = new A(); // A1
A a2 = new A(); // A2
```

`A1` and `A2` are two object values that are allocated in heap memory. Their memory addresses are saved into `a1` and `a2` (which are two reference variables created on the stack), respectively. Thus, we can use `a1` to refer to `A1`, and similarly, `a2` to refer to `A2`. In other words, `a1` points to `A1` and `a2` points to `A2`. Let $\overline{\texttt{PTS}}(v)$ be the context-insensitive points-to set of v. Then we have $\overline{\texttt{PTS}}(\texttt{a1}) = \{\texttt{A1}\}$ and $\overline{\texttt{PTS}}(\texttt{a2}) = \{\texttt{A2}\}$. Note that in Java, reference variables cannot refer to stack values but only to heap objects. Such a different variable model makes Java pointer analysis substantially different from other languages'.

The points-to relation plays a significant role in static program analysis. For example, in a data race analysis, suppose the two statements below run concurrently (without being protected by a lock discipline):

```
v1.f = a1; || v2.f = a2;
```

if we know that `v1` and `v2` are aliases, i.e., $\overline{\texttt{PTS}}(\texttt{v1}) \cap \overline{\texttt{PTS}}(\texttt{v2}) \neq \emptyset$, we can report a data race warning between the two statements since they may possibly write the memory addresses of `A1` and `A2` to the `f` field of a common heap object concurrently. Moreover, the points-to relation can also be used to construct a precise callgraph. Consider a virtual call to `showName()` whose base variable `v3` is of type **Fruit**, denoted as `v3` : **Fruit**:

```
v3.showName();
```

if we know what objects (and thus their corresponding types) `v3` may point to, we could resolve the target methods invoked and build the callgraph edges precisely. In this case, suppose $\overline{\texttt{PTS}}(v3) = \{\texttt{O1}, \texttt{O2}\}$, where `O1` : **Apple** and `O2` : **Mango**, with **Apple** and **Mango** being subtypes of **Fruit**, we could correctly resolve the two targets to be **Apple** :: `showName`() and **Mango** :: `showName`(), so that spurious call targets such as **Pear** :: `showName`() and **Grape** :: `showName`() have been successfully eliminated.

1.1.2 Context Sensitivity

Most pointer analysis techniques (especially for object-oriented languages like Java) rely on *context sensitivity*, which distinguishes the reference variables declared and heap objects allocated locally in a method under different calling contexts for developing highly precise pointer analyses.

Consider an *identity* function below:

```
Object id(Object q) {
```

```
    return q;
}
```

which will return whatever its parameter `q` receives. Suppose this function is invoked twice with `O1` and `O2` as the corresponding arguments:

```
w1 = id(new Object()); // O1
w2 = id(new Object()); // O2
```

Context-insensitively, we have $\overline{\texttt{PTS}}(\texttt{q}) = \{\texttt{O1},\texttt{O2}\}$, and thus obtain $\overline{\texttt{PTS}}(\texttt{w1}) = \overline{\texttt{PTS}}(\texttt{w2}) = \{\texttt{O1},\texttt{O2}\}$ soundly but imprecisely.

To eliminate the spurious pointed-to targets (e.g., `O2` in $\overline{\texttt{PTS}}(\texttt{w1})$ and `O1` in $\overline{\texttt{PTS}}(\texttt{w2})$), we thus need *context sensitivity*. Suppose the two invocations to the *identity* function are distinguished by two different calling contexts, c' and c'', respectively, and let $\texttt{PTS}(v, c) = \{(o_1, c_1), \cdots, (o_n, c_n)\}$ be one context-sensitive points-to set of v, where each pointed-to object o_i is identified by its heap context c_i. Then we have $\texttt{PTS}(\texttt{w1}, []) = \texttt{PTS}(\texttt{q}, c') = \{(\texttt{O1}, [])\}$ and $\texttt{PTS}(\texttt{w2}, []) = \texttt{PTS}(\texttt{q}, c'') = \{(\texttt{O2}, [])\}$, where $[]$ represents an empty context. By dropping the contexts, we obtain the context-insensitive points-to information as $\overline{\texttt{PTS}}(\texttt{q}) = \{\texttt{O1},\texttt{O2}\}$, $\overline{\texttt{PTS}}(\texttt{w1}) = \{\texttt{O1}\}$ and $\overline{\texttt{PTS}}(\texttt{w2}) = \{\texttt{O2}\}$, which is now precise.

As a kind of value flow analysis, *pointer analysis* could be graphically represented by using a so-called *pointer assignment graph* (PAG) [36]. With PAG, *pointer analysis* is reduced to a graph reachability problem. In the previous case, Figure 1.1 illustrates how heap objects flow in the program. In the context-insensitive analysis (Figure 1.1(a)), `O1` and `O2` from different callsites are merged at `q` and later propagated to both `w1` and `w2`. In the context-sensitive analysis (Figure 1.1(b)), `q` is distinguished by its two contexts c' and c''. As a result, the original cross value flow paths in Figure 1.1(a) are now split into two separate value flow paths

(a) Context-insensitive PAG. (b) Context-sensitive PAG.

Figure 1.1: A graphical illustration of value flow across `id()`.

in Figure 1.1(b). Context sensitivity enlarges the scale of PAG but substantially improves the precision of pointer analysis.

In the literature, there are several kinds of *context sensitivity*, e.g., *call-site-sensitivity* [70], *object-sensitivity* [53, 54], *type-sensitivity* [73], and *hibrid sensitivity* [31]. We will formally define them in Section 2.3.3. Here, we just give a brief description. Different types of context-sensitivity are distinguished by different context elements used. For example, k-call-site-sensitivity distinguishes the contexts of a method by its k-most-recent call sites and k-object-sensitivity distinguishes the contexts of a method by its receiver object's k-most-recent allocation sites. In this thesis, we focus on *call-site-sensitivity* [70] and *object-sensitivity* [53,54] as they are widely used in practice.

1.1.3 CFL-Reachability

The standard forms of many static analyses can be formulated as context-free language reachability (CFL-reachability) analyses [62]. A context-free language is defined by a 4-tuple $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{R}, S)$, where $\mathcal{V}$ is the nonterminal set, Σ is the terminal set, $\mathcal{R}$ is the production set, and S is the start nonterminal. The general form of a production rule is $\alpha \to \beta$, where $\alpha \in \mathcal{V}$ is a nonterminal and $\beta \in (\mathcal{V} \cup \Sigma)^*$ is a string of nonterminals and/or terminals. For example, the following production

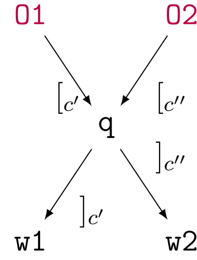


Figure 1.2: The labeled PAG for the `id()` example in Section 1.1.2.

rules are all valid:

$$A \rightarrow B\ C\ D \quad |\ A \rightarrow C \quad |\ A \rightarrow a$$

Dyck-CFL is a special kind of context-free language for handling balanced parentheses. Consider an alphabet Σ over the set of opening parentheses $\hat{A} = \{\hat{c_1}, \hat{c_2}, \cdots, \hat{c_k}\}$ and the set of their matching closing parentheses $\check{A} = \{\check{c_1}, \check{c_2}, \cdots, \check{c_k}\}$. The Dyck language of size k (i.e., k kinds of parentheses) is defined by the following context-free grammar:

$$S \longrightarrow S\ S \mid \hat{c_1}\ S\ \check{c_1} \mid \cdots \mid \hat{c_k}\ S\ \check{c_k} \mid \epsilon \tag{1.1}$$

where S is the start symbol and ϵ is the empty string.

Let G be a directed graph with edge labels taken from alphabet Σ, and let L be a context-free language over Σ. Each path p in G is labeled with a string $s(p)$ in Σ^*, obtained by concatenating edge labels in order. We say p is an L-path if $s(p) \in L$. Given nodes u and v, if G contains an L-path from u to v, we say v is L-reachable from u, denoted by $L(u, v)$. For a node n in G, we write $L(u, v)^n$ if n appears on $L(u, v)$. For a path p in G such that its label is $L(p) = \ell_1, \cdots, \ell_r$ in L, the inverse of p, i.e., $\overline{p}$ has the label $L(\overline{p}) = \overline{\ell_r}, \cdots, \overline{\ell_1}$.

Figure 1.2 gives a labeled PAG for the `id()` example described in Section 1.1.2. Let $S \rightarrow S\ S \mid [_{c'}\ S\ ]_{c'} \mid [_{c''}\ S\ ]_{c''} \mid \epsilon$ be a Dyck language defined on this PAG. Then we have $L(\texttt{O1}, \texttt{w1})$:

$$\texttt{O1} \xrightarrow{[_{c'}} \texttt{q} \xrightarrow{]_{c'}} \texttt{w1}$$

and $L(\texttt{O2}, \texttt{w2})$:

$$\texttt{O2} \xrightarrow{[_{c''}} \texttt{q} \xrightarrow{]_{c''}} \texttt{w2}$$

are valid flow paths, but $L(\texttt{O1}, \texttt{w2})$:

$$\texttt{O1} \xrightarrow{[_{c'}} \texttt{q} \xrightarrow{]_{c''}} \texttt{w2}$$

and $L(\texttt{O2}, \texttt{w1})$:

$$\texttt{O2} \xrightarrow{[_{c''}} \texttt{q} \xrightarrow{]_{c'}} \texttt{w1}$$

are invalid paths. Here, the Dyck language plays the same role as context-sensitivity and enables `O1` and `O2` in the labeled PAG (Figure 1.2) to flow precisely. In addition, we sometimes also denote $L(\texttt{O1}, \texttt{w1})$ as $L(\texttt{O1}, \texttt{w1})^{\texttt{q}}$ when variable `q` appears on $L(\texttt{O1}, \texttt{w1})$. Let path $p = L(\texttt{O1}, \texttt{w1})$. Then we have $L(p) = [_{c'},]_{c'}$ and $L(\overline{p}) = \overline{]_{c'}}, \overline{[_{c'}}$.

1.2 Contribution Overview

The goal of this thesis is to explore the design space of *fine-grained context-sensitivity* for accelerating existing *pointer analysis* techniques while preserving all or most of their precision.

1.2.1 Accelerating kOBJ by Exploiting Object Containment and Reachability

Recently, there are several selective pointer analysis techniques on improving efficiency of kOBJ [22, 29, 40, 47, 49, 74].

ZIPPER [40], as a non-precision-preserving representative of *method-level* techniques [22, 29, 40, 74], selects heuristically a set of context-sensitive methods to include some but not all the precision-critical variables/objects and also some precision-uncritical variables/objects in the program. As a result, ZIPPER sometimes improves the efficiency of kOBJ significantly but at the expense of introducing a substantial loss of precision for some programs.

In contrast, EAGLE [47, 49], as the precision-preserving representative of *partial* techniques, selects the set of context-sensitive variables/objects as a superset of the set of precision-critical variables/objects in a program by conservatively reasoning about CFL (Context-Free-Language) reachability in the program, thereby limiting its potential speedups achieved.

As the first contribution, this thesis introduces TURNER [24], which represents a sweet spot between EAGLE and ZIPPER: TURNER enables kOBJ to run significantly faster than EAGLE while achieving substantially better precision than ZIPPER. TURNER achieves this by two novel aspects in its design. First, we exploit a key observation that some precision-uncritical objects can be approximated initially based on the object-containment relationship. This approximation turns out to be practically accurate, as it introduces a small degree yet the only source of imprecision into the final points-to information computed. Second, leveraging this initial approximation, we present a simple DFA (Deterministic Finite Automaton) to reason about object reachability across a method (from its entry to its exit) intra-procedurally to finalize all its precision-critical variables/objects selected.

1.2.2 Context Debloating for Object-Sensitive Pointer Analysis

Currently, kOBJ does not scale well for reasonably large programs when $k \geqslant 3$ and is often time-consuming when it is scalable [27, 73, 82, 86]. When designing TURNER, we observe that many objects allocated in a method are used independently of its calling contexts. Distinguishing these objects context-sensitively, as often done in the past, will only increase the number of calling contexts analyzed without any precision improvement. Although the object containment relation exploited in TURNER is already practical, it may still miss thousands of context-independent objects in some programs.

To further tap the potential for performance improvement, this thesis introduces *context debloating* as its second contribution. We propose a new approach, CONCH [25], for finding more context-independent objects (than TURNER) based on three critical observations regarding context-dependability for the objects used practically in real-world object-oriented programs. By allowing only context-dependent objects to be handled context-sensitively, CONCH can significantly limit the explosive growth of the number of contexts and achieve substantially improved efficiency and scalability.

1.2.3 Precision-Preserving Acceleration for kCFA

This thesis introduces the third fine-grained approach, P3CTX, as an application of a complete CFL-reachability formulation of callsite-sensitive pointer analysis with built-in on-the-fly callgraph construction.

When designing SELECTX [48], we observe that the traditional CFL-reachability formulation for k-callsite-sensitive pointer analysis (k-CFA) models field accesses

and calling contexts only but relies on a separate algorithm for call graph construction [76, 78], which may cause k-CFA to lose precision even if it uses the most precise call graph for a program, built in advance or on the fly. In addition, the SELECTX-guided k-CFA, whose pre-analysis reasons about CFL-reachability (based on the traditional incomplete formulation) to select precision-critical variables and objects, could not preserve k-CFA's precision as it is disconnected to the value-flow paths traversed by the call graph construction algorithm.

We overcome these two limitations by presenting the first complete CFL-reachability formulation of k-CFA for Java with built-in on-the-fly call graph construction. Based on this new CFL-reachability formulation, we present P3CTX, the first precision-preserving acceleration technique for k-CFA with fine-grained context-sensitivity.

1.3 Publications and Organization

This thesis contains text and material from several publications:

- Our first fine-grained pointer analysis technique presented in Chapter 3 is based on "Accelerating Object-Sensitive Pointer Analysis by Exploiting Object Containment and Reachability", which is appeared in ECOOP 2021.
- The second fine-grained pointer analysis technique, *context debloating*, proposed in Chapter 4 is derived from "Context Debloating for Object-Sensitive Pointer Analysis" published in ASE 2021.
- Our third fine-grained techniques presented in Chapter 5 is established on "A CFL-Reachability Formulation of Callsite-Sensitive Pointer Analysis with Built-in On-the-Fly Call Graph Construction", which is now in submission.

Lastly, we organize other chapters as follows. In Chapter 2, we will review some background knowledge for this thesis. Chapter 6 will discuss some more related work, and finally, we summarize this thesis and suggest some future research directions in Chapter 7.

Chapter 2

Background

This chapter introduces some background knowledge for this thesis. Section 2.1 gives a simplified Java language for formalizing pointer analyses. Section 2.2 briefly reviews the concept and measurements of pointer analyses. Section 2.3 reviews the inclusion-based formulations for Java pointer analyses, including the standard Andersen's analysis [3, 36] and its context-sensitive incarnations. We also give a fine-grained formulation [1] as the basis of this thesis. Section 2.4 reviews two CFL-reachability formulations for Java pointer analyses.

2.1 A Simplified Java Language

Figure 2.1 gives a simplified Java language, in which a program consists of a set of classes, where each class consists of instance fields and methods. Methods are stylised to have a total of 5 kinds of basic statements, i.e., New, Assign, Store, Load, and Call, in their body, and always end with a single return statement. Constructors are regarded as regular instance methods. For exam-

[1]This formulation is systematically introduced in our work "Qilin: A New Framework for Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis" for the first time.

ple, “x = **new** T(...)” in the standard Java language is modeled as “x = **new** T; x.`<init>`(...)” in Figure 2.1, where `<init>`(...) is the corresponding constructor invoked. Allocation sites and call sites are identified by their labels (e.g., line numbers). The control flow statements are not considered because our formalizations discussed in this thesis are flow-insensitive.

$$
\begin{array}{rcll}
\textsf{Class} & ::= & \text{class } C \text{ extends } C \; \{ \; \overline{C\ f} \; ; \; \overline{M} \; \} & \\
M \in \textsf{Method} & ::= & C\ m\ (\ \overline{C\ v}\ )\ \{\ \overline{C\ v};\ \overline{S};\ \text{return } v;\ \} & \\
S \in \textsf{Statement} & ::= & v = \text{new } C\ ^{o} & [\textsc{New}] \\
& | & v = v & [\textsc{Assign}] \\
& | & v.f = v \mid C.f = v & [\textsc{Store}] \\
& | & v = v.f \mid v = C.f & [\textsc{Load}] \\
& | & v = v.m(\overline{v})\ ^{c} & [\textsc{Call}]
\end{array}
$$

$$
\begin{array}{rl}
v \in & \mathbb{V} \text{ is the domain of variable names} \\
C \in & \mathbb{T} \text{ is the domain of class names} \\
m \in & \mathbb{M} \text{ is the domain of method names} \\
f \in & \mathbb{F} \text{ is the domain of field names} \\
o \in & \mathbb{H} \text{ is the domain of heap objects} \\
c \in & \mathbb{L} \text{ is the domain of labels for call sites}
\end{array}
$$

Figure 2.1: A simplified Java language.

In the formalisation of this thesis, the parameters of a method m are often uniquely identified as follows. The “`this`” variable, i-th parameter, and the return variable of m are denoted by $\texttt{this}^m$, p_i^m, and $\texttt{ret}^m$, respectively.

As is standard, the method/field (variable) names in distinct classes (methods) are assumed to be distinct. In addition, all variables in a method are assumed to be in SSA form [14]. Static methods/fields may be handled differently in different chapters and will be discussed wherever appropriate.

2.2 Pointer Analysis: Concepts and Measurements

Pointer analysis is a fundamental static analysis that determines what objects a pointer may point to, where objects refer to abstractions for a set of objects typically allocated at an allocation site at runtime, and pointers refer to variables and object field references. In essence, a specific pointer analysis aims to establish a many-to-many mapping between the set of pointers and the set of objects in a program.

There are three measurements on judging the effectiveness and quality of a pointer analysis: soundness, precision, and scalability.

Soundness The soundness of a pointer analysis is measured by assessing how many false negatives that should exist in the points-to results compared with ground truths. Generally, the fewer the false negatives, the higher the soundness. A pointer analysis technique is sound if it could infer all the ground truths in the program. In practice, a sound pointer analysis algorithm is almost impossible due to ubiquitous dynamic language features in Java such as reflections and the native codes [45]. In this case, a new term named *soundiness* is introduced to describe a sound analysis. A *soundy* analysis aims to be as sound as possible without excessively compromising precision and/or scalability.

Precision The precision of a pointer analysis is measured by assessing how many false positives exist in the points-to results compared with ground truths. Generally, the fewer the false positives, the higher the precision. Specifically, if a pointer analysis is sound, then the higher precision is often reflected by fewer established points-to relations. Based on this fact, the following five metrics are commonly used in measuring the precision of a pointer analysis in practice [24, 40, 49, 73, 82]:

- "#call-edges": the number of call graph edges discovered
- "#poly-calls": the number of polymorphic calls discovered
- "#fail-casts": the number of type casts that may fail
- "#reachables": the number of reachable methods.
- "#avg-pts": the average number of objects pointed by a variable by considering only the local variables in the Java methods (being analyzed).

For all metrics, the smaller they are, the higher precision a pointer analysis is.

Scalability We measure the scalability of a pointer analysis by the efficiency (in elapsed time and memory consumption) of a pointer analysis.

In Java, field sensitivity and context sensitivity are two main factors that affect the precision and scalability of pointer analyses and are thus mainly considered in this thesis. Shortly, we will review how to support field-sensitivity and context-sensitivity in two different formulations of pointer analysis, i.e., inclusion-based formulation (Section 2.3) and CFL-reachability formulation (Section 2.4).

2.3 Inclusion-based Formulation

In this section, we first introduce a few commonly used notations (Section 2.3.1) and then review the inclusion-based formulations for Java pointer analysis (Section 2.3.2 and Section 2.3.3). Lastly, we present a new formulation for supporting fine-grained context-sensitive pointer analysis (Section 2.3.4).

2.3.1 Notations

Let $\mathbb{C}$ be the universe of contexts. Given a context $ctx = [c_1, \ldots, c_n] \in \mathbb{C}$, we write $\lceil ctx \rceil_k$ for $[c_1, \ldots, c_k]$ and write $c +\!\!+ \; ctx$ for $[c, c_1, \ldots, c_n]$, implying a new context constructed by appending the context element c in front of ctx. The following auxiliary functions will be used shortly:

- $\texttt{PTS} : (\mathbb{V} \cup \mathbb{H} \times \mathbb{F}) \times \mathbb{C} \rightarrow \wp(\mathbb{H} \times \mathbb{C})$
- $\overline{\texttt{PTS}} : (\mathbb{V} \cup \mathbb{H} \times \mathbb{F}) \rightarrow \wp(\mathbb{H})$
- $\texttt{MethodCtx} : \mathbb{M} \rightarrow \wp(\mathbb{C})$
- $\texttt{Dispatch} : \mathbb{M} \times \mathbb{H} \rightarrow \mathbb{M}$
- $\texttt{Cons} : \mathbb{H} \times \mathbb{C} \times \mathbb{L} \times \mathbb{C} \rightarrow \mathbb{C}$
- $\mathsf{HCons} : \mathbb{C} \mapsto \mathbb{C}$

where PTS (as already defined in Section 1.1.2) records the points-to information found context-sensitively for a variable or an object's field, $\overline{\texttt{PTS}}$ is the context-insensitive version of PTS, MethodCtx maintains the contexts used for analyzing a method, Dispatch resolves a virtual call to a target method based on the dynamic type of the receiver object, and Cons and HCons are two significant parameters in context-sensitive pointer analyses used for describing how to construct a new context for a method and heap, respectively.

2.3.2 Andersen-Style Inclusion-based Formulation

Andersen's analysis [3, 36] is also known as a context-insensitive, field-sensitive pointer analysis. It is less precise but very efficient in practice. Thus, it is widely

used by other program analysis techniques and often used as a pre-analysis in developing other precise and efficient pointer analyses.

Figure 2.2 presents the inference rules of Anderson's analysis. Each rule handles value flow for each basic statement. In [A-New], $O \in \mathbb{H}$ is an abstract heap object. [A-Assign] handles direct value flow. [A-Load] and [A-Store] handle indirect value flow. Loads and stores to the elements of an array are modeled by collapsing all the elements into a special field `arr` of the array. Finally, [A-Call] handles inter-procedural value flow. Here, m' is a target method resolved by `Dispatch`for a receiver object O at callsite `c` (based on the dynamic type of O). Thus, this rule is also responsible for performing on-the-fly call graph construction during the pointer analysis.

$$\frac{\texttt{x} = \textbf{new}\ \texttt{T}\ //\ O}{O \in \overline{\texttt{PTS}}(\texttt{x})}\ \text{[A-New]} \qquad \frac{\texttt{x} = \texttt{y}}{\overline{\texttt{PTS}}(\texttt{y}) \subseteq \overline{\texttt{PTS}}(\texttt{x})}\ \text{[A-Assign]}$$

$$\frac{\texttt{x} = \texttt{y.f} \quad O \in \overline{\texttt{PTS}}(\texttt{y})}{\overline{\texttt{PTS}}(O.\texttt{f}) \subseteq \overline{\texttt{PTS}}(\texttt{x})}\ \text{[A-Load]} \qquad \frac{\texttt{x.f} = \texttt{y} \quad O \in \overline{\texttt{PTS}}(\texttt{x})}{\overline{\texttt{PTS}}(\texttt{y}) \subseteq \overline{\texttt{PTS}}(O.\texttt{f})}\ \text{[A-Store]}$$

$$\frac{\texttt{x} = \texttt{r.m}(a_1, \ldots, a_n)\ //\ \texttt{c} \quad O \in \overline{\texttt{PTS}}(\texttt{r}) \quad m' = \texttt{Dispatch}(\texttt{m}, O)}{O \in \overline{\texttt{PTS}}(\texttt{this}^{m'}) \quad \forall i \in [1, n] : \overline{\texttt{PTS}}(a_i) \subseteq \overline{\texttt{PTS}}(p_i^{m'}) \quad \overline{\texttt{PTS}}(\texttt{ret}^{m'}) \subseteq \overline{\texttt{PTS}}(\texttt{x})}\ \text{[A-Call]}$$

Figure 2.2: Anderson-style Inclusion-based Formulation.

Note that the receiver variable `r` and the other arguments $a_1, \ldots, a_n$ are handled differently. A receiver object flows only to the method dispatched on itself while the objects pointed to by the other arguments flow to all the methods dispatched at this callsite. Finally, static calls can be regarded as special virtual calls without the receiver variables. Therefore, its rule is subsumed by [A-Call].

2.3.3 Inclusion-based Formulation with Context Sensitivity

Traditionally, context-sensitive approaches analyze a method separately under different calling contexts that abstract its different run-time invocations. Under such *method-level context-sensitivity*, whenever a method is analyzed for a given context, all its variables and objects are qualified by, i.e., analyzed under that context.

Figure 2.3 (extended from Figure 2.2) gives the inclusion-based formulation for method-level context-sensitive pointer analysis [31, 74, 86]. In this formulation, context-sensitivity is achieved by parameterizing contexts (such as ctx and htx) as modifiers to the basic abstractions (i.e., variables and objects). In [I-NEW], HCons is used to create the heap contexts. Rules [I-ASSIGN], [I-LOAD], and [I-STORE] for handling assignments, loads, and stores are standard. In [I-CALL], m' is a target method dynamically resolved in the same way as [A-CALL]. ctx' constructed by `Cons` represents a callee context of m' and $ctx' \in$ `MethodCtx`(m') reveals how the contexts of a method are maintained. Initially, the contexts of all the entry methods are set to be empty, e.g., `MethodCtx`(“`main`”) $= \{[\,]\}$.

In the literature, four flavors of context-sensitivity are proposed: callsite-sensitivity [70] (which distinguishes the contexts of a method by its callsites) and object-sensitivity [53, 54] (which distinguishes the contexts of a method by its receiver's allocation sites) are the two most popular ones. The two other variations are type-sensitivity [73] and hybrid sensitivity [31].

Below, we describe a few significant instantiations of `Cons` for the four common types of context-sensitivity, where a calling context of a method is abstracted by its last k context elements (under k-limiting).

- Callsite. A callsite-sensitive pointer analysis [70], known also as *control-flow analysis* (CFA) [71], uses a callsite c as a context element. Therefore, the

$$\frac{\texttt{x} = \textbf{new}\ \texttt{T}\ //\ O \quad ctx \in \texttt{MethodCtx(M)} \quad htx = \textsf{HCons}(ctx)}{\langle O, htx\rangle \in \texttt{PTS}(\texttt{x}, ctx)} \quad \text{[I-NEW]}$$

$$\frac{\texttt{x} = \texttt{y} \quad ctx \in \texttt{MethodCtx(M)}}{\texttt{PTS}(\texttt{y}, ctx) \subseteq \texttt{PTS}(\texttt{x}, ctx)} \quad \text{[I-ASSIGN]}$$

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

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

$$\frac{\begin{array}{c}\texttt{x} = \texttt{r.m}(a_1, \ldots, a_n)\ //\ c \quad ctx \in \texttt{MethodCtx(M)} \quad \langle O, htx\rangle \in \texttt{PTS}(\texttt{r}, ctx) \\ m' = \texttt{Dispatch}(\texttt{m}, O) \quad ctx' = \texttt{Cons}(O, htx, c, ctx)\end{array}}{\begin{array}{c}ctx' \in \texttt{MethodCtx}(m') \quad \texttt{PTS}(\texttt{ret}^{m'}, ctx') \subseteq \texttt{PTS}(\texttt{x}, ctx) \\ \langle O, htx\rangle \in \texttt{PTS}(\texttt{this}^{m'}, ctx') \quad \forall i \in [1, n] : \texttt{PTS}(a_i, ctx) \subseteq \texttt{PTS}(p_i^{m'}, ctx')\end{array}} \quad \text{[I-VCALL]}$$

Figure 2.3: Inclusion-based formulation with context-sensitivity (M is the containing method of a statement being analyzed).

context constructor is:

$$\texttt{Cons}_{\text{CFA}}(o, htx, c, ctx) = \lceil c +\!\!+ ctx \rceil_k \tag{2.1}$$

- Object. An object-sensitive pointer analysis [53, 54] uses a receiver object o as a context element. Thus, the context constructor simply becomes:

$$\texttt{Cons}_{\text{OBJ}}(o, htx, c, ctx) = \lceil o +\!\!+ htx \rceil_k \tag{2.2}$$

The context constructed here for analyzing a method m is represented by a sequence of k context elements (under k-limiting), $[o_1, ..., o_k]$, where o_1 is the receiver object of m and o_i is the receiver object of a method in which o_{i-1} is allocated [73]. So o_i is an *allocator* of o_{i-1}.

For object-oriented languages, object-sensitive pointer analysis is regarded as providing highly useful precision [27, 49, 73, 82, 86] and thus widely adopted in

several pointer analysis frameworks for Java, such as Soot [89], Doop [72], and Wala [26].

- Type. A type-sensitive pointer analysis [73], which is a more scalable but less precise alternative of an object-sensitive pointer analysis, resorts to the class type containing the method where a receiver object o is allocated, denoted as $\texttt{TypeContg}(o)$. Thus, we have:

$$\texttt{Cons}_{\text{TYPE}}(o, htx, c, ctx) = \lceil \texttt{TypeContg}(o) +\!\!+\ htx \rceil_k \tag{2.3}$$

- Hybrid. A hybrid pointer analysis [31] distinguishes static and dynamic call sites:

$$\texttt{Cons}_{\text{HYB}}(o, htx, c, ctx) = \begin{cases} \lceil o +\!\!+\ htx \rceil_k & c \notin \texttt{SC} \\ \lceil \texttt{car}(ctx) +\!\!+\ c +\!\!+\ \texttt{cdr}(ctx) \rceil_k & c \in \texttt{SC} \end{cases} \tag{2.4}$$

  where SC is the set of all static call sites in the program. Here, `car` and `cdr` are standard, with `car` pulling the first element of a list and `cdr` returning the list without the `car`.

The heap constructor could be simply defined as below:

$$\mathsf{HCons(ctx)} = \lceil ctx \rceil_{hk} \tag{2.5}$$

where hk represents the (heap) context length for a heap object. In practice, $hk = k - 1$ is usually used [27, 41, 74, 82].

Let Csm be the subset of (precision-critical) methods in a program that will be analyzed context-sensitively. Figure 2.3 could also be adapted to support method-level selective context-sensitive pointer analysis techniques [29, 40, 74] by redefining

its context constructor (i.e. Cons) as follows:

$$\texttt{Cons}_{\textsc{sel}}(o, htx, c, ctx) = \begin{cases} [\,] & m' \notin \textsc{Csm} \\ \texttt{Cons}(o, htx, c, ctx) & m' \in \textsc{Csm} \end{cases} \tag{2.6}$$

where m' is a target method resolved at callsite c.

2.3.4 Fine-Grained Context-sensitive Pointer Analysis

In this section, we go one step further by generalizing the inclusion-based formulation for traditional context-sensitive pointer analyses in Figure 2.3 to a new formulation for supporting the fine-grained context-sensitive pointer analysis.

$$\frac{\texttt{x} = \textbf{new}\ \texttt{T}\ //\ O \quad ctx \in \texttt{MethodCtx(M)}}{\langle O, \texttt{Sel}(O, ctx)\rangle \in \texttt{PTS}(\texttt{x}, \texttt{Sel}(\texttt{x}, ctx))} \quad \text{[F-New]}$$

$$\frac{\texttt{x} = \texttt{y} \quad ctx \in \texttt{MethodCtx(M)}}{\texttt{PTS}(\texttt{y}, \texttt{Sel}(\texttt{y}, ctx)) \subseteq \texttt{PTS}(\texttt{x}, \texttt{Sel}(\texttt{x}, ctx))} \quad \text{[F-Assign]}$$

$$\frac{\texttt{x} = \texttt{y.f} \quad ctx \in \texttt{MethodCtx(M)} \quad (O, htx) \in \texttt{PTS}(\texttt{y}, \texttt{Sel}(\texttt{y}, ctx))}{\texttt{PTS}(O.\texttt{f}, htx) \subseteq \texttt{PTS}(\texttt{x}, \texttt{Sel}(\texttt{x}, ctx))} \quad \text{[F-Load]}$$

$$\frac{\texttt{x.f} = \texttt{y} \quad ctx \in \texttt{MethodCtx(M)} \quad (O, htx) \in \texttt{PTS}(\texttt{x}, \texttt{Sel}(\texttt{x}, ctx))}{\texttt{PTS}(\texttt{y}, \texttt{Sel}(\texttt{y}, ctx)) \subseteq \texttt{PTS}(O.\texttt{f}, htx)} \quad \text{[F-Store]}$$

$$\frac{\begin{array}{c}\texttt{x} = \texttt{r.m}(a_1, \ldots, a_n)\ //\ c \quad ctx \in \texttt{MethodCtx(M)} \quad m' = \texttt{Dispatch}(\texttt{m}, O) \\ (O, htx) \in \texttt{PTS}(\texttt{r}, \texttt{Sel}(\texttt{r}, ctx)) \quad ctx' = \texttt{Cons}(O, htx, c, ctx)\end{array}}{\begin{array}{c} ctx' \in \texttt{MethodCtx}(m') \quad (O, htx) \in \texttt{PTS}(\texttt{this}^{m'}, \texttt{Sel}(\texttt{this}^{m'}, ctx')) \\ \forall i \in [1, n] : \texttt{PTS}(a_i, \texttt{Sel}(a_i, ctx)) \subseteq \texttt{PTS}(p_i^{m'}, \texttt{Sel}(p_i^{m'}, ctx')) \\ \texttt{PTS}(\texttt{ret}^{m'}, \texttt{Sel}(\texttt{ret}^{m'}, ctx')) \subseteq \texttt{PTS}(\texttt{x}, \texttt{Sel}(\texttt{x}, ctx))\end{array}} \quad \text{[F-Call]}$$

Figure 2.4: Fine-grained Context-sensitive pointer analysis (M is the containing method of a statement being analyzed).

Figure 2.4 presents the new formulation, in which the contexts of each variable and object are no longer required to be the same as the contexts of their containing

methods but are selected from their containing methods' contexts by using a particular selector, `Sel`. By defining different instantiations of `Sel`, we could define different fine-grained context-sensitive pointer analysis techniques. For example, we could apply contexts to only a subset of (precision-critical) variables by defining the selector as follows:

$$\texttt{Sel}(v, ctx) = \begin{cases} [\,] & v \notin \textsc{Csv} \\ ctx & v \in \textsc{Csv} \end{cases} \tag{2.7}$$

where Csv is the set of variables and objects selected to be analyzed context-sensitively. Thus, Figure 2.4 lays a foundation of the three fine-grained pointer analysis techniques introduced later in this thesis.

2.4 CFL-Reachability Formulation

In the CFL-reachability formulation of pointer analysis, a Java program is represented by a directed graph, called *Pointer Assignment Graph* (PAG), where nodes represent the variables and objects in the program and its edges represent value flow through the assignments in the program.

Below, we will review two different kinds of CFL-reachability formulations for callsite-sensitive and object-sensitive pointer analyses, respectively.

2.4.1 Callsite-based CFL-Reachability Formulation

Traditionally, k-callsite-sensitive pointer analysis (denoted k-CFA) [76] could also be solved by reasoning about CFL-reachability on its PAG (Pointer Assignment Graph) representation [36]. Figure 2.5 gives five 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* edges (for modeling regular assignments without a below-edge label) and *inter-procedural* edges (for modeling parameter passing with a below-edge label representing a callsite).

In such a PAG, passing arguments to parameters at callsites is modeled by inter-procedural assign edges. For example, in [P-VCALL], $\hat{c}$ ($\check{c}$) signifies an inter-procedural value-flow entering into (exiting from) m' at callsite c, where m' represents a virtual method discovered by a separate call graph construction algorithm (either in advance [5, 16, 81] or on the fly [76, 78]). Thus, $\hat{c}$ ($\check{c}$) is also known as an *entry* (*exit*) context. Static calls are subsumed by [P-VCALL] by just ignoring receiver-variable-related edges (e.g., $\texttt{r} \xrightarrow[\hat{c}]{\textsf{assign}} \texttt{this}^{\texttt{m}'}$).

For a PAG edge $x \xrightarrow[c]{\ell} y$, its *inverse edge*, which is omitted in Figure 2.5 but required by the formulation, 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.

$$\frac{\texttt{x} = \textbf{new}\ \texttt{T}\ //\ O}{O \xrightarrow{\textsf{new}} \texttt{x}}\ \text{[P-NEW]} \qquad \frac{\texttt{x} = \texttt{y}}{\texttt{y} \xrightarrow{\textsf{assign}} \texttt{x}}\ \text{[P-ASSIGN]}$$

$$\frac{\texttt{x} = \texttt{y.f}}{\texttt{y} \xrightarrow{\textsf{load[f]}} \texttt{x}}\ \text{[P-LOAD]} \qquad \frac{\texttt{x.f} = \texttt{y}}{\texttt{y} \xrightarrow{\textsf{store[f]}} \texttt{x}}\ \text{[P-STORE]}$$

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

Figure 2.5: Inference Rules for building the PAG required by the traditional callsite-based CFL-reachability formulation [76].

In the traditional callsite-based CFL-reachability formulation [76], k-CFA is solved by reasoning about the intersection of two CFLs, $L_{FC} = L_F \cap L_C$, with L_F defined over the PAG's above-edge labels and L_C over the PAG's below-edge labels. Let L be a CFL over Σ formed by the above-edge (below-edge) labels. Each

path p in the PAG has a string $L(p)$ in Σ^* formed by concatenating in order the above-edge (below-edge) labels in p. A node v in the PAG is said to be *L-reachable* from a node u in the PAG if there exists a path p from u to v, known as *L-path*, such that $L(p) \in L$.

L_F requires all field accesses to be field-sensitive (with stores and loads being matched as balanced parentheses):

$$
\begin{aligned}
\mathsf{flowsto} &\longrightarrow \mathsf{new}\ \mathsf{flows}^* \\
\mathsf{flows} &\longrightarrow \mathsf{assign} \mid \mathsf{store[f]}\ \mathit{alias}\ \mathsf{load[f]} \\
\mathit{alias} &\longrightarrow \overline{\mathsf{flowsto}}\ \mathsf{flowsto} \\
\overline{\mathsf{flowsto}} &\longrightarrow \overline{\mathsf{flows}}^*\ \overline{\mathsf{new}} \\
\overline{\mathsf{flows}} &\longrightarrow \overline{\mathsf{assign}} \mid \overline{\mathsf{load[f]}}\ \mathit{alias}\ \overline{\mathsf{store[f]}}
\end{aligned}
\tag{2.8}
$$

If O $\mathsf{flowsto}$ v, then v is L_F-reachable from O. In addition, O $\mathsf{flowsto}$ v iff v $\overline{\mathsf{flowsto}}$ O, meaning that $\overline{\mathsf{flowsto}}$ actually represents the standard points-to relation. Finally, u *alias* v iff u $\overline{\mathsf{flowsto}}$ O $\mathsf{flowsto}$ v for some object O.

Figure 2.6 depicts a code snippet (consisting of local variables only), together with its PAG. Here, $L_F(\mathtt{O}, \mathtt{v})$, i.e., O flowsto v, implying that v points to O, which holds due to the following flowsto path:

$$
\mathtt{O} \xrightarrow{\mathsf{new}} \mathtt{u} \xrightarrow{\mathsf{store[f]}} \mathtt{p} \xrightarrow{\overline{\mathsf{new}}} \mathtt{A} \xrightarrow{\mathsf{new}} \mathtt{p} \xrightarrow{\mathsf{assign}} \mathtt{q} \xrightarrow{\mathsf{load[f]}} \mathtt{v}
\tag{2.9}
$$

By inverting all the edges in this flowsto path, a $\overline{\mathsf{flowsto}}$ path showing v $\overline{\mathsf{flowsto}}$ O is obtained.

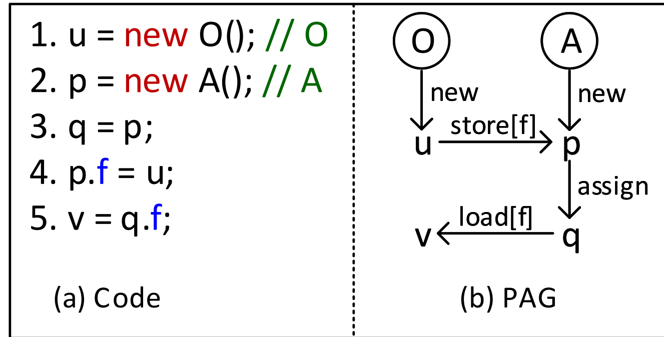


Figure 2.6: The PAG for a code snippet.

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

$$\begin{array}{rcl} \mathsf{realizable} & \longrightarrow & \mathsf{exit}\ \mathsf{entry} \\ \mathsf{exit} & \longrightarrow & \mathsf{exit}\ \mathsf{balanced} \mid \mathsf{exit}\ \check{c} \mid \epsilon \\ \mathsf{entry} & \longrightarrow & \mathsf{entry}\ \mathsf{balanced} \mid \mathsf{entry}\ \hat{c} \mid \epsilon \\ \mathsf{balanced} & \longrightarrow & \mathsf{balanced}\ \mathsf{balanced} \mid \hat{c}\ \mathsf{balanced}\ \check{c} \mid \epsilon \end{array} \tag{2.10}$$

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

Finally, a variable v points to an object O iff there exists a path p from O to v in the PAG, 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). In this thesis, such a path is referred to as an L_{FC}*-path*.

The limitation of L_{FC} is that it does not have a built-in call graph construction mechanism. In other words, when handling the inter-context value flow at a virtual call site, the language itself does not know where to propagate next. As a result, SELECTX [48], a recent L_{FC}-based approach specially designed to select precision-critical variables/objects for k-CFA, could not guarantee to preserve precision. In Chapter 5, we will propose a new CFL-reachability formulation for k-CFA with a built-in callgraph dispatching mechanism in the language. We have also developed a precision-preserving approach, P3CTX, for accelerating k-CFA.

$$\frac{\texttt{x} = \textbf{new}\ \texttt{T}\ //\ \texttt{O}}{\texttt{O} \xrightarrow{\textsf{new}} \texttt{x}}\ \text{[O-NEW]} \qquad \frac{\texttt{x} = \texttt{y}}{\texttt{y} \xrightarrow{\textsf{assign}} \texttt{x}}\ \text{[O-ASSIGN]}$$

$$\frac{\texttt{x.f} = \texttt{y} \quad O \in \overline{\texttt{PTS}}(\texttt{x})}{\texttt{y} \xrightarrow{\textsf{store[f]}} \texttt{x} \quad O \xrightarrow[\hat{o}]{\textsf{hload[f]}} \texttt{f}}\ \text{[O-STORE]} \qquad \frac{\texttt{x} = \texttt{y.f} \quad O \in \overline{\texttt{PTS}}(\texttt{y})}{\texttt{y} \xrightarrow{\textsf{load[f]}} \texttt{x} \quad \texttt{f} \xrightarrow[\check{o}]{\textsf{hstore[f]}} O}\ \text{[O-LOAD]}$$

$$\frac{\texttt{x} = \texttt{r.m}(a_1, \ldots, a_n)\ //\ \texttt{c} \qquad O \in \overline{\texttt{PTS}}(\texttt{r}) \quad m' = \textsf{dispatch}(\texttt{m}, O)}{\forall\ i : a_i \xrightarrow{\textsf{store}[p_i^{m'}]} \texttt{r} \quad \texttt{r} \xrightarrow{\textsf{load}[\texttt{ret}^{m'}]} x \quad \forall\ i : O \xrightarrow[\hat{o}]{\textsf{hload}[p_i^{m'}]} p_i^{m'} \quad \texttt{ret}^{m'} \xrightarrow[\check{o}]{\textsf{hstore}[\texttt{ret}^{m'}]} O}\ \text{[O-CALL]}$$

Figure 2.7: Rules for building the PAG required by L^o_{FC}.

2.4.2 Object-based CFL-Reachability Formulation

Recently, Lu et. at [47, 49] propose a new CFL reachability formulation for object-sensitive pointer analysis, which is also formalized as the intersection of two context-free languages: $L^o_{FC} = L^o_F \cap L^o_C$ on top of a new PAG.

Figure 2.7 gives the PAG construction rules. [O-NEW] and [O-ASSIGN] are standard. In [O-STORE], when a store edge $\texttt{y} \xrightarrow{\textsf{store[f]}} \texttt{x}$ is added, its corresponding heap load edges, i.e., $O \xrightarrow[\hat{o}]{\textsf{hload[f]}} \texttt{f}$ for each $O \in \overline{\texttt{PTS}}(\texttt{x})$ are also added. In [O-LOAD], for each added load edge $\texttt{y} \xrightarrow{\textsf{load[f]}} \texttt{x}$, its heap store edges, i.e., $\texttt{f} \xrightarrow[\check{o}]{\textsf{hstore[f]}} O$ for each $O \in \overline{\texttt{PTS}}(\texttt{y})$ are added accordingly. Finally, in [O-CALL], the parameter passing for each argument a_i is simply modelled as a store, i.e., $\texttt{r}.p_i^{m'} = a_i$, resulting in the edges added by apply [O-STORE]. Similarly, the returned value for $\texttt{x}$ is modelled as a load, i.e.,$\texttt{x} = \texttt{r.ret}^{m'}$, resulting in the edges added by applying [O-LOAD].

Again, the *inverse edge* of a PAG edge $x \xrightarrow[o]{\ell} y$, which is omitted in Figure 2.7 but required by L^o_{FC}, is defined as $y \xrightarrow[\overline{o}]{\overline{\ell}} x$. For a below-edge label $\hat{o}$ or $\check{o}$, $\overline{\hat{o}} = \check{o}$ and $\overline{\check{o}} = \hat{o}$, implying that the concepts of entry and exit contexts for inter-procedural $\textsf{assign}$ edges are swapped if they are traversed inversely.

Equation (2.11) defines the L_F^o, which realises a context-insensitive field-based pointer analysis for kOBJ:

$$\begin{aligned} \mathsf{flowsto} &\longrightarrow \mathsf{new}\ \mathsf{flows}^* \\ \overline{\mathsf{flowsto}} &\longrightarrow \overline{\mathsf{flows}}^*\ \overline{\mathsf{new}} \\ \mathsf{flows} &\longrightarrow \mathsf{assign} \mid \mathsf{store[f]}\ \overline{\mathsf{flowsto}}\ \mathsf{hload[f]} \mid \mathsf{hstore[f]}\ \mathsf{flowsto}\ \mathsf{load[f]} \\ \overline{\mathsf{flows}} &\longrightarrow \overline{\mathsf{assign}} \mid \overline{\mathsf{load[f]}}\ \overline{\mathsf{flowsto}}\ \overline{\mathsf{hstore[f]}} \mid \overline{\mathsf{hload[f]}}\ \mathsf{flowsto}\ \overline{\mathsf{store[f]}} \end{aligned} \tag{2.11}$$

L_F^o requires $\mathsf{store[f]}$ matched by $\mathsf{hload[f]}$, $\mathsf{hstore[f]}$ matched by $\mathsf{load[f]}$, $\overline{\mathsf{hload[f]}}$ matched by $\overline{\mathsf{store[f]}}$, and $\overline{\mathsf{load[f]}}$ matched by $\overline{\mathsf{hstore[f]}}$. By creatively treating parameters as special fields of receiver objects, L_F^o enables a uniformly and object-sensitively handling for both method calls and field accesses.

L_C^o given below looks the same as L_C grammatically except that each context element is now an object instead, i.e., $o_i (i \in [1, n])$:

$$\begin{aligned} \mathsf{realisable} &\longrightarrow \mathsf{exit}\ \mathsf{entry} \\ \mathsf{exit} &\longrightarrow \mathsf{exit}\ \mathsf{balanced} \mid \mathsf{exit}\ \check{o}_i \mid \epsilon \\ \mathsf{entry} &\longrightarrow \mathsf{entry}\ \mathsf{balanced} \mid \mathsf{entry}\ \hat{o}_i \mid \epsilon \\ \mathsf{balanced} &\longrightarrow \mathsf{balanced}\ \mathsf{balanced} \mid \hat{o}_i\ \mathsf{balanced}\ \check{o}_i \mid \epsilon \end{aligned} \tag{2.12}$$

L_C^o enforces the object-sensitive context-sensitivity in kOBJ for both method calls and field accesses (by turning L_F^o from a field-based analysis into a field-sensitive analysis).

L_{FC}^o provides a theoretical basis for determining whether a variable or an object should be context-sensitive or not. In Chapter 4, CONCH will leverage three practical conditions to approximate these theoretical conditions in order to identify effectively context-independent objects, which will be later used for developing

an efficient pointer analysis algorithm by context debloating, i.e., debloating the contexts for existing object-sensitive pointer analysis algorithms.

Chapter 3

Accelerating kOBJ by Exploiting Object Containment and Reachability

In the previous chapter, we have introduced some background knowledge about pointer analysis, such as the inclusion-based and CFL-reachability formulations for supporting field- and context-sensitivity. We have also presented the inference rules of fine-grained pointer analysis. We are ready to introduce our fine-grained pointer analysis approaches.

This chapter presents TURNER, our first fine-grained pointer analysis technique that was published at the ECOOP conference [24]. The objective is to advance existing pointer analysis techniques in terms of efficiency and precision trade-offs made. TURNER will enable kOBJ to run significantly faster than the precision-preserving state-of-the-art while achieving substantially better precision than the non-precision-preserving state-of-the-art.

The outline of this chapter is organized as follows. Section 3.1 gives an overview of TURNER. Section 3.2 motivates our TURNER approach. We formalize our TURNER approach in Section 3.3 and evaluate TURNER against the state of the arts in Section 3.4. Finally, Section 3.5 concludes this chapter.

3.1 Overview

Traditional k-object-sensitive pointer analysis, denoted kOBJ, blindly applies the k-limiting context abstraction uniformly to a program, which can cause the number of contexts handled to blow up exponentially (often without improving precision much). In this chapter, we address the problem of developing a pre-analysis for a Java program to enable kOBJ to apply fine-grained context-sensitivity (i.e, a k-limited context abstraction) only to some of its variables/objects selected and context-insensitivity to all the rest in the program.

Definition 3.1 *A variable/object n in a program is* precision-critical *if kOBJ loses precision in terms of the points-to information obtained (for some value of k) when n is analyzed by kOBJ context-insensitively instead of context-sensitively.*

A pre-analysis is said to be *precision-preserving* if it can identify the precision-critical variables/objects in a program precisely or over-approximately as being context-sensitive, and *non-precision-preserving* otherwise. Unfortunately, making such selections precisely is out of question as solving kOBJ without k-limiting is undecidable [64]. When designing a practical pre-analysis, which aims to select the set of context-sensitive variables/objects, C_{ideal}, in the program, the main challenge are to ensure that (1) C_{ideal} includes as many precision-critical variables/objects as possible but as few precision-uncritical variables/objects as possible, (2) C_{ideal}

results in no or little precision loss, and (3) C_{ideal} is found in a lightweight manner to ensure that the pre-analysis overhead introduced is negligible (relative to kOBJ).

Recently, several pre-analyses have been proposed [22,29,40,47,49,74]. Broadly speaking, two approaches exist. EAGLE [47, 49] represents a precision-preserving acceleration of kOBJ by reasoning about CFL (Context-Free-Language) reachability in the program. Designed to be precision-preserving, EAGLE analyzes conservatively and often efficiently the value flows reaching a variable/object and selects the set of context-sensitive variables/objects as a superset of the set of precision-critical variables/objects in the program over-approximately, thereby limiting the potential speedups achieved. On the other hand, ZIPPER [40], as a non-precision-preserving representative of the remaining pre-analyses [22,29,40,74], examines the value flows reaching a variable/object heuristically and often efficiently by selecting the set of context-sensitive variables/objects to include some but not all the precision-critical variables/objects and also some precision-uncritical variables/objects in the program. As a result, ZIPPER can sometimes improve the efficiency of kOBJ more significantly than EAGLE but at the expense of introducing a substantial loss of precision for some programs.

In this chapter, we introduce a new approach, named TURNER, that represents a sweet spot between EAGLE and ZIPPER . TURNER enables kOBJ to run significantly faster than EAGLE while achieving significantly better precision than ZIPPER. Despite losing a small precision in the average points-to set size (#avg-pts), TURNER achieves exactly the same precision for the other three commonly used precision metrics [27, 47, 49, 73, 82, 86], call graph construction (#call-edges), may-fail casting (#fail-casts) and polymorphic call detection (#poly-calls), for a set of 12 popular Java benchmarks and applications evaluated. TURNER is simple, lightweight yet effective due to two novel aspects in its design. First, we exploit a

key observation that some precision-uncritical objects can be approximated initially based on the object-containment relationship that is inferred from the points-to information pre-computed by Andersen's analysis [3]. This approximation turns out to be practically accurate, as it introduces a small degree yet the only source of imprecision into the final points-to information computed. Second, leveraging this initial approximation, we introduce a simple DFA (Deterministic Finite Automaton) to reason about object reachability across a method (from its entry to its exit) intra-procedurally along all the possible value flows established by its statements to finalize all its precision-critical variables/objects selected.

We have validated TURNER with an implementation in SOOT against EAGLE and ZIPPER using a set of 12 Java benchmarks and applications. In general, TURNER enables kOBJ to run significantly faster than EAGLE due to fewer precision-uncritical variables/objects analyzed context-sensitively and achieve significantly better precision than ZIPPER due to more precision-critical variables/objects analyzed context-sensitively than ZIPPER.

In summary, this chapter makes the following contributions:

- We introduce a new approach, TURNER, that can accelerate k-object-sensitive pointer analysis (i.e., kOBJ) for Java programs significantly with negligible precision loss.

- We propose to first approximate the precision-criticality of the objects in a program based on object containment and then decide whether the variables/objects in the program should be context-sensitive or not by conducting an object reachability analysis intra-procedurally with a DFA, which turns out to be simple, lightweight and effective.

- Turner enables $k$obj to run significantly faster than Eagle and achieve significantly better precision than Zipper for a set of 12 popular Java benchmarks and applications evaluated in terms of four common precision metrics, #avg-pts, #call-edges, #fail-casts, and #poly-calls (with Turner losing no precision for the last three metrics).

3.2 Motivation

We motivate Turner in the context of the two state-of-the-art pre-analyses, Eagle [47, 49] and Zipper [40]. Eagle supports partial context-sensitivity as it enables $k$obj to analyze only a subset of variables/objects in a method context-sensitively. On the other hand, Zipper allows $k$obj to analyze a method either fully context-sensitively or fully context-insensitively. Like Eagle, Turner also supports partial context-sensitivity in order to maximize the potential speedups attainable. As in Eagle and Zipper, Turner also relies on the points-to information in a program pre-computed by Andersen's analysis [3] (context-insensitively).

In Section 3.2.1, we examine the main challenges faced in developing a pre-analysis for accelerating $k$obj and discuss the methodological differences between Turner and two existing approaches, Eagle and Zipper. In Section 3.2.2, we introduce a motivating example abstracted from real code by highlighting the effects of these differences on the context-sensitivity selectively applied to $k$obj. In Section 3.2.3, we describe the basic idea behind Turner (including our insights and trade-offs).

3.2.1 Challenges

A variable/object n in a program is precision-critical if kOBJ loses precision when it analyzes n context-insensitively instead of context-sensitively (Definition 3.1). In the case of a precision loss, there must exist some variable v in the program such that its context-insensitive points-to information becomes less precise. In this case, $\overline{\mathtt{PTS}}(v)$ will contain not only the pointed-to objects found before (when n is analyzed context-sensitively) but also some spurious pointed-to objects introduced now (when n is analyzed context-insensitively). As n and v can be further apart in the program, separated by a long sequence of method calls (with complex field accesses on n along the way), designing a practical pre-analysis P, which selects a set of variables/objects in a program for kOBJ to analyze context-sensitively, is challenging (since solving kOBJ without k-limiting is undecidable [64]). For a program, let C_{ideal} be the set of precision-critical variables/objects specified by Definition 3.1 and C_P the set of context-sensitive variables/objects selected by P. The main challenges lie in how to ensure that (1) $|C_{\text{ideal}} - C_P|$ is minimized (so that as many precision-critical variables/objects are selected) and $|C_P - C_{\text{ideal}}|$ is minimized (so that as few precision-uncritical variables/objects are selected), (2) C_P causes kOBJ to lose no or little precision, and (3) C_P is selected in a lightweight manner (so that P introduces negligible overhead relative to kOBJ).

A pre-analysis for kOBJ relies on the following fact to identify a precision-critical variable/object, with its accesses possibly triggered by statements outside its containing method. Without loss of generality, a method is assumed to contain only one return statement of the form "**return** r", where r a local variable in the method (referred to as its *return variable*).

Fact 3.1 *Consider a program being analyzed object-sensitively with the parameters and the return variable of a method modeled as its (special) fields as in [47, 49]. A*

variable/object n in a method M in the program is considered to be precision-critical only if, *during program execution, there is a value flow entering and leaving M via a parameter or the return variable of M, by passing through n (i.e., by first writing into n via an access path and then reading it from the same access path), where n may be the parameter or the return variable itself.*

In this case, analyzing n context-sensitively will allow several such value flows to be tracked separately based on their calling contexts. Otherwise, some precision may be potentially lost.

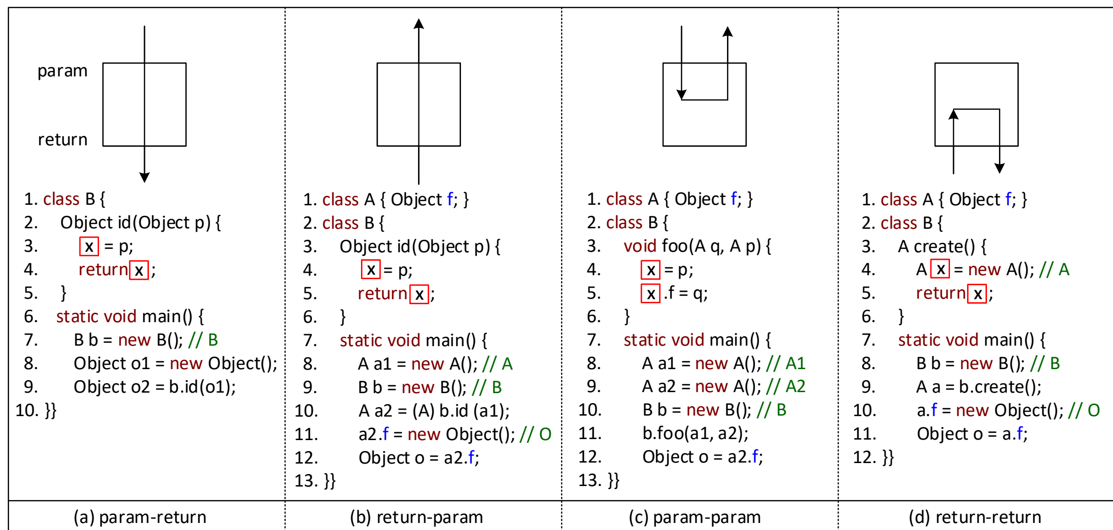


Figure 3.1: A total of four possible value-flow patterns for determinng whether a variable x should be precision-critical or not.

A pre-analysis, as illustrated in Figure 3.1, should identify a (local) variable `x` as precision-critical by considering a total of four possible value-flow patterns passing through `x` (classified according to whether the two end points of a value-flow are a parameter or the return variable of its containing method [49, 76]). The same four patterns are also applicable to a locally allocated object. In "*param-return*" (Figure 3.1(a)), the pre-analysis should recognize that the object created in line 8 will flow into `x` in `id()` via its parameter `p` and then out of `id()` via a

return variable, which happens to be `x` itself. In "*return-param*" (Figure 3.1(b)), the pre-analysis, when checking whether the object created in line 11 will flow into `o` in line 12 or not, will first need to find out what `a2` points to. This will entail reasoning about the value flow of `a2` in reverse order, by entering `id()` via its return statement (variable) and leaving `id()` from its parameter `p`. In "*param-param*" (Figure 3.1(c)), the object `A1` created in line 8 will flow into `x` (or `x.f` precisely) in `foo()` via its parameter `q` and then out of `foo()` via its parameter `p`. In "*return-return*" (Figure 3.1(d)), the pre-analysis, when checking whether the object created in line 10 can flow into `o` in line 11 or not, will need to find what `a` points to, by entering and exiting `create()` from its return variable and visiting `x` in between.

We can now discuss how Turner differs from Eagle [47,49] and Zipper [40] methodologically. To start with, all the three are relatively lightweight with respect to kOBJ. Below we examine these pre-analyses in terms of their efficiency and precision tradeoffs made on approximating C_{ideal}. There are two caveats. First, C_{ideal} is conceptual but cannot be found exactly in a program. Second, some precision-critical variables/objects affect the precision and/or efficiency of kOBJ more profoundly than others, but they cannot be easily identified. How to do so approximately can be an interesting research topic in future work.

Eagle is precision-preserving, since it accounts for all the four value-flow patterns given in Figure 3.1 by reasoning about CFL reachability in the program inter-procedurally to ensure that $C_{\text{ideal}} - C_{\text{EAGLE}} = \emptyset$. For some programs, Eagle may conservatively mis-classify many precision-uncritical variables/objects as being precision-critical, thereby causing $C_{\text{EAGLE}} - C_{\text{ideal}}$ to be unduly large, and consequently, limiting the speedups attainable.

ZIPPER is not precision-preserving (implying that $C_{\text{ideal}} - C_{\text{ZIPPER}} \neq \emptyset$, in general), since it considers only the "*param-return*" and "*return-param*" patterns in Figure 3.1 heuristically by pattern-matching and ignores "*param-param*" (according to its authors [40]) and "*return-return*" (according to its open-source implementation). For some programs, ZIPPER can achieve greater speedups than EAGLE but at a precision loss, since it has misclassified some precision- yet performance-critical variables/objects as context-insensitive.

In this chapter, TURNER is designed to strike a good balance between EAGLE and ZIPPER. We aim to ensure that $|C_{\text{TURNER}} - C_{\text{ideal}}| < |C_{\text{EAGLE}} - C_{\text{ideal}}|$ so that TURNER can enable *k*OBJ to run significantly faster than EAGLE (due to fewer precision-uncritical variable/objects selected for *k*OBJ to analyze context-sensitively). At the same time, we aim to ensure that $|C_{\text{ideal}} - C_{\text{TURNER}}| < |C_{\text{ideal}} - C_{\text{ZIPPER}}|$ so that TURNER can also enable *k*OBJ to achieve significantly better precision than ZIPPER (due to more precision-critical variable/objects selected for *k*OBJ to analyze context-sensitively). We accomplish this by exploiting object containment to approximate the precision-criticality of objects and then reasoning about object reachability by considering all the four value-flow patterns in Figure 3.1 intra-procedurally.

3.2.2 Example

Figure 3.2 gives a Java program abstracted from real code developed based on JDK. In lines 1-25, a simplified `HashMap` class is defined. In lines 26-42, class `A` represents a use case of `HashMap`. In `foo()`, two instances of `HashMap`, `M1` and `M2`, and two instances of `java.lang.Object`, `O1` and `O2`, are created. Afterwards, `O1` (`O2`), pointed to by `v1` (`v2`), is deposited into `M1` (`M2`), pointed to by `map1` (`map2`), with `O` (received from its parameter `k`) as the corresponding key, and later retrieved

and saved in `w1` (`w2`). In `main()`, n instances of `A`, $\texttt{A}_1, ..., \texttt{A}_n$, are created (where $n > 1$) and then used as the receivers for invoking `foo()`.

```
1. class Entry {
2.    Object key, value;
3.    Entry(Object p, Object q) {
4.       this.key = p;
5.       this.value = q;
6. }}

7.  class HashMap {
8.    Entry[] table;
9.    Object get(Object k){
10.      int idx = k.hashCode;
11.      Entry[] t = this.table;
12.      Entry e = t[idx];
13.      Object r = e.value;
14.      return r;
15.   }
16.   void put(Object k, Object v) {
17.      int idx = k.hashCode;
18.      Entry e = new Entry(k, v);  // E
19.      Entry[] t = this.table;
20.      t[idx] = e;
21.   }
22.   HashMap() {
23.      Entry[] t = new Entry[16]; // @
24.      this.table = t;
25. }}

26. class A {
27.   void foo(Object k) {
28.      HashMap map1 = new HashMap(); // M1
29.      HashMap map2 = new HashMap(); // M2
30.      Object v1 = new Object(); // O1
31.      Object v2 = new Object(); // O2
32.      map1.put(k, v1);
33.      map2.put(k, v2);
34.      Object w1 = map1.get(k);
35.      Object w2 = map2.get(k);
36.   }
37.   public static void main(String args[]) {
38.      Object k = new Object(); // O
39.      A ai = new A(); // Ai   ┐
40.      ai.foo(k);              ├ 1 ≤ i ≤ n
41.      ...                     ┘
42. }}
```

Figure 3.2: A Java program abstracted from real code using the standard JDK library.

Table 3.1 lists the contexts used for analyzing this program by the four main analyses, 2OBJ, E-2OBJ, Z-2OBJ, and T-2OBJ. Here, P-2OBJ denotes the version of 2OBJ that adopts the selective context-sensitivity prescribed by $P \in \{\text{E (for EAGLE)}, \text{Z (for ZIPPER)}, \text{T (for TURNER)}\}$. EAGLE is always precision-preserving. For this program, ZIPPER happens to be also precision-preserving since Z-2OBJ behaves exactly as 2OBJ does. TURNER also happens to be precision-preserving but T-2OBJ differs from 2OBJ/Z-2OBJ and E-2OBJ substantially. Below we focus on examining how the context-insensitive points-to information for `w1` and `w2` in `foo()`, $\overline{\texttt{PTS}}(\texttt{w1}) = \{\texttt{O1}\}$ and $\overline{\texttt{PTS}}(\texttt{w2}) = \{\texttt{O2}\}$, is obtained by each of

the four main analyses. For reasons of symmetry, Figure 3.3 illustrates only how $\overline{\texttt{PTS}}(\texttt{w1}) = \{\texttt{O1}\}$ is obtained.

Table 3.1: The contexts used for analyzing the variables/objects in Figure 3.2 by 2OBJ, E-2OBJ, Z-2OBJ, and T-2OBJ (where i in each context containing $\texttt{A}_i/\texttt{a}_i$ ranges over $[1, n]$).

<table>
<tr><th>Method</th><th>Variables/Objects</th><th>2OBJ / Z-2OBJ</th><th>E-2OBJ</th><th>T-2OBJ</th></tr>
<tr><td>Entry</td><td>p, q, this</td><td>[E, M1], [E, M2]</td><td>[E, M1], [E, M2]</td><td>[E, M1], [E, M2]</td></tr>
<tr><td rowspan="2">get</td><td>k</td><td rowspan="2">[M1, A_i], [M2, A_i]</td><td>[]</td><td>[]</td></tr>
<tr><td>e, r, this, t</td><td>[M1, A_i], [M2, A_i]</td><td>[M1], [M2]</td></tr>
<tr><td rowspan="2">put</td><td>k, v, e, this, t</td><td>[M1, A_i], [M2, A_i]</td><td>[M1, A_i], [M2, A_i]</td><td rowspan="2">[M1], [M2]</td></tr>
<tr><td>E</td><td>[M1], [M2]</td><td>[M1], [M2]</td></tr>
<tr><td rowspan="2">HashMap</td><td>this, t</td><td>[M1, A_i], [M2, A_i]</td><td>[M1, A_i], [M2, A_i]</td><td rowspan="2">[M1], [M2]</td></tr>
<tr><td>@</td><td>[M1], [M2]</td><td>[M1], [M2]</td></tr>
<tr><td rowspan="4">foo</td><td>v1, v2, w1, w2</td><td rowspan="4">[A_i]</td><td rowspan="2">[]</td><td rowspan="4">[]</td></tr>
<tr><td>O1, O2</td></tr>
<tr><td>k, map1, map2</td><td rowspan="2">[A_i]</td></tr>
<tr><td>M1, M2</td></tr>
<tr><td rowspan="2">main</td><td>k, a_i</td><td rowspan="2">[]</td><td rowspan="2">[]</td><td rowspan="2">[]</td></tr>
<tr><td>O, A_i</td></tr>
</table>

First of all, 2OBJ analyzes `foo()` for a total of n times by identifying its variables/objects under the i-th invocation with its receiver $\texttt{A}_i$ (Figure 3.3(a)). Thus, $\forall\ 1 \leqslant i \leqslant n : \texttt{PTS}(\texttt{w1}, [\texttt{A}_i]) = \{\texttt{O1}, [\texttt{A}_i]\} \wedge \texttt{PTS}(\texttt{w2}, [\texttt{A}_i]) = \{\texttt{O2}, [\texttt{A}_i]\}$ context-sensitively. By projecting out all the contexts, 2OBJ obtains $\overline{\texttt{PTS}}(\texttt{w1}) = \{\texttt{O1}\}$ and $\overline{\texttt{PTS}}(\texttt{w2}) = \{\texttt{O2}\}$ context-insensitively, as desired.

For this particular program, Z-2OBJ is equivalent to 2OBJ (Table 3.1 and Figure 3.3(a)). However, it is easy to modify it slightly so that Z-2OBJ will behave differently while suffering from a loss of precision (as it does not consider the last two patterns given in Figure 3.1).

E-2OBJ enables 2OBJ to support partial context-sensitivity without losing any precision. The variables/objects in $\{\texttt{v1}, \texttt{v2}, \texttt{w1}, \texttt{w2}, \texttt{O1}, \texttt{O2}\}$ in `foo()` and variable `k` in `get()` will now be context-insensitive. In the case of `foo()`, however, `k`, `map1`,

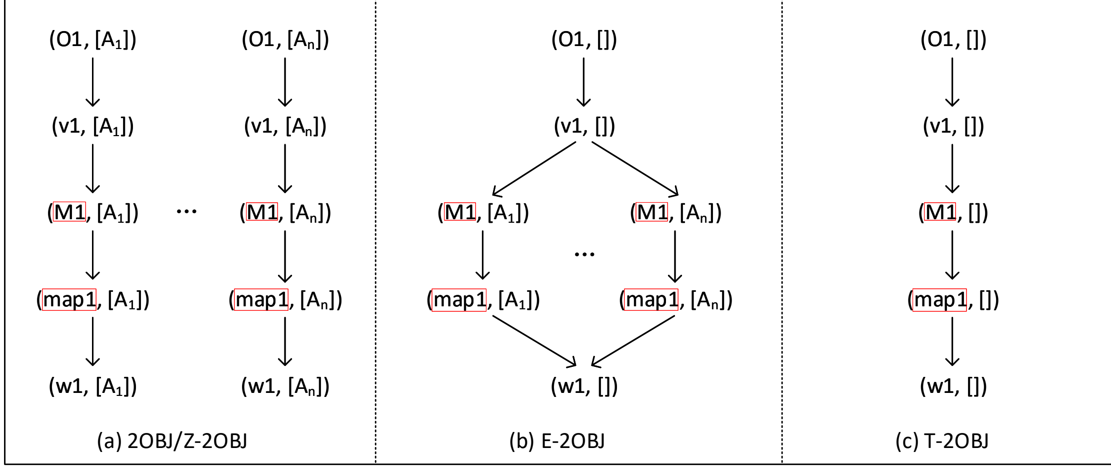


Figure 3.3: Computing $\overline{\texttt{PTS}}(\texttt{w1}) = \{\texttt{O1}\}$ for Figure 3.2 by 2obj, E-2obj, Z-2obj and T-2obj.

`map2`, `M1` and `M2` must still be analyzed context-sensitively due to a spurious "*param-return*" pattern established by the facts that (1) `k` is a parameter, (2) `put()` can write into `M1`/`M2`, and (3) `get()` can read from `M1`/`M2`. As a result, as illustrated in Figure 3.3(b), E-2OBJ will still need to analyze `foo()` for a total of n times, since it must distinguish the two `HashMap` objects `M1` and `M2` created in `foo()` context-sensitively as in 2OBJ, except that it can now analyze the two objects, `O1` and `O2`, created in `foo()` context-insensitively. Thus, E-2OBJ obtains directly that $\texttt{PTS}(\texttt{w1}, [\,]) = \{\texttt{O1}, [\,]\}$ and $\texttt{PTS}(\texttt{w2}, [\,]) = \{\texttt{O2}, [\,]\}$, i.e., $\overline{\texttt{PTS}}(\texttt{w1}) = \{\texttt{O1}\}$ and $\overline{\texttt{PTS}}(\texttt{w2}) = \{\texttt{O2}\}$.

T-2OBJ, as illustrated in Figure 3.3(c), goes beyond E-2OBJ (for this particular program) by modeling `M1` and `M2` also context-insensitively. As a result, `foo()` is analyzed context-insensitively only once. As in the case of E-2OBJ, T-2OBJ also obtains directly that $\texttt{PTS}(\texttt{w1}, [\,]) = \{\texttt{O1}, [\,]\}$ and $\texttt{PTS}(\texttt{w2}, [\,]) = \{\texttt{O2}, [\,]\}$, i.e., $\overline{\texttt{PTS}}(\texttt{w1}) = \{\texttt{O1}\}$ and $\overline{\texttt{PTS}}(\texttt{w2}) = \{\texttt{O2}\}$.

3.2.3 Turner: Our Approach

TURNER is designed to accelerate *k*OBJ with partial context-sensitivity at a negligible loss of precision. Unlike EAGLE [47, 49] and ZIPPER [40], TURNER works by exploiting both object containment and object reachability to enable *k*OBJ to strike a better balance between efficiency and precision. In principle, TURNER may lose precision in its first stage only but will always preserve precision in its second stage if it does not lose precision in its first stage.

3.2.3.1 Object Containment

To start with, we exploit a key insight stated below to identify some precision-uncritical objects approximately based on the object containment relationship that is inferred from the points-to information pre-computed (context-insensitively) by Anderson's analysis [3].

Observation 3.1 *A* top container *is an object that is pointed to by neither (1) another object (which may be the container itself) via a field of a declared type of C or $C[]$, where C is a class type nor (2) the return variable of the method in which the container is allocated.*

A bottom container *is an object that does not point to another object (which may be the container itself) via a field of a declared type of C or $C[]$, where C is a class type.*

Given a program, its top and bottom containers are considered as being precision-uncritical.

Definition 3.2 *Observation 3.1 is said to be precision-preserving for a program if kOBJ does not lose precision when it analyzes the precision-uncritical objects*

identified in the program context-insensitively and the remaining variables/objects exactly as before.

Therefore, an object created by a factory method (regarded here as a method that returns its own allocated objects via its return variable) is not a top container. Such an object will be considered as being precision-uncritical iff it is a bottom container. For a program, the precision-uncritical objects identified here will be analyzed by kOBJ context-insensitively (for the reasons given below) and the remaining objects will be further classified as either precision-critical or precision-uncritical by an object reachability analysis (Section 3.2.3.2).

Consider `create()` in Figure 3.1(d). The object `A` created inside is not regarded as a top container, since it is pointed to by its return variable. In object-sensitive pointer analysis, when `create()` called on receiver object `B` in line 9 is analyzed, returning `A` to this caller is actually modeled as `this.ret = x` (line 5) and `a = b.ret` (line 9), where both `this` and `b` point to `B`, and `ret` can be understood as a special return variable introduced for `create()` (Section 3.3.2.1) [47, 49]. Conceptually, `A` is not a top container. In this example, `A` is not a bottom container either, since `A.f = O` in line 10, where `O` is an instance of `java.lang.Object`. As a result, `A` is considered as being precision-critical. However, if lines 10-11 were not present, then `A` would be deemed as being precision-uncritical as it is now a bottom container.

Consider Figure 3.2 (which is free of factory methods), where a total of $n + 7$ objects can be found: `E`, `@`, `M1`, `M2`, `O1`, `O2`, `O`, $\mathtt{A}_1, ..., \mathtt{A}_n$. According to the object containment relationship inferred from Andersen's analysis, `M1` and `M2` contain `@`, which contains `E`, which contains `O1`, `O2` and `O`. By Observation 3.1, the set of top containers is given by $\{\mathtt{M1}, \mathtt{M2}, \mathtt{A}_1, ..., \mathtt{A}_n\}$ and the set of bottom containers is given by $\{\mathtt{O1}, \mathtt{O2}, \mathtt{O}, \mathtt{A}_1, ..., \mathtt{A}_n\}$. Note that both sets of containers are not necessarily

disjoint. Thus, the $n+5$ objects in $\{\texttt{M1}, \texttt{M2}, \texttt{O1}, \texttt{O2}, \texttt{O}, \texttt{A}_1, ..., \texttt{A}_n\}$ are considered as being precision-uncritical and will thus be analyzed by kOBJ context-insensitively.

In our approach, Observation 3.1 (made based on object containment) represents the only source of imprecision in TURNER, which may propagate into its object reachability analysis. TURNER will suffer only a slight loss of precision in #avg-pts computed by T-kOBJ when some top or bottom containers that should be context-sensitive are mis-classified as being precision-uncritical, and consequently, analyzed by T-kOBJ context-insensitively. However, this does not affect the precision of #call-edges, #fail-casts, and #poly-calls for the set of 12 popular Java programs evaluated (at least). The set of top containers consists of the objects that are allocated and used locally in a method, such as `M1` and `M2` (two `HashMap` objects) in `foo()` in Figure 3.2. These objects do not require context-sensitivity, since their encapsulated data does not usually flow out of its containing methods via their parameters or return variables. On the other hand, a bottom container also does not usually require context-sensitivity, as it represents an object that typically encapsulates its primitive data (if any), including arrays of primitive types if it ever contains pointers, such as `O`, `O1` and `O2` (three field-less `java.lang.Object` objects) in Figure 3.2. In Section 3.4.3, we will examine two examples to explain why TURNER loses some small precision in #avg-pts but preserves precision in #call-edges, #fail-casts, and #poly-calls in real code.

3.2.3.2 Object Reachability

Given a program, TURNER relies on a simple DFA (Deterministic Finite Automaton) to reason about implicitly the four value-flow patterns in Figure 3.1 in a method to select its variables/objects to be analyzed by T-kOBJ context-sensitively. By design, the precision-uncritical objects identified by Observation 3.1 in the pro-

gram are deemed context-insensitive. The remaining objects in the program will be classified by the DFA as either precision-critical (context-sensitive) or precision-uncritical (context-insensitive). Simultaneously, the variables in the program are classified. TURNER's intra-procedural analysis will be precision-preserving if Observation 3.1 is precision-preserving, as it is designed to over-approximate the precision-critical variables/objects in the program.

For our example in Figure 3.2, Table 3.1 gives the contexts selected by TURNER for kOBJ, i.e., T-2OBJ. We discuss only their differences with the contexts selected by EAGLE for kOBJ, i.e., E-2OBJ. By exploiting object containment as discussed in Section 3.2.3.1, `M1`, `M2`, `O1`, `O2`, and `O` have been identified as being precision-uncritical and will thus be analyzed context-insensitively. Given that `M1` and `M2`, are now context-insensitive, `k`, `map1`, and `map2` will also be identified as being context-insensitive by our DFA, as the spurious "*param-param*" pattern that causes EAGLE to flag `M1`, `M2`, `k`, `map1`, and `map2` in `foo()` as being context-sensitive no longer exists (Section 3.2.2). As `M1` and `M2` are context-insensitive, the contexts $[\texttt{M1}, \texttt{A}_i]$ and $[\texttt{M2}, \texttt{A}_i]$ listed under E-2OBJ have been shortened to [`M1`] and [`M2`] under T-2OBJ (Table 3.1).

3.3 Turner

We describe the two stages of TURNER, object containment (Section 3.3.1) and reachability (Section 3.3.2), by focusing predominantly on formalizing our object reachability analysis.

3.3.1 Object Containment

In this first stage on object containment analysis, we identify some precision-uncritical objects in a program based on the points-to information pre-computed by Andersen's analysis [3] according to Observation 3.1. For an object o, we write ret_o to denote the return variable in the method where o is allocated. For two objects o_1 and o_2, we write $o_1 \xrightarrow{class-type(f)} o_2$ if $o_1.f = o_2$ for some field f whose declared type is either C or $C[]$, where C is some class type. As a result, the set of precision-uncritical objects in a program can be found by:

$$\mathsf{CI}^{\text{OBS}}_{\text{TURNER}} = \mathsf{TopCon} \cup \mathsf{BotCon} \tag{3.1}$$

where the sets of top and bottom containers in the program are identified as follows:

$$\begin{aligned}\mathsf{TopCon} &= \left\{ o \;\middle|\; \left(\nexists\, (o', f) \in \mathbb{H} \times \mathbb{F} : o' \xrightarrow{class-type(f)} o\right) \land o \notin \overline{\mathsf{PTS}}(ret_o) \right\} \\ \mathsf{BotCon} &= \left\{ o \;\middle|\; \nexists\, (o', f) \in \mathbb{H} \times \mathbb{F} : o \xrightarrow{class-type(f)} o' \right\}\end{aligned} \tag{3.2}$$

3.3.2 Object Reachability

In this second stage on object reachability analysis, we make use of a DFA to determine intra-procedurally whether a variable/object requires context-sensitivity or not. Let CI_{TURNER} be the set of context-insensitive variables/objects that are finally selected by TURNER to support fine-grained selective context-sensitivity (Figure 2.4). By design, $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}} \subseteq CI_{\text{TURNER}}$, i.e., the precision-uncritical objects selected earlier will always be analyzed context-insensitively. The remaining objects and all the variables in the program will be further classified as either context-sensitive or context-insensitive according to the DFA, by leveraging $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}}$.

We start by using L_F (Equation (2.8)) [69, 78], renamed to L_0 here, to depict value flows and perform pointer analysis intra-procedurally for parameterless methods that contain no calls inside. We reuse [P-New], [P-Assign], [P-Store], and [P-Load] in Figure 2.5 for constructing PAG for L_0. For clarity, we rename them as [T-New], [T-Assign], [T-Store], and [T-Load] respectively.

Below, we introduce how to evolve L_0 incrementally to obtain a regular grammar, i.e., a DFA to decide intra-procedurally whether a variable/object requires context-sensitivity or not.

3.3.2.1 Ignoring Context-Insensitive Value Flows

Instead of computing points-to information in a program directly, Turner is designed to analyze the context-sensitive value flows across the parameters or return variables of its methods (Fact 3.1). Thus, we will ignore the global statements and the precision-uncritical objects in $\mathsf{CI}^{\text{OBS}}_{\text{Turner}}$, as all the value-flows passing through them are context-insensitive.

$$\frac{\texttt{x} = \textbf{new}\ \texttt{T}\ //\ O \quad O \notin \mathsf{CI}^{\text{OBS}}_{\text{Turner}}}{O \xrightarrow{\mathsf{cs\text{-}likely}} O}\ \text{[T-Object]}$$

Figure 3.4: Rule for treating all the objects in $\mathsf{CI}^{\text{OBS}}_{\text{Turner}}$ as context-insensitive.

To handle the objects in $\mathsf{CI}^{\text{OBS}}_{\text{Turner}}$ context-insensitively as global variables, as shown in Figure 3.4, we have added a self-loop edge label, named cs-likely, for each object that is not in $\mathsf{CI}^{\text{OBS}}_{\text{Turner}}$ to indicate that it is currently treated as being potentially context-sensitive but will be classified later as being either context-

sensitive or context-insensitive by our reachability analysis. By adding one new terminal cs-likely to the grammar for defining L_0, we obtain:

$$L_1 : \begin{cases} \mathsf{flowsto} \longrightarrow \mathsf{new}\ \mathsf{flows}^* \\ \mathsf{flows} \longrightarrow \mathsf{assign} \mid \mathsf{store[f]}\ \mathit{alias}\ \mathsf{load[f]} \\ \mathit{alias} \longrightarrow \overline{\mathsf{flowsto}}\ \mathsf{cs\text{-}likely}\ \mathsf{flowsto} \\ \overline{\mathsf{flowsto}} \longrightarrow \overline{\mathsf{flows}}^*\ \overline{\mathsf{new}} \\ \overline{\mathsf{flows}} \longrightarrow \overline{\mathsf{assign}} \mid \overline{\mathsf{load[f]}}\ \mathit{alias}\ \overline{\mathsf{store[f]}} \end{cases} \tag{3.3}$$

We will discuss how to handle method parameters and method calls shortly below.

Let us consider Figure 2.6 again by supposing A to be a cs-likely object, then $L_1(\mathtt{O}, \mathtt{v})$ can also be established as before, since we have:

$$\mathtt{O} \xrightarrow{\mathsf{new}} \mathtt{u} \xrightarrow{\mathsf{store[f]}} \mathtt{p} \xrightarrow{\overline{\mathsf{new}}} \mathtt{A} \xrightarrow{\mathsf{cs\text{-}likely}} \mathtt{A} \xrightarrow{\mathsf{new}} \mathtt{p} \xrightarrow{\mathsf{assign}} \mathtt{q} \xrightarrow{\mathsf{load[f]}} \mathtt{v} \tag{3.4}$$

Otherwise, $L_1(\mathtt{O}, \mathtt{v})$ will no longer be possible due to the absence of $\mathtt{A} \xrightarrow{\mathsf{cs\text{-}likely}} \mathtt{A}$.

To simplify matters, returning values from a method can be treated identically as passing parameters for the method. In object-sensitive pointer analysis [27, 47, 49, 73, 82, 86], a method M is analyzed context-sensitively under different receiver objects. Thus, its return statement "**return** r" can be modeled as "`this.ret` $= r$", where `ret` is a fresh local variable (interpreted now as the *return variable* of M) and the return values in "`this.ret`" can be retrieved by its callers via its receiver objects. Given this simple transformation, the four value-flow patterns given in Figure 3.1 can be unified as one "*param-param*" pattern.

Lemma 3.1 *A variable/object n in a method M requires context-sensitivity only if n lies on a sequence of statements, $s_1, ..., s_r$, such that (1) s_i and s_{i+1} form a def-use chain involving only local variables and* cs-likely *objects, (2) s_1 represents a use of*

either a cs-likely *object or a parameter of M, and (3) s_r represents a definition of $P.f$, where P is a parameter of M (including* **this**) *and f is a field of the objects pointed by P (including M's return variable (***ret***)).*

PROOF. Follows directly from Fact 3.1 and the definition of cs-likely objects. □

In this case, n should be context-sensitive, since the modification effects of different definitions of n on $P.f$ under different calling contexts of M must be separated context-sensitively.

3.3.2.2 Approximating the Value Flows Spanning across Method Calls

We now consider how to handle a method call made in a method being analyzed. TURNER will over-approximate the context-sensitive value flows spanning across a call site without analyzing its invoked methods. With L_1, we can only reason about CFL reachability starting from an object. With L_2 given below, we can also start from a variable (Lemma 3.1):

$$L_2 : \begin{cases} \mathsf{flows} \longrightarrow (\mathsf{new} \mid \mathsf{assign} \mid \mathsf{store[f]}\ \mathit{alias}\ \mathsf{load[f]})^* \\ \mathit{alias} \longrightarrow \overline{\mathsf{flows}}\ \mathsf{cs\text{-}likely}\ \mathsf{flows} \\ \overline{\mathsf{flows}} \longrightarrow (\overline{\mathsf{new}} \mid \overline{\mathsf{assign}} \mid \overline{\mathsf{load[f]}}\ \mathit{alias}\ \overline{\mathsf{store[f]}})^* \end{cases} \tag{3.5}$$

Lemma 3.2 $L_2 \supseteq L_1$.

PROOF. Follows simply from examining the structural differences in their productions. □

In both languages, the aliases between two variables are established in exactly the same way.

Next, we over-approximate L_2 to obtain L_3 by abstracting the field accesses with 1-limited access paths and handling aliases more conservatively (as explained shortly below):

$$L_3 : \begin{cases} \textsf{flows} \longrightarrow (\textsf{new} \mid \textsf{assign} \mid \textsf{load} \mid \textsf{store}\ \mathit{alias})^* \\ \mathit{alias} \longrightarrow \overline{\textsf{flows}}\ \textsf{cs-likely flows} \\ \overline{\textsf{flows}} \longrightarrow (\overline{\textsf{new}} \mid \overline{\textsf{assign}} \mid \overline{\textsf{load}} \mid \mathit{alias}\ \overline{\textsf{store}})^* \end{cases} \tag{3.6}$$

Thus, the fields in loads and stores are ignored, and loads and assignments become indistinguishable, but stores are treated differently (i.e., unsymmetrically as loads) in order to keep track of aliases as desired. Note that L_3 is still a CFL, since (1) a store is required to match a $\overline{\textsf{new}}$, $\overline{\textsf{assign}}$ or $\overline{\textsf{load}}$, and (2) a $\overline{\textsf{store}}$ is required to match a new, assign or load. This balanced-parentheses property is somehow hidden in the *alias*-production.

For the code given in Figure 2.6, $L_3(\texttt{O}, \texttt{v})$ will still hold even if, say, `v = q.f` is replaced by `v = q.g` due to the existence of the following flowsto path:

$$\texttt{O} \xrightarrow{\textsf{new}} \texttt{u} \xrightarrow{\textsf{store}} \texttt{p} \xrightarrow{\overline{\textsf{new}}} \texttt{A} \xrightarrow{\textsf{cs-likely}} \texttt{A} \xrightarrow{\textsf{new}} \texttt{p} \xrightarrow{\textsf{assign}} \texttt{q} \xrightarrow{\textsf{load}} \texttt{v} \tag{3.7}$$

Lemma 3.3 *$L_3 \supseteq L_2$.*

Proof. In L_3, the first and third production can be expressed equivalently as $\textsf{flows} \longrightarrow (\textsf{new} \mid \textsf{assign} \mid \textsf{load} \mid \textsf{store}\ \mathit{alias}\ \textsf{load}?)^*$ and $\overline{\textsf{flows}} \longrightarrow (\overline{\textsf{new}} \mid \overline{\textsf{assign}} \mid \overline{\textsf{load}} \mid \overline{\textsf{load}}?\ \mathit{alias}\ \overline{\textsf{store}})^*$, respectively. Here, (s)? indicates that s is optional, where '(' and ')' can be omitted if s represents one symbol. We can conclude that $L_3 \supseteq L_2$ by noting that the field access paths in L_3 are 1-limited. □

In L_3, a store can now also be matched with a $\overline{\text{store}}$ when looking for aliases:

$$\text{flows} \Longrightarrow^{+} \ldots\ \text{store}\ \overline{\text{flows}}\ \text{cs-likely}\ \text{flows}\ \overline{\text{store}}\ \ldots \tag{3.8}$$

For the code given in Figure 2.6, $L_3(\texttt{O}, \texttt{v})$ will thus still hold if we (1) replace `v = q.f` by `q.g = v` and (2) add `v = new V()`, where the allocated object, `V`, is assumed to be cs-likely:

$$\texttt{O} \xrightarrow{\text{new}} \texttt{u} \xrightarrow{\text{store}} \texttt{p} \xrightarrow{\overline{\text{new}}} \texttt{A} \xrightarrow{\text{cs-likely}} \texttt{A} \xrightarrow{\text{new}} \texttt{p} \xrightarrow{\text{assign}} \texttt{q} \xrightarrow{\overline{\text{store}}} \texttt{v} \xrightarrow{\overline{\text{new}}} \texttt{V} \xrightarrow{\text{cs-likely}} \texttt{V} \xrightarrow{\text{new}} \texttt{v} \tag{3.9}$$

We discuss below how to exploit this property to avoid analyzing the methods invoked at a call site while still keeping track of all context-sensitive value flows spanning the call site.

$$\frac{x \;=\; a_0.m(a_1, ..., a_r)}{\forall\, i : a_i \xrightarrow{\text{store}[p_i^{m'}]} a_0 \quad \forall\, i : a_0 \xrightarrow{\overline{\text{store}[p_i^{m'}]}} a_i \quad a_0 \xrightarrow{\text{load}[\texttt{ret}^{m'}]} x \quad x \xrightarrow{\overline{\text{load}[\texttt{ret}^{m'}]}} a_0} \;\text{[T-CALL]}$$

Figure 3.5: Rule for analyzing a method call.

Consider how kOBJ analyzes a method call $x = a_0.m(a_1, ..., a_r)$, with a target method m' resolved when a_0 points to a receiver object O. Let its $r+1$ parameters be $p_0^{m'}, ..., p_r^{m'}$, where $p_0^{m'}$ represents $\texttt{this}^{m'}$. Let its return variable $ret^{m'}$ be introduced as described in Section 3.3.2.1. Object-sensitively, $p_0^{m'}, ..., p_r^{m'}$ and $\texttt{ret}^{m'}$ are handled as if they were special fields of O [47, 49]: $\forall\ i : a_0.p_i^{m'} = a_i$ for passing parameters and $x = a_0.\texttt{ret}^{m'}$ (for retrieving return values). As a result, Figure 3.5 gives a rule, [T-CALL], for adding the PAG edges required for a method call according to [T-LOAD] and [T-STORE]. When m' is analyzed by kOBJ, where its $\texttt{this}^{m'}$ variable points to O, its parameters will be initialized as $\forall\ i : p_i^{m'} = \texttt{this}^{m'}.p_i^{m'}$ and its return values will be made available in $\texttt{this}^{m'}.\texttt{ret}^{m'}$.

Given how $x = a_0.m(a_1, ..., a_r)$ is modeled above, we can determine whether or not a context-sensitive value flow that enters one of its invoked methods via a parameter can also exit it via another parameter without actually analyzing the invoked method itself, by enforcing $L_3(a_i, a_j)$ conservatively, i.e., ensuring that whatever flows into a_i flows also into a_j, if necessary. As will be clear in Section 3.3.2.3 below, $x = a_0.m(a_1, ..., a_r)$ needs to be approximated this way if a_0 may point to at least one cs-likely object and can be ignored otherwise.

Lemma 3.4 *Given a method M (where how its parameters are modeled is irrelevant here), when analyzing a call $x = a_0.m(a_1, ..., a_r)$ contained in M, $L_3(a_i, a_j)$ is established iff a_0 points to at least one cs-likely object.*

PROOF. Let O be an object pointed by a_0. By [T-CALL], passing a_i and a_j to a target method m' at the call site is modeled by two stores $a_0.p_i^{m'} = a_i$ and $a_0.p_i^{m'} = a_j$. Thus, we have:

$$\mathsf{flows} \Longrightarrow^{+} \ldots\ a_i \xrightarrow{\mathsf{store}} a_0\ \overline{\mathsf{flows}}\ O \cdots O\ \mathsf{flows}\ a_0 \xrightarrow{\overline{\mathsf{store}}} a_j\ \ldots \tag{3.10}$$

As a result, $L_3(a_i, a_j)$ is established (as far as this particular call site is concerned, regardless of its truthhood established elsewhere) iff O is a cs-likely object, in which case the "$\cdots$" that sits between the two occurrences of O can be replaced by $\xrightarrow{\text{cs-likely}}$. □

3.3.2.3 Approximating the Incoming Value Flows from Parameters

We discuss now how to handle the parameters of a method when it is analyzed. It is not computationally feasible to formulate our pre-analysis for a method in terms of L_3 directly (even after its parameters are modeled in a certain way). As L_3 is a CFL (with balanced parentheses), the worst-time complexity for finding

the points-to set of a variable is $O(N^3\Gamma_{L_3}^3)$, where N is the number of nodes in the PAG and Γ_{L_3} is the size of L_3 [32, 63].

We now over-approximate L_3 by turning it into a regular language L_4 defined by:

$$L_4 : \begin{cases} \mathsf{flows} \longrightarrow (\mathsf{new} \mid \mathsf{assign} \mid \mathsf{load})^*((\mathsf{store} \mid \overline{\mathsf{store}})\ \overline{\mathsf{flows}})? \\ \overline{\mathsf{flows}} \longrightarrow (\overline{\mathsf{new}} \mid \overline{\mathsf{assign}} \mid \overline{\mathsf{load}})^*(\mathsf{cs\text{-}likely\ flows})? \end{cases} \tag{3.11}$$

Lemma 3.5 $L_4 \supseteq L_3$.

PROOF. L_4 is regularized from L_3 by no longer distinguishing store and $\overline{\mathsf{store}}$. □

Thus, we are now even more conservative in abstracting aliases in L_4 than in L_3. If we replace `p.f = u` with `u.f = p` in Figure 2.6, $L_3(\mathtt{O}, \mathtt{v})$ will not hold but $L_4(\mathtt{O}, \mathtt{v})$ will, since

$$\mathtt{O} \xrightarrow{\mathsf{new}} \mathtt{u} \xrightarrow{\overline{\mathsf{store}}} \mathtt{p} \xrightarrow{\overline{\mathsf{new}}} \mathtt{A} \xrightarrow{\mathsf{cs\text{-}likely}} \mathtt{A} \xrightarrow{\mathsf{new}} \mathtt{p} \xrightarrow{\mathsf{assign}} \mathtt{q} \xrightarrow{\mathsf{load}} \mathtt{v} \tag{3.12}$$

We are now ready to describe our final regular language L_5 used to decide if a variable/object in a method should be context-sensitive or not. By exploiting the fact that store and $\overline{\mathsf{store}}$ are treated identically in L_4, we have obtained L_5:

$$L_5 : \begin{cases} s \longrightarrow \mathsf{param\ flows} \\ \mathsf{flows} \longrightarrow (\mathsf{new} \mid \mathsf{assign} \mid \mathsf{load})^*((\mathsf{store} \mid \overline{\mathsf{store}})\ \overline{\mathsf{flows}})? \\ \overline{\mathsf{flows}} \longrightarrow (\overline{\mathsf{new}} \mid \overline{\mathsf{assign}} \mid \overline{\mathsf{load}})^*(\mathsf{cs\text{-}likely\ flows})? \mid \overline{\mathsf{param}}\ e \\ e \longrightarrow \epsilon \end{cases} \tag{3.13}$$

where param and $\overline{\text{param}}$ are terminals of self-loop edges which are added to the PAG for each parameter of a method according to a rule, [T-PARAM], given in Figure 3.6.

$$\frac{p \text{ is a parameter}}{p \xrightarrow{\text{param}} p \quad p \xrightarrow{\overline{\text{param}}} p}\ \text{[P-PARAM]}$$

Figure 3.6: Rule for adding the PAG edges for parameters.

We can now analyze a method without knowing what its parameters may point to, by treating it effectively as a parameterless method, so that all the results developed so far are applicable.

Lemma 3.6 *Let P_1 and P_2 be two (not necessarily different) parameters of a method. Then $L_4(P_1, P_2) \iff L_5(P_1, P_2)$.*

PROOF. Follows straightforwardly by noting the minor differences in their productions. □

As discussed in Section 1.1.3, if L is a CFL, $L(u, v)^n$ holds if $L(u, v)$ holds due to an L-path that contains a node n. Thus, CI_{TURNER} can now be defined as:

$$CI_{\text{TURNER}} = \{n \mid M \in \mathbb{M}, \nexists P_1, P_2 \in \mathit{param}(M) : L_5^{G_M}(P_1, P_2)^n\} \tag{3.14}$$

where $\mathit{param}(M)$ is the set of parameters of a method M, L_5 is superscripted with the PAG, G_M, built for M, and n is a node in G_M. By construction, $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}} \subseteq CI_{\text{TURNER}}$ holds due to the absence of a self-loop edge, labeled cs-likely, around each object in $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}}$. In addition, all the global variables will be context-insensitive regardless.

Let us apply TURNER to the four examples in Figure 3.1 to see how it has successfully selected `x` to be context-sensitive (where "return `x`" in each example

has been replaced by "`this.ret = x`" and the object `A` created in Figure 3.1(d) is assumed to be a cs-likely object):

- **Figure 3.1(a) and 3.1(b):** $L_5(\texttt{p}, \texttt{this})^{\texttt{x}}$: $\texttt{p} \xrightarrow{\textsf{assign}} \texttt{x} \xrightarrow{\textsf{store}} \texttt{this}$.
- **Figure 3.1(c).** $L_5(\texttt{p}, \texttt{q})^{\texttt{x}}$: $\texttt{p} \xrightarrow{\textsf{assign}} \texttt{x} \xrightarrow{\overline{\textsf{store}}} \texttt{q}$.
- **Figure 3.1(d):** $L_5(\texttt{this}, \texttt{this})^{\texttt{x}}$: $\texttt{this} \xrightarrow{\overline{\textsf{store}}} \texttt{x} \xrightarrow{\overline{\textsf{new}}} \texttt{A} \xrightarrow{\textsf{cs-likely}} \texttt{A} \xrightarrow{\textsf{new}} \texttt{x} \xrightarrow{\textsf{store}} \texttt{this}$.

Finally, we show that TURNER is precision-preserving if Observation 3.1 is precision-preserving. The basic idea is to show that if a variable/object is context-sensitive according to Theorem 3.1, i.e., Fact 3.1 (Figure 3.1), then it must reside on an L_5-path.

Theorem 3.7 *Suppose Observation 3.1 is precision-preserving. Let G be the PAG built for a method M by using rules [T-NEW], [T-ASSIGN], [T-STORE], [T-LOAD], [T-OBJECT], and [T-CALL]. If a variable/object n in M is context-sensitive by Lemma 3.1, then $L_5(P_1, P_2)^n$, where P_1 and P_2 are two (not necessarily different) parameters of M.*

PROOF. Our proof proceeds in the following three steps:

1. We assume that M is analyzed equivalently under one cs-likely receiver object, O_M. Let M' be obtained from M by augmenting it with (1) "$\texttt{this}^M$ = **new** T // O_M" and (2) "$P = \texttt{this}^M.P$" for every parameter P of M. Let G' be the resulting PAG augmented from G. For every parameter P of M, we now have $P \xrightarrow{\overline{\textsf{assign}}} \texttt{this}^M \xrightarrow{\overline{\textsf{new}}} O_M \xrightarrow{\textsf{cs-likely}} O_M \xrightarrow{\textsf{new}} \texttt{this}^M \xrightarrow{\textsf{assign}} P$. Thus, $L_5(P_1, P_2)^n$ holds over G, where P_1 and P_2 are two parameters of M iff $L_5(P', P')^n$ holds over G', where P' is a parameter of M. In L_5, every

variable will now be guaranteed to point to at least one object, which can be O_M.

2. We show now that all the context-sensitive value flows that enter M under its different calling contexts are tracked in L_5 if they pass through a method call $b = a_0.m_0(a_1, ..., a_r)$ (via $a_0, ..., a_r$). Thus, it suffices to consider each call site in M in isolation. Note that the loads and stores in a program can always be modeled as getters and setters.

   By Lemmas 3.5 and 3.6, Theorem 3.4 applies also to L_5: $L_5(a_i, a_j)$ is established in analyzing $b = a_0.m_0(a_1, ..., a_r)$ iff a_0 points to at least one cs-likely object. Thus, we only need to argue that if a_0 points to only context-insensitive objects, recorded in F_{a_0}, then each invoked method at this call site can be ignored in this sense. In this case (where $O_M \notin F_{a_0}$ as O_M is context-sensitive by construction), if some pointed-to objects of a_0 are missing in F_{a_0} (as our pre-analysis is intra-procedural), then there must exist a call chain, $a_0 = x_1.m_1(...), x_1 = x_2.m_2(...), ..., x_{t-1} = x_t.m_t(...)$ (modeled effectively as $a_0 = x_1 = ... = x_t$ in L_5), where all the pointed-to objects of x_t in the program are found intra-procedurally (under the assumption that all the receiver objects of M are abstracted by one single context-sensitive object, O_M, as explained in Step 1).

   Since Observation 3.1 is assumed to be precision-preserving, the value flows that enter M under its different calling contexts (i.e., receiver objects) need not be tracked, i.e., separated context-sensitively at each call site $m_i()$. To prove this claim inductively, let us write $x_{-1} = x_0.m_0(...)$ to represent $b = a_0.m_0(...)$. Now, let R_{m_i} be the set of objects returned by $m_i()$ but missed by L_5, as $m_i()$ is not analyzed. Our claim is true for $x_{t-1} = x_t.m_t(...)$, since all the objects pointed to by x_t in the program are context-insensitive. This

also implies that the objects in R_{m_t} are all conflated under different calling contexts of M. Suppose that our claim holds for $m_i()$, in which case, the objects in R_{m_i} are all conflated. Let us consider $x_{i-2} = x_{i-1}.m_{i-1}(...)$. As x_{i-1} can only point to either some context-insensitive objects in F_{a_0} found intra-procedurally by L_5 or the conflated objects in R_{m_i}, our claim must also be true for $m_{i-1}()$.

3. If a variable n is context-sensitive by Lemma 3.1, there must exist a cs-likely O due to Step 1 such that $L_1(O,P)^n : O$ flows $n' \xrightarrow{\text{store}} P$, which contains n, where n' is a variable (which may be n) and P is a parameter of M. By applying Lemmas 3.2 – 3.6 and the result established in Step 2, we must have $L_5(O,P)^n : O$ flows $n' \xrightarrow{\text{store}} P$ (passing through n). As a result, $L_5(P,P)^n$: $P \xrightarrow{\overline{\text{store}}} n'\ \overline{\text{flows}}\ O \xrightarrow{\text{cs-likely}} O$ flows $n' \xrightarrow{\text{store}} P$ holds. If an object n is context-sensitive by Lemma 3.1, $L_5(P,P)^n$ can be established similarly.

□

3.3.2.4 Computing CI_{Turner} with a DFA

We give an efficient algorithm for computing CI_{TURNER} with a DFA (Figure 3.7) obtained equivalently from the regular grammar for L_5. Our algorithm proceeds in linear time of the number of nodes in the PAG by exploiting a special property in our DFA.

The DFA is a quintuple $\mathcal{A} = (Q, \Sigma, \delta, s, e)$, where $Q = \{s, \mathsf{flows}, \overline{\mathsf{flows}}, e\}$ is the set of states, $\Sigma = \{\mathsf{param}, \overline{\mathsf{param}}, \mathsf{new}, \overline{\mathsf{new}}, \mathsf{assign}, \overline{\mathsf{assign}}, \mathsf{load}, \overline{\mathsf{load}}, \mathsf{store}, \overline{\mathsf{store}}, \text{cs-likely}\}$ is the alphabet, $\delta : Q \times \Sigma \mapsto Q$ is the state transition function, s is the start state, and e is the accepting, i.e., final state.

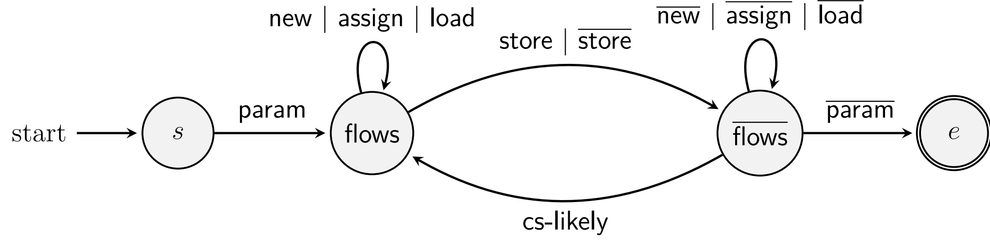


Figure 3.7: The DFA as an equivalent representation of the grammar for defining L_5.

Definition 3.3 *Given a PAG edge $n_1 \xrightarrow{\sigma} n_2$ with a corresponding state transition $\delta(q_1, \sigma) = q_2$, we define $(n_1, q_1) \rightarrowtail (n_2, q_2)$ as a one-step transition. The transitive closure of $\rightarrowtail$, denoted by $\rightarrowtail^+$, represents a multiple-step transition.*

We describe an important property of our DFA in Lemmas 3.8 and 3.9 below.

Lemma 3.8 *Let n_1 and n_2 be two PAG nodes. We have (1) $(n_1, s) \rightarrowtail^+ (n_2, \mathsf{flows}) \implies (n_2, \overline{\mathsf{flows}}) \rightarrowtail^+ (n_1, e)$ and (2) $(n_1, s) \rightarrowtail^+ (n_2, \overline{\mathsf{flows}}) \implies (n_2, \mathsf{flows}) \rightarrowtail^+ (n_1, e)$.*

Proof. To prove (1), we note that n_1 flows $n_2 \implies n_2$ $\overline{\mathsf{flows}}$ n_1 in L_5. To prove (2), we note that n_1 flows $n \xrightarrow{\mathsf{store}\,|\,\overline{\mathsf{store}}} n_2 \implies n_2 \xrightarrow{\overline{\mathsf{store}}\,|\,\mathsf{store}} n$ $\overline{\mathsf{flows}}$ n_1 in L_5, where n is a PAG node. □

Lemma 3.9 *Let n_1 and n_2 be two PAG nodes. We have $(n_2, \overline{\mathsf{flows}}) \rightarrowtail^+ (n_1, e) \implies (n_1, s) \rightarrowtail^+ (n_2, \mathsf{flows})$ and $(n_2, \mathsf{flows}) \rightarrowtail^+ (n_1, e) \implies (n_1, s) \rightarrowtail^+ (n_2, \overline{\mathsf{flows}})$.*

Proof. Proceeds as in the proof of Theorem 3.8 by noting [T-Param] given in Figure 3.6. □

In (3.14), we include a variable/object n in a method M (with its PAG denoted by G_M) into CI_{TURNER} if $L_5^{G_M}(P_1, P_2)^n$ does not hold for any two parameters P_1 and P_2 of M. In terms of our DFA, $L_5^{G_M}(P_1, P_2)^n$ holds iff $(P_1, s) \rightarrowtail^+ (n, q) \rightarrowtail^+ (P_2, e)$, where $q \in \{\mathsf{flows}, \overline{\mathsf{flows}}\}$.

Theorem 3.10 *Let n be a variable/object in a method with P_1 and P_2 as its two parameters.* $(P_1, s) \rightarrowtail^+ (n, q) \rightarrowtail^+ (P_2, e) \iff (P_2, s) \rightarrowtail^+ (n, \overline{q}) \rightarrowtail^+ (P_1, e)$, *where* $q \in \{\mathsf{flows}, \overline{\mathsf{flows}}\}$.

Proof. Lemmas 3.8 and 3.9. □

As a result, we have designed an efficient algorithm for verifying $L_5^{G_M}(P_1, P_2)^n$ by verifying $n \in R_M(\mathsf{flows}) \cap R_M(\overline{\mathsf{flows}})$ (Theorem 3.10) for a method M (with G_M as its PAG), in which, $R : Q \mapsto \wp(\mathbb{V} \cup \mathbb{H})$ returns a set of nodes in G_M reached at a given state $q \in Q$ and $R^{-1} : \mathbb{V} \cup \mathbb{H} \mapsto \wp(Q)$ is the inverse of R. These two functions are computed according to the two rules given in Figure 3.8. The two rules are simple: [A-I] performs the initializations needed while [A-II] computes a fixed point for each function iteratively.

$$\frac{n \in N_M}{n \in R_M(s) \quad s \in R_M^{-1}(n)} \qquad \text{[A-I]}$$

$$\frac{n_1 \xrightarrow{\sigma} n_2 \in E_M \quad q_1 \in R_M^{-1}(n_1) \quad \delta(q_1, \sigma) = q_2 \quad q_2 \notin R_M^{-1}(n_2)}{n_2 \in R_M(q_2) \quad q_2 \in R_M^{-1}(n_2)} \qquad \text{[A-II]}$$

Figure 3.8: Rules for computing R_M and R_M^{-1} for a method M with $G_M = (N_M, E_M)$.

Given R_M computed above, we can now obtain CI_{TURNER} efficiently as follows:

$$CI_{\text{TURNER}} = \{n \mid M \in \mathbb{M}, n \text{ is a node in } G_M, n \notin R_M(\mathsf{flows}) \cap R_M(\overline{\mathsf{flows}})\} \quad (3.15)$$

3.3.3 Time Complexity

The worst-case time complexity of Turner in analyzing a program is linear in terms of its number of statements, for two reasons. First, $\mathsf{CI}_{\text{TURNER}}^{\text{OBS}}$ given in (3.1) and (3.2) can be found in $O(|\mathbb{H}|)$ based on the points-to information already computed

by Andersen's analysis [3]. Second, R_M used in (3.15) for a method M, with its PAG denoted $G_M = (N_M, E_M)$, can be computed by the rules in Figure 3.8 in $O(|E_M| \times |Q|)$, where $|E_M|$ is the number of edges in G_M (constructed linearly based on the number of statements in M according to the rules [T-NEW], [T-ASSIGN], [T-STORE], [T-LOAD] and those in Figures 3.4–3.6) and $|Q|$, i.e., the number of states in the DFA (Figure 3.7), is 4.

3.4 Evaluation

We demonstrate that TURNER can accelerate kOBJ significantly with only negligible precision loss, by being both substantially faster than EAGLE [47,49] (the currently best precision-preserving pre-analysis) and substantially more precise than ZIPPER [40] (the currently best non-precision-preserving pre-analysis). We address the following three research questions:

- RQ1. Is TURNER precise?
- RQ2. Is TURNER efficient?
- RQ3. Is TURNER effective (by exploiting object containment and reachability)?

We have implemented TURNER in SOOT [89], a program analysis and optimization framework for Java, on top of its context-insensitive Andersen's pointer analysis, SPARK [36], and an object-sensitive version of SPARK (i.e., kOBJ) developed by ourselves. Our pre-analysis is implemented in 1000 lines of Java code, which has been released as an open-source tool at http://www.cse.unsw.edu.au/~corg/turner. To compare TURNER with EAGLE [49] and ZIPPER [40], we have

implemented EAGLE based on its three rules (in 600 lines of Java code) and used ZIPPER's latest version (b83b038).

As ZIPPER is evaluated in DOOP [72], we have used an experimental setting that is as close as possible to its original one in several major aspects. First, objects such as `StringBuilder`, `StringBuffer` and `Throwable` objects are merged in terms of their dynamic types and then analyzed context-insensitively as is often done in DOOP [11] and WALA [26]. Second, we perform an exception analysis together with kOBJ as in DOOP by handling exception objects caught in terms of so-called exception-catch links [10]. Third, for type-filtering purposes performed on the elements of an array, we use the declared type of its elements instead of `java.lang.Object`. Finally, we use the summaries provided in SOOT to handle native code.

We have carried out all the experiments on an Intel(R) Xeon(R) CPU E5-2637 3.5GHz machine with 512GB of RAM. We have selected a set of 12 popular Java programs, including 9 benchmarks from DaCapo2006 [7], and 3 Java applications (`checkstyle`, `JPC` and `findbugs`), which are commonly used in evaluating kOBJ [27, 29, 74, 82, 84]. The Java library used is `JRE1.6.0_45` (as the DaCapo2006 benchmarks rely only on an older version of JRE). We use TAMIFLEX [9], a dynamic reflection analysis tool, to resolve Java reflection as is often done in the pointer analysis literature [40, 47, 49, 73, 74, 82].

The time budget used for running each object-sensitive pointer analysis on a program is set as 24 hours. The analysis time of a program is an average of three runs.

Table 3.2 presents our main results. We compare TURNER with EAGLE and ZIPPER in terms of their efficiency and precision tradeoffs made on improving kOBJ. For each $k \in \{2, 3\}$ considered, kOBJ is the baseline, Z-kOBJ, E-kOBJ and T-kOBJ

are the versions of kOBJ for performing selective context-sensitivity under ZIPPER, EAGLE and TURNER, respectively.

3.4.1 RQ1: Precision

Table 3.2 lists four common metrics used for measuring the precision of a context-sensitive pointer analysis: #fail-casts, #call-edges, #poly-calls, and #avg-pts. EAGLE is designed to be precision-preserving by ensuring that E-kOBJ produces exactly the same context-insensitive points-to information as kOBJ. Thus, E-2OBJ and E-3OBJ achieve trivially the same precision in all the four metrics. ZIPPER is designed to accelerate kOBJ heuristically as much as possible (by also ignoring the last two value-flow patterns in Figure 3.1) while allowing sometimes a significant loss of precision. For 2OBJ, Z-2OBJ has caused its #avg-pts to increase by 18.1% on average, resulting in the average percentage precision losses of 7.8%, 0.7%, and 1.7% for #fail-casts, #call-edges, and #poly-calls, respectively. For 3OBJ, Z-3OBJ has caused its #avg-pts to increase by 16.2% on average, resulting in the average percentage precision losses of 10.8%, 0.7%, and 2.0% for #fail-casts, #call-edges, and #poly-calls, respectively. In this thesis, TURNER is designed to trade only a slight loss of precision for efficiency (by reasoning all the four value-flow patterns in Figure 3.1 (implicitly) using a DFA based on object containment and reachability). Despite some slightly imprecise points-to information produced (with #avg-pts increasing by 0.6% and 0.5% under T-2OBJ and T-3OBJ, respectively), both T-2OBJ and T-3OBJ preserve the precision for #fail-casts, #call-edges, and #poly-calls across all the 12 programs.

Table 3.2: Main results. For a given $k \in \{2, 3\}$, the speedups of E-kOBJ, Z-kOBJ, and T-kOBJ are normalized with kOBJ as the baseline. For all the metrics except "Speedup", smaller is better.

	Metrics	2OBJ	E-2OBJ	Z-2OBJ	T-2OBJ	3OBJ	E-3OBJ	Z-3OBJ	T-3OBJ
antlr	Time (s)	24.5	12.4	12.7	6.8	628.9	570.8	141.4	196.5
	Speedup	-	2.0x	1.9x	3.6x	-	1.1x	4.4x	3.2x
	#fail-casts	516	516	565	516	456	456	513	456
	#call-edges	50975	50975	51203	50975	50948	50948	51176	50948
	#poly-calls	1607	1607	1629	1607	1600	1600	1622	1600
	#avg-pts	6.110	6.110	6.585	6.125	4.927	4.927	5.427	4.945
bloat	Time (s)	412.6	290.9	324.2	138.9	10648.2	6994.7	6878.9	1902.8
	Speedup	-	1.4x	1.3x	3.0x	-	1.5x	1.5x	5.6x
	#fail-casts	1295	1295	1349	1295	1198	1198	1256	1198
	#call-edges	56488	56488	56988	56488	56258	56258	56837	56258
	#poly-calls	1549	1549	1587	1549	1535	1535	1577	1535
	#avg-pts	14.796	14.796	15.672	14.816	13.995	13.995	14.802	14.019
chart	Time (s)	206.2	107.5	28.3	75.1	*OoM*	12346.4	522.7	7886.1
	Speedup	-	1.9x	7.3x	2.7x	-	-	-	-
	#fail-casts	1339	1339	1410	1339	-	1239	1316	1239
	#call-edges	72426	72426	73009	72426	-	71987	72640	71987
	#poly-calls	1988	1988	2011	1988	-	1962	1989	1962
	#avg-pts	4.905	4.905	5.363	4.971	-	4.149	4.799	4.168
eclipse	Time (s)	10680.5	5885.3	4122.8	4686.0	*OoM*	*OoM*	*OoM*	*OoM*
	Speedup	-	1.8x	2.6x	2.3x	-	-	-	-
	#fail-casts	3551	3551	3718	3551	-	-	-	-
	#call-edges	162208	162208	163186	162208	-	-	-	-
	#poly-calls	9525	9525	9572	9525	-	-	-	-
	#avg-pts	17.334	17.334	19.691	17.519	-	-	-	-
fop	Time (s)	18.7	10.2	6.9	5.2	728.1	651.6	123.8	187.3
	Speedup	-	1.8x	2.7x	3.6x	-	1.1x	5.9x	3.9x
	#fail-casts	414	414	460	414	362	362	416	362
	#call-edges	34173	34173	34406	34173	34146	34146	34379	34146
	#poly-calls	816	816	841	816	809	809	834	809
	#avg-pts	3.577	3.577	4.132	3.597	3.359	3.359	3.942	3.383
luindex	Time (s)	15.7	9.4	6.3	4.6	596.3	532.6	131.7	185.0
	Speedup	-	1.7x	2.5x	3.4x	-	1.1x	4.5x	3.2x
	#fail-casts	402	402	455	402	348	348	405	348
	#call-edges	33449	33449	33689	33449	33422	33422	33662	33422
	#poly-calls	905	905	932	905	898	898	925	898
	#avg-pts	3.595	3.595	4.285	3.612	3.352	3.352	4.072	3.374
lusearch	Time (s)	22.3	15.8	11.1	10.4	1968.0	1736.8	523.5	881.1
	Speedup	-	1.4x	2.0x	2.1x	-	1.1x	3.8x	2.2x
	#fail-casts	417	417	473	417	366	366	425	366
	#call-edges	36247	36247	36485	36247	36220	36220	36458	36220
	#poly-calls	1103	1103	1131	1103	1096	1096	1124	1096
	#avg-pts	3.611	3.611	4.229	3.627	3.358	3.358	3.959	3.381
pmd	Time (s)	42.1	23.9	23.8	18.3	1504.0	1380.1	358.6	266.2
	Speedup	-	1.8x	1.8x	2.3x	-	1.1x	4.2x	5.7x
	#fail-casts	1174	1174	1252	1174	1116	1116	1199	1116
	#call-edges	59664	59664	59832	59664	59599	59599	59767	59599
	#poly-calls	2329	2329	2354	2329	2322	2322	2347	2322
	#avg-pts	4.943	4.943	6.378	4.954	4.684	4.684	5.973	4.698
xalan	Time (s)	243.2	121.8	54.2	90.9	25424.4	6771.9	694.2	1386.4
	Speedup	-	2.0x	4.5x	2.7x	-	3.8x	36.6x	18.3x
	#fail-casts	569	569	629	569	516	516	582	516
	#call-edges	45916	45916	46113	45916	45884	45884	46086	45884
	#poly-calls	1589	1589	1611	1589	1582	1582	1604	1582
	#avg-pts	4.253	4.253	5.258	4.272	4.096	4.096	5.014	4.119
checkstyle	Time (s)	1240.6	710.2	484.3	339.3	*OoM*	*OoM*	*OoM*	*OoM*
	Speedup	-	1.7x	2.6x	3.7x	-	-	-	-
	#fail-casts	1129	1129	1203	1129	-	-	-	-
	#call-edges	66702	66702	67528	66702	-	-	-	-
	#poly-calls	2188	2188	2246	2188	-	-	-	-
	#avg-pts	6.380	6.380	10.070	6.491	-	-	-	-
JPC	Time (s)	101.9	59.2	31.0	44.0	2371.1	1172.9	175.9	316.8
	Speedup	-	1.7x	3.3x	2.3x	-	2.0x	13.5x	7.5x
	#fail-casts	1364	1364	1438	1364	1209	1209	1281	1209
	#call-edges	81003	81003	81590	81003	79315	79315	79893	79315
	#poly-calls	4255	4255	4301	4255	4115	4115	4159	4115
	#avg-pts	5.050	5.050	5.486	5.065	4.434	4.434	4.752	4.458
findbugs	Time (s)	1820.6	681.1	128.7	150.9	*OoM*	*OoM*	2133.8	1947.0
	Speedup	-	2.7x	14.1x	12.1x	-	-	-	-
	#fail-casts	2037	2037	2100	2037	-	-	1884	1650
	#call-edges	87532	87532	88134	87532	-	-	87289	86599
	#poly-calls	3472	3472	3487	3472	-	-	3463	3441
	#avg-pts	8.011	8.011	8.804	8.058	-	-	7.203	6.632

3.4.2 RQ2: Efficiency

On average, as shown in Table 3.2, T-kOBJ is faster than E-kOBJ but slower than Z-kOBJ. By adopting the context selections prescribed by each of the three pre-analyses, kOBJ runs faster under all the configurations. We compare TURNER with EAGLE and ZIPPER below.

- **T-kOBJ vs. E-kOBJ.** Both achieve the same precision for #fail-casts, #call-edges, and #poly-calls across the 12 benchmarks for $k \in \{2, 3\}$, but T-kOBJ is faster in each case. For $k = 2$, the speedups of T-2OBJ over 2OBJ range from 2.1x (for `lusearch`) to 12.1x (for `findbugs`) with an average of 3.6x. In contrast, the speedups of E-2OBJ over 2OBJ range from 1.4x (for `bloat` and `lusearch`) to 2.7x (for `findbugs`) with an average of 1.8x only. For $k = 3$, the speedups of T-3OBJ over 3OBJ range from 2.2x (for `lusearch`) to 18.3x (for `xalan`) with an average of 6.2x, while the speedups of E-3OBJ over 3OBJ range from 1.1x (for `antlr`, `fop`, `luindex`, `lusearch`, and `pmd`) to 3.8x (for `xalan`) with an average of 1.6x only. Thus, the speedups of T-kOBJ over E-kOBJ are 1.9x when $k = 2$ and 3.4x (with `chart` included even though 3OBJ is unscalable) when $k = 3$.

  In addition, T-kOBJ exhibits better scalability than E-kOBJ. For the four benchmarks, `chart`, `eclipse`, `checkstyle` and `findbugs`, that are unscalable under 3OBJ, T-3OBJ can now analyze `chart` and `findbugs` successfully but E-3OBJ can analyze `chart` only.

- **T-kOBJ vs. Z-kOBJ.** Despite its substantially better precision, T-kOBJ is faster in seven programs when $k = 2$ and three when $k = 3$. Compared with the kOBJ baseline, the average speedups achieved by T-kOBJ and Z-kOBJ are 3.6x and 3.9x, respectively, when $k = 2$, and 6.2x and 9.3x, respectively,

when $k = 3$. As a result, Z-kOBJ is faster than T-kOBJ by 1.1x when $k = 2$ and 2.7x (with `chart` and `findbugs` included) when $k = 3$, on average. In terms of scalability, T-kOBJ is on par with Z-kOBJ for $k \in \{2, 3\}$.

Table 3.3 gives the numbers of context-sensitive facts established by kOBJ, E-kOBJ, Z-kOBJ and T-kOBJ, with #cs-gpts, #cs-pts and #cs-fpts representing the numbers of context-sensitive objects pointed by global variables (i.e., static fields), local variables and instance fields, respectively, and #cs-calls representing the number of context-sensitive call edges. In general, the speedups of a pointer analysis over a baseline come from a significant reduction in the number of context-sensitive facts computed by the baseline. For example, Z-3OBJ is significantly faster than T-3OBJ and E-3OBJ for `chart` as its number of context-sensitive facts is significantly less than the other two. Similarly, T-3OBJ is also much faster than E-3OBJ and Z-3OBJ for `bloat`. However, the analysis time of a pointer analysis is not linearly proportional to the number of context-sensitive facts computed [86]. For example, T-3OBJ is faster than 3OBJ by 3.2x for `antlr` but achieves a percentage time reduction of only 49.7%.

Table 3.4 gives the times spent by SPARK [36] (an implementation of context-insensitive Andersen's analysis [3]) and the three pre-analyses, EAGLE, ZIPPER and TURNER. As discussed earlier, each pre-analysis relies on the points-to information computed by SPARK to make its context selection decisions. TURNER is significantly faster than EAGLE and ZIPPER across all the 12 programs. On average, we have 1.1 seconds (TURNER), 8.9 seconds (EAGLE) and 12.2 seconds (ZIPPER). EAGLE is a single-threaded pre-analysis, ZIPPER is multi-threaded (with 16 threads used in our experiments), TURNER is currently single-threaded but is embarrassingly parallel, as it is intra-procedural. Without any parallelization, TURNER ex-

Table 3.3: Context-sensitive facts (in millions). For all the metrics, smaller is better.

	Metrics	2OBJ	E-2OBJ	Z-2OBJ	T-2OBJ	3OBJ	E-3OBJ	Z-3OBJ	T-3OBJ
antlr	#cs-gpts	4.0K	3.8K	4.8K	2.2K	6.6K	6.0K	12.2K	2.8K
	#cs-pts	8.7M	4.9M	8.8M	1.5M	83.4M	63.4M	72.4M	33.3M
	#cs-fpts	0.4M	0.3M	0.4M	0.2M	10.2M	9.9M	10.3M	8.0M
	#cs-calls	2.4M	1.8M	1.0M	0.7M	38.5M	33.5M	6.8M	25.1M
	Total	11.5M	7.1M	10.2M	2.4M	132.1M	106.7M	89.6M	66.4M
bloat	#cs-gpts	3.2K	3.0K	4.0K	2.2K	5.1K	4.3K	11.3K	3.1K
	#cs-pts	120.4M	82.4M	111.1M	36.9M	1196.0M	856.5M	1137.5M	230.8M
	#cs-fpts	4.0M	4.0M	5.1M	3.7M	35.8M	35.4M	51.3M	30.6M
	#cs-calls	35.5M	32.1M	29.5M	15.0M	371.7M	340.5M	298.2M	109.9M
	Total	159.9M	118.4M	145.7M	55.6M	1603.6M	1232.5M	1487.0M	371.3M
chart	#cs-gpts	14.3K	13.0K	10.8K	8.2K	-	34.5K	26.3K	22.0K
	#cs-pts	64.3M	36.7M	17.0M	19.9M	-	1378.0M	171.2M	1005.7M
	#cs-fpts	1.5M	1.1M	0.8M	1.0M	-	55.4M	24.8M	48.8M
	#cs-calls	20.5M	12.2M	2.5M	8.7M	-	356.0M	23.9M	260.8M
	Total	86.4M	49.9M	20.4M	29.7M	-	1789.4M	220.0M	1315.3M
eclipse	#cs-gpts	40.6K	39.9K	28.8K	10.0K	-	-	-	-
	#cs-pts	991.9M	742.7M	744.5M	565.5M	-	-	-	-
	#cs-fpts	21.8M	21.4M	20.4M	16.2M	-	-	-	-
	#cs-calls	609.1M	342.7M	188.6M	296.5M	-	-	-	-
	Total	1622.8M	1106.8M	953.6M	878.2M	-	-	-	-
fop	#cs-gpts	3.1K	2.9K	3.7K	2.1K	4.5K	3.8K	9.8K	2.7K
	#cs-pts	3.7M	2.1M	3.6M	1.0M	70.3M	56.1M	48.8M	33.5M
	#cs-fpts	0.2M	0.2M	0.2M	0.2M	9.7M	9.4M	9.4M	7.9M
	#cs-calls	1.1M	0.9M	0.5M	0.5M	33.7M	29.8M	4.2M	25.0M
	Total	5.0M	3.2M	4.2M	1.6M	113.7M	95.3M	62.5M	66.4M
luindex	#cs-gpts	2.8K	2.6K	3.8K	1.9K	4.5K	3.9K	11.0K	2.7K
	#cs-pts	3.8M	2.2M	4.2M	1.1M	67.6M	54.2M	56.5M	33.2M
	#cs-fpts	0.2M	0.2M	0.2M	0.2M	9.7M	9.4M	10.8M	8.0M
	#cs-calls	1.1M	0.9M	0.5M	0.5M	33.1M	29.6M	4.7M	25.1M
	Total	5.2M	3.3M	4.9M	1.7M	110.4M	93.2M	72.0M	66.3M
lusearch	#cs-gpts	3.0K	2.7K	3.8K	1.9K	4.2K	3.5K	10.3K	2.5K
	#cs-pts	5.8M	3.9M	5.1M	2.2M	167.7M	151.6M	115.3M	92.2M
	#cs-fpts	0.3M	0.2M	0.2M	0.2M	11.2M	11.0M	11.0M	9.4M
	#cs-calls	2.3M	1.9M	1.0M	1.4M	108.1M	94.9M	40.5M	80.8M
	Total	8.4M	6.0M	6.4M	3.8M	287.1M	257.5M	166.9M	182.4M
pmd	#cs-gpts	3.9K	3.6K	5.9K	2.5K	5.6K	4.9K	23.8K	3.4K
	#cs-pts	12.2M	7.6M	15.1M	4.1M	144.6M	108.8M	184.5M	45.5M
	#cs-fpts	1.1M	1.0M	1.1M	0.9M	15.9M	15.3M	19.0M	11.7M
	#cs-calls	3.6M	2.6M	2.1M	1.7M	58.5M	49.0M	17.0M	33.3M
	Total	16.9M	11.1M	18.4M	6.7M	219.0M	173.1M	220.5M	90.6M
xalan	#cs-gpts	3.9K	3.6K	3.6K	2.4K	15.5K	13.5K	10.3K	6.1K
	#cs-pts	99.1M	45.9M	20.1M	14.3M	1795.3M	987.3M	253.0M	104.5M
	#cs-fpts	2.5M	2.4M	1.8M	1.9M	70.9M	63.6M	18.8M	27.0M
	#cs-calls	26.0M	19.3M	4.7M	17.2M	432.4M	300.8M	35.3M	168.1M
	Total	127.6M	67.6M	26.6M	33.3M	2298.6M	1351.7M	307.1M	299.6M
checkstyle	#cs-gpts	7.8K	7.5K	11.5K	3.9K	-	-	-	-
	#cs-pts	145.0M	107.2M	118.2M	38.0M	-	-	-	-
	#cs-fpts	2.5M	2.3M	3.0M	1.6M	-	-	-	-
	#cs-calls	78.6M	34.5M	23.2M	21.1M	-	-	-	-
	Total	226.1M	144.0M	144.4M	60.7M	-	-	-	-
JPC	#cs-gpts	7.9K	7.1K	7.7K	5.7K	22.1K	19.5K	17.5K	10.2K
	#cs-pts	28.7M	18.8M	13.9M	12.1M	618.1M	319.8M	68.6M	69.1M
	#cs-fpts	1.2M	0.9M	1.0M	0.9M	22.8M	20.0M	13.0M	13.0M
	#cs-calls	9.6M	7.1M	2.7M	5.8M	95.2M	61.4M	7.2M	38.4M
	Total	39.6M	26.9M	17.6M	18.8M	736.1M	401.3M	88.8M	120.5M
findbugs	#cs-gpts	33.5K	32.9K	10.7K	4.0K	-	-	45.6K	6.0K
	#cs-pts	326.4M	245.0M	57.2M	37.8M	-	-	545.9M	183.3M
	#cs-fpts	15.7M	15.5M	4.7M	1.1M	-	-	59.4M	26.6M
	#cs-calls	120.0M	58.3M	11.9M	9.6M	-	-	96.4M	138.5M
	Total	462.0M	318.9M	73.8M	48.5M	-	-	701.7M	348.5M

hibits already negligible analysis times as it runs linearly in terms of the number of statements in a program.

Table 3.4: Times spent by Spark and the three pre-analyses in seconds.

	antlr	bloat	chart	eclipse	fop	luindex	lusearch	pmd	xalan	checkstyle	JPC	findbugs	Avg
SPARK	9.0	10.7	17.2	38.6	8.1	7.4	7.9	13.5	9.5	16.8	19.3	19.8	14.8
EAGLE	3.5	3.8	9.9	34.6	2.8	2.7	3.0	9.3	6.1	9.2	9.6	12.1	8.9
ZIPPER	5.4	6.5	17.1	38.9	4.4	4.2	4.6	9.5	9.0	17.9	11.5	17.4	12.2
TURNER	0.8	0.9	1.4	2.4	0.5	0.5	0.5	1.1	0.8	1.2	1.2	1.3	1.1

3.4.3 RQ3: Effectiveness

TURNER relies on object containment and reachability to make its context selections. In order to understand roughly their percentage contributions to the speedups achieved by T-kOBJ over kOBJ, let us consider two versions of T-kOBJ: (1) T-kOBJ^C, where only object containment is exploited, i.e., the objects in $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}}$ are context-insensitive and all the rest (the variables/objects in $(\mathbb{V} \cup \mathbb{G} \cup \mathbb{H}) \setminus \mathsf{CI}^{\text{OBS}}_{\text{TURNER}}$) are handled as in kOBJ, and (2) T-kOBJ^R, where only object reachability is exploited by assuming $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}} = \emptyset$. Let T-$kOBJ^S_{\text{Speedup}}$ be the speedup obtained by T-kOBJ^S over kOBJ, where $S \in \{C, R, \epsilon\}$, for a program. Certainly, T-kOBJ$^C_{\text{Speedup}}$ + T-kOBJ$^R_{\text{Speedup}}$ = T-kOBJ$_{\text{Speedup}}$ is not expected for a program, as the common contribution made by T-kOBJ^C and T-kOBJ^R towards T-kOBJ$_{\text{Speedup}}$ cannot be meaningfully isolated. Instead, we consider T-kOBJ$^S_{\text{Speedup}}$/(T-kOBJ$^C_{\text{Speedup}}$ + T-kOBJ$^R_{\text{Speedup}}$), where $S \in \{C, R\}$, as the relative percentage contribution made by T-kOBJ^S towards T-kOBJ$_{\text{Speedup}}$ in order to gain a rough understanding about whether both stages are indispensable. Figure 3.9 illustrates the case for accelerating 2OBJ by T-2OBJ, demonstrating that both object containment and object reachability are indeed exploited beneficially for real-world programs.

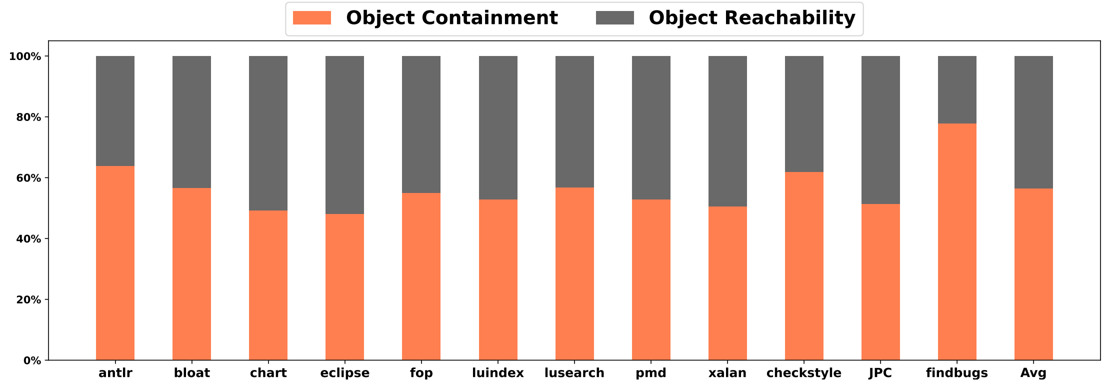


Figure 3.9: Percentage contributions made by Turner's two analysis stages for the speedups of T-2OBJ over 2OBJ.

Our work is largely driven by our insight stated in Observation 3.1. Therefore, Turner is designed to exploit both object containment and reachability to classify the objects, and consequently, the variables in a program as context-sensitive or context-insensitive.

Figure 3.10 gives a Venn diagram showing how Turner classifies the containers, i.e., objects in a program. Based on object containment (Observation 3.1), $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}} = \mathsf{TopCon} \cup \mathsf{BotCon}$ gives the set of precision-uncritical, i.e., context-insensitive objects identified. Based on object reachability (performed by our DFA), $CI^{DFA}_{\text{TURNER}} \subseteq \mathbb{H} \backslash \mathsf{CI}^{\text{OBS}}_{\text{TURNER}}$ gives an additional set of context-insensitive sets identified. Thus, $CS_{\text{TURNER}} = \mathbb{H} \setminus (\mathsf{CI}^{\text{OBS}}_{\text{TURNER}} \cup CI^{DFA}_{\text{TURNER}})$ represents the set of context-sensitive objects identified. On average, across the 12 programs evaluated, the ratios of $|\mathsf{CI}^{\text{OBS}}_{\text{TURNER}}|$, $|CI^{DFA}_{\text{TURNER}}|$ and $|CS_{\text{TURNER}}|$ over $|\mathbb{H}|$ are 61.3%, 4.9%, and 33.8%, respectively. As the performance benefits of making different objects context-insensitive can be drastically different (which are hard to measure individually), these ratios, together with Figure 3.9, demonstrate again the effectiveness of Turner's two analysis stages.

Finally, we give two examples abstracted from the JDK library to explain why Turner does not lose any precision in #call-edges, #fail-casts, and #poly-calls

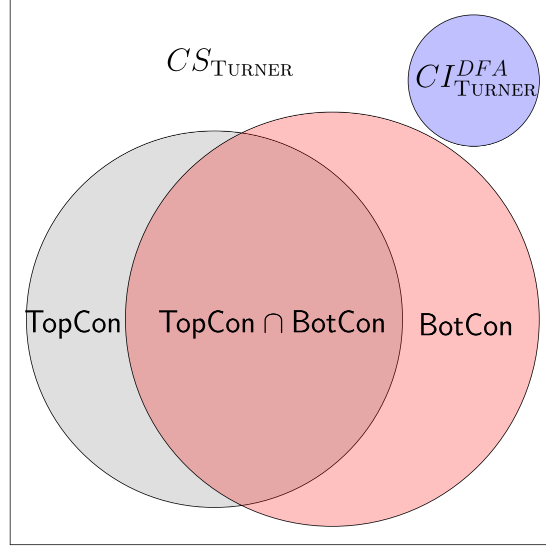


Figure 3.10: The Venn diagram of the objects in a program.

even though it suffers from a small loss of precision in #avg-pts across the 12 programs evaluated. TURNER can render some points-to sets imprecise when some top/bottom containers that are classified as precision-uncritical in $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}}$ should have been analyzed context-sensitively.

Figure 3.11 illustrates a case in which whether the object `P` created in line 4 (a top container according to Observation 3.1) is analyzed context-sensitively or not affects $\overline{\texttt{PTS}}(\texttt{str})$ obtained in line 23. Consider 2OBJ, which will analyze `P` context-sensitively. When analyzing lines 19–22, we find that $\texttt{PTS}(\texttt{u}i, [\,]) = \{(\texttt{U}i, [\,])\} \wedge \texttt{PTS}(\texttt{U}i\texttt{.file}, [\,]) = \texttt{PTS}(\texttt{P.path}, [\texttt{U}i]) = \{(\texttt{S}i, [\,])\}$, where $1 \leqslant i \leqslant 2$. When analyzing line 23, we find that $\texttt{PTS}(\texttt{str}, [\,]) = \{(\texttt{S1}, [\,])\}$. Context-insensitively, 2OBJ thus obtains $\overline{\texttt{PTS}}(\texttt{str}) = \{\texttt{S1}\}$. In the case of T-2OBJ, however, $\texttt{P} \in \mathsf{CI}^{\text{OBS}}_{\text{TURNER}}$ will be analyzed context-insensitively instead. When analyzing lines 19–22, we have $\texttt{PTS}(\texttt{u}i, [\,]) = \{(\texttt{U}i, [\,])\} \wedge \texttt{PTS}(\texttt{U}i\texttt{.file}, [\,]) = \texttt{PTS}(\texttt{P.path}, [\,]) = \{(\texttt{S1}, [\,]), (\texttt{S2}, [\,])\}$, where $1 \leqslant i \leqslant 2$. As `P` is context-insensitive, analyzing line 23 this time will give rise to $\texttt{PTS}(\texttt{str}, [\,]) = \{(\texttt{S1}, [\,]), (\texttt{S2}, [\,])\}$. Thus, context-insensitively, T-2OBJ obtains $\overline{\texttt{PTS}}(\texttt{str}) = \{\texttt{S1}, \texttt{S2}\}$, which contains a spurious target `S2` introduced for

str. Despite this loss of precision in #avg-pts, however, T-2OBJ does not lose any precision in #fail-casts, #call-edges, and #poly-calls, as both S1 and S2 have exactly the same type, java.lang.String.

```
1. class URL {
2.     String file;
3.     URL(String s) {
4.         Parts parts = new Parts(s); // P
5.         this.file = parts.getPath();
6. }
7.     String getFile() {
8.         return this.file;
9. }}
10. class Parts {
11.     String path;
12.     Parts(String p) {
13.         this.path = p;
14. }
15. String getPath() {
16.         return this.path;
17. }}

18. void main() {
19.     String s1 = new String(); // S1
20.     String s2 = new String(); // S2
21.     URL u1 = new URL(s1); // U1
22.     URL u2 = new URL(s2); // U2
23.     String str = u1.getFile();
24.     InputStream in = new FileInputStream(str);
25.     // parse content of the Stream.
26.     in.close();
27. }
```

Figure 3.11: Imprecise points-to information computed by T-2OBJ for a top container P.

```
1. class DerInputBuffer {
2.     byte[] buf;
3.     DerInputBuffer (byte[] p) {
4.         this.buf = p;
5.     }

6.     Date getTime() {
7.         byte[] t = this.buf;
8.         long l = t[0];
9.         return new Date(l);
10. }}
11. class DerValue {
12.     DerInputBuffer buffer;
13.     DerValue(byte[] buf) {
14.         this.buffer = new DerInputBuffer(buf); // D
15. }}
16. void main() {
17.     byte[] b1 = new byte[10]; // B1
18.     byte[] b2 = new byte[10]; // B2
19.     DerValue v1 = new DerValue(b1); // V1
20.     DerValue v2 = new DerValue(b2); // V2
21.     Date d1 = v1.buffer.getTime();
22. }
```

Figure 3.12: Imprecise points-to information computed by T-2OBJ for a bottom container D.

Figure 3.12 illustrates another case in which whether the object D created in line 14 (a bottom container according to Observation 3.1) is analyzed context-sensitively or not affects $\overline{\texttt{PTS}}(\texttt{t})$ obtained in line 7. Consider 2OBJ, which will an-

alyze D context-sensitively. When analyzing lines 17–20, we find that $\texttt{PTS}(\texttt{v}i, [\,]) = \{(\texttt{V}i, [\,])\} \wedge \texttt{PTS}(\texttt{V}i.\texttt{buffer}, [\,]) = \{(\texttt{D}, [\texttt{V}i])\} \wedge \texttt{PTS}(\texttt{D.buf}, [\texttt{V}i]) = \{(\texttt{B}i, [\,])\}$, where $1 \leqslant i \leqslant 2$. When analyzing line 7, we find that $\texttt{PTS}(\texttt{t}, [\texttt{D}, \texttt{V1}]) = \{(\texttt{B1}, [\,])\}$. Context-insensitively, 2OBJ thus obtains $\overline{\texttt{PTS}}(\texttt{t}) = \{\texttt{B1}\}$. In the case of T-2OBJ, however, $\texttt{D} \in \mathsf{CI}^{\text{OBS}}_{\text{TURNER}}$ will be analyzed context-insensitively instead. When analyzing lines 17–20, we have $\texttt{PTS}(\texttt{v}i, [\,]) = \{(\texttt{V}i, [\,])\} \wedge \texttt{PTS}(\texttt{V}i.\texttt{buffer}, [\,]) = \{(\texttt{D}, [\,])\} \wedge \texttt{PTS}(\texttt{D.buf}, [\,]) = \{(\texttt{B}i, [\,])\}$, where $1 \leqslant i \leqslant 2$. As `t` is context-insensitive, analyzing line 7 will give rise to $\texttt{PTS}(\texttt{t}, [\,]) = \{(\texttt{B1}, [\,]), (\texttt{B2}, [\,])\}$. Thus, context-insensitively, T-2OBJ obtains $\overline{\texttt{PTS}}(\texttt{t}) = \{\texttt{B1}, \texttt{B2}\}$, which contains a spurious target `B2` introduced for `t`. Despite this loss of precision in #avg-pts, T-2OBJ loses no precision in #fail-casts, #call-edges, and #poly-calls, as both `B1` and `B2` have exactly the same type, `java.lang.byte[]`, and in addition, each array object pointed by `t` is used in line 8 for obtaining a long integer only.

3.5 Conclusion

In this chapter, we have introduced TURNER, a simple, lightweight yet effective pre-analysis technique that can accelerate object-sensitive pointer analysis for Java programs with negligible precision loss. We exploit a key insight that many precision-uncritical objects in a program can be identified based on a pre-computed object containment relationship. Leveraging this approximation, we can reason about object reachability intra-procedurally to determine whether the remaining objects, together with all the variables, in the program, are precision-critical or not. As a result, we have obtained a novel pre-analysis that can improve the efficiency of object-sensitive pointer analysis significantly while suffering only a small loss of precision in the points-to information produced. Our evaluation shows that TURNER

could preserve the precision of object-sensitive pointer analysis for three important clients, call graph construction, may-fail casting, and polymorphic call detection over a set of 12 popular Java programs evaluated.

Chapter 4

Context Debloating for Object-sensitive Pointer Analysis

In the last chapter, we have introduced Turner as our first fine-grained pointer analysis technique. When designing Turner, we realize that the most significant factor affecting the efficiency of an object-sensitive pointer analysis is the context combinatorial explosion. To mitigate the context explosion issue, this chapter introduces our second fine-grained technique, i.e., the context debloating approach, that was accepted by ASE 2021 [25]. We mitigate context explosion by eliminating the context explosion problem completely for context-independent objects. The key to achieving this lies in how to effectively identify context-independent objects. To address this problem, we introduce Conch, a novel approach that enables us to find thousands of more context-independent objects than Turner (which identifies context-independent objects by simply exploiting object containment). Our context debloating approach is orthogonal to all existing object-sensitive pointer analyses. Thus, it can be used to substantially accelerate all existing object-sensitive pointer analysis at the cost of negligible precision loss.

This chapter is organized as follows. Section 4.1 gives an overview. Section 4.2 motivates our approach. Section 4.3 formalizes context debloating. We present our CONCH approach in Section 4.4 and evaluate the effectiveness of CONCH in terms of context debloating in Section 4.5. Finally, Section 4.6 concludes this chapter.

4.1 Overview

Currently, kOBJ does not scale well for reasonably large programs when $k \geqslant 3$ and is often time-consuming when it is scalable [27, 73, 82, 86]. As k increases, the number of contexts analyzed for a method often blows up exponentially without improving precision much. To alleviate this issue, several recent research efforts [22, 29, 40, 49, 74] focus on selective context-sensitivity, which first conducts a pre-analysis to the program and then instructs kOBJ to apply context-sensitivity only to some of its methods selected. A number of attempts have been made, including client-specific machine learning techniques [29] (guided by improving the precision of a given client, e.g., may-fail-casting) and general-purpose techniques, such as user-supplied hints [22, 74], pattern matching [40], and CFL (Context-Free Language) reachability [24, 47, 49]. Despite some performance improvements obtained (at no or a noticeable loss of precision), these existing selective context-sensitive pointer analysis algorithms still suffer from an unreasonable explosion of contexts.

We introduce a new approach, CONCH, for debloating contexts for all object-sensitive pointer analysis algorithms, including kOBJ and its various incarnations for performing selective context-sensitivity [24, 29, 40, 47, 74], by boosting their performance significantly with negligible loss in precision. In object-oriented programs, we observe that a large number of objects that are allocated in a method are used independently of its calling contexts. Distinguishing these objects context-sensitively,

as often done in the past, will serve to increase only the number of calling contexts analyzed for the methods invoked on these objects (as receivers) without any precision improvement.

Our key insight is to approximate a recently proposed set of two necessary conditions for an object to be context-sensitive, i.e., *context-dependent* [47,49] (whose precise verification is undecidable [64]) with a set of three linearly verifiable necessary conditions (in terms of the number of statements in the program), based on three key observations regarding context-dependability for the objects used practically in real-world object-oriented programs. To create a practical implementation for CONCH, we have developed a new lightweight IFDS-based algorithm [65] for verifying these conditions (governing object reachability). By allowing only context-dependent objects to be handled context-sensitively, CONCH can significantly limit the explosive growth of the number of contexts and achieve substantially improved efficiency and scalability.

We have implemented CONCH on top of the SOOT framework [89] and evaluated it with 12 popular Java benchmarks and applications. Compared with kOBJ [54] and ZIPPER [40] (a representative of selective context-sensitive pointer analyses [40,49,74]), CONCH can speed up the two baselines together substantially (3.1x on average with a maximum of 15.9x) and analyze 7 more programs scalably, but at no loss of precision for 10 programs and only a negligible loss of precision (less than 0.1%) for the remaining two.

In summary, this chapter makes the following contributions:

- We present context debloating, a new approach for accelerating all object-sensitive pointer analysis algorithms.

- We give a set of three mostly necesssary conditions for determining an object's context-dependability and propose a new lightweight IFDS-based algorithm for verifying them on the PAG representation [36] of a program.
- We have implemented CONCH in the SOOT framework and have released it as an open-source tool at `http://www.cse.unsw.edu.au/~corg/conch`.
- We have extensively evaluated the effectiveness of CONCH (using several popular metrics) and demonstrated its practical significance for real-world programs.

4.2 Motivation

We first review object sensitivity as a context abstraction (Section 4.2.1). We then examine the limitations of existing object-sensitive pointer analysis algorithms (Section 4.2.2). Finally, we motivate context debloating, by describing the basic idea behind this new approach, examining the main challenges faced in realizing it efficiently and effectively, and discussing our solution for addressing these challenges (Section 4.2.3).

4.2.1 Object Sensitivity

We briefly review object-sensitive pointer analysis with an example given in Figure 4.1. In lines 7-11, we define class `A`, which has a field `f` and its corresponding setter and getter methods. In lines 12-28, we define class `B`, which has a field `g`, a constructor, and two regular methods (foo() and bar()). In foo() (bar()) of class `B`, an instance of `java.lang.Object`, `O1` (`O2`) is created. Later, `O1` (`O2`) is firstly stored into `A.f` and then loaded into `v1` (`v2`) via the `setF()` and `getF()` methods,

respectively. In `main()`, two instances of `B`, `B1` and `B2`, are created and used as the receivers for invoking foo() and bar(), respectively.

```
1  void main() {
2    B b1 = new B();// B1
3    b1.foo();
4    B b2 = new B();// B2
5    b2.bar();
6  }
7  class A {
8    Object f;
9    void setF(Object o) { this.f = o; }
10   Object getF() { return this.f; }
11 }
12 class B {
13   A g;
14   B() {
15     this.g = new A();// A
16   }
17   void foo() {
18     Object o1 = new Object();// O1
19     A a1 = this.g;
20     a1.setF(o1);
21     Object v1 = a1.getF();
22   }
23   void bar() {
24     Object o2 = new Object();// O2
25     A a2 = this.g;
26     a2.setF(o2);
27     Object v2 = a2.getF();
28 }}
```

Figure 4.1: An example for illustrating object sensitivity.

In a context-insensitive Andersen's analysis [3, 36], every method is analyzed only once under an empty context, [ ]. Let $\overline{\texttt{PTS}}(v)$ denote the points-to set of a variable v thus computed. As illustrated in Figure 4.2(a), `O1` and `O2` are merged at `o` (line 9) and will later flow spuriously to `v2` and `v1`, respectively. Hence, we have $\overline{\texttt{PTS}}(\texttt{v1}) = \overline{\texttt{PTS}}(\texttt{v2}) = \{\texttt{O1},\texttt{O2}\}$.

In a k-object-sensitive pointer analysis (kOBJ), denoted A, the calling contexts of a method are distinguished by its receiver objects, with each being abstracted by its k-most-recent allocation sites [53, 54]. We write $\texttt{PTS}_A(v, c)$ to represent the points-to set of a variable v thus computed under a context c. In the case of 2OBJ (i.e., kOBJ with $k = 2$), `setF()` (`getF()`) will be analyzed differently for its two invocations in lines 20 and 26 (lines 21 and 27) under two different contexts, [`A`, `B1`] and [`A`, `B2`]. As a result, `O1` (created under context [`B1`]) and `O2` (created under context [`B2`]) will flow along two separate paths to `v1` and `v2`, respectively (Figure 4.2(b)). Hence, $\texttt{PTS}_{2\text{OBJ}}(\texttt{v1}, [\texttt{B1}]) = \{(\texttt{O1}, [\texttt{B1}])\}$ and $\texttt{PTS}_{2\text{OBJ}}(\texttt{v2}, [\texttt{B2}]) =$

{(O2, [B2])}, without the spurious points-to information generated by Andersen's analysis.

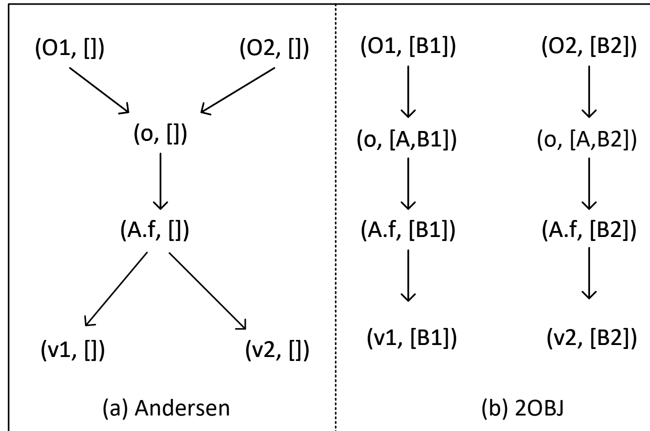


Figure 4.2: Computing the points-to information for `v1` and `v2` in Figure 4.1 by applying Andersen's analysis and 2obj.

4.2.2 Limitations of Existing Algorithms

We now use an example in Figure 4.3, which reuses class `B` from Figure 4.1, to reveal the limitations of kOBJ [53, 54] and existing approaches for selective context-sensitivity [22, 29, 40, 49, 74] in analyzing real-world programs.

In lines 29-51, we define class C with a total of 2^{n+1} methods. In lines 30-38, where $0 \leqslant j < 2^{i-1}$ ($2^{i-1} \leqslant j < 2^i$), a method, $\text{foo}_{i,j}()$ ($\text{bar}_{i,j}()$), is defined, in which an object, $C_{i,j}$, is created and used as the receiver to invoke $\text{foo}_{i-1,\frac{j}{2}}()$ ($\text{bar}_{i-1,\frac{j}{2}}()$). In lines 39-51, we define $\text{foo}_{0,0}$`()` ($\text{bar}_{0,0}$`()`), where an instance of `B` (defined in Figure 4.1), `B3` (`B4`), is created and used to invoke `foo()` (`bar()`). In `main()` (lines 53-65), 2^n instances of `C`, denoted as $\text{C}_{n,j}$, where $0 \leqslant j < 2^n$, are created and used as the receivers to call $\text{foo}_{n-1,\frac{j}{2}}()$ when $j < 2^{n-1}$ and $\text{bar}_{n-1,\frac{j}{2}}()$ when $j \geq 2^{n-1}$.

Figure 4.4 depicts the OAG (Object Allocation Graph) [84], where an edge $O \to O'$ signifies that O is an allocator of O'. For kOBJ [54, 73], the contexts of

```
29  class C {
30    void foo_{i,j}() { // j < 2^{i-1}
31      C c_{i,j} = new C();{// C_{i,j}
32      c_{i,j}.foo_{i-1,j/2}();
33    }
34    D bar_{i,j}(D d) { // 2^{i-1} ≤ j
35      C c_{i,j} = new C(); // C_{i,j}
36      c_{i,j}.bar_{i-1,j/2}(d);
37      return d;
38    }
39    void foo_{0,0}() {
40      B b3 = new B(); // B3
41      b3.foo();
42    }
43    D bar_{0,0}(D d) {
44      B b4 = new B(); // B4
49      b4.bar();
50      return d;
51  }}
52  class D {}
53  void main() {
54    D d = new D(); // D
55    C c = new C(); // C_{n,0}
56    c.foo_{n-1,0}();
57    ...
58    C c = new C(); // C_{n,2^{n-1}-1}
59    c.foo_{n-1,2^{n-2}-1}();
60    C c = new C(); // C_{n,2^{n-1}}
61    c.bar_{n-1,2^{n-2}}(d);
62    ...
63    C c = new C(); // C_{n,2^n-1}
64    c.bar_{n-1,2^{n-1}-1}(d);
65  }
```

Figure 4.3: An example for motivating Conch ($1 \leqslant i \leqslant n$ and $0 \leqslant j < 2^i$), reusing class `B` defined in lines 12-28 in Figure 4.1.

a method can be directly read off from this graph by starting from its receiver object and then retrieving the next $k-1$ objects backwards. For example, the contexts of foo() and bar() are $\{[\texttt{B3}, \mathrm{C}_{1,\frac{j}{2^{k-2}}}, \cdots, \mathrm{C}_{k-2,\frac{j}{2}}, \mathrm{C}_{k-1,j}] \mid 0 \leqslant j < 2^{k-2}\}$ and $\{[\texttt{B4}, \mathrm{C}_{1,\frac{j}{2^{k-2}}}, \cdots, \mathrm{C}_{k-2,\frac{j}{2}}, \mathrm{C}_{k-1,j}] \mid 2^{k-2} \leqslant j < 2^{k-1}\}$, respectively. Let $C_j(X) = [\mathrm{A}, X, \mathrm{C}_{1,\frac{j}{2^{k-3}}}, \cdots, \mathrm{C}_{k-3,\frac{j}{2}}, \mathrm{C}_{k-2,j}]$. Both `setF()` and `getF()` share the contexts in $\{C_j(\texttt{B3}) \mid 0 \leqslant j < 2^{k-3}\} \cup \{C_j(\texttt{B4}) \mid 2^{k-3} \leqslant j < 2^{k-2}\}$.

In practice, the number of contexts for analyzing a method can be exponential. For example, there are a total of 2^{k-2} contexts for `foo()`, `bar()`, `setF()` and `getF()`. As k increases, such a method becomes exponentially expensive to analyze, consuming more and more memory and analysis time.

Existing approaches for selective context-sensitivity [22, 29, 40, 49, 74] can improve the efficiency and scalability of kOBJ. For example, ZIPPER [40], which does not preserve the precision of kOBJ, will select `main()`, `B()`, `foo()`, `bar()`, and $\texttt{foo}_{i,j}()$ (where $j < \frac{r^i}{2}$) to be analyzed context-insensitively. However, the context

explosion problem still remains for $\texttt{bar}_{i,j}$`()`, `setF()` and `getF()`. EAGLE [47, 49], which preserves the precision of kOBJ, is worse as it will also analyze `B()`, `foo()` and `bar()` partially context-sensitively.

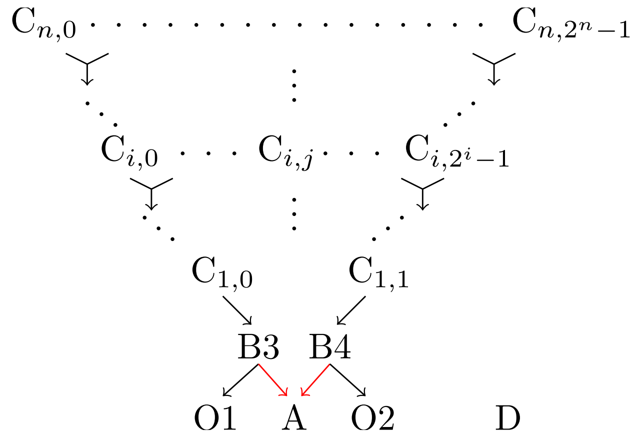


Figure 4.4: The object allocation graph (OAG) for Figure 4.3, where only the two edges in red will remain after context debloating.

4.2.3 Conch: Our Context Debloating Approach

Basic Idea We offer a new approach to mitigating the context explosion problem. Our approach, named CONCH (**CON**text-dependability **CH**ecking), aims to debloat contexts during the pointer analysis and thus complements the prior work on selective context-sensitivity. CONCH can be plugged into all object-sensitive analysis algorithms, including kOBJ and its various incarnations for supporting selective context-sensitivity [22, 29, 40, 49, 74], to boost their performance significantly with negligible loss in precision. For our motivating example, only `A` is context-dependent. Handling any of the other objects context-sensitively will cost an exponential increase in analysis time without any precision benefit.

To illustrate context debloating using the OAG in Figure 4.4, we will remove all the allocators of a context-independent object so that the exponential growth of contexts for the object is avoided completely. Under CONCH, only the two edges

in red will remain, as `A` is the only context-dependent object in the example. This implies that only `setF`() and `getF`() will be analyzed context-sensitively under [`A`,`B3`] and [`A`,`B4`]. All the other methods will be analyzed context-insensitively. For this example, debloating contexts can help kOBJ and its variants reduce their analysis times and memory consumption significantly without losing any precision.

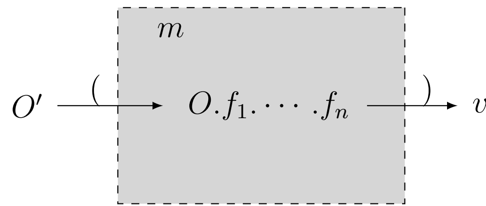


Figure 4.5: Illustrating the conditions for an object to be context-dependent.

Challenges To debloat contexts, we must find context-dependent objects. Recently, the following two necessary conditions (as graphically illustrated in Figure 4.5) are given for determining the context-dependability of an object O allocated in a method m based on a new CFL-reachability formulation for object-sensitive pointer analysis (Section 2.4.2), requiring us to check the existence of a write into and a read from an access path $O.f_1.\cdots.f_n$ context-sensitively (where the two accesses often happen outside m) [47, 49]:

- $A \xrightarrow{(_c} O.f_1.\cdots.f_n$: there exists an object A that flows into m from outside and ends up being stored later into $O.f_1.\cdots.f_n$ under a calling context c of m, and
- $O.f_1.\cdots.f_n \xrightarrow{)_c} v$: there exists a load of $O.f_1.\cdots.f_n$ flowing into a variable v outside m under also c.

where context matching is formulated by solving the standard balanced parentheses problem [65]. If these two conditions hold, O must be context-dependent. Other-

wise, different objects A flowing into $O.f_1.\cdots.f_n$ under different calling contexts of m will be conflated, causing them to flow into different variables v spuriously. In object-sensitive pointer analysis, the parameters and return variable of a method are also conceptually regarded as special fields of its receiver objects [47,49]. Thus, in the access path above, a field f_i can be either a real Java field or one of such special fields.

Unfortunately, verifying these two conditions precisely is undecidable [63], as it requires us to solve kOBJ fully context-sensitively (with $k = \infty$). In addition, weakening these two conditions [47, 49] will over-approximate unduly the number of context-dependent objects found but approximating them heuristically [22, 29, 40,74] may cut it down significantly but at the expense of some significant precision loss.

Our Solution To identify context-dependent objects efficiently and effectively, our key insight is to approximate the two aforementioned necessary conditions with the three conditions that are linearly verifiable (in terms of the number of statements) and mostly necessary for real code, based on three key observations governing how objects are used.

Like the prior work on selective context-sensitivity [22, 29, 40, 74], CONCH also relies on the points-to information pre-computed by Andersen's analysis.

Observation 4.1 *A context-dependent object O often has at least one instance field $O.f$ that is both written into ($x_1.f = \cdots$) and read from ($\cdots = x_2.f$), where $O \in \overline{PTS}(x_1) \land O \in \overline{PTS}(x_2)$, x_1 and x_2 are not necessarily different.*

There can be rare cases, as illustrated in Figure 4.6, where Obs 4.1 may not be valid for some context-dependent objects, such as `B`. Under object-sensitivity [47, 49], `O` pointed to by `p` is first written into `B.q` and then returned and stored into

```
1  void main() {
2    A a = new A(); // A
3    Object o = new Object(); // O
4    Object v = a.wrapId(o);
5  }
6  class B {
7    Object id(Object q) {
8      return q;
9  }}
10 class A {
11   Object wrapId(Object p) {
12     B b = new B(); // B
13     return b.id(p);
14 }}
```

Figure 4.6: A context-dependent object `B` violating Obs 4.1.

`v`. As discussed in Section 4.2.3, `q` is considered as a special field of `B`. Such cases are rare in real-world object-oriented programs, as Conch loses little precision (Section 4.5).

Observation 4.2 *A context-dependent object O, pointed to by a variable or a field of some object, usually flows out of its containing method (for allocating O).*

```
1  Vector(int size) {
2    this.elems = new Object[size];
3  }
```

(a) Case 1 from `Vector`

```
1  Iterator iterator() {
2    return new KeyIterator();
3  }
```

(b) Case 2 from `HashMap`

```
1  void SunJCE_e_a(...) {
2    BufferedReader br = new BufferedReader();
3    this.f = new StreamTokenizer(br);
4  }
```

(c) Case 3 from `SunJCE_e`

Figure 4.7: Three common cases abstracted from JDK for Obs 4.2.

Figure 4.7 gives three representative cases abstracted from the JDK where Obs 4.2 holds. In Figure 4.7(a), the array object created flows out of the constructor via a store. In Figure 4.7(b), the `KeyIterator` object created flows out of `iterator()` directly via a return. In Figure 4.7(c), we have a slightly more complicated case. The `BufferedReader` object created flows out of its containing method as it is stored into the `input` field of the `StreamTokenizer` object, which

flows out of the containing method via a store. The objects that cannot flow out of their containing methods are usually context-independent as they are often created and used locally.

Observation 4.3 *A context-dependent object O tends to have a store statement* `x.f = y` *in a method m', where $O \in \overline{PTS}(x)$. Let m be the method where O is allocated if m' is a constructor (i.e., the constructor for creating O) and m' otherwise. Then y* ***(a)*** *is data-dependent on a parameter of m or* ***(b)*** *points to a context-dependent object.*

```
1 ArrayList() {
2   this.elems = new Object[5];
3 }
4 void set(int idx, E e) {
5   this.elems[idx] = e;
6 }
```

(a) Case 1 from `ArrayList`

```
1 void addEntry(int idx, K k, V v) {
2   this.table[idx] = new Entry(k,v);
3 }
4 Entry(K k, V v) {
5   this.key = k; this.value = v;
6 }
```

(b) Case 2 from `HashMap`.

```
1 HashSet() {
2   this.map = new HashMap();
3 }
```

```
4 HashMap(...) {
5   this.table = new Entry[10];
6 }
```

(c) Case 3 from `HashSet` and `HashMap`.

Figure 4.8: Three common cases abstracted from JDK for Obs 4.3.

Figure 4.8 gives three representative cases abstracted from the JDK where Obs 4.3 holds. In Figure 4.8(a), O is the `Object[]` object allocated in line 2 and `x.f = y` is `this.elems[idx] = e`, which is modeled as `this.elems.arr = e`, where `arr` is a special field introduced to represent all the elements of an array (Section 2.3.2). In this case, $m = m' =$ `set()`. Here, `e` satisfies Obs 4.3(a) trivially. In Figure 4.8(b), O is the `Entry` object allocated in line 2, `x.f = y` is `this.key = k`/`this.value = v`, $m' =$ `Entry()`, and $m =$ `addEntry()`. Here, `k`/`v` (in line 5)

also satisfies Obs 4.3(a) trivially. In Figure 4.8(c), O is the `HashMap` object allocated in line 2, `x.f = y` is `this.table = new Entry[10]`, m' = `HashMap()`, and m = `HashSet()`. As `new Entry[10]` is context-dependent by Obs 4.2 (as well as Obs 4.1 and Obs 4.3 if the entire code is considered), the `HashMap` object in line 2 is also context-dependent by Obs 4.3(b). In Obs 4.3(b), the circular dependences on context-dependability are solved optimistically in Algorithm 1.

Motivating Example For this example given in Figure 4.3 (with class `B` from Figure 4.1), CONCH will identify `A` as the only context-dependent object. Let us examine Figure 4.1, where `A` is created in line 15. `A` is context-dependent as it satisfies all the three observations: (1) `A` has an instance field `f`, which has a write and a read in lines 9 and 10, respectively (Obs 4.1), (2) `A` can flow out of `B()` via the store statement in line 15 (Obs 4.2), and (3) `o` is stored into `A.f` in line 9, where `o` happens to be a parameter of `setF()` (Obs 4.3). Let us now consider `B3` and `B4` created in Figure 4.3. Both are context-independent as both satisfy Obs 4.1 (with an instance field `g` of `B3`/`B4` stored in `B()` and loaded in `foo()`/`bar()` in Figure 4.1) and Obs 4.3 (due to the existence of `this.g = new A(); // A` in line 15, where `A` is context-dependent) but not Obs 4.2 (as `B3`/`B4` does not flow out of its containing method $\texttt{foo}_{0,0}$`()`/$\texttt{bar}_{0,0}$`()`). Finally, all the other objects are context-independent as they do not contain instance fields and are used only locally, failing to satisfy any of the three observations stated.

Discussion CONCH relies on Obs 4.1– Obs 4.3 to generate three corresponding linearly verifiable conditions for determining the context-dependability of an object. In Section 4.4, we introduce a lightweight IFDS-based algorithm for verifying these conditions efficiently. In Section 4.5, we demonstrate CONCH is highly effective for real-world programs.

4.3 Context Debloating

To debloat contexts, we assume that $\mathcal{D}$ represents the set of context-independent objects found by CONCH. Thus, the objects in $\mathbb{H} \setminus \mathcal{D}$ are context-dependent. To support context debloating, we have defined a new context constructor for kOBJ:

$$\texttt{Cons}(O, htx, l, ctx) = \begin{cases} O & \text{if } O \in \mathcal{D} \\ \lceil O +\!\!+ htx \rceil_k & \text{if } O \in \mathbb{H} \setminus \mathcal{D} \end{cases} \tag{4.1}$$

For a context-dependent receiver object, we proceed identically as before. For a context-independent receiver object, we no longer distinguish it under its different allocators, by setting its heap context as $htx = [\,]$, eliminating the context explosion problem that would otherwise have occurred when it is used to construct the context of an invoked method.

CONCH is conceptually simple, algorithmically easy to plug into any existing object-sensitive pointer analysis, and practically effective as validated during our extensive evaluation.

4.4 Conch

We introduce an IFDS-based algorithm [65] for verifying efficiently the three mostly necessary conditions stated in Obs 4.1 – Obs 4.3 to find the context-dependent objects in a program. As these conditions are not sufficient, we may mis-classify context-independent objects as being context-dependent (but err on the side of preserving precision). As these conditions are mostly but not strictly necessary (Figure 4.6), we may occasionally mis-classify context-dependent objects as being context-independent (at a small loss of precision). We use the points-to information

pre-computed by Andersen's analysis [3, 36] (Figure 2.2). We first give a high-level overview of Algorithm 1 and then discuss how to verify these conditions.

CONCH takes a program P as input and returns $\mathcal{D}$ as the set of context-independent objects in P for context-debloating. Some additional notations are in order. For a given object O, $\mathsf{fieldsOf}(O)$ denotes the set of the fields of O. In addition, $\mathsf{hasLoad}(O, f)$ ($\mathsf{hasStore}(O, f)$) holds if P contains a load $\cdots = x.f$ (store $x.f = \cdots$) such that $O \in \overline{\mathsf{PTS}}(x)$. CI and CD, which are initialized to be $\emptyset$ (line 1), represent the sets of context-independent and context-dependent objects found so far, respectively. There are two stages, with the first stage (lines 2-16) for verifying Obs 4.1, Obs 4.2 and Obs 4.3(a) and the second stage (lines 17-23) for verifying Obs 4.3(b).

4.4.1 Verifying Observation 4.1

In lines 3-4, an object O_l is classified as being context-independent (and inserted into CI) if it does not satisfy Obs 4.1. Otherwise, we will proceed to verify Obs 4.2 and Obs 4.3.

4.4.2 Verifying Observation 4.2

In lines 5-6, an object O_l is classified as being context-independent (and inserted into CI) if it does not satisfy Obs 4.2, i.e., $O_l \notin \mathsf{leakObjects}$, where $\mathsf{leakObjects}$ contains the set of objects that can flow out of their containing methods by Obs 4.2. Otherwise, we will proceed to verify Obs 4.3.

We introduce an IFDS-based algorithm given in Figure 4.13 for computing $\mathsf{leakObjects}$ in P context-sensitively, based on the DFA (Deterministic Finite Automaton) given in Figure 4.12. Computing $\mathsf{leakObjects}$ entails reasoning about object reachability in P. Let us describe it incrementally.

Algorithm 1: CONCH: context debloating.

```
   Input: P                                                // Input program
   Output: 𝒟.                                // Set of Context-Indep Objects
 1 CI ← CD ← ∅
 2 for O_l ∈ ℍ do
 3 │ if ∄ f ∈ fieldsOf(O_l) s.t hasLoad(O_l, f) ∧ hasStore(O_l, f) then
 4 │ │ CI = CI ∪ {O_l}                                                 // Obs 1
 5 │ else if O_l ∉ leakObjects then
 6 │ └ CI = CI ∪ {O_l}                                                 // Obs 2
 7 │ else
 8 │ │ R(O_l) = {l' : x.f = y in P | O_l ∈ PTS‾(x)}
 9 │ │ for l' : x.f = y ∈ R(O_l) do
10 │ │ │ if MethodOf(l') is a constructor of O_l then
11 │ │ │ │ m = MethodOf(l)
12 │ │ │ else
13 │ │ │ └ m = MethodOf(l')
14 │ │ │ if depOnParam(y, m) then
15 │ │ │ │ CD = CD ∪ {O_l}                                            // Obs 3(a)
16 │ └ └ │ break
   └
17 UK ← ℍ \ (CI ∪ CD), changed ← true
18 while changed do
19 │ changed ← false
20 │ for O_l ∈ UK do
21 │ │ if ∃ l' : x.f = y ∈ R(O_l) s.t. PTS‾(O_l.f) ∩ CD ≠ ∅ then
22 │ │ │ CD = CD ∪ {O_l}                                                // Obs 3(b)
23 │ └ │ changed ← true
   └
24 𝒟 = CI ∪ (UK \ CD);
25 return 𝒟
```

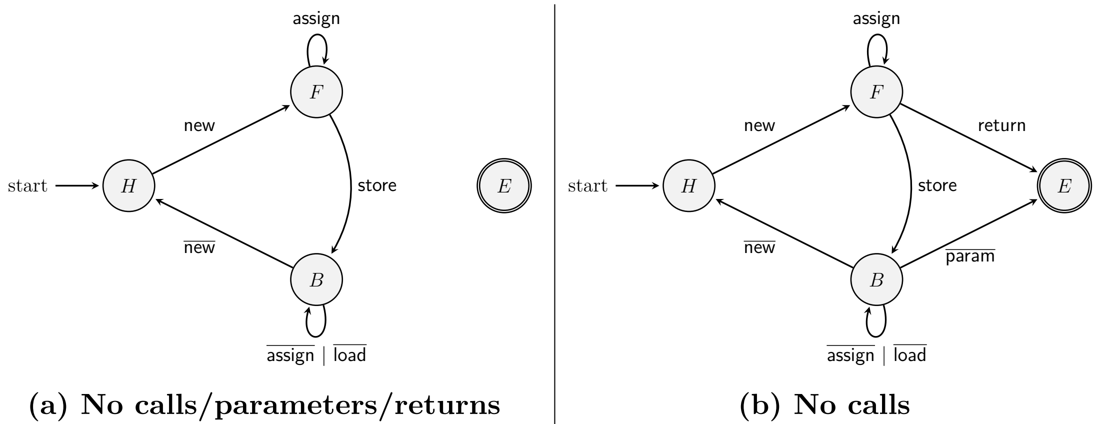
Initially, we start with a parameterless method containing no calls. Its PAG [36] can be built by the rules in Figure 4.9. Our analysis is field-insensitive, as reflected by [D-LOAD] and [D-STORE].

Figure 4.10(a) gives a DFA for tracing approximately how an object O allocated in a method flows over the PAG. There are four states: H (starting at a heap object), F (moving forwards in the PAG), B (moving backwards in the PAG), and E (exiting from the allocating method). Due to the absence of parameters and returns, no object can flow out of a method, once it is allocated inside, as indicated by the lack of transitions into the final state E.

$$\frac{\texttt{x} = \textbf{new}\ \texttt{T}\ //\ O}{O \xrightarrow{\mathsf{new}} \texttt{x} \quad \texttt{x} \xrightarrow{\overline{\mathsf{new}}} O}\ [\text{D-New}] \qquad \frac{\texttt{x} = \texttt{y}}{\texttt{y} \xrightarrow{\mathsf{assign}} \texttt{x} \quad \texttt{x} \xrightarrow{\overline{\mathsf{assign}}} \texttt{y}}\ [\text{D-Assign}]$$

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

Figure 4.9: PAG edges for a parameterless method with no calls.



(a) No calls/parameters/returns **(b) No calls**

Figure 4.10: Two intermediate DFAs for the DFA in Figure 4.12.

Let us explain the object reachability analysis supported by this DFA (Figure 4.12(a)). If the DFA starts with an object O under state H and transits to a node x under state F by following a sequence of PAG edges, then either O flows directly to x (via a new edge and possibly some assign edges) or O first flows into an access path $O'.f_1.\cdots.f_n = O$, where O', which is a locally allocated object, flows to x. If the DFA starts with an object O under state H and transits to a node y under state B, then either O is stored directly into an access path of y, i.e., $y.f_1.\cdots.f_n = O$, or O is firstly stored into an access path of some locally allocated object O' and then O' is stored into an access path of y, i.e., $y.f_1.\cdots.f_n = O'$. In this DFA, the load edges in the PAG are ignored as we track where O rather than its pointed-to objects flow to (but are used by the DFA in Figure 4.14 for computing depOnParam). In addition, the DFA also ignores the $\overline{\text{store}}$ edges in the PAG, as we assume that a method rarely contains a store and a load operating on the same field of an object (which is often accessed via its `getter` and `setter`). In the rare cases where this fails to hold, CONCH may classify a context-dependent object as being context-independent, causing the underlying pointer analysis to lose some precision.

To support parameters and return variables, we add their self-loop edges using the rules in Figure 4.11 and transform the DFA in Figure 4.10(a) into the one in Figure 4.10(b). Once an object allocated in a method flows to a parameter (suggested by $\overline{\text{param}}$) or the return variable (suggested by return) under state E, it has leaked.

$$\frac{p \text{ is a parameter}}{p \xrightarrow{\overline{\text{param}}} p} \text{[D-PARAM]} \qquad \frac{\texttt{ret} \text{ is a return variable}}{\texttt{ret} \xrightarrow{\text{return}} \texttt{ret}} \text{[D-RETURN]}$$

Figure 4.11: PAG edges for parameters and return variables.

The final DFA is presented in Figure 4.12, where the three dotted transitions are added for handling call statements. While each method has its own PAG, some summary edges are added to its PAG for its call sites to capture the inter-procedural value-flows across these call sites context-sensitively, along the three dotted transitions. The call graph is built using $\overline{\texttt{PTS}}$.

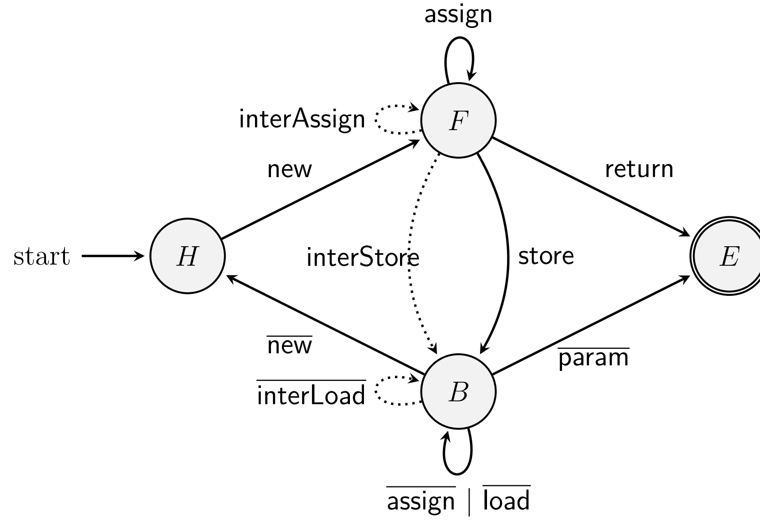


Figure 4.12: The DFA for verifying Obs 4.2.

Given a call statement $l : x = a_0.m(a_1, ..., a_r)$ contained in method M, let m' be a resolved target method (with $p_i^{m'}$ being its i-th parameter and $\texttt{ret}^{m'}$ being its return variable). Let n_1 and n_2 be two PAG nodes. We write $\langle n_1, S_1\rangle \rightarrow \langle n_2, S_2\rangle$ (known as a path edge in [65]) to indicate that node n_1 at state S_1 can reach node n_2 at state S_2. Let us write G_M as the PAG of M. There are four cases considered when m' is analyzed:

- $\langle p_i^{m'}, F\rangle \rightarrow \langle p_j^{m'}, E\rangle$: $p_i^{m'}$ is saved into some access path of $p_j^{m'}$, i.e., $p_j^{m'}.f_1.\cdots.f_n = p_i^{m'}$. Thus, we add a summary edge, $a_i \xrightarrow{\text{interStore}} a_j$ (i.e., $a_j.f = a_i$), to G_M to propagate this reachability fact inter-procedurally.

- $\langle p_i^{m'}, F\rangle \rightarrow \langle \texttt{ret}^{m'}, E\rangle$: $p_i^{m'}$ is saved into some access path of a locally allocated object O in m', i.e., $O.f_1.\cdots.f_n = p_i^{m'}$, and then O flows out of m' via its return. Thus, we add a summary edge, $a_i \xrightarrow{\textsf{interAssign}} x$, to G_M to reflect this reachability fact inter-procedurally.

- $\langle \texttt{ret}^{m'}, B\rangle \rightarrow \langle p_i^{m'}, E\rangle$: $\texttt{ret}^{m'}$ is loaded from some access path of $p_i^{m'}$, i.e., $\texttt{ret}^{m'} = p_i^{m'}.f_1.\cdots.f_n$. Thus, we add a summary edge, $x \xrightarrow{\overline{\textsf{interLoad}}} a_i$ (i.e., $x = a_i.f$), to G_M to propagate this reachability fact inter-procedurally.

- $\langle O, H\rangle \rightarrow \langle \texttt{ret}^{m'}, E\rangle$: O, which is allocated in m', flows out of m' via its return. We introduce a symbolic object Sym_l to abstract all the possible objects returned from the call site l and continue our analysis in M.

Figure 4.13 gives our IFDS-based algorithm [65] for computing leakObjects, operating on a PAG instead of a CFG representation of a program. The rules in [Seeds] inject three kinds of path edges, where the first one is for tracing leak objects while the other two are for finding summary edges (which are not injected on-demand in order to improve parallelism in a parallel implementation of our algorithm). The rules in [Propagate] perform the reachability analysis according to the DFA in Figure 4.12. Note that the three dotted transitions in the DFA are implicitly handled by the summary edges generated in [Summary]. Finally, we collect the objects that can reach the final state, E, by using [Collect].

4.4.3 Verifying Observation 4.3

In lines 8-16, we verify if an object O_l satisfies Obs 4.3(a). In the case of a positive answer, O_l is considered immediately as being context-dependent (and thus inserted into CD), since O_l has already satisfied both Obs 4.1 and Obs 4.2 at this point. Otherwise, we proceed to verify Obs 4.3(b) in lines 17-23.

$$\frac{}{\langle O_l, H\rangle \rightarrow \langle O_l, H\rangle} \quad \frac{}{\langle p_i^{m'}, F\rangle \rightarrow \langle p_i^{m'}, F\rangle} \quad \frac{}{\langle \mathtt{ret}^{m'}, B\rangle \rightarrow \langle \mathtt{ret}^{m'}, B\rangle} \qquad \text{[SEEDS]}$$

$$\frac{\langle n_1, S_1\rangle \rightarrow \langle O_l, H\rangle \quad l : n_2 = \text{new } T}{\langle n_1, S_1\rangle \rightarrow \langle n_2, F\rangle} \quad \frac{\langle n_1, S_1\rangle \rightarrow \langle n_2, F\rangle \quad l : n_3 = n_2}{\langle n_1, S_1\rangle \rightarrow \langle n_3, F\rangle}$$

$$\frac{\langle n_1, S_1\rangle \rightarrow \langle n_2, F\rangle \quad l : n_3.f = n_2}{\langle n_1, S_1\rangle \rightarrow \langle n_3, B\rangle} \quad \frac{\langle n_1, S_1\rangle \rightarrow \langle n_2, B\rangle \quad l : n_2 = n_3 \mid n_3.f}{\langle n_1, S_1\rangle \rightarrow \langle n_3, B\rangle} \qquad \text{[PROPAGATE]}$$

$$\frac{\langle n_1, S_1\rangle \rightarrow \langle n_2, B\rangle \quad l : n_2 = \text{new } T \quad S_1 \neq B}{\langle n_1, S_1\rangle \rightarrow \langle O_l, H\rangle} \quad \frac{\langle n_1, S_1\rangle \rightarrow \langle n_2, S_2\rangle \quad \langle n_2, S_2\rangle \rightarrow \langle n_3, S_3\rangle \in \mathsf{Sum}}{\langle n_1, S_1\rangle \rightarrow \langle n_3, S_3\rangle}$$

$$\frac{\langle n_1, S_1\rangle \rightarrow \langle \mathtt{ret}^{m'}, F\rangle}{\langle n_1, S_1\rangle \rightarrow \langle \mathtt{ret}^{m'}, E\rangle} \quad \frac{\langle n_1, S_1\rangle \rightarrow \langle p_i^{m'}, B\rangle}{\langle n_1, S_1\rangle \rightarrow \langle p_i^{m'}, E\rangle}$$

$$\frac{\langle p_i^{m'}, F\rangle \rightarrow \langle p_j^{m'}, E\rangle \quad p_i^{m'} \neq p_j^{m'} \quad l : x = a_0.m(a_1, \cdots, a_r) \quad O \in \overline{\mathtt{PTS}}(a_0) \quad m' = \mathtt{Dispatch}(m, O)}{\langle a_i, F\rangle \rightarrow \langle a_j, B\rangle \in \mathsf{Sum}}$$

$$\frac{\langle p_i^{m'}, F\rangle \rightarrow \langle \mathtt{ret}^{m'}, E\rangle \quad l : x = a_0.m(a_1, \cdots, a_r) \quad O \in \overline{\mathtt{PTS}}(a_0) \quad m' = \mathtt{Dispatch}(m, O)}{\langle a_i, F\rangle \rightarrow \langle x, F\rangle \in \mathsf{Sum}} \qquad \text{[SUMMARY]}$$

$$\frac{\langle \mathtt{ret}^{m'}, B\rangle \rightarrow \langle p_i^{m'}, E\rangle \quad l : x = a_0.m(a_1, \cdots, a_r) \quad O \in \overline{\mathtt{PTS}}(a_0) \quad m' = \mathtt{Dispatch}(m, O)}{\langle x, B\rangle \rightarrow \langle a_i, B\rangle \in \mathsf{Sum}}$$

$$\frac{\langle O, H\rangle \rightarrow \langle \mathtt{ret}^{m'}, F\rangle \quad l : x = a_0.m(a_1, \cdots, a_r) \quad O \in \overline{\mathtt{PTS}}(a_0) \quad m' = \mathtt{Dispatch}(m, O)}{\langle x, B\rangle \rightarrow \langle Sym_l, H\rangle \in \mathsf{Sum} \quad \langle Sym_l, H\rangle \rightarrow \langle x, F\rangle \in \mathsf{Sum}}$$

$$\frac{\langle O_l, H\rangle \rightarrow \langle p_i^{m'}, E\rangle}{O_l \in \mathsf{leakObjects}} \quad \frac{\langle O_l, H\rangle \rightarrow \langle \mathtt{ret}^{m'}, E\rangle}{O_l \in \mathsf{leakObjects}} \qquad \text{[COLLECT]}$$

Figure 4.13: Rules for computing leakObjects, i.e., the set of objects that can flow out of their containing methods for verifying Obs 4.2. $S_i \in \{H, F, B\}$, where $i \in \{1, 2, 3\}$ and Sym_l is a symbolic object abstracting all objects returned from call site l.

The key to verifying Obs 4.3(a) lies in $\mathsf{depOnParam}(y, m)$, which returns true if y is data-dependent on any parameter of method m. We have also designed and implemented an IFDS-based algorithm for computing depOnParam, in a similar manner as how we have computed leakObjects in Figure 4.13, by making use of a simpler DFA given in Figure 4.14.

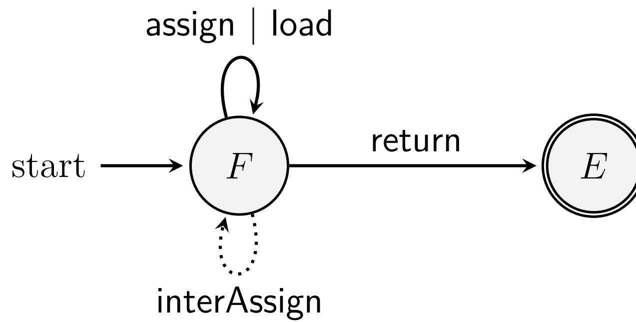


Figure 4.14: The DFA used for computing depOnParam.

This DFA has only two states, F and E, recognizing only four types of PAG edges, where interAssign is a summary edge introduced for supporting call statements. Given a call statement $x = a_0.m(a_1, \cdots, a_r)$ in method M. Let m' be a target method invoked. When $\langle p_i^{m'}, F\rangle \to \langle \mathtt{ret}^{m'}, E\rangle$ happens, $\mathtt{ret}^{m'}$ is recognized to be data-dependent on $p_i^{m'}$ (i.e., $\mathtt{ret}^{m'} = p_i^{m'}.f_1.\cdots.f_n$). Thus, we add a summary edge, $a_i \xrightarrow{\mathsf{interAssign}} x$, to the PAG of m to propagate this reachability fact inter-procedurally from the callee m' to the caller M.

Our algorithm for computing depOnParam, which proceeds forwards from method parameters, is a simplified version of the one in Figure 4.13. For [SEEDS], only the parameters need to be injected. The rules for [PROPAGATE], are similar. For [SUMMARY], we use the summary edges added as discussed above. Finally, let $\mathsf{dps}(v, m_v) = \{p_i^{m_v} \mid \langle p_i^{m_v}, F\rangle \to \langle y, F\rangle\}$, where v is a variable defined in its containing method m_v and $p_i^{m_v}$ is some (i-th) parameter of m_v. Then $\mathsf{depOnParam}(y, m)$

can be defined recursively as (by taking care of chained constructors, in practice):

$$\mathsf{depOnParam}(y,m) = \begin{cases} \mathsf{dps}(y,m_y) \neq \emptyset & \text{if } m = m_y \\ \bigvee_{p_i^{m_y} \in \mathsf{dps}(y,m_y)} \mathsf{depOnParam}(a_i, m) & \text{otherwise} \end{cases} \tag{4.2}$$

where a_i is the corresponding argument of $p_i^{m_y}$.

Finally, Obs 4.3(b) can be verified straightforwardly. At this point, CI and CD contain the sets of context-independent and context-dependent objects found so far. Let O be an object in $\mathbb{H} \setminus (\text{CI} \cup \text{CD})$. O is regarded as being context-dependent if it can point to any context-dependent object (found so far) transitively and context-independent otherwise.

4.4.4 Soundness and Time Complexity

Conch is sound as it may mis-classify some context-dependent objects as being context-independent and thus cause the underlying pointer analysis to produce over-approximated points-to information, resulting in some loss of precision.

The worst-case time complexity of Conch in analyzing a program P is linear to the number of its statements, for three reasons. First, leakObjects can be computed according to Figure 4.13 in $O(ED^3)$ [65], where E is the number of PAG edges in P, which are constructed linearly to the number of statements in P according to Figures 4.9 and 4.11, and $D = 4$ is the number of states of the DFA in Figure 4.12. Second, the first stage of Algorithm 1 (lines 2-16) runs in $O(|\mathbb{L}|)$, where $\mathbb{L}$ is the set of statements in P. Finally, the second stage of Algorithm 1 (lines 17-23) can be efficiently performed in $O(|\mathbb{H}|)$, where $\mathbb{H}$ is the set of heap objects in P.

4.5 Evaluation

We demonstrate the effectiveness of our CONCH approach by addressing the following two research questions:

- **RQ1.** Is CONCH precise and efficient?
- **RQ2.** Can CONCH speed up existing object-sensitive analysis algorithms significantly?

Implementation. We have implemented CONCH in SOOT [89], a program analysis and optimization framework for Java, on top of its context-insensitive Andersen's pointer analysis, SPARK [36] (for computing $\overline{\mathsf{PTS}}$). CONCH is implemented in about 1500 lines of Java code, which has been released as an open-source tool at http://www.cse.unsw.edu.au/~corg/conch along with a reproducible artifact in the form of a Docker image. As described in Section 4.2, CONCH aims to boost the performance of all object-sensitive pointer analysis algorithms. We report and analyze our results by applying CONCH to debloat two representative baselines, kOBJ (an object-sensitive version of SPARK) and ZIPPER [40] (the latest version b83b038, which can deliver the arguably best speedups for kOBJ among the recent algorithms for supporting selective context-sensitivity [22, 29, 40, 49, 74] in our experimental setting).

Experimental Setting. kOBJ is a standard in-house implementation of SPARK in SOOT [19]. As for ZIPPER (originally released in DOOP [72] but used here to accelerate kOBJ in SOOT), we have used an analysis setting that is as close as possible to the one used by ZIPPER in several major aspects. First, we perform an exception analysis on the fly with kOBJ as in DOOP by handling exceptions along the so-called exception-catch links [10]. Second, we use the declared type of an array element instead of `java.lang.Object` to filter type-incompatible points-to

objects. Third, we handle native code by using the summaries provided in SOOT. Fourth, we analyze a static method m by using the contexts of m's closest callers that are instance methods (on the call stack) and resolve Java reflection by using the reflection log generated by TAMIFLEX [9] as is often done in the pointer analysis literature [40,73,74,82]. Finally, objects that are instantiated from `StringBuilder` and `StringBuffer` as well as `Throwable` (including its subtypes) are distinguished per dynamic type and then analyzed context-insensitively as is done in DOOP [11] and WALA [26].

We have conducted our experiments on an Intel(R) Xeon(R) CPU E5-1660 3.2GHz machine with 256GB of RAM. We have selected a set of 12 popular Java programs, including 9 benchmarks from DaCapo [7], and 3 Java applications (`checkstyle`, `JPC` and `findbugs`). The Java library used is `jre1.6.0_45`. These are the standard Java programs that are frequently used for evaluating pointer analysis algorithms [40, 73, 74, 82]. The time budget used for running each pointer analysis on a program is set as 12 hours. The analysis time of a program is an average of three runs.

4.5.1 RQ1: Is Conch Precise and Efficient?

Given Base (a baseline pointer analysis) and Base+D (Base with its contexts debloated by CONCH), we measure the precision of CONCH in terms of precision loss incurred with respect to a given metric (Metric) when both Base and Base+D are applied to analyze the same program:

$$\Delta = \frac{\mathsf{Metric}(\mathsf{Base{+}D}) - \mathsf{Metric}(\mathsf{Base})}{\mathsf{Metric}(\mathsf{Base})} \tag{4.3}$$

where Metric(Base) and Metric(Base+D) are the metric numbers obtained by Base and Base+D, respectively. We use four common metrics for measuring the preci-

sion of a context-sensitive pointer analysis [40, 49, 73, 86]: #fail-casts, #call-edges, #poly-calls, and #reachables.

Table 4.1 gives our main results. For kOBJ, Z-kOBJ denotes the version of kOBJ with selective context-sensitivity provided by ZIPPER. All the baselines (where $k \in \{2, 3\}$) and their debloated versions are compared over the 12 Java programs.

CONCH is very precise in terms of supporting context debloating while losing negligible precision. Our approach preserves the precision of all the baselines for 10 programs consisting of the 9 DaCapo benchmarks and `findbugs`. For `checkstyle` and `JPC`, our approach suffers from an average precision loss of only less than 0.1% (across the four metrics). This happens since a `PropertyChangeEvent` object created in method `firePropertyChange(...)` of class `java.beans.PropertyChangeSupport` and a `LineReader` object created in method `load(InputStream)` of `java.util.Properties` have been misclassified as being context-independent by CONCH as they do not satisfy Obs 4.2.

CONCH is also highly efficient (as a pre-analysis). Table 4.2 gives the times spent by SPARK [36], ZIPPER [40] and CONCH. Note that both ZIPPER and CONCH are designed to be multi-threaded (with 8 threads used in our experiments). CONCH is slightly faster than ZIPPER and SPARK across all the 12 programs. On average, we have 2.6 seconds (CONCH), 10.4 seconds (ZIPPER) and 12.5 seconds (SPARK). Thus, CONCH is efficient enough for supporting context debloating.

4.5.2 RQ2: Can Conch Speed Up Baseline Analyses?

Table 4.1 also gives the analysis times of all the analyses. CONCH deliver significant speedups (geometric means) over all the baselines. For kOBJ, the speedups of 2OBJ+D over 2OBJ range from 1.7x (for `pmd`) to 9.0x (for `findbugs`) with an average of 2.6x. When $k = 3$, the speedups of 3OBJ+D over 3OBJ are more

Table 4.1: Main results. In all metrics (except for speedups), smaller is better. Given an analysis Base, Base+D is its debloated version by Conch. OoM stands for "Out of Memory".

Prog	Metrics	Classic kOBJ				Selective kOBJ			
		2OBJ	2OBJ+D	3OBJ	3OBJ+D	Z2OBJ	Z2OBJ+D	Z3OBJ	Z3OBJ+D
antlr	Time (s)	45.4	13.9 **(3.3x)**	1049.3	185.4 **(5.7x)**	20.8	7.8 **(2.7x)**	337.9	32.1 **(10.5x)**
	#fail-casts	509	509	449	449	559	559	507	507
	#call-edges	51176	51176	51149	51149	51394	51394	51367	51367
	#poly-calls	1622	1622	1615	1615	1643	1643	1636	1636
	#reachables	7804	7804	7803	7803	7842	7842	7841	7841
bloat	Time (s)	743.8	359.5 **(2.1x)**	> **12h**	4093.7	519.9	279.7 **(1.9x)**	OoM	2771.2
	#fail-casts	1314	1314	-	1221	1368	1368	-	1279
	#call-edges	56699	56699	-	56464	57192	57192	-	57036
	#poly-calls	1695	1695	-	1675	1732	1732	-	1716
	#reachables	9021	9021	-	9005	9093	9093	-	9085
chart	Time (s)	253.0	85.5 **(3.0x)**	OoM	4215.9	34.6	20.3 **(1.7x)**	573.6	178.3 **(3.2x)**
	#fail-casts	1348	1348	-	1241	1418	1418	1323	1323
	#call-edges	72457	72457	-	72023	73123	73123	72738	72738
	#poly-calls	2032	2032	-	2008	2060	2060	2040	2040
	#reachables	15143	15143	-	15113	15269	15269	15247	15247
eclipse	Time (s)	> **12h**	4113.5	OoM	OoM	2956.3	2487.7 **(1.2x)**	OoM	OoM
	#fail-casts	-	3215	-	-	3357	3357	-	-
	#call-edges	-	145763	-	-	146492	146492	-	-
	#poly-calls	-	8720	-	-	8737	8737	-	-
	#reachables	-	19916	-	-	19985	19985	-	-
fop	Time (s)	18.6	10.5 **(1.8x)**	572.3	177.8 **(3.2x)**	9.2	5.1 **(1.8x)**	113.3	28.1 **(4.0x)**
	#fail-casts	395	395	336	336	444	444	400	400
	#call-edges	34120	34120	34100	34100	34343	34343	34323	34323
	#poly-calls	808	808	802	802	832	832	826	826
	#reachables	7582	7582	7582	7582	7620	7620	7620	7620
luindex	Time (s)	19.4	8.7 **(2.2x)**	555.3	192.6 **(2.9x)**	9.4	4.9 **(1.9x)**	129.5	31.0 **(4.2x)**
	#fail-casts	394	394	340	340	448	448	398	398
	#call-edges	33495	33495	33468	33468	33728	33728	33701	33701
	#poly-calls	918	918	911	911	944	944	937	937
	#reachables	7017	7017	7016	7016	7057	7057	7056	7056
lusearch	Time (s)	30.4	11.8 **(2.6x)**	2225.7	252.1 **(8.8x)**	13.2	5.2 **(2.5x)**	622.7	39.2 **(15.9x)**
	#fail-casts	409	409	357	357	466	466	418	418
	#call-edges	36377	36377	36350	36350	36605	36605	36578	36578
	#poly-calls	1116	1116	1109	1109	1143	1143	1136	1136
	#reachables	7669	7669	7668	7668	7707	7707	7706	7706
pmd	Time (s)	41.6	24.2 **(1.7x)**	1236.1	257.0 **(4.8x)**	23.9	14.9 **(1.6x)**	344.7	52.5 **(6.6x)**
	#fail-casts	1432	1432	1367	1367	1514	1514	1461	1461
	#call-edges	59864	59864	59805	59805	60029	60029	59970	59970
	#poly-calls	2357	2357	2351	2351	2382	2382	2376	2376
	#reachables	11841	11841	11841	11841	11880	11880	11880	11880
xalan	Time (s)	565.3	298.2 **(1.9x)**	OoM	1632.1	230.7	227.5 **(1.0x)**	2487.7	1125.6 **(2.2x)**
	#fail-casts	600	600	-	546	657	657	609	609
	#call-edges	46653	46653	-	46621	46842	46842	46815	46815
	#poly-calls	1613	1613	-	1606	1636	1636	1629	1629
	#reachables	9659	9659	-	9657	9701	9701	9700	9700
checkstyle	Time (s)	1014.6	349.1 **(2.9x)**	> **12h**	OoM	404.4	236.1 **(1.7x)**	OoM	4887.4
	#fail-casts	1130	1130	-	-	1206	1206	-	1117
	#call-edges	67039	67041	-	-	67854	67854	-	66892
	#poly-calls	2210	2210	-	-	2268	2268	-	2211
	#reachables	12314	12314	-	-	12383	12383	-	12342
JPC	Time (s)	106.1	54.6 **(1.9x)**	2163.3	240.6 **(9.0x)**	34.4	26.3 **(1.3x)**	181.0	44.8 **(4.0x)**
	#fail-casts	1356	1356	1206	1206	1431	1431	1278	1278
	#call-edges	80965	80978	79297	79310	81616	81629	79932	79945
	#poly-calls	4263	4264	4127	4128	4324	4325	4187	4188
	#reachables	15508	15508	15161	15161	15582	15582	15232	15232
findbugs	Time (s)	1629.6	180.1 **(9.0x)**	OoM	936.3	131.3	50.3 **(2.6x)**	1890.0	186.8 **(10.1x)**
	#fail-casts	2072	2072	-	1696	2144	2144	1956	1956
	#call-edges	87915	87915	-	86993	88567	88567	87741	87741
	#poly-calls	3655	3655	-	3621	3670	3670	3643	3643
	#reachables	16266	16266	-	16219	16315	16315	16287	16287

Table 4.2: Times spent by pre-analyses in seconds.

	antlr	bloat	chart	eclipse	fop	luindex	lusearch	pmd	xalan	checkstyle	JPC	findbugs
SPARK	8.6	8.9	15.6	25.9	7.6	6.9	7.8	11.7	8.7	14.0	17.4	16.7
ZIPPER	4.6	6.4	16.4	25.5	4.0	3.7	4.3	9.5	10.2	14.5	9.8	16.2
CONCH	1.7	2.0	3.2	6.6	1.5	1.4	1.4	2.2	2.5	2.9	2.6	3.1

impressive, ranging from 2.9x (for `luindex`) to 9.0x (for `JPC`) with an average of 5.2x. For ZIPPER, the speedups of Z2OBJ+D over Z-2OBJ range from 1.0x (for `xalan`) to 2.7x (for `antlr`) with an average of 1.8x. When $k = 3$, the speedups of Z3OBJ+D over Z-3OBJ are also more impressive, ranging from 2.2x (for `xalan`) to 15.9x (for `lusearch`) with an average of 5.6x.

These results suggest that the speedups delivered by CONCH increase as k increases, implying that CONCH can help all the baselines improve their scalability. In particular, 2OBJ+D scales one more benchmark, i.e., `eclipse` than 2OBJ, 3OBJ+D can scale 4 more benchmarks (`bloat`, `chart`, `xalan`, and `findbugs`) than 3OBJ, and Z3OBJ+D can scale 2 more benchmarks (`bloat` and `checkstyle`) than Z-3OBJ. In general, an analysis may be unscalable due to running either out of memory ("OoM") or the time budget ("> **12h**").

Therefore, CONCH can accelerate existing object-sensitive pointer analyses significantly with negligible loss in precision. These include not only kOBJ (the standard algorithm) but also its variants enabled by, e.g., ZIPPER [40] (one recent attempt on applying selective context-sensitivity to improve the performance of kOBJ).

Below we analyze in detail why context debloating can enable baseline analyses, kOBJ and Z-kOBJ, to improve their efficiency and scalability (as reported in Table 4.1).

Figure 4.15 depicts the percentage distribution of context-dependent objects and context-independent objects classified by CONCH. CONCH has successfully identified a large percentage of context-independent objects in all the programs,

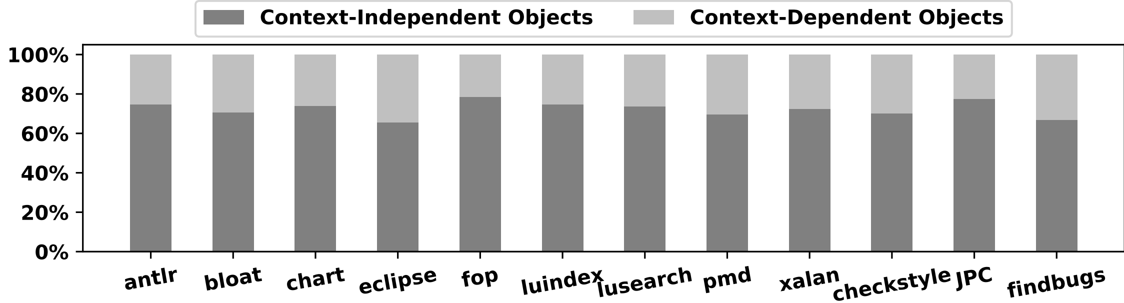


Figure 4.15: Percentage distribution of the two types of objects.

Table 4.3: Average number of contexts analyzed for a method by kOBJ, kOBJ+D, ZkOBJ and ZkOBJ+D, where $k \in \{2, 3\}$.

	antlr	bloat	chart	eclipse	fop	luindex	lusearch	pmd	xalan	checkstyle	JPC	findbugs
2OBJ	27.1	30.3	36.5	-	15.6	17.1	20.0	17.3	50.8	66.4	24.7	37.9
2OBJ+D	13.1	18.3	20.8	31.1	9.2	9.9	10.3	10.3	31.6	41.4	16.0	18.8
Z2OBJ	8.1	14.3	6.9	14.8	4.9	5.6	6.0	6.1	15.3	18.7	7.4	9.6
Z2OBJ+D	4.8	8.7	5.2	12.3	3.6	4.0	4.0	4.4	11.8	14.4	6.1	7.3
3OBJ	99.8	-	-	-	53.0	58.1	91.5	53.5	-	-	87.2	-
3OBJ+D	24.6	39.0	78.9	-	19.1	21.3	22.2	18.7	61.6	-	26.1	29.5
Z3OBJ	26.5	-	21.7	-	14.3	16.7	23.5	17.6	60.7	-	14.5	25.7
Z3OBJ+D	8.2	17.3	12.5	-	6.4	7.1	7.3	7.1	22.6	57.5	7.7	10.8

ranging from 65.6% (in `eclipse`) to 78.7% (in `fop`) with an average of 72.4%. Thus, a large amount of precision-irrelevant contexts has been eliminated via context debloating.

Table 4.3 compares the baseline analyses (i.e., kOBJ and Z-kOBJ) and their debloated counterparts (i.e., kOBJ+D and ZkOBJ+D) in terms of the average number of contexts analyzed for a method, where $k \in \{2, 3\}$. The debloated analyses have achieved a substantial reduction in terms of this important metric across all the programs, providing the reasons behind the improved efficiency and scalability via context debloating.

Finally, we can also understand the effectiveness of Conch from a substantial reduction it has achieved in the number of context-sensitive facts inferred. In Table 4.4, #cs-gpts, #cs-pts and #cs-fpts represent the numbers of context-sensitive objects pointed by global variables (i.e., static fields), local variables and instance

Table 4.4: Context-sensitive facts. For all the metrics, smaller is better.

Prog	Metrics	Classic kOBJ				Selective kOBJ			
		2OBJ	2OBJ+D	3OBJ	3OBJ+D	Z2OBJ	Z2OBJ+D	Z3OBJ	Z3OBJ+D
antlr	#cs-gpts	4.9K	2.1K	12.1K	2.5K	5.7K	2.3K	17.6K	2.7K
	#cs-pts	19.8M	3.6M	228.8M	32.1M	18.6M	3.3M	205.1M	10.4M
	#cs-fpts	0.6M	0.1M	13.6M	6.3M	0.6M	0.1M	13.7M	6.3M
	#cs-calls	5.4M	1.3M	87.5M	22.7M	1.9M	0.5M	22.7M	1.1M
	Total	25.8M	5.1M	329.8M	61.1M	21.1M	3.9M	241.6M	17.8M
bloat	#cs-gpts	3.1K	1.9K	-	2.3K	3.9K	2.0K	-	2.4K
	#cs-pts	159.8M	68.0M	-	325.0M	140.9M	53.6M	-	235.0M
	#cs-fpts	5.7M	4.6M	-	28.8M	6.9M	4.6M	-	28.0M
	#cs-calls	47.1M	20.9M	-	112.0M	38.2M	16.4M	-	74.0M
	Total	212.7M	93.5M	-	465.8M	186.0M	74.5M	-	336.9M
chart	#cs-gpts	12.5K	6.9K	-	11.3K	10.1K	5.5K	24.6K	6.9K
	#cs-pts	56.9M	20.8M	-	944.2M	16.2M	6.9M	166.8M	55.1M
	#cs-fpts	1.1M	0.4M	-	19.6M	0.7M	0.3M	21.7M	14.0M
	#cs-calls	20.0M	8.5M	-	332.8M	2.5M	1.4M	26.7M	10.1M
	Total	78.0M	29.7M	-	1296.6M	19.5M	8.6M	215.3M	79.1M
eclipse	#cs-gpts	-	7.8K	-	-	21.9K	8.0K	-	-
	#cs-pts	-	585.7M	-	-	601.8M	512.5M	-	-
	#cs-fpts	-	12.8M	-	-	16.7M	13.5M	-	-
	#cs-calls	-	345.2M	-	-	161.2M	147.3M	-	-
	Total	-	943.7M	-	-	779.7M	673.4M	-	-
fop	#cs-gpts	2.9K	1.8K	4.3K	2.0K	3.4K	1.9K	9.1K	2.2K
	#cs-pts	4.1M	1.2M	67.8M	27.5M	3.7M	1.1M	47.0M	7.9M
	#cs-fpts	0.2M	71.4K	8.0M	5.8M	0.2M	76.4K	8.2M	6.2M
	#cs-calls	1.3M	0.5M	31.0M	20.0M	0.5M	0.2M	5.1M	0.7M
	Total	5.6M	1.8M	106.7M	53.2M	4.4M	1.4M	60.4M	14.8M
luindex	#cs-gpts	2.8K	1.6K	4.5K	2.0K	3.7K	1.8K	10.6K	2.2K
	#cs-pts	4.4M	1.4M	72.6M	31.4M	4.1M	1.2M	53.0M	8.5M
	#cs-fpts	0.2M	73.0K	9.0M	6.6M	0.2M	77.3K	9.0M	6.6M
	#cs-calls	1.4M	0.6M	34.1M	22.9M	0.6M	0.3M	5.6M	0.8M
	Total	6.0M	2.0M	115.6M	60.9M	4.9M	1.6M	67.7M	15.9M
lusearch	#cs-gpts	2.9K	1.6K	4.2K	1.8K	3.7K	1.8K	10.3K	2.1K
	#cs-pts	6.8M	1.6M	193.6M	37.8M	5.4M	1.4M	116.5M	10.1M
	#cs-fpts	0.2M	77.4K	11.0M	7.9M	0.2M	82.7K	10.3M	7.9M
	#cs-calls	3.1M	0.7M	149.3M	27.8M	1.1M	0.3M	41.8M	1.0M
	Total	10.1M	2.4M	353.9M	73.6M	6.7M	1.8M	168.6M	19.0M
pmd	#cs-gpts	3.4K	1.9K	5.1K	2.1K	5.3K	2.1K	21.3K	2.4K
	#cs-pts	12.7M	5.1M	142.9M	42.0M	14.9M	4.8M	171.1M	14.8M
	#cs-fpts	0.6M	0.3M	13.1M	8.4M	1.1M	0.4M	17.0M	9.0M
	#cs-calls	3.9M	2.0M	56.8M	29.1M	2.2M	1.0M	17.4M	1.9M
	Total	17.2M	7.3M	212.8M	79.6M	18.2M	6.2M	205.5M	25.7M
xalan	#cs-gpts	4.9K	2.9K	-	3.2K	4.2K	2.8K	10.0K	3.2K
	#cs-pts	160.4M	49.0M	-	161.0M	51.1M	41.5M	517.5M	123.6M
	#cs-fpts	6.3M	4.3M	-	15.7M	5.4M	4.5M	33.1M	16.0M
	#cs-calls	49.6M	21.6M	-	103.4M	14.6M	13.7M	86.0M	52.8M
	Total	216.3M	74.9M	-	280.2M	71.2M	59.7M	636.6M	192.3M
checkstyle	#cs-gpts	7.7K	3.5K	-	-	10.8K	4.3K	-	5.2K
	#cs-pts	166.2M	44.7M	-	-	130.8M	38.3M	-	353.9M
	#cs-fpts	1.5M	0.4M	-	-	2.8M	0.6M	-	141.6M
	#cs-calls	86.5M	23.2M	-	-	24.1M	9.0M	-	79.8M
	Total	254.2M	68.3M	-	-	157.6M	47.9M	-	575.3M
JPC	#cs-gpts	7.3K	4.1K	21.3K	5.7K	7.0K	3.8K	16.4K	4.3K
	#cs-pts	27.8M	11.9M	606.3M	48.0M	13.2M	7.0M	67.3M	12.2M
	#cs-fpts	0.9M	0.3M	19.3M	7.2M	0.8M	0.3M	11.2M	6.7M
	#cs-calls	9.8M	5.5M	93.8M	28.8M	2.8M	2.0M	8.2M	2.0M
	Total	38.5M	17.7M	719.5M	84.1M	16.8M	9.4M	86.7M	20.8M
findbugs	#cs-gpts	34.1K	4.5K	-	6.0K	11.0K	4.5K	43.8K	5.9K
	#cs-pts	358.2M	41.2M	-	126.9M	58.6M	19.5M	553.2M	38.6M
	#cs-fpts	18.0M	1.0M	-	23.1M	5.0M	1.0M	61.1M	23.9M
	#cs-calls	147.2M	13.3M	-	84.9M	13.2M	5.8M	101.5M	5.9M
	Total	523.5M	55.5M	-	234.8M	76.8M	26.2M	715.9M	68.4M

fields, respectively, and #cs-calls represents the number of context-sensitive call edges. In general, the speedups of a pointer analysis over a baseline come from a significant reduction in the number of context-sensitive facts computed by the baseline. For example, 2OBJ+D is significantly faster than 2OBJ for `findbugs` as its number of context-sensitive facts is significantly less than 2OBJ. Similarly, Z3OBJ+D is also much faster than Z-3OBJ for `lusearch`. However, the analysis time of a pointer analysis is not linearly proportional to the number of context-sensitive facts computed [86]. Consider `xalan`. Z2OBJ+D has achieved a reduction of 16.2% over Z-2OBJ in terms of the number of facts inferred but their analysis times are comparable.

4.6 Conclusion

In this chapter, we have partially addressed the context explosion problem in kOBJ when analyzing large object-oriented programs by context debloating. Our key insight is to replace a set of two existing necessary conditions (whose verification is undecidable) by a set of three necessary conditions that can be linearly verifiable in terms of the number of statements in the program for determining the context-dependability of any object. Our evaluation shows that our new approach, CONCH, can improve significantly the efficiency and scalability of not only kOBJ but also existing approaches to selective context-sensitivity that can already accelerate the performance of kOBJ.

Chapter 5

Precision-Preserving Acceleration for k-CFA

While designing the selective approach, SELECTX [48], for accelerating k-CFA, we did not succeed in preserving the precision of k-CFA since the value flows related to call graph construction are missing in the traditional CFL-reachability formulation [76] (which SELECTX is based on). How to develop a precision-preserving selective approach to accelerate k-CFA remains to be a challenging research problem, motivating us to design P3CTX, the third fine-grained pointer analysis technique introduced in this thesis.

The rest of this chapter is organized as follows. Section 5.1 gives an overview. Section 5.2 motivates the development of $\mathcal{L}_{FCR}$ by highlighting several challenges faced in its design. Section 5.3 introduces $\mathcal{L}_{FCR}$ by explaining how we address these challenges and providing some insights in understanding its design. Section 5.4 introduces our $\mathcal{L}_{FCR}$-enabled pre-analysis for accelerating k-CFA, and Section 5.5 evaluates P3CTX. Finally, Section 5.6 concludes this chapter.

5.1 Overview

For object-oriented languages such as Java, k-callsite-sensitive pointer analysis (abbreviated to k-CFA) is either inclusion-based [3] or founded on context-free language (CFL) reachability [63].

Andersen-style inclusion-based formulation for k-CFA [31, 74, 86] has been adopted in several pointer analysis frameworks for Java, such as Doop [11], Soot [89], Wala [26], and Jchord [56]. Given a program, its statements are modeled as points-to set constraints, its methods' calling contexts (abstracted by their last k callsites) are tracked by parameterizing these constraints with context abstractions, and its call graph is often constructed on the fly in order to achieve the best precision and efficiency possible [21,36,38,44,67,73].

On the other hand, the CFL-reachability formulation for k-CFA [76] has also been used extensively in understanding and developing a wide range of pointer analysis algorithms, such as demand-driven pointer/alias analysis [69,76,78,92,96], context transformations [86], specification inference [6], library-code summarization [69,85], information flow analysis [42,51], and selective context-sensitivity [40,48]. Given a program, its points-to information is computed by solving a graph reachability problem based on a so-called *pointer assignment graph* (PAG) [36]. This consists of reasoning about the intersection of two CFLs, $L_{FC} = L_F \cap L_C$, where L_F describes field accesses as balanced parentheses and L_C enforces callsite-sensitivity by matching method calls and returns as also balanced parentheses. However, a separate algorithm (outside L_{FC}) is used for call graph construction.

Compared with Andersen-style inclusion-based formulation, this L_{FC}-based CFL-reachability formulation for specifying k-CFA suffers from two serious limitations due to the lack of a built-in call graph construction mechanism. First, k-CFA may lose precision even if it uses the most precise call graph for a program

(built in advance or on the fly). Second, any (meta) analysis that reasons about CFL-reachability in terms of L_{FC} will fail to make a connection with the value-flow paths traversed by a separate call graph construction algorithm used, and consequently, may make optimization decisions that reduce the precision of k-CFA (among others).

In this chapter, we overcome these two limitations by introducing a complete CFL-reachability formulation of k-CFA for Java with on-the-fly call graph construction being built-in. We are not aware of any earlier attempt on this in the literature. Our formulation consists of reasoning about the intersection of three CFLs, $\mathcal{L}_{FCR} = \mathcal{L}_F \cap \mathcal{L}_C \cap \mathcal{L}_R$, where $\mathcal{L}_F$ specifies not only field accesses as in L_F but also dynamic method dispatch, $\mathcal{L}_C$ enforces callsite-sensitivity exactly as in L_C, and $\mathcal{L}_R$ supports parameter passing in the presence of built-in on-the-fly call graph construction. We will discuss several challenges faced in designing $\mathcal{L}_{FCR}$ and provide some insights for understanding our formulation. Note that the Melski-Reps reduction [50] cannot be used to convert the inclusion-based formulation into $\mathcal{L}_{FCR}$ since $\mathcal{L}_{FCR}$ is the intersection of three CFLs (rather than just one single CFL).

$\mathcal{L}_{FCR}$ can be applied to all the applications that benefit from L_{FC}. By formulating k-CFA completely in terms of CFL-reachability, $\mathcal{L}_{FCR}$ can be potentially more useful than L_{FC} for several recently-studied CFL-reachability-based analyses: specification inference [6], library-code summarization [69, 85], information flow analysis [42, 51], and selective context-sensitivity [40, 48]. To demonstrate its utility, we introduce our third fine-grained pointer analysis technique, P3Ctx, the first $\mathcal{L}_{FCR}$-enabled precision-preserving pre-analysis for accelerating k-CFA for Java with selective context-sensitivity, which also serves to validate the correctness of $\mathcal{L}_{FCR}$. In contrast, a recently proposed pre-analysis, Selectx [48], will always lose precision as it is developed based on L_{FC} [76].

In summary, this chapter makes two major contributions:

- We introduce a CFL-reachability formulation $\mathcal{L}_{FCR}$ of k-CFA for object-oriented languages with on-the-fly call graph construction being built into the formulation itself.
- We introduce an $\mathcal{L}_{FCR}$-enabled precision-preserving pre-analysis for accelerating k-CFA for object-oriented languages with selective context-sensitivity. Our evaluation, which is conducted in Soot [89] using a set of 12 representative Java benchmarks and applications, shows that our pre-analysis enables k-CFA to achieve an average speedup of 3.1x while incurring negligible pre-analysis overheads (0.8 seconds on average).

5.2 Motivation

To motivate our work, we use a small program (Section 5.2.1). We start with Andersen-style inclusion-based formulation that comes with its own on-the-fly call graph construction mechanism (Section 5.2.2). We then examine the limitations of L_{FC} when such a built-in mechanism is absent (Section 5.2.3). Finally, we discuss several challenges faced in designing our new CFL-reachability formulation, $\mathcal{L}_{FCR}$, with on-the-fly call graph construction being built-in (Section 5.2.4).

5.2.1 Example

Consider a Java program in Figure 5.1. Given a class `T`, we write `T:foo()` for method `foo()` defined in `T`.

There are five classes, `A`, `B`, `C`, `D` and `O`, defined (lines 1-13). `B` and `C` are the subclasses of `A`, both overriding method `foo()` defined in `A`. Method `bar()` (lines 14-18) is a wrapper method which first stores whatever object pointed by its parameter

```
 1  class A {                        14  static void bar(A x, O o) {
 2    void foo(D p) {                15    D d = new D(); // D1
 3      Object v = p.f;              16    d.f = o;
 4    }                              17    x.foo(d); // c3
 5  }                                18  }
 6  class B extends A {              19  static void main() {
 7    void foo(D q) { }              20    O o1 = new O(); // O1
 8  }                                21    O o2 = new O(); // O2
 9  class C extends A {              22    A a = new A(); // A1
10    void foo(D r) { }              23    A b = new B(); // B1
11  }                                24    bar(a, o1); // c1
12  class D { Object f; }            25    bar(b, o2); // c2
13  class O { }                      26  }
```

Figure 5.1: A motivating example.

`o` into `D1.f` and then invokes `A:foo()` or `B:foo()`, depending on the dynamic type of the object pointed by its parameter `x`. In `main()`, four objects, `O1`, `O2`, `A1` and `B1`, are created, in which `A1` and `O1` (`B1` and `O2`) are passed into `bar()` as its first and second arguments, respectively, at callsite `c1` (`c2`).

Note that `C:foo()` may be regarded as being called conservatively in line 17 by a pointer analysis algorithm even though this cannot happen during program execution.

5.2.2 Andersen-Style Inclusion-based Formulation

According to Figure 2.3, [I-VCALL] not only discovers dynamically the target methods dispatched at a virtual callsite but also propagates iteratively the points-to information inter-procedurally across the call graph thus built on the fly.

Table 5.1 lists the points-to results computed for the program in Figure 5.1 by 2-CFA according to the rules in Figure 2.3. 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 $\texttt{PTS}(\texttt{x}, [\texttt{c1}]) = \{\langle \texttt{A1}, [\,]\rangle\}$, $\texttt{PTS}(\texttt{d}, [\texttt{c1}]) =$

Table 5.1: The points-to results for the program in Figure 5.1 computed by 2-CFA according to the rules in Figure 2.3.

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

{⟨D1, []⟩}, and PTS(D1.f, [c1]) = PTS(o, [c1]) = {⟨O1, []⟩}. Then A:foo() is found to be the target invoked by x.foo() at callsite c3 in line 17 ([I-VCall]). Thus, PTS(p, [c3, c1]) = {⟨D1, [c1]⟩} and PTS(v, [c3, c1]) = {⟨O1, []⟩}. Similarly, when bar() is analyzed under [c2], we have PTS(x, [c2]) = {⟨B1, []⟩}. Thus, x.foo() at callsite c3 is now resolved to B:foo(). Note that 2-CFA is precise enough by not resolving C:foo() as a target at callsite c3.

5.2.3 L_{FC}-based CFL-Reachability Formulation

In this traditional L_{FC}-based framework for solving k-CFA [76], a separate algorithm for call graph construction is used. Thus, for a virtual callsite, parameter passing that is prescribed by L_{FC} is disconnected both conceptually and algorithmically with the dynamic dispatch process done at the callsite. We discuss the resulting limitations by considering whether the call graph is constructed in advance and on the fly.

L_{FC} + a Pre-Built Call Graph k-CFA may lose precision even if L_{FC} uses the mostly precise pre-built call graph possible. In this case, the methods invoked at

"`x.foo(d); // c3`" (line 17) in Figure 5.1 are found to be `A:foo()` and `B:foo()`. However, due to the existence of the two L_{FC}-paths:

$$\texttt{O1} \xrightarrow{\textsf{new}} \texttt{o1} \xrightarrow[\hat{c1}]{\textsf{assign}} \texttt{o} \xrightarrow{\textsf{store}[\texttt{f}]} \texttt{d} \xrightarrow{\overline{\textsf{new}}} \texttt{D1} \xrightarrow{\textsf{new}} \texttt{d} \xrightarrow[\hat{c3}]{\textsf{assign}} \texttt{p} \xrightarrow{\textsf{load}[\texttt{f}]} \texttt{v} \tag{5.1}$$

$$\texttt{O2} \xrightarrow{\textsf{new}} \texttt{o2} \xrightarrow[\hat{c2}]{\textsf{assign}} \texttt{o} \xrightarrow{\textsf{store}[\texttt{f}]} \texttt{d} \xrightarrow{\overline{\textsf{new}}} \texttt{D1} \xrightarrow{\textsf{new}} \texttt{d} \xrightarrow[\hat{c3}]{\textsf{assign}} \texttt{p} \xrightarrow{\textsf{load}[\texttt{f}]} \texttt{v} \tag{5.2}$$

This L_{FC}-based CFL-reachability pointer analysis will conclude that `v` point to both `O1` and `O2` although `v` points to `O1` only by 2-CFA (Table 5.1), meaning that `O2` is spurious.

Why is the precision loss? In L_{FC}, parameter passing for a virtual callsite ([P-VCall]) is modeled identically as a static callsite ([P-SCall]) in the form of inter-procedural assign edges as shown in the two L_{FC}-paths given above, without being CFL-reachability-related to the receiver objects at the callsite. As a result, L_{FC} does not really understand that the first (second) L_{FC}-path above can be established only when `x` points to the receiver `A1` (`B1`) under context [`c1`] ([`c2`]).

If L_{FC} uses a less precise call graph, which is pre-built by, say, CHA [16], then `C:foo()` will also be identified as a target method at callsite `c3` (line 17). In this case, `r` will also be found to point to `D1` due to `D1` $\xrightarrow{\textsf{new}}$ `d` $\xrightarrow[\hat{c3}]{\textsf{assign}}$ `r`. However, `r`'s points-to set is empty by 2-CFA (not listed in Table 5.1).

L_{FC} + On-the-Fly Call Graph Construction In solving k-CFA with L_{FC} on-demand [69, 76, 92], the methods at a virtual callsite are dispatched only under a specific context, resulting in on-the-fly call graph construction.

Consider again "`x.foo(d); // c3`" (line 17) in Figure 5.1. We can now establish the L_{FC}-path in Equation (5.1) as before but not the one in Equation (5.2) any more, thereby concluding that `v` points to `O1` only. In the former case, we reach `d`

under context [c1] and then issue a points-to query to find what x points to under [c1]. As x is found to point to A1 (causing A:foo() to be invoked at callsite c3), we will continue traversing the remaining L_{FC}-path from d and conclude that v points to O1. In the latter case, reaching d under [c2] reveals B:foo() as the target at callsite c3 instead (as x points to B1 under [c2]), thereby causing $\xrightarrow[\hat{c3}]{\text{assign}} \texttt{p} \xrightarrow{\text{load[f]}} \texttt{v}$ not to be traversed.

While L_{FC} can be used to solve k-CFA on-demand (more precisely than if a pre-built call graph is used), some precision loss may occur when a callsite has several dispatch targets under a common calling context. Consider the following code snippet (which reuses classes A, B, and D from Figure 5.1):

```
D d = new D(); // D1
if (...)
  d.f = a = new A(); // A1
else
  d.f = b = new B(); // B1
A x = d.f;
x.foo(null); // c
```

If we ask a separate call graph construction algorithm to find on-demand the target methods at "x.foo(null); // c" under a currently given context invoking this piece of code, A:foo() and B:foo() will be returned. If we then reason about CFL-reachability with L_{FC}, we will obtain:

$$\texttt{A1} \xrightarrow{\text{new}} \texttt{a} \xrightarrow{\text{store[f]}} \texttt{d} \xrightarrow{\overline{\text{new}}} \texttt{D1} \xrightarrow{\text{new}} \texttt{d} \xrightarrow{\text{load[f]}} \texttt{x} \xrightarrow[\hat{\texttt{c}}]{\text{assign}} \texttt{this}^{\texttt{A:foo()}} \tag{5.3}$$

$$\texttt{B1} \xrightarrow{\text{new}} \texttt{b} \xrightarrow{\text{store[f]}} \texttt{d} \xrightarrow{\overline{\text{new}}} \texttt{D1} \xrightarrow{\text{new}} \texttt{d} \xrightarrow{\text{load[f]}} \texttt{x} \xrightarrow[\hat{\texttt{c}}]{\text{assign}} \texttt{this}^{\texttt{A:foo()}} \tag{5.4}$$

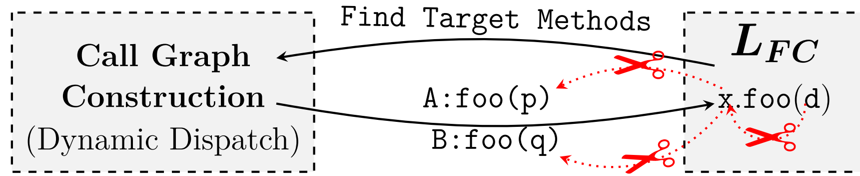


Figure 5.2: Disconnection in the value flows between parameter passing in L_{FC} and dynamic dispatch at a virtual callsite.

Both A1 and B1 will flow to $\texttt{this}^{\texttt{A:foo}}$ although B1 is spurious by [I-VCall] (since it cannot be a receiver of `A:foo()`).

We see a loss of precision at such a virtual callsite since L_{FC} does not handle its receiver variable differently from its other arguments ([P-VCall] in Figure 2.5) unlike Andersen-style inclusion-based formulation ([I-VCall]). Removing spurious receiver objects such as B1 as discussed above with brute force, which is specified formally by neither L_{FC} nor the call graph construction algorithm used, is ad hoc. Indeed, the L_{FC}-based on-demand algorithm for solving k-CFA (released by the authors of L_{FC} [76] in Soot [89] and used by many others [69, 91] in the past 15 years) suffers still from this problem.

L_{FC} + Separate Call Graph Construction L_{FC} relies a separate algorithm for call graph construction. In addition to cause k-CFA to lose precision as discussed above, L_{FC} suffers from another limitation, as it fails to capture all the value-flow paths traversed during the analysis (regardless of whether the call graph is built in advance or on the fly).

Consider again how "`x.foo(d)`" (line 17) in Figure 5.1 is analyzed. To pass `d` to its corresponding parameter for every method invoked at the callsite according to Andersen-style inclusion-based formulation ([I-VCall]), we must first find the virtual methods dispatched on the receiver objects pointed by `x` and then perform the actual parameter passing (from `d` to `p` if `A:foo()` is dispatched or `d` to `q` if

`B:foo()` is dispatched). However, with L_{FC}, as illustrated in Figure 5.2, parameter passing (realized by inter-procedural assign edges ([P-VCALL] in Figure 2.5)) is both conceptually and algorithmically disconnected (✂) with dynamic dispatch at the callsite, without being CFL-reachability-related to its receiver objects.

As L_{FC} is incomplete, any (meta) analysis that reasons about CFL-reachability in terms of L_{FC} may make some optimization decisions that reduce the precision of k-CFA (among others). For example, a recent pre-analysis [48] that is developed based on L_{FC} for accelerating k-CFA with selective context-sensitivity will cause k-CFA to lose precision.

5.2.4 $\mathcal{L}_{FCR}$: Necessity and Challenges

We introduce $\mathcal{L}_{FCR}$ as the first complete CFL-reachability formulation of k-CFA with built-in call graph construction, overcoming the limitations of $\mathcal{L}_{FC}$ discussed above. In particular, this enables us to develop the first precision-preserving pre-analysis for accelerating k-CFA with selective context-sensitivity.

We have designed $\mathcal{L}_{FCR}$ as the intersection of three CFLs to facilitate $\mathcal{L}_{FCR}$-enabled CFL-reachability analyses. By noting that ∞-CFA is undecidable and thus both the L_{FC}- and $\mathcal{L}_{FCR}$-path problems are undecidable (implying that either is a context-sensitive language rather than a single CFL) [64], we list below three challenges in handling virtual callsites:

- **CHL1**. How do we handle parameter passing for a receiver variable at a virtual callsite precisely by avoiding the precision loss illustrated by the code given in Sec. 5.2.3?

- **CHL2**. How do we perform dynamic dispatch at a virtual callsite during parameter passing for its non-receiver arguments by reasoning about CFL-reachability

from these arguments to its receiver variable (as such a path does not exist in the L_{FC}-based formulation [76])?

- **CHL3**. How do we ensure that parameter passing at a virtual callsite (with dynamic dispatch performed by $\mathcal{L}_{FCR}$ itself) happens correctly with respect to k-CFA [71]?

5.3 $\mathcal{L}_{FCR}$: Design and Insights

When solving a CFL-reachability problem with a CFL, the CFL and its underlying graph structure are always inter-related and carefully designed together. To break their cyclic dependencies, we first describe a new `PAG` representation used for representing the value flows in a program (Section 5.3.1). We then formalize $\mathcal{L}_{FCR}$ by explaining how we address the three challenges (**CHL1** – **CHL3**) and providing some insights in understanding its design (Section 5.3.2).

5.3.1 Pointer Assignment Graph

Figure 5.3 gives the rules for building the `PAG` for a program. As in the case of L_{FC} (Figure 2.5), the inverse of a PAG edge is not given explicitly. For each `PAG` edge $x \xrightarrow[c]{\ell} y$, its inverse edge is defined as $y \xrightarrow[\bar{c}]{\bar{\ell}} x$ as in L_{FC} exactly (Section 5.2.3), except that a below-edge label can also be $\boxed{\hat{c}}$ or $\boxed{\check{c}}$ (in addition to $\hat{c}$ and $\check{c}$), in which case, $\overline{\boxed{\hat{c}}} = \boxed{\check{c}}$ and $\overline{\boxed{\check{c}}} = \boxed{\hat{c}}$, where c signifies a callsite. To trigger dynamic dispatch, an edge with a boxed below-edge label also represents conceptually a new kind of inter-procedural value-flow entering into (marked by $\boxed{\hat{c}}$) or exiting from (marked by $\boxed{\check{c}}$) from a method invoked at c.

Our PAG representation (for supporting $\mathcal{L}_{FCR}$) differs from that for supporting L_{FC} (Figure 2.5) in how virtual callsites are handled. Therefore, [C-Assign], [C-

$$\frac{\texttt{x} = \textbf{new}\ \texttt{T}\ //\ \texttt{O}}{O \xrightarrow{\mathsf{new}[\texttt{T}]} \texttt{x}}\ \text{[C-New]} \qquad \frac{\texttt{x} = \texttt{y}}{\texttt{y} \xrightarrow{\mathsf{assign}} \texttt{x}}\ \text{[C-Assign]}$$

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

$$\frac{\texttt{x} = \texttt{m}(a_1, \ldots, a_n)\ //\ \texttt{c}}{\forall\ i \in [1, n] : a_i \xrightarrow[\hat{c}]{\mathsf{assign}} p_i^{\texttt{m}} \quad \texttt{ret}^{\texttt{m}} \xrightarrow[\check{c}]{\mathsf{assign}} \texttt{x}}\ \text{[C-SCall]}$$

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

$$\frac{\texttt{M}\ \text{is an instance method}}{\texttt{this}^{\texttt{M}} \xrightarrow{\mathsf{load}[i]} p_i^{\texttt{M}}}\ \text{[C-Param]} \qquad \frac{\texttt{M}\ \text{is an instance method}}{\texttt{ret}^{\texttt{M}} \xrightarrow{\mathsf{store}[0]} \texttt{this}^{\texttt{M}}}\ \text{[C-Ret]}$$

Figure 5.3: Rules for building the PAG required by $\mathcal{L}_{FCR}$.

Load], and [C-Store] are identical to [P-Assign], [P-Load], and [P-Store], respectively. In addition, [C-SCall], which behaves also identically as [P-SCall], handles parameter passing at a static callsite c simply as assignments in terms of inter-procedural assign edges, with its entry (exit) context being $\hat{c}$ ($\check{c}$).

[C-New], [C-VCall], [C-Param], and [C-Ret] build the PAG edges together to enable $\mathcal{L}_{FCR}$ to perform its own on-the-fly call graph construction at virtual callsites. In [C-New] (unlike [P-New] in Figure 2.5), $O \xrightarrow{\mathsf{new}[\texttt{t}]} \texttt{x}$ encodes the dynamic type t of O for supporting dynamic dispatch on O.

Given an instance method M (with $\texttt{this}^{\texttt{M}}$ denoting its this variable), its i-th (non-this) parameter $p_i^{\texttt{M}}$ (where i starts from 1) and its return variable $\texttt{ret}^{\texttt{M}}$ are modeled as special fields of $\texttt{this}^{\texttt{M}}$ (identified by their offsets). Thus, we can initialize $p_i^{\texttt{M}}$ with a load $\texttt{this}^{\texttt{M}} \xrightarrow{\mathsf{load}[i]} p_i^{\texttt{M}}$ ([C-Param]) and $\texttt{this}^{\texttt{M}}.0$ with a store $\texttt{ret}^{\texttt{M}} \xrightarrow{\mathsf{store}[0]} \texttt{this}^{\texttt{M}}$ ([C-Ret]).

[C-VCall] differs fundamentally from [P-VCall] (Figure 2.5) in handling a virtual callsite "$\texttt{x} = \texttt{r.m}(a_1, \ldots, a_n)$ // c". For convenience, we make use of $\texttt{r\#c}$ (a temporary) to identify uniquely this particular occurrence of $\texttt{r}$ at this callsite (as $\texttt{r}$ may also be used as a receiver variable elsewhere). When building the PAG, we over-approximate the set of target methods invoked at each callsite (and consequently, the call graph for the program) by using class hierarchy analysis (CHA) [16] (due to $\texttt{t} <: \texttt{TypeContg}(\texttt{r})$ and $\texttt{m}' = \texttt{Dispatch}(\texttt{m}, \texttt{t})$). Note that $\mathcal{L}_{FCR}$ will perform its on-the-fly call graph construction over such an over-approximated PAG. To pass a non-receiver argument a_i ($1 \leqslant i \leqslant n$) to its parameter $p_i^{\texttt{m}'}$ of a target method, $\texttt{m}'$, we make use of a store $a_i \xrightarrow[\hat{\boxed{c}}]{\mathsf{store}[i]} \texttt{r}$ introduced in this rule and a matching load $\texttt{this}^{\texttt{m}'} \xrightarrow{\mathsf{load}[i]} p_i^{\texttt{m}'}$ ([C-Param]). By performing CFL-reachability under $\mathcal{L}_{FCR}$, traversing such an edge will trigger a search for the dynamic type of every receiver object pointed to by $\texttt{r}$ (marked by $\hat{\boxed{c}}$). Encountering $\texttt{r} \xrightarrow[\check{\boxed{c}}]{\mathsf{assign}} \texttt{r\#c} \xrightarrow[\hat{c}]{\mathsf{dispatch}[\texttt{t}]} \texttt{this}^{\texttt{m}'}$ (introduced in this rule) later signifies that one such a dynamic type $\texttt{t}$ has been found (marked by $\check{\boxed{c}}$) so that $\texttt{m}'$, where $\texttt{m}' = \texttt{Dispatch}(\texttt{m}, \texttt{t})$, can be dispatched with $\hat{c}$ as its entry context (as desired), where c is recovered from $\boxed{c}$. By definition, a $\mathsf{dispatch}$ edge also serves as an assign edge. As for the receiver variable $\texttt{r}$, we use $\texttt{r} \xrightarrow{\mathsf{assign}} \texttt{r}\#c$ (without a need for searching itself). Finally, we assign $\texttt{ret}^{\texttt{m}'}$ (saved earlier in $\texttt{this}^{\texttt{m}}.0$) ([C-Ret]) to $\texttt{x}$ via a load $a_0 \xrightarrow[\check{\boxed{c}}]{\mathsf{load}[0]} \texttt{x}$ (introduced in this rule), where $\check{\boxed{c}}$ signifies the end of dynamic dispatch on $\texttt{r}$ on exit from callsite c.

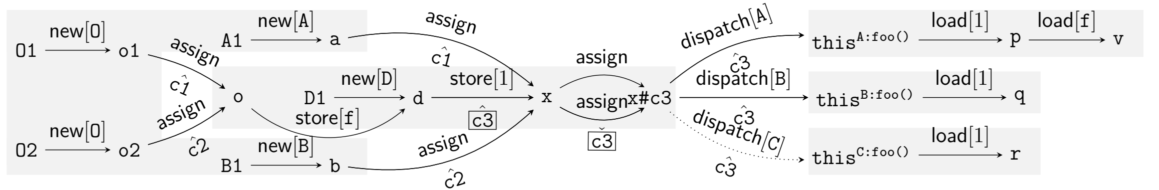


Figure 5.4: The PAG constructed for the program given in Figure 5.1.

Figure 5.4 depicts the PAG for the program in Figure 5.1.

5.3.2 $\mathcal{L}_{FCR}$

We express $\mathcal{L}_{FCR}$ as the intersection of three CFLs, $\mathcal{L}_{FCR} = \mathcal{L}_F \cap \mathcal{L}_C \cap \mathcal{L}_R$, with each specifying a different aspect of k-CFA. In Section 5.3.2.1, we first introduce $\mathcal{L}_F$, which describes field accesses and dynamic dispatch, by addressing **CHL1** and **CHL2** (Section 5.3.2.1). $\mathcal{L}_C$, which is the same L_C (Equation (2.10)) using the below-edge terminals $\hat{c}$ and $\check{c}$, enforces callsite-sensitivity in the standard manner. In Section 5.3.2.2, we introduce $\mathcal{L}_R$ (defined over the terminals $\hat{c}$ and $\check{c}$, $\hat{\boxed{c}}$, and $\check{\boxed{c}}$), which supports parameter passing in the presence of on-the-fly call graph construction, by addressing **CHL3**.

We shall speak of an $\mathcal{L}_{FC}$-path as we do for L_{FC}-path (Section 5.2.3). Similarly, we shall also speak of an $\mathcal{L}_{FCR}$-path (p such that $\mathcal{L}_F(p) \in \mathcal{L}_F$, $\mathcal{L}_C(p) \in \mathcal{L}_C$, and $\mathcal{L}_R(p) \in \mathcal{L}_R$).

Given a virtual callsite $\texttt{x} = \texttt{r}.\texttt{m}(a_1, \ldots, a_n)$, the basic idea in addressing **CHL1** – **CHL3** (facilitated by the PAG designed in Figure 5.3) is to first store a_i into $\texttt{r}.i$ (at its special field i), then discover the dynamic type $\texttt{t}$ of every receiver object pointed to by $\texttt{r}$ and propagate $\texttt{t}$ to this callsite, where $\texttt{m}' = \texttt{Dispatch}(\texttt{m}, \texttt{t})$, and finally, assign $\texttt{this}^{\texttt{m}'}.i$ to $p_i^{\texttt{m}'}$. As discussed earlier, method returns are handled similarly.

Let L_{FC}^{dd} be the demand-driven formulation of L_{FC} for solving k-CFA by using a separate algorithm for on-the-fly construction under the assumption that it can handle parameter passing for receiver variables correctly (without suffering from the precision loss discussed in Sec. 5.2.3). $\mathcal{L}_{FCR}$ is designed to solve k-CFA with CFL-reachability fully by computing the same points-to information as L_{FC}^{dd}. Consider our motivating example in Figure 5.1. As discussed in Sec. 5.2.3, L_{FC}^{dd} will assert that v points to O1 only due to the existence of the L_{FC}-path in Equation (5.1).

With $\mathcal{L}_{FCR}$, we will reach the same conclusion according to the $\mathcal{L}_{FCR}$-path:

$$
\begin{array}{l}
\color{red}{\texttt{O1}} \xrightarrow{\textsf{new[O]}} \texttt{o1} \xrightarrow[\hat{\texttt{c1}}]{\textsf{assign}} \texttt{o} \xrightarrow{\textsf{store[f]}} \texttt{d} \xrightarrow{\overline{\textsf{new[D]}}} \color{red}{\texttt{D1}} \xrightarrow{\textsf{new[D]}} \texttt{d} \\
\xrightarrow[\boxed{\hat{\texttt{c3}}}]{\textsf{store[1]}} \texttt{x} \xrightarrow[\check{\texttt{c1}}]{\overline{\textsf{assign}}} \texttt{a} \xrightarrow{\overline{\textsf{new[A]}}} \color{red}{\texttt{A1}} \xrightarrow{\textsf{new[A]}} \texttt{a} \xrightarrow[\hat{\texttt{c1}}]{\textsf{assign}} \texttt{x} \xrightarrow[\boxed{\check{\texttt{c3}}}]{\textsf{assign}} \texttt{x\#c3} \\
\xrightarrow[\hat{\texttt{c3}}]{\textsf{dispatch[A]}} \texttt{this}^{\texttt{A:foo()}} \xrightarrow{\textsf{load[1]}} \texttt{p} \xrightarrow{\textsf{load[f]}} \texttt{v}
\end{array}
\tag{5.5}
$$

O1 can flow to v only when `bar()` is called at callsite c1. Between $\boxed{\hat{\texttt{c3}}}$ and $\boxed{\check{\texttt{c3}}}$, x points to A1 of type A under context [c1]. We then dispatch `A:foo()` via `x#c3` $\xrightarrow[\hat{\texttt{c3}}]{\textsf{dispatch[A]}} \texttt{this}^{\texttt{A:foo()}}$ so that d can be passed to p under [c3, c1], with c3 recovered from $\boxed{\texttt{c3}}$. While L^{dd}_{FC} [76] uses [c3] for passing d to p (Equation (5.1)), $\mathcal{L}_{FCR}$ uses [c3, c1] more precisely to indicate that this happens only when x points to A1 under [c1].

5.3.2.1 The $\mathcal{L}_F$ Language

This CFL describes not only field(-sensitive) accesses as balanced parentheses as in L_F given in Equation (2.8) but also dynamic dispatch in the language itself:

$$
\begin{array}{rcl}
\textsf{flowsto} & \longrightarrow & \textsf{new[t]}\ (\textsf{flows} \mid \textsf{dispatch[t]})^* \\
\textsf{flows} & \longrightarrow & \textsf{assign} \mid \textsf{store[f]}\ \mathit{alias}\ \textsf{load[f]} \\
\mathit{alias} & \longrightarrow & \overline{\textsf{flowsto}}\ \textsf{flowsto} \\
\overline{\textsf{flowsto}} & \longrightarrow & (\overline{\textsf{dispatch[t]}} \mid \overline{\textsf{flows}})^*\ \overline{\textsf{new[t]}} \\
\overline{\textsf{flows}} & \longrightarrow & \overline{\textsf{assign}} \mid \overline{\textsf{load[f]}}\ \mathit{alias}\ \overline{\textsf{store[f]}}
\end{array}
\tag{5.6}
$$

where the set of terminals includes all the above-edge labels (of both regular and their inverse edges) in the PAG.

In designing $\mathcal{L}_F$, we have extended L_F [76, 78] by preserving its capability in handling field accesses as balanced parentheses and adding a new capability in supporting dynamic dispatch, and consequently, on-the-fly call graph construction. The key novelty in addressing **CHL1** and **CHL2** is to propagate the type information of a receiver object to where dynamic dispatch is triggered by parameter passing.

CHL1 In addressing this challenge concerning parameter passing at a virtual callsite, we must distinguish its receiver variable from its other arguments to ensure that the receiver objects pointed by the receiver variable can only be passed to the `this` variable of a method that can be dispatched on these receiver objects. Consider `x.foo(null)` in the code snippet discussed in Sec. 5.2.3, where `x` may point to both `A1` and `B1`. The traditional L_{FC}-based formulation that uses a separate algorithm for call graph construction (Figure 2.5) will end up passing both `A1` and `B1` to $\texttt{this}^{\texttt{A:foo()}}$ due to the existence of the two L_{FC}-paths given in Equation (5.3) and Equation (5.4), although `B1` is spurious (Figure 2.3).

In $\mathcal{L}_F$, we make explicit the dynamic types of objects in the four kinds of terminals, $\mathsf{new}[\mathtt{t}]$, $\overline{\mathsf{new}[\mathtt{t}]}$, $\mathsf{dispatch}[\mathtt{t}]$, and $\overline{\mathsf{dispatch}[\mathtt{t}]}$. During a $\mathsf{flowsto}$ ($\overline{\mathsf{flowsto}}$) traversal, we require the type information in $\mathsf{dispatch}[\mathtt{t}]$ ($\overline{\mathsf{dispatch}[\mathtt{t}]}$) to be consistent with that in its corresponding $\mathsf{new}[\mathtt{t}]$ ($\overline{\mathsf{new}[\mathtt{t}]}$). As a result, the two L_{FC}-paths in Equation (5.3) and Equation (5.4) become:

$$\mathtt{A1} \xrightarrow{\mathsf{new}[\mathtt{A}]} \mathtt{a} \xrightarrow{\mathsf{store}[\mathtt{f}]} \mathtt{d} \xrightarrow{\overline{\mathsf{new}[\mathtt{D}]}} \mathtt{D1} \xrightarrow{\mathsf{new}[\mathtt{D}]} \mathtt{d} \xrightarrow{\mathsf{load}[\mathtt{f}]} \mathtt{x} \xrightarrow{\mathsf{assign}} \mathtt{x\#c} \xrightarrow[\hat{\mathtt{c}}]{\mathsf{dispatch}[\mathtt{A}]} \texttt{this}^{\texttt{A:foo()}}$$

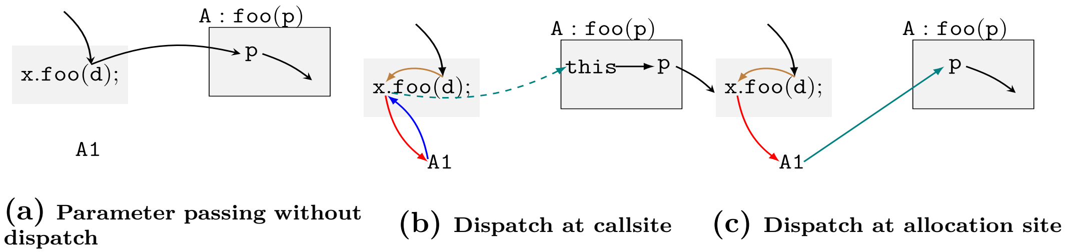
$$\mathtt{B1} \xrightarrow{\mathsf{new}[\mathtt{B}]} \mathtt{b} \xrightarrow{\mathsf{store}[\mathtt{f}]} \mathtt{d} \xrightarrow{\overline{\mathsf{new}[\mathtt{D}]}} \mathtt{D1} \xrightarrow{\mathsf{new}[\mathtt{D}]} \mathtt{d} \xrightarrow{\mathsf{load}[\mathtt{f}]} \mathtt{x} \xrightarrow{\mathsf{assign}} \mathtt{x\#c} \xrightarrow[\hat{\mathtt{c}}]{\mathsf{dispatch}[\mathtt{A}]} \texttt{this}^{\texttt{A:foo()}}$$

The first is an $\mathcal{L}_F$-path since $\mathsf{new}[\mathtt{A}]\ \mathsf{flows}^*\ \mathsf{dispatch}[\mathtt{A}] \in \mathcal{L}_F$ but the second is not since $\mathsf{new}[\mathtt{B}]\ \mathsf{flows}^*\ \mathsf{dispatch}[\mathtt{A}] \notin \mathcal{L}_F$. Thus, `B1` cannot flow to $\texttt{this}^{\texttt{A:foo()}}$ spuriously.

Lemma 5.1 *Consider a virtual callsite* $\texttt{x} = \texttt{r.m}(a_1, \ldots, a_n)$*. In* $\mathcal{L}_F$*, every object pointed to by* `r` *flows only to the* ***this*** *variable of a method that can be dispatched on the object.*

Proof Sketch. $\mathcal{L}_F$ requires the type of an object pointed to by `r` to be used for dynamic dispatch at the virtual callsite. □

CHL2 In addressing this second challenge, we must decide how to trigger dynamic dispatch during parameter passing at a virtual callsite. Figure 5.5 illustrates three different approaches for handling `x.foo(d)` in Figure 5.1.



(a) **Parameter passing without dispatch** (b) **Dispatch at callsite** (c) **Dispatch at allocation site**

Figure 5.5: Three different approaches for performing dynamic dispatch at a virtual callsite during parameter passing.

As discussed in Section 5.2.3, L_{FC} [76] solves k-CFA by using a separate algorithm for call graph construction (Figure 5.5a) and may thus cause k-CFA to lose precision (either directly (Sec. 5.2.3 and Sec. 5.2.3) or when assisted by a pre-analysis [48] (illustrated in Figure 5.2 in Sec. 5.2.3)).

In $\mathcal{L}_F$, passing `d` to a parameter at `x.foo(d)` will trigger immediately a $\overline{\mathsf{flowsto}}$ traversal looking for a receiver object of `x`, as symbolized by a red arrow (→) in Figure 5.5b (for performing dynamic dispatch at this callsite) and Figure 5.5c (for performing dynamic dispatch at the allocation site of the receiver object). $\mathcal{L}_F$ adopts the callsite-based approach since the allocation-site-based approach is infeasible.

To understand our approach, we examine the $\mathcal{L}_{FCR}$-path in Equation (5.5) by focusing on its sub-path starting from argument `d` and ending at parameter `p` of `A:foo()`. By considering only its above-edge labels in $\mathcal{L}_F$, we start with a store $\texttt{d} \xrightarrow{\mathsf{store}[\mathtt{1}]} \texttt{x}$ ($\rightarrow$) to trigger a $\overline{\mathsf{flowsto}}$ traversal via $\texttt{x} \xrightarrow{\overline{\mathsf{assign}}} \texttt{a} \xrightarrow{\overline{\mathsf{new}[\mathtt{A}]}} \texttt{A1}$ ($\rightarrow$), return to `x` via $\texttt{A1} \xrightarrow{\mathsf{new}[\mathtt{A}]} \texttt{a} \xrightarrow{\mathsf{assign}} \texttt{x}$ ($\rightarrow$), and finally, dispatch at the callsite via $\texttt{x} \xrightarrow{\mathsf{assign}} \texttt{x\#c3} \xrightarrow{\mathsf{dispatch}[\mathtt{A}]} \texttt{this}^{\texttt{A:foo()}}$ ($\dashrightarrow$). If we consider now the below-edge labels of the entire path in Equation (5.5) (by ignoring $\boxed{\hat{\texttt{c3}}}$ and $\boxed{\check{\texttt{c3}}}$ for now), we find that `O1` flows to `v` under [`c3`,`c1`] (as in Equation (5.1)) and `d` flows to `p` also under [`c3`,`c1`] (indicating that this parameter passing happens only when `x` point to `A1` under [`c1`]).

Let us explain why the allocation-site-based dispatch (Figure 5.5c) is infeasible. To handle parameter passing only (without considering method returns), we need to extend $\mathcal{L}_F$ to:

$$
\begin{array}{lcl}
\mathsf{flows} & \longrightarrow & \cdots \mid \mathsf{store}[\mathtt{m}:\mathtt{i}]\ \overline{\mathsf{flowsto}}\ \mathsf{load}[\mathtt{m}:\mathtt{i}] \\
\overline{\mathsf{flows}} & \longrightarrow & \cdots \mid \overline{\mathsf{load}[\mathtt{m}:\mathtt{i}]}\ \overline{\mathsf{flowsto}}\ \overline{\mathsf{store}[\mathtt{m}:\mathtt{i}]}
\end{array}
\tag{5.7}
$$

where $\mathsf{store}[\mathtt{m}:\mathtt{i}]$ ($\mathsf{load}[\mathtt{m}:\mathtt{i}]$) is used to replace $\mathsf{store}[\mathtt{i}]$ ($\mathsf{load}[\mathtt{i}]$) in Figure 5.3 in order to encode also the signature of a method invoked at $\texttt{r.m}(a_1, \ldots, a_n)$. With this modified $\mathcal{L}_F$ language, the L_{FC}-path in Equation (5.1) becomes:

$$
\begin{array}{l}
\texttt{O1} \xrightarrow{\mathsf{new}[\mathtt{O}]} \texttt{o1} \xrightarrow[\hat{\texttt{c1}}]{\mathsf{assign}} \texttt{o} \xrightarrow{\mathsf{store}[\mathtt{f}]} \texttt{d} \xrightarrow{\overline{\mathsf{new}[\mathtt{D}]}} \texttt{D1} \xrightarrow{\mathsf{new}[\mathtt{D}]} \texttt{d} \\
\xrightarrow{\mathsf{store}[\mathtt{foo}:\mathtt{1}]} \texttt{x} \xrightarrow[\check{\texttt{c1}}]{\overline{\mathsf{assign}}} \texttt{a} \xrightarrow{\overline{\mathsf{new}[\mathtt{A}]}} \texttt{A1} \xrightarrow{\mathsf{load}[\mathtt{foo}:\mathtt{1}]} \texttt{p} \xrightarrow{\mathsf{load}[\mathtt{f}]} \texttt{v}
\end{array}
\tag{5.8}
$$

where we have $\texttt{d} \xrightarrow{\mathsf{store}[\mathtt{foo}:\mathtt{1}]} \texttt{x}$ ($\rightarrow$), $\texttt{x} \xrightarrow[\check{\texttt{c1}}]{\overline{\mathsf{assign}}} \texttt{a} \xrightarrow{\overline{\mathsf{new}[\mathtt{A}]}} \texttt{A1}$ ($\rightarrow$), and $\texttt{A1} \xrightarrow{\mathsf{load}[\mathtt{foo}:\mathtt{1}]} \texttt{p}$ ($\rightarrow$). However, `O1` flows to `v` under [ ] incorrectly (with $\hat{\texttt{c1}}$ and $\check{\texttt{c1}}$ being matched).

5.3.2.2 The $\mathcal{L}_R$ Language

Given $\mathcal{L}_F$ (defined in Equation (5.6)) and $\mathcal{L}_C = L_C$ (defined in Equation (2.10)), we have obtained a new language: $\mathcal{L}_{FC} = \mathcal{L}_F \cap \mathcal{L}_C$. However, $\mathcal{L}_{FC}$ solves k-CFA soundly but less precisely than L_{FC}^{dd} (Lemma 5.2).

We can obtain the points-to set of a variable v, $\texttt{PTS}(v, c_v)$, from $\mathcal{L}_{FC}$ as follows. Given an $\mathcal{L}_C$-path p with its label being $\mathcal{L}_C(p) = \ell_1, \ldots, \ell_n$, the inverse of p, i.e., $\overline{p}$ has the label $\mathcal{L}_C(\overline{p}) = \overline{\ell_n}, \ldots, \overline{\ell_1}$. By splitting p into a sub-path p^{ex} followed by a sub-path p^{en}, we can define ${\mathcal{L}_C}^{\mathsf{ex}}(p) = \mathcal{L}_C(p^{\mathsf{ex}})$ and ${\mathcal{L}_C}^{\mathsf{en}}(p) = \mathcal{L}_C(p^{\mathsf{en}})$, where $\mathcal{L}_C(p) = {\mathcal{L}_C}^{\mathsf{ex}}(p){\mathcal{L}_C}^{\mathsf{en}}(p)$, such that ${\mathcal{L}_C}^{\mathsf{ex}}(p)$ (${\mathcal{L}_C}^{\mathsf{en}}(p)$) is derived from exit (entry) in $\mathcal{L}_C$'s grammar (Equation (2.10)). Let $s \in \mathcal{L}_C$. Let $\mathscr{B}(s)$ return the canonical form of s with all its balanced contexts (i.e., parentheses) removed. If c is a string of exit contexts of the form $\check{c_1} \ldots \check{c_n}$, we write $\mathscr{E}(c) = [c_1, \ldots, c_n]$ to turn it into a context representation (by noting that $\mathscr{E}(\epsilon) = [\,]$).

Given an $\mathcal{L}_{FC}$-path p starting from an object O to a variable v, we can deduce the following points-to relation:

$$\langle O, \mathscr{E}(\mathscr{B}({\mathcal{L}_C}^{\mathsf{ex}}(p)))\rangle \in \texttt{PTS}(v, \mathscr{E}(\overline{\mathscr{B}({\mathcal{L}_C}^{\mathsf{en}}(p))})) \tag{5.9}$$

Let $p_{\texttt{O1},\texttt{v}}$ be the $\mathcal{L}_{FC}$-path in Equation (5.5) (by ignoring $\boxed{\hat{\texttt{c3}}}$ and $\boxed{\check{\texttt{c3}}}$). By definition, $\mathcal{L}_C(p_{\texttt{O1},\texttt{v}}) = \hat{\texttt{c1}}\check{\texttt{c1}}\hat{\texttt{c1}}\hat{\texttt{c3}}$, where $p^{\mathsf{ex}}_{\texttt{O1},\texttt{v}}$ can be interpreted as the sub-path from $\texttt{O1}$ to $\texttt{A1}$ and $p^{\mathsf{en}}_{\texttt{O1},\texttt{v}}$ as the sub-path from $\texttt{A1}$ to $\texttt{v}$. Thus, ${\mathcal{L}_C}^{\mathsf{ex}}(p_{\texttt{O1},\texttt{v}}) = \hat{\texttt{c1}}\check{\texttt{c1}}$ and ${\mathcal{L}_C}^{\mathsf{en}}(p_{\texttt{O1},\texttt{v}}) = \hat{\texttt{c1}}\hat{\texttt{c3}}$. Since $\mathscr{E}(\mathscr{B}(\hat{\texttt{c1}}\check{\texttt{c1}})) = \mathscr{E}(\epsilon) = [\,]$ and $\mathscr{E}(\overline{\mathscr{B}(\hat{\texttt{c1}}\hat{\texttt{c3}})}) = \mathscr{E}(\overline{\hat{\texttt{c1}}\hat{\texttt{c3}}}) = \mathscr{E}(\check{\texttt{c3}}\check{\texttt{c1}}) = [\texttt{c3}, \texttt{c1}]$, we have:

$$\langle \texttt{O1}, [\,]\rangle \in \texttt{PTS}(\texttt{v}, [\texttt{c3}, \texttt{c1}])$$

However, relying on $\mathcal{L}_{FC}$ for solving k-CFA is insufficient.

Lemma 5.2 *$\mathcal{L}_{FC}$ is sound but less precise than L_{FC}^{dd}.*

Proof Sketch. In terms of their PAGs used, $\mathcal{L}_{FCR}$ differs from L_{FC}^{dd} (i.e., L_{FC}) only in how virtual callsites are handled. Given a program, its PAG used by $\mathcal{L}_{FC}$ (Figure 5.3), together with $\mathcal{L}_F$, ensures that for every virtual callsite $\mathtt{x} = \mathtt{r.m}(a_1, \dots, a_n)$, there always exists an $\mathcal{L}_{FC}$-path from its arguments (return variable) to their parameters (x) for all possible methods invoked at the callsite. Let $\mathtt{PTS}^{L_{FC}^{dd}}(v, c_v)$ be computed by L_{FC}^{dd}. By applying Lemma 5.1 further and noting the definition of L_{FC}^{dd} (stated at the beginning of Section 5.3), we obtain $\mathtt{PTS}(v, c_v) \supseteq \mathtt{PTS}^{L_{FC}^{dd}}(v, c_v)$, where $\supseteq$ can be strict, i.e., $\supset$. $\square$

$\mathcal{L}_{FC}$ can lose precision since, for some $\mathcal{L}_{FC}$-paths, its sub-paths responsible for performing dynamic dispatch can be spurious. Consider a virtual callsite $\mathtt{x} = \mathtt{r.m}(a_1, \dots, a_n)$ at a callsite c. Before passing an argument a_i into (or receiving a return value from) a method invoked at this callsite, $\mathcal{L}_{FC}$ performs dynamic dispatch by carrying out the following alias-related traversal on its receiver variable r:

$$\cdots \xrightarrow[\hat{\boxed{\mathtt{c}}}]{\ell} \mathtt{r}\ \overline{\mathsf{flowsto}}\ O\ \mathsf{flowsto}\ \mathtt{r}' \xrightarrow[\check{\boxed{c'}}]{\mathsf{assign}} \mathtt{r}'\#c' \xrightarrow[\hat{c'}]{\mathsf{dispatch[_]}} \cdots \tag{5.10}$$

where ℓ is store[i] (in passing a_i) or $\overline{\mathsf{load[0]}}$ (in retrieving a return value). Such a path, which starts from $\hat{\boxed{\mathtt{c}}}$ and ends at $\check{\boxed{c'}}$, is called a *dispatch path*. A dispatch path is said to be *valid* if the following two conditions are met:

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

However, $\mathcal{L}_{FC}$ can only ensure that r and r′ are aliases with no guarantee for the validity of this dispatch path.

To filter out all $\mathcal{L}_{FC}$-paths containing invalid dispatch paths, we use a third CFL $\mathcal{L}_R$ to enforce `DP-C1` and `DP-C2`, thereby addressing **CHL3** by restoring the context of `r`:

$$
\begin{array}{rcl}
W & \longrightarrow & W\ \hat{c} \mid W\ \check{c} \mid W\ R \mid \epsilon \\
R & \longrightarrow & \hat{\boxed{c}}\ Y\ \check{\boxed{c}} \\
Y & \longrightarrow & B\ Y \mid Y\ B \mid \check{c}\ Y\ \hat{c} \mid \epsilon \\
B & \longrightarrow & B\ B \mid \hat{c}\ B\ \check{c} \mid R \mid \epsilon
\end{array}
\tag{5.11}
$$

where its terminals are the below-edge labels, $\hat{c}$ and $\check{c}$, $\hat{\boxed{c}}$, and $\check{\boxed{c}}$, in the `PAG`. The R-production enforces `DP-C1` and the set of Y- and B-productions enforces `DP-C2`.

Imprecision of $\mathcal{L}_{FC}$ We look at two examples to understand why $\mathcal{L}_{FC}$ fails to enforce `DP-C1` and `DP-C2`.

Consider the following simple code snippet:

```
D d1 = new D(); // D1
D d2 = new D(); // D2
A a = new A(); // A1
a.foo(d1); // c1
a.foo(d2); // c2
```

where classes `A` and `D` are from Figure 5.1. The following path is accepted by $\mathcal{L}_{FC}$ (as an $\mathcal{L}_{FC}$-path) but rejected by $\mathcal{L}_{FCR}$:

$$
\texttt{D1} \xrightarrow{\textsf{new[D]}} \texttt{d1} \xrightarrow[\hat{\boxed{\texttt{c1}}}]{\textsf{store[1]}} \texttt{a} \xrightarrow{\overline{\textsf{new[A]}}} \texttt{A1} \xrightarrow{\textsf{new[A]}} \texttt{a} \xrightarrow[\check{\boxed{\texttt{c2}}}]{\textsf{assign}} \texttt{a\#c2} \xrightarrow[\hat{\texttt{c2}}]{\textsf{dispatch[A]}} \texttt{this}^{\texttt{foo}} \xrightarrow{\textsf{load[1]}} \texttt{p}
\tag{5.12}
$$

Its dispatch path is invalid since it violates `DP-C1` (due to `c1` $\neq$ `c2`). As a result, $\mathcal{L}_{FC}$ allows `D1` to flow to `p` under a wrong context [`c2`], causing potentially precision loss.

Consider the second slightly more complex example:

```
1  class A {                              8  static void main() {
2    O id(O p) { return p; } }            9    A a1 = new A(); // A1
3  class O { }                           10    O o1 = new O(); // O1
4  static O wid(A a, O o) {              11    O o2 = new O(); // O2
5    O v = a.id(o); // c3                12    O v1 = wid(a1, o1); // c1
6    return v;                           13    O v2 = wid(a1, o2); // c2
7  }                                     14  }
```

The following path is accepted by $\mathcal{L}_{FC}$ but not by $\mathcal{L}_{FCR}$:

$$
\begin{array}{l}
\texttt{O1} \xrightarrow{\mathsf{new}[\texttt{O}]} \texttt{o1} \xrightarrow[\hat{\texttt{c1}}]{\mathsf{assign}} \texttt{o} \xrightarrow[\boxed{\hat{\texttt{c3}}}]{\mathsf{store}[1]} \texttt{a} \xrightarrow[\check{\texttt{c1}}]{\overline{\mathsf{assign}}} \texttt{a1} \xrightarrow{\overline{\mathsf{new}[\texttt{A}]}} \texttt{A1} \xrightarrow{\mathsf{new}[\texttt{A}]} \texttt{a1} \xrightarrow[\hat{\texttt{c2}}]{\mathsf{assign}} \texttt{a} \\
\xrightarrow[\boxed{\check{\texttt{c3}}}]{\mathsf{assign}} \texttt{a\#c3} \xrightarrow[\hat{\texttt{c3}}]{\mathsf{dispatch}[\texttt{A}]} \texttt{this}^{\texttt{id}} \xrightarrow{\mathsf{load}[1]} \texttt{p} \xrightarrow{\mathsf{store}[0]} \texttt{this}^{\texttt{id}} \xrightarrow[\check{\texttt{c3}}]{\overline{\mathsf{dispatch}[\texttt{A}]}} \texttt{a\#c3} \\
\xrightarrow[\boxed{\hat{\texttt{c3}}}]{\overline{\mathsf{assign}}} \texttt{a} \xrightarrow[\check{\texttt{c2}}]{\overline{\mathsf{assign}}} \texttt{a1} \xrightarrow{\overline{\mathsf{new}[\texttt{A}]}} \texttt{A1} \xrightarrow{\mathsf{new}[\texttt{A}]} \texttt{a1} \xrightarrow[\hat{\texttt{c2}}]{\mathsf{assign}} \texttt{a} \xrightarrow[\boxed{\check{\texttt{c3}}}]{\mathsf{load}[0]} \texttt{v} \xrightarrow[\check{\texttt{c2}}]{\mathsf{assign}} \texttt{v2}
\end{array}
\tag{5.13}
$$

This path contains two dispatch paths for "`a.id(o) //c3`", one for passing `o` to `p` and one for returning `p` back to the same callsite. The first one is invalid, since `a` starts with pointing to `A1` under [`c1`] during its $\overline{\mathsf{flowsto}}$ traversal but ends up with pointing to `A1` under [`c2`] during the ensuing $\mathsf{flowsto}$ traversal, violating `DP-C2`. Thus, $\mathcal{L}_{FC}$ enables `O1` passed at callsite `c1` to flow into `v2` at callsite `c2` spuriously.

Precision of $\mathcal{L}_{FCR}$ We have designed $\mathcal{L}_R$ to filter out all $\mathcal{L}_{FC}$-paths containing invalid dispatch paths. Let us examine its productions by considering a generic dispatch path given in Equation (5.10). The start symbol W would define a language that contains $\mathcal{L}_C$ if its alternative WR were changed to W. Therefore, $\mathcal{L}_R$ comes into play only when a dispatch path is traversed by enforcing simply `DP-C1` and `DP-C2`.

To enforce DP-C1, the R-production, $R \longrightarrow \hat{\boxed{c}}\ Y\ \check{\boxed{c}}$, states that if we start a dispatch process at a callsite (flagged by $\hat{\boxed{c}}$), we must return to the same callsite (flagged by $\check{\boxed{c}}$). For the dispatch path in Equation (5.10), we are therefore guaranteed that $c = c'$, and consequently, $\mathtt{r} = \mathtt{r}'$. As a result, once $\hat{\boxed{c}}$ and $\check{\boxed{c}}$ are matched, c is recovered to appear at the ensuing dispatch edge so that dynamic dispatch can be performed at exactly the same callsite, i.e., c. Let us return to the path in Equation (5.12) (Sec. 5.3.2.2) but modified now with its two occurrences of c2 being replaced by c1. Due to the R-production, $\mathcal{L}_{FCR}$ will accept this modified path as an $\mathcal{L}_{FCR}$-path.

To enforce DP-C2, we rely on the Y- and B-productions, of which the Y-production, $Y \longrightarrow \check{c}\ Y\ \hat{c}$, plays the key role. Let us explain its theoretical basis by referring to a generic dispatch path given in Equation (5.10) again. We can express DP-C2 equivalently as follows. Let $p_{\mathtt{r},O}$ be the $\overline{\mathsf{flowsto}}$ path from $\mathtt{r}$ to O. Its inverse $\overline{p_{\mathtt{r},O}}$ is naturally a flowsto path. Let $p_{O,\mathtt{r}'}$ be the flowsto path from O to $\mathtt{r}'$. By Equation (5.9), we obtain:

$$\begin{array}{c} \langle O, \mathscr{E}(\mathscr{B}(\mathcal{L}_C{}^{\mathsf{ex}}(\overline{p_{\mathtt{r},O}})))\rangle \in \mathtt{PTS}(\mathtt{r}, \mathscr{E}(\overline{\mathscr{B}(\mathcal{L}_C{}^{\mathsf{en}}(\overline{p_{\mathtt{r},O}}))})) \\ \langle O, \mathscr{E}(\mathscr{B}(\mathcal{L}_C{}^{\mathsf{ex}}(p_{O,\mathtt{r}'})))\rangle \in \mathtt{PTS}(\mathtt{r}', \mathscr{E}(\overline{\mathscr{B}(\mathcal{L}_C{}^{\mathsf{en}}(p_{O,\mathtt{r}'}))})) \end{array} \tag{5.14}$$

As aliases, $\mathtt{r}$ and $\mathtt{r}'$ point to O with the same heap context:

$$\mathscr{E}(\mathscr{B}(\mathcal{L}_C{}^{\mathsf{ex}}(\overline{p_{\mathtt{r},O}}))) = \mathscr{E}(\mathscr{B}(\mathcal{L}_C{}^{\mathsf{ex}}(p_{O,\mathtt{r}'}))) \tag{5.15}$$

implying that the entry contexts in $\overline{\mathscr{B}(\mathcal{L}_C{}^{\mathsf{ex}}(\overline{p_{\mathtt{r},O}}))}$ are fully balanced out by the exit contexts in $\mathscr{B}(\mathcal{L}_C{}^{\mathsf{ex}}(p_{O,\mathtt{r}'}))$ in $\mathcal{L}_C$:

$$\mathscr{B}\Big(\overline{\mathscr{B}(\mathcal{L}_C{}^{\mathsf{ex}}(\overline{p_{\mathtt{r},O}}))}\mathscr{B}(\mathcal{L}_C{}^{\mathsf{ex}}(p_{O,\mathtt{r}'}))\Big) \quad = \quad \epsilon \tag{5.16}$$

Recall that exit and entry are inverses of each other (Equation (2.10)).

Now, r and r′ have the same context (needed by DP-C2) iff

$$\mathscr{E}(\overline{\mathscr{B}(\mathcal{L}_C{}^{\mathsf{en}}(\overline{p_{\mathtt{r},O}}))}) = \mathscr{E}(\overline{\mathscr{B}(\mathcal{L}_C{}^{\mathsf{en}}(p_{O,\mathtt{r}'}))}) \tag{5.17}$$

In $\mathcal{L}_R$, the exit contexts in $\overline{\mathscr{B}(\mathcal{L}_C{}^{\mathsf{en}}(\overline{p_{\mathtt{r},O}}))}$ are thus needed to be balanced out by the entry contexts in $\mathscr{B}(\mathcal{L}_C{}^{\mathsf{en}}(p_{O,\mathtt{r}'})$:

$$\mathscr{B}\Big(\mathscr{B}(\mathcal{L}_C{}^{\mathsf{en}}(p_{O,\mathtt{r}'})\overline{\mathscr{B}(\mathcal{L}_C{}^{\mathsf{en}}(\overline{p_{\mathtt{r},O}}))}\Big) \quad = \quad \epsilon \tag{5.18}$$

We are now ready to explain the Y- and B- productions. When traversing a dispatch path (Equation (5.10)), $Y \longrightarrow \check{c}\ Y\ \hat{c}$ serves to enforce DP-C2 via Equation (5.18), $\mathsf{B} \rightarrow R$ is used to start traversing another dispatch path (recursively), the remaining productions serve to skip all balanced contexts. Informally, if we write down all the unmatched exit contexts we see when moving from r to O (r $\overline{\text{flowsto}}$ O) as $\check{c_1}, \ldots, \check{c_n}$, then all the unmatched entry contexts we see in returning from O to r′ (O flowsto r′) must be $\hat{c_n}, \ldots, \hat{c_1}$. ($\mathtt{r} = \mathtt{r}'$ due to DP-C1.)

For $\mathcal{L}_{FCR}$, the points-to relation given in Equation (5.9) remains unchanged except that only $\mathcal{L}_{FCR}$-paths are considered.

Let us return to Equation (5.13) (Sec. 5.3.2.2). For its first dispatch path launched at callsite c3 starting from a and ending at a#c3, $\overline{\mathscr{B}(\mathcal{L}_C{}^{\mathsf{en}}(\overline{p_{\mathtt{a},\mathtt{A1}}}))} = \check{\mathtt{c1}}$ and $\mathscr{B}(\mathcal{L}_C{}^{\mathsf{en}}(p_{\mathtt{A1},\mathtt{a}})) = \hat{\mathtt{c2}}$. It is invalid as $\check{\mathtt{c1}}\hat{\mathtt{c2}}$ cannot balance out according to $Y \longrightarrow \check{c}\ Y\ \hat{c}$. However, the path in Equation (5.13), once modified with its three occurrences of c2 replaced by c1, will be an $\mathcal{L}_{FCR}$-path, implying that its first dispatch path will also become valid.

Lemma 5.3 *At every virtual callsite, $\mathcal{L}_{FCR}$ passes arguments into and receives return values from exactly the same set of target methods found at the callsite as L^{dd}_{FC} does.*

PROOF SKETCH. $\mathcal{L}_R$ filters out all and only $\mathcal{L}_{FC}$-paths containing invalid dispatch paths (discussed and proved above). □

Theorem 5.4 *$\mathcal{L}_{FCR}$ and L^{dd}_{FC} are equally precise.*

PROOF. Lemmas 5.2 and 5.3. □

We can now apply $\mathcal{L}_{FCR}$ to compute the points-to information in our motivating example (Figure 5.1). We have discussed several times earlier one of its $\mathcal{L}_{FCR}$-paths given in Equation (5.5). Note that `C:foo()`, which appears in its PAG (Figure 5.4), can never be called due to on-the-fly call graph construction.

Finally, we conclude this section with one caveat. L^{dd}_{FC} (introduced in [76] and released in SOOT [89]) suffers from the precision loss discussed in Sec. 5.2.3 but is assumed to be free of this issue to facilitate our presentation.

5.3.3 Time Complexities

In general, the $\mathcal{L}_{FCR}$-reachability problem is undecidable as it is the intersection of three CFLs interleaved with each other [64]. For a single CFL language $L \in \{\mathcal{L}_F, \mathcal{L}_C, \mathcal{L}_R\}$, the time complexity for solving its L-reachability problem is bounded by $O(m^3n^3)$ from above, where m is its grammar size and n is the number of the `PAG` nodes. $\mathcal{L}_C$ is a standard Dyck-CFL and its complexity can be reduced to $O(mn^3)$ [33].

5.4 P3Ctx: An $\mathcal{L}_{FCR}$ Application

To demonstrate the utility of $\mathcal{L}_{FCR}$, we consider one significant application by introducing the first $\mathcal{L}_{FCR}$-enabled pre-analysis, P3Ctx, for accelerating k-CFA (implemented as a whole-program analysis in terms of the rules in Figure 2.3) with selective context-sensitivity while always preserving its precision. This also serves to validate the correctness of $\mathcal{L}_{FCR}$. In contrast, Selectx, a recently proposed pre-analysis developed based on L_{FC} [48], is not precision-preserving.

5.4.1 CFL-Reachability-Guided Selections

We have developed P3Ctx by following the same basic principle introduced in [48] for developing Selectx. The basic idea in applying L_{FC} to develop Selectx [48] is simple. Let $p_{O,n,v}$ be a flowsto path operated by L_{FC} from some object O to some variable v, where n is a variable/object accessed in a method `M`. Let $p_{O,n}$ be its sub-path from O to n and $p_{n,v}$ its sub-path from n to v. Then n requires context-sensitivity (to prevent k-CFA from potentially losing precision) *only if*:

$$\begin{aligned} &\texttt{CS-C1}: L_F(p_{O,n,v}) \in L_F \\ &\texttt{CS-C2}: \wedge\, L_C(p_{O,n}) \in L_C \wedge L_C(p_{n,v}) \in L_C \\ &\texttt{CS-C3}: \wedge\, L_C^{\mathsf{en}}(p_{O,n}) \neq \epsilon \wedge L_C^{\mathsf{ex}}(p_{n,v}) \neq \epsilon \end{aligned} \tag{5.19}$$

In this case, 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.

Selectx will select n to be context-sensitive *if* `CS-C1`– `CS-C3` hold. By interpreting these conditions as being sufficient (rather than just necessary), Selectx

is conservative as it may select some n to be context-sensitive even though k-CFA loses no precision if it is analyzed context-insensitively.

However, SELECTX may cause k-CFA to lose precision. Consider our motivating example (Figure 5.1), for which whether `v` points to `O2` spuriously or not hinges on whether `d`, `o`, `x`, and `D1` in `bar()` (containing a virtual callsite `x.foo(d)`) are analyzed context-sensitively or not. SELECTX will select all the four to be context-insensitive (causing `v` to point to `O2`), as none can flow out of `bar()` via its parameter `x` (which is also the receiver variable of `x.foo(d)`) in the PAG operated by L_{FC} (as revealed by Equation (5.1) and Equation (5.2) for `d`, `o`, and `D1` and the paths for `x` that are not given explicitly but can be constructed easily). Thus, `CS-C3` fails to hold. In L_{FC}, which uses a separate algorithm for call graph construction, its PAG representation contains no dispatch paths that allow these four variables/objects to flow outside `bar()` via `x` (Figure 5.2).

P3CTX will always be precision-preserving as it leverages `CS-C1`– `CS-C3` by setting $L_C = \mathcal{L}_C$ but substituting $\mathcal{L}_F$ for L_F, with $\mathcal{L}_{FC}$ operating on a PAG representation including explicitly the dispatch paths for all virtual callsites in the program. Consider our motivating example again. In $\mathcal{L}_F$, parameter passing for `d` at `x.foo(d)` is CFL-reachability-related to its receiver variable `x`. Let $p_{\texttt{O1},n,\texttt{v}}$ be the path in Equation (5.5) (which happens to be an $\mathcal{L}_{FC}$-path). Let $n \in \{\texttt{d}, \texttt{o}, \texttt{x}, \texttt{D1}\}$. P3CTX will select every n to be context-sensitive, since $p_{\texttt{O1},n,\texttt{v}}$ is an $\mathcal{L}_F$-path (`CS-C1`), and both $p_{\texttt{O1},n}$ and $p_{n,v}$ are $\mathcal{L}_C$-paths (`CS-C2`), and $\mathcal{L}_C{}^{\textsf{en}}(p_{\texttt{O1},n}) = \hat{c}1 \neq \epsilon$ and $\mathcal{L}_C{}^{\textsf{ex}}(p_{n,v}) = \check{c}1 \neq \epsilon$ (`CS-C3`).

5.4.2 Regularization of $\mathcal{L}_F$ into $\mathcal{L}_{F^r}$

As $\mathcal{L}_{FC} \supseteq \mathcal{L}_{FCR}$ (Lemma 5.2 and Theorem 5.4), it suffices to use $\mathcal{L}_{FC}$ in place of L_{FC} in Equation (5.19) in developing our precision-preserving pre-analysis (by

noting that $\mathcal{L}_C = L_C$). As the $\mathcal{L}_{FC}$-problem is also undecidable [64], we follow [48] to regularize $\mathcal{L}_F$ into $\mathcal{L}_{F^r}$ and thus over-approximate $\mathcal{L}_{FC}$ to $\mathcal{L}_{F^rC} = \mathcal{L}_{F^r} \cap \mathcal{L}_C$, so that we can verify CS-C1– CS-C3 efficiently by using $\mathcal{L}_{F^rC}$.

We start with $\mathcal{L}_0 = \mathcal{L}_F$, where $\mathcal{L}_F$ is given in Equation (5.6). Next, we over-approximate $\mathcal{L}_0$ by disregarding its field-sensitivity requirement and thus obtain $\mathcal{L}_1$ given below:

$$\begin{aligned} \mathsf{flowsto} &\longrightarrow \mathsf{new}\ (\mathsf{flows} \mid \mathsf{dispatch})^* \\ \mathsf{flows} &\longrightarrow \mathsf{assign} \mid \mathsf{store}\ \overline{\mathsf{flowsto}}\ \mathsf{flowsto}\ \mathsf{load} \\ \overline{\mathsf{flowsto}} &\longrightarrow (\overline{\mathsf{dispatch}} \mid \overline{\mathsf{flows}})^*\ \overline{\mathsf{new}} \\ \overline{\mathsf{flows}} &\longrightarrow \overline{\mathsf{assign}} \mid \overline{\mathsf{load}}\ \overline{\mathsf{flowsto}}\ \mathsf{flowsto}\ \overline{\mathsf{store}} \end{aligned} \tag{5.20}$$

In the absence of field-sensitivity, a 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 $\mathcal{L}_2$ below:

$$\begin{aligned} \mathsf{flowsto} &\longrightarrow \mathsf{new}\ \mathsf{flows}^* \\ \overline{\mathsf{flowsto}} &\longrightarrow \overline{\mathsf{flows}}^*\ \overline{\mathsf{new}} \\ \mathsf{flows} &\longrightarrow \mathsf{assign} \mid \mathsf{store}\ \overline{\mathsf{flowsto}}\ \mathsf{flowsto}\ \mathsf{load} \\ \overline{\mathsf{flows}} &\longrightarrow \overline{\mathsf{assign}} \mid \overline{\mathsf{load}}\ \overline{\mathsf{flowsto}}\ \mathsf{flowsto}\ \overline{\mathsf{store}} \end{aligned} \tag{5.21}$$

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 $\mathcal{L}_3$ below:

$$\begin{aligned} \mathsf{flowsto} &\longrightarrow \mathsf{new}\ \mathsf{flows}^* \\ \overline{\mathsf{flowsto}} &\longrightarrow \overline{\mathsf{flows}}^*\ \overline{\mathsf{new}} \\ \mathsf{flows} &\longrightarrow \mathsf{assign} \mid \mathsf{store}\ \overline{\mathsf{flowsto}}\ \mathsf{flowsto} \\ \overline{\mathsf{flows}} &\longrightarrow \overline{\mathsf{assign}} \mid \overline{\mathsf{flowsto}}\ \mathsf{flowsto}\ \overline{\mathsf{store}} \end{aligned} \tag{5.22}$$

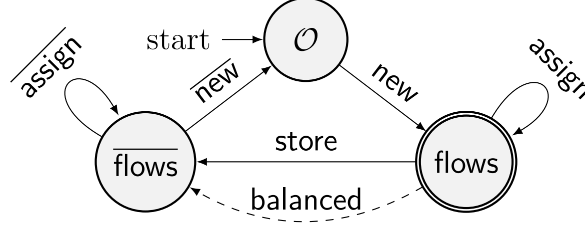


Figure 5.6: A DFA for accepting $\mathcal{L}_{F^r}$.

Finally, we obtain $\mathcal{L}_{F^r} = \mathcal{L}_4$ given below by no longer distinguishing a store edge from its inverse, $\overline{\text{store}}$, so that we can represent both types of edges as a store edge:

$$
\begin{aligned}
\text{flowsto} &\longrightarrow \text{new flows}^* \\
\overline{\text{flowsto}} &\longrightarrow \overline{\text{flows}}^*\ \overline{\text{new}} \\
\text{flows} &\longrightarrow \text{assign} \mid \text{store}\ \overline{\text{assign}}^*\ \overline{\text{new}}\ \text{new} \\
\overline{\text{flows}} &\longrightarrow \overline{\text{assign}} \mid \overline{\text{new}}\ \text{new}\ \text{assign}^*\ \text{store}
\end{aligned}
\tag{5.23}
$$

Lemma 5.5 $\mathcal{L}_F \subset \mathcal{L}_{F^r}$.

PROOF. Follows from the fact that $\mathcal{L}_i \subseteq \mathcal{L}_{i+1}$. □

While $\mathcal{L}_{F^r}$ is identical to L_R regularized from L_F in SELECTX [48], our PAG representation (Figure 5.3), which makes all dynamic dispatch paths explicitly, differs fundamentally from the one operated by L_{FC} (Figure 2.5). Let $G = (N, E)$ be the PAG of a program. We use Andersen's algorithm [3] instead of CHA [16] to build its underlying call graph in order to sharpen the precision of P3CTX.

5.4.3 P3Ctx

We use a simple DFA in Figure 5.6 designed to accept $\mathcal{L}_{F^r}$ exactly. P3CTX is inter-procedural running in linear time of the number of the PAG edges in G. To deal with $\mathcal{L}_C$, we make use of summary edges added into the PAG (facilitated by the transition labeled by balanced).

Let $Q = \{\mathcal{O}, \mathsf{flows}, \overline{\mathsf{flows}}\}$ be the set of states and $\delta : Q \times \Sigma \mapsto Q$ the state transition function. Given a PAG edge $n_1 \xrightarrow{\ell} n_2 \in E$ in G with its state transition $\delta(q_1, \ell) = q_2$, we define $(n_1, q_1) \rightarrowtail (n_2, q_2)$ as a one-step transition. The transitive closure of $\rightarrowtail$, denoted by $\rightarrowtail^+$, represents a multiple-step transition. As $\mathsf{flowsto}$ and $\overline{\mathsf{flowsto}}$ in $\mathcal{L}_{F^r}$ are symmetric, the following two properties about this DFA are immediate:

- `PROP-O`. Let O be an object created in a method `M`. Then $\langle \mathtt{this}^{\mathtt{M}}, \mathsf{flows}\rangle \rightarrowtail^+ \langle O, \mathcal{O}\rangle \iff \langle O, \mathcal{O}\rangle \rightarrowtail^+ \langle \mathtt{this}^{\mathtt{M}}, \overline{\mathsf{flows}}\rangle$.

- `PROP-V`. Let v be a variable defined in a method `M`. Then $\langle \mathtt{this}^{\mathtt{M}}, \mathsf{flows}\rangle \rightarrowtail^+ \langle v, q\rangle \iff \langle v, \overline{q}\rangle \rightarrowtail^+ \langle \mathtt{this}^{\mathtt{M}}, \overline{\mathsf{flows}}\rangle$, where $q \in \{\mathsf{flows}, \overline{\mathsf{flows}}\}$ (since v is a variable).

To handle static callsites (methods) uniformly as virtual callsites (methods), we assume that a static callsite is invoked on a unique dummy receiver object. Thus, in our PAG representation (Figure 5.3), passing arguments and receiving return values for a method must all flow through its `this` variable.

To verify `CS-C1` in Equation (5.19), where L_F is now replaced by $\mathcal{L}_{F^r}$, we do not have to start from an object to track its $\mathsf{flowsto}$ paths. For each method, we can start from its `this` variable by assuming reasonably and over-approximately that there always exists some object O that can flow into it.

To verify `CS-C2`, we take advantage of summary edges to verify the balanced-parentheses property in $\mathcal{L}_C$-paths.

To verify `CS-C3`, we check if there exists $q \in Q$ such that

$$\langle \mathtt{this}^{\mathtt{M}}, \mathsf{flows}\rangle \rightarrowtail^+ \langle n, q\rangle \rightarrowtail^+ \langle \mathtt{this}^{\mathtt{M}}, \overline{\mathsf{flows}}\rangle \tag{5.24}$$

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

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

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

Figure 5.7: Rules for conducting P3Ctx over $G = (N, E)$.

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

Figure 5.7 gives our P3CTX pre-analysis, which checks `CS-C1`– `CS-C3` by checking the following condition efficiently:

$$n \in R(\mathcal{O}) \vee n \in R(\mathsf{flows}) \cap R(\overline{\mathsf{flows}}) \tag{5.25}$$

where $R : Q \mapsto \wp(N)$ returns the set of nodes in G reached at a given state $q \in Q$ and $R^{-1} : N \mapsto \wp(Q)$ is the inverse of R, which are computed by performing a simple inter-procedural reachability analysis in G. [F-INIT] does the initializations needed, [F-PROPA] computes the reachable states for each node iteratively, and [F-SUM] performs a standard context-sensitive summary for a callsite invoking `M` [66] by adding a summary edge $n_1 \xrightarrow{\mathsf{balanced}} n_2$ in G to capture inter-procedural reachability across the callsite.

P3CTX checks `CS-C1`– `CS-C3` as follows. For `CS-C1`, we rely on a simple DFA for accepting $\mathcal{L}_{F^r}$. For `CS-C2`, we verify the balanced-parentheses property in $\mathcal{L}_C$ by using summary edges. For `CS-C3` (i.e., Equation (5.24)), we resort to Equation (5.25). Its first disjunct says that if an object n is in $R(\mathcal{O})$, then Equation (5.24) holds due

to PROP-O. Its second disjunct says that if a variable n is in $R(\mathsf{flows}) \cap R(\overline{\mathsf{flows}})$, then Equation (5.24) holds (due to PROP-V).

Theorem 5.6 *k-CFA (performed in terms of the rules in Figure 2.3) produces exactly the same points-to information when performed with selective context-sensitivity under P3CTX.*

PROOF. Follows from the facts that (1) Equation (5.19) provides a set of necessary conditions for supporting selective context-sensitive, (2) $\mathcal{L}_{FCR}$ provides a complete specification of k-CFA via CFL-reachability, (3) $\mathcal{L}_{F^rC} \supseteq \mathcal{L}_{FC} \supseteq \mathcal{L}_{FCR}$, 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 $|E|$, where $|Q| = 3$ is the number of states in our DFA.

5.5 Evaluation

We demonstrate that P3CTX can enable k-CFA to run substantially faster while preserving its precision. We do not compare with SELECTX [48] as it is not precision-preserving.

5.5.1 Experimental Setup

Implementation We have implemented k-CFA and P3CTX (Figure 5.7) in SOOT [89] on top of its context-insensitive Andersen's pointer analysis, SPARK [36], which is used for building the PAG of a program (including its call graph) for P3CTX. We follow a few common practices adopted in the literature [1, 2, 47–49, 59, 82]. We use

a reflection log generated by a dynamic reflection analysis tool, TamiFlex [9] for resolving Java reflection. For native code, we use the method summaries provided in Soot. String factory objects and exception-like objects are distinguished per dynamic type and analyzed context-insensitively. Our implementation of P3Ctx will soon be released as an open-source tool at `http://www.cse.unsw.edu.au/~corg/p3ctx`.

Benchmarks We have selected 12 large Java programs, including 9 benchmarks from DaCapo-2006-10-MR2 [7] and 3 popular Java applications, `checkstyle`, `findbugs`, and `JPC`, together with a large Java library (`JDK1.6.0_45`), which are frequently used in evaluating pointer analysis algorithms for Java [31,40,74,82]. For DaCapo, `luindex` is excluded as it is similar to `lusearch`. We have also excluded `jython` as its reflection log is overly conservative [86].

Platform We have carried out all our experiments on an Intel(R) Xeon(R) W-2245 3.90GHz machine with 512GB of RAM, running on Ubuntu 20.04.3 LTS (Focal Fossa).

5.5.2 Results

Table 5.2 contains the results for k-CFA, and Pk-CFA (i.e., k-CFA accelerated by P3Ctx), where $k \in \{1, 2\}$. Table 5.3 gives the analysis times of Spark and P3Ctx.

Precision We measure the precision of a pointer analysis by considering four commonly used metrics [24, 40, 49, 73, 82]: "#call-edges", "#poly-calls", "#fail-casts", and "#avg-pts".

Table 5.2: The precision and efficiency of k-CFA, and Pk-CFA. For all the metrics except speedup, smaller is better.

Prog.	Metrics	1CFA	$P1$-CFA	2CFA	$P2$-CFA
antlr	Time(secs)	10.5	3.2 (3.3x)	292.0	99.1 (2.9x)
	#call-edges	55069	55069	54212	54212
	#poly-calls	1922	1922	1861	1861
	#fail-casts	910	910	841	841
	#avg-pts	10.69	10.69	9.50	9.50
bloat	Time(secs)	18.2	8.2 (2.2x)	902.6	572.7 (1.6x)
	#call-edges	64253	64253	63160	63160
	#poly-calls	2054	2054	1993	1993
	#fail-casts	1871	1871	1793	1793
	#avg-pts	47.70	47.70	45.63	45.63
chart	Time(secs)	28.6	8.7 (3.3x)	671.0	181.2 (3.7x)
	#call-edges	72489	72489	71080	71080
	#poly-calls	2380	2380	2290	2290
	#fail-casts	1925	1925	1819	1819
	#avg-pts	51.00	51.00	45.95	45.95
eclipse	Time(secs)	14.5	4.5 (3.2x)	453.9	126.9 (3.6x)
	#call-edges	53274	53274	52069	52069
	#poly-calls	1620	1620	1551	1551
	#fail-casts	1309	1309	1237	1237
	#avg-pts	22.82	22.82	21.39	21.39
fop	Time(secs)	9.9	3.0 (3.3x)	364.3	101.1 (3.6x)
	#call-edges	38116	38116	37264	37264
	#poly-calls	1143	1143	1082	1082
	#fail-casts	694	694	637	637
	#avg-pts	16.99	16.99	15.06	15.06
hsqldb	Time(secs)	8.8	2.5 (3.5x)	285.1	99.2 (2.9x)
	#call-edges	38733	38733	37565	37565
	#poly-calls	1136	1136	1065	1065
	#fail-casts	689	689	635	635
	#avg-pts	11.90	11.90	10.13	10.13
lusearch	Time(secs)	7.0	2.4 (2.9x)	239.1	91.5 (2.6x)
	#call-edges	35012	35012	34159	34159
	#poly-calls	1092	1092	1031	1031
	#fail-casts	659	659	602	602
	#avg-pts	11.73	11.73	10.35	10.35
pmd	Time(secs)	28.2	6.4 (4.4x)	923.4	280.6 (3.3x)
	#call-edges	66865	66865	65877	65877
	#poly-calls	2840	2840	2782	2782
	#fail-casts	2010	2010	1941	1941
	#avg-pts	26.74	26.74	24.38	24.38
xalan	Time(secs)	9.9	3.1 (3.2x)	296.5	102.1 (2.9x)
	#call-edges	41525	41525	40645	40645
	#poly-calls	1321	1321	1260	1260
	#fail-casts	800	800	742	742
	#avg-pts	17.51	17.51	15.72	15.72
checkstyle	Time(secs)	25.3	7.4 (3.4x)	876.6	335.7 (2.6x)
	#call-edges	76463	76463	74792	74792
	#poly-calls	2630	2630	2564	2564
	#fail-casts	1640	1640	1549	1549
	#avg-pts	24.84	24.84	21.85	21.85
findbugs	Time(secs)	35.9	8.8 (4.1x)	944.7	289.3 (3.3x)
	#call-edges	79361	79361	77133	77133
	#poly-calls	3183	3183	3043	3043
	#fail-casts	2091	2091	1972	1972
	#avg-pts	33.41	33.41	30.84	30.84
JPC	Time(secs)	25.7	7.1 (3.6x)	530.5	158.7 (3.3x)
	#call-edges	69137	69137	67989	67989
	#poly-calls	2761	2761	2667	2667
	#fail-casts	1768	1768	1658	1658
	#avg-pts	32.13	32.13	29.27	29.27

Pk-CFA produces the same points-to results as k-CFA as proved in Theorem 5.6 and validated in Table 5.2.

Efficiency We measure the efficiency of a pointer analysis by the time elapsed in analyzing a program (as an average of 3 runs). As shown in Table 5.2, Pk-CFA delivers significant speedups (geometric means) over k-CFA while preserving its precision. The speedups of $P1$-CFA over 1-CFA range from 2.2x (for bloat) to 4.4x (for pmd) with an average of 3.3x. When $k = 2$, the speedups are also impressive, ranging from 1.6x (for bloat) to 3.7x (for chart) with an average of 3.0x. Overall, an average speedup of 3.1x is achieved.

Table 5.3: The analysis times of Spark and P3Ctx in seconds.

Prog.	antlr	bloat	chart	eclipse	fop	hsqldb	lusearch	pmd	xalan	checkstyle	findbugs	JPC
SPARK	4.8	5.6	8.5	5.7	4.3	4.1	3.9	7.6	4.6	8.0	9.1	7.7
P3CTX	0.7	0.8	1.0	0.8	0.6	0.6	0.5	0.9	0.7	1.0	1.0	0.9

Effectiveness According to Table 5.3, P3CTX is effective as a pre-analysis as it is lightweight (running in linear time of the PAG edges in a program). In terms of the average analysis time spent on analyzing the 12 programs, we have 5.9 seconds for SPARK but only 0.8 seconds for P3CTX.

5.6 Conclusion

In this chapter, we have introduced $\mathcal{L}_{FCR}$, a new CFL-reachability formulation for supporting k-callsite-based context-sensitive pointer analysis (k-CFA) with its own built-in call graph construction mechanism for handling dynamic dispatch. Based on this new CFL-reachability formulation, we have also introduced P3CTX for accelerating k-CFA while preserving its precision.

Chapter 6

Related Work

There is a great deal of work related to this thesis. Along the way, we have discussed some of the most relevant work. In this chapter, we briefly review more related work.

6.1 Selective Context-Sensitivity

Context sensitivity is a significant factor in developing highly precise pointer analysis techniques for object-oriented languages like Java. The traditional approaches [31, 38, 53, 54, 73], which blindly apply context-sensitivity to all analyzed methods, are less efficient in analyzing reasonable large applications.

To further tap the performance potential of context-sensitive analysis, many approaches on selective context-sensitivity, which select a subset of precision-critical methods/variables for context-sensitive analysis, have been proposed recently [22, 24, 27, 29, 40, 41, 47–49, 74]. Based on their designing principles, these approaches can be roughly classified into three categories: (1) heuristic- or pattern-based approaches [22, 40, 41, 74], (2) data-driven approaches [27, 29], and (3) CFL-reachability-guided approaches [24, 47–49].

Heuristic- or pattern-based approaches rely on empirical heuristics or commonly used patterns for selecting the set of precision-critical methods/variables to be analyzed context-sensitively. For example, [22, 74] leverage manually-selected metrics to define some heuristics to guide context selection. ZIPPER [40, 41] makes its context selection by determining whether a method contains variables/objects on some paths of specific value-flow patterns.

Data-driven approaches [27,29] apply machine learning to learn a set of heuristic formulas from small test cases for guiding context selection. Unlike the tools introduced in this thesis (e.g., TURNER, CONCH, and P3CTX), which consume only some negligible amount of time in their pre-analysis stages, data-driven approaches can cost substantially longer training times before their main analyses.

CFL-reachability-guided approaches [24, 47–49] rely on some CFL formulations of pointer analysis for making context selection systematically. EAGLE [47, 49] represents a precision-preserving acceleration for object-sensitivity [53, 54]. In Chapter 3, TURNER [24] explores a better efficiency and precision trade-off by exploiting object containment and reachability. SELECTX [48] represents the first attempt for accelerating k-CFA with CFL-reachability. However, it cannot preserve the baseline's precision due to a lack of a call graph construction mechanism built-into the conventional CFL formulation [76, 78]. In Chapter 5, we have designed a new CFL-based formulation to overcome this limitation and also has implemented P3CTX, the first precision-preserving acceleration technique for k-CFA.

6.2 Other Efficient Pointer Analysis Techniques

There are other types of efficient pointer analysis techniques for Java.

MAHJONG [82] sacrifices the precision of alias analysis (by merging objects of the same dynamic type) in order to improve the efficiency of pointer analyses at a small loss of precision for a class of so-called type-dependent clients, such as call graph construction, may-fail casting, and polymorphic call detection.

Thiessen and Lhoták [86] propose to use context transformations rather than context strings as a new context abstraction for pointer analyses, making it theoretically possible for pointer analyses to run more efficiently with better precision.

Tian et al. [83] propose retrieving back the precision of kOBJ by unleashing the precision potential of a set of selective context-sensitive pointer analyses which may lose precision but in an orthogonal manner. In contrast, all the approaches introduced in this thesis are almost precision-preserving for a few commonly used precision metrics in the literature.

Jeon et al. [28] have recently undertaken some research investigating whether object sensitivity is indeed superior to call-site sensitivity for object-oriented programs.

Finally, unlike whole-program analyses [11, 36, 39, 54, 73, 90] considered in this thesis, demand-driven pointer analyses [69, 75, 76, 78, 80, 92] improve the efficiency by typically only computing the points-to information for program points that may affect a particular site of interest for specific clients.

6.3 CFL-Reachability

In program analysis, CFL-reachability [63, 66] was initially introduced for supporting inter-procedural dataflow analysis. It has since been used in tacking many other problems such as pointer analysis [12, 47, 49, 69, 76, 78, 91, 92, 94, 96], information flow [42, 51, 52], and type inference [58, 60]. For callsite-based context-sensitive

pointer analysis [69, 76, 92], the CFL-reachability formulation used so far relies on a separate mechanism for call graph construction (in advance or on the fly). In this thesis, we introduce a CFL-reachability formulation with such a mechanism built in, by using a new language $\mathcal{L}_{FCR}$ expressed as the intersection of three CFLs.

Another line of research on CFL-reachability focuses on its computational complexity. In general, the all-pairs CFL-reachability problem can be solved in $O(m^3n^3)$ time, where m is the size of its underlying CFL grammar and n is the number of its underlying graph nodes. Kodumal et. al [33] solve the Dyck-CFL-reachability more efficiently in $O(mn^3)$. Later, Chaudhuri [13] shows that the general CFL-reachability algorithm can be optimized into a subcubic one by exploiting the well-known Four Russians' Trick [34]. Recently, Zhang et. al [94] show that bidirected Dyck-CFL reachability can be solved in $O(n + p \ \log \ p)$ (where p is the number of its underlying graph edges) by noting that the reachability relation in a bidirected graph is an equivalence relation and Chatterjee et. al [12] improve the problem further by proposing an optimal algorithm solved in $O(p + n \cdot \alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function. In Chapter 5, we introduce P3Ctx as an $\mathcal{L}_{FCR}$-enabled pre-analysis that is linear in terms of the number of PAG edges in a program for accelerating k-CFA without any precision loss.

For a CFL-reachability-based formulation [49] proposed recently for supporting object-sensitive pointer analysis [53, 54], call graph construction is built-in naturally since object-sensitivity uses receiver objects as context elements. For callsite-sensitivity, however, incorporating call graph construction into the traditional CFL-reachability formulation [69, 76, 92] is non-trivial (Section 5.2). To the best of our knowledge, $\mathcal{L}_{FCR}$ represents the first such a solution.

6.4 IFDS Analysis

IFDS dataflow analyses are widely used in software testing, program verification, understanding and maintenance, and compiler optimization. Reps et al. [66] initially introduced an efficient framework for solving the IFDS problems and subsequently generalized it to the IDE framework [68] for interprocedural distributed environment problems, where the dataflow facts are maps ("environments") from some finite set of symbols to some (possibly infinite) set of values. Later, Naeem et al. [55] give several extensions, making it applicable to a broader class of interprocedural dataflow problems, and also introduced a concurrent alternative implemented based on Scala's actor framework. WALA [26] contains a memory-efficient bit-vector-based IFDS algorithm. Recently, Bodden [8] has provided a generic (multi-threaded) implementation of a generic IFDS/IDE solver in Soot [89] and our previous work [23] exploits the sparsity of flow graph to improve IFDS algorithm's performance in an orthogonal way. In this thesis, we apply the IFDS algorithm to a different underlying graph structure (e.g., PAG) to assist CONCH to identify context-independent objects efficiently.

Chapter 7

Summary and Future Directions

Designing a precise and efficient pointer analysis for the real world is urgently required but challenging. The theoretical formulations for Java pointer analysis, either in inclusion-style or CFL-reachability style, have been proven to be undecidable [64]. That is why practical implementations have to resort to regularization in order to boost performance. For example, the inclusion-based formulation regularizes context-sensitivity by applying the k-limiting technique [54, 54, 73], and the CFL-reachability formulation normalizes L_F by losing partial field sensitivity [47,49,76,78]. Despite such approximations, existing pointer analysis algorithms are still impractical for some large programs, in practice.

Regarding the precision of pointer analysis for object-oriented languages. field sensitivity is more significant than context sensitivity, which is more effective than flow sensitivity. To make pointer analysis as practical as possible, the mainstream pointer analysis frameworks choose to support field sensitivity fully and context sensitivity locally (by limiting context stacks with k most recent context elements) but no flow sensitivity. However, the local context sensitivity (by k-limiting) can only be considered as a good heuristic at best. As observed by many existing

works [29,40,49,74], only a small set of methods, variables, and objects benefit from context-sensitivity. Among these, the context lengths required by them may also vary drastically across different parts of the program. Blindly applying contexts to all variables and objects with a uniform context length often makes existing context-sensitive pointer analyses less efficient and practically unusable without introducing much additional precision.

This thesis makes its contributions by proposing three efficient fine-grained pointer analysis techniques for Java. They all achieve promising speedups than the state of the art by selecting a large set of precision-uncritical variables/objects to be analyzed context-insensitively. In Chapter 3, we propose the first intra-procedural pre-analysis for selecting precision-uncritical variables/objects by exploiting object containment and reachability. In Chapter 4, we partially address the context explosion problem in kOBJ by context debloating and propose three linearly verifiable conditions for identifying context-independent objects. In Chapter 5, we propose the first precision-preserving acceleration technique for k-CFA based on a newly designed CFL-reachability formulation for supporting k-CFA with built-in call graph construction mechanism for handling dynamic dispatch.

Despite the advances made in this thesis, Java pointer analysis is far from practically usable for some large programs. We want to highlight that the three tools, i.e., TURNER, CONCH, and P3CTX, are all designed as open-source tools and have been/will be released at:

- TURNER: http://www.cse.unsw.edu.au/~corg/turner
- CONCH: http://www.cse.unsw.edu.au/~corg/conch
- P3CTX: http://www.cse.unsw.edu.au/~corg/p3ctx

We hope the researchers and practitioners in the program analysis community will find these tools to be useful in their future research.

Below, we discuss some future directions we would like to pursue. We also hope that these ideas can inspire future researchers to work in this challenging area.

7.1 Fine-Grained Pointer Analysis with Variable-Level Context Lengths

The three fine-grained pointer analysis techniques proposed in this thesis analyze a subset of variables and objects context-sensitively while others context-insensitively. A further extension could be specifying different context lengths for different variables/objects, thus potentially enabling better performance speedups to be achieved in the future.

A theoretical context length required by a variable/object could be deduced from the CFL-reachability formulations. Let $L_{FC}(O, v)^n$ be an L_{FC} path from O to v that passes through n. The context transformation [86] (i.e., the realized context string with balanced context elements elided out) from O to n and from n to v are respectively denoted as $\check{C}_O\hat{C}_n$ and $\check{C}_n\hat{C}_v$. Then, the context length required by n along this specific path is the number of context elements in $\hat{C}_n$ that would be balanced out in $\check{C}_n$. Given that all L_{FC} paths that pass through n are available, the context length of n is naturally defined as the context length of the path with the maximum number of balanced-out context elements.

However, developing a practical pre-analysis for determining the context length of each variable/object is quite challenging: L_{FC} is undecidable, and overapproximation approaches (like regularization) could not give rise to desirable results. Based on our existing research experience on Java pointer analysis, the vari-

ables/objects that require variable-level context lengths should generally reside in a small part of a given program. In the future, it would be interesting to develop an approach to find such code snippets and apply L_{FC} only to them to further improve the performance of pointer analysis without much pre-analysis overhead.

7.2 Design-Pattern-based Acceleration technique for Pointer Analysis

Real-world Java applications are usually designed by following different kinds of design patterns. For example, compiler parsers often use visiter and factory patterns, and the iterator patterns and proxy patterns are prevalent in the standard JDK library.

While many real-world Java programs are becoming increasingly large and complex, the design patterns used in these programs are finite (e.g., the popular GitHub repository, `java-design-patterns`[1], so far only record 148 patterns). The finite design patterns may suggest that the cases where context sensitivity is necessary are also finite. Therefore, it may be possible to develop an efficient and precise pointer analysis technique based on the features from these design patterns.

In the future, it would be interesting to develop a design-pattern-based acceleration technique for Java pointer analysis.

7.3 Client-Oriented Pointer Analysis

Pointer analysis should be a means to an end, not a stand-alone research field. The goal of pointer analysis is to provide alias information and points-to informa-

[1] https://github.com/iluwatar/java-design-patterns

tion for many clients, such as compiler optimization, bug detection and security analysis. For a specific client, its unique properties and characteristics may offer opportunities for pointer analysis to deliver enough precise points-to information while consuming some acceptable amounts of time/memory resources.

Unfortunately, except for MAHJONG [82] (which is designed for type-dependent clients), almost all other Java pointer analysis techniques are designed for general purposes, resulting in some client-specific optimization opportunities to be missed.

In the future, we foresee that more client-oriented pointer analysis techniques will be designed to satisfy the maximum precision requirement of clients while significantly cutting down analysis overhead.

7.4 Context Debloating for Other Context-Sensitive Program Analysis

The context debloating technique proposed in Chapter 4 is for object-sensitive pointer analysis only. However, the idea of this approach may be also effective in other context-sensitive analyses like the callsite-sensitive pointer analysis [70], context-transformation-based pointer analysis [86], taint analysis [4] and data-dependence analysis [95]. Thus, it may be worth investigating context debloating techniques for these context-sensitive analyses in order to improve their efficiency and scalability, particularly for large codebases.

7.5 Other Potential Directions

There are potentially some other directions of interest.

When designing our fine-grained techniques, we have observed that variables and objects selected by our tools do not affect equally the performance of pointer analysis. Only a small part of variables and objects analyzed context-insensitively can improve significantly the efficiency of pointer analysis. In the future, it would be interesting to identify what these variables/objects are and why they affect substantially the performance of pointer analysis.

In addition, we hope that $\mathcal{L}_{FCR}$ (introduced in Chapter 5) can provide some new insights on understanding k-CFA, especially its demand-driven incarnations [76, 78, 92], and developing new algorithmic solutions. In addition to selective context-sensitivity, leveraging $\mathcal{L}_{FCR}$ in library-code summarization [69, 85] and information flow analysis [42, 51] also merits extensive investigation.

Bibliography

[1] K. Ali and O. Lhoták. Application-only call graph construction. In *ECOOP 2012 – Object-Oriented Programming*, pages 688–712, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[2] K. Ali and O. Lhoták. Averroes: Whole-program analysis without the whole program. In *European Conference on Object-Oriented Programming*, pages 378–400, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[3] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 259–269, New York, NY, USA, 2014. Association for Computing Machinery.

[5] D. F. Bacon and P. 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.

[6] O. Bastani, S. Anand, and A. Aiken. Specification inference using context-free language reachability. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 553–566, New York, NY, USA, 2015. Association for Computing Machinery.

[7] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 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.

[8] E. Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 3–8, 2012.

[9] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. 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.

[10] M. Bravenboer and Y. Smaragdakis. Exception analysis and points-to analysis: Better together. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, page 1–12, New York, NY, USA, 2009. Association for Computing Machinery.

[11] M. Bravenboer and Y. 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.

[12] K. Chatterjee, B. Choudhary, and A. Pavlogiannis. Optimal dyck reachability for data-dependence and alias analysis. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2017.

[13] S. 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.

[14] F. Chow, S. Chan, S.-M. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In *International Conference on Compiler Construction*, pages 253–267, 1996.

[15] J. Da Silva and J. G. Steffan. A probabilistic pointer analysis for speculative optimizations. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, page 416–425, New York, NY, USA, 2006. Association for Computing Machinery.

[16] J. Dean, D. Grove, and C. 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.

[17] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, and C. Zhang. Smoke: Scalable path-sensitive memory leak detection for millions of lines of code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 72–82, New York, NY, USA, 2019. IEEE.

[18] X. Fan, Y. Sui, X. Liao, and J. Xue. Boosting the precision of virtual call integrity protection with partial pointer analysis for c++. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 329–340, New York, NY, USA, 2017. Association for Computing Machinery.

[19] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of Android applications in DroidSafe. In *NDSS*, volume 15, page 110, USA, 2015. The Internet Society.

[20] N. Grech and Y. Smaragdakis. P/taint: Unified points-to and taint analysis. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), Oct. 2017.

[21] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):685–746, 2001.

[22] B. Hassanshahi, R. K. Ramesh, P. Krishnan, B. Scholz, and Y. 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*, page 13–18, New York, NY, USA, 2017. Association for Computing Machinery.

[23] D. He, H. Li, L. Wang, H. Meng, H. Zheng, J. Liu, S. Hu, L. Li, and J. Xue. Performance-boosting sparsification of the ifds algorithm with applications to

taint analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 267–279, San Diego, CA, USA, 2019. IEEE.

[24] D. He, J. Lu, Y. Gao, and J. 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 fuer Informatik.

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

[26] IBM. WALA: T.J. Watson Libraries for Analysis, 2020.

[27] M. Jeon, S. Jeong, and H. Oh. Precise and scalable points-to analysis via data-driven context tunneling. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.

[28] M. Jeon and H. Oh. Return of cfa: Call-site sensitivity can be superior to object sensitivity even for object-oriented programs. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–28, 2022.

[29] S. Jeong, M. Jeon, S. Cha, and H. Oh. Data-driven context-sensitivity for points-to analysis. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):100, 2017.

[30] T. Kapus and C. Cadar. A segmented memory model for symbolic execution. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software*

Engineering Conference and Symposium on the Foundations of Software Engineering, page 774–784, New York, NY, USA, 2019. Association for Computing Machinery.

[31] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 423–434, New York, NY, USA, 2013. Association for Computing Machinery.

[32] J. Kodumal and A. Aiken. The set constraint/cfl reachability connection in practice. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 207–218, New York, NY, USA, 2004. ACM.

[33] J. Kodumal and A. Aiken. The set constraint/cfl reachability connection in practice. *ACM Sigplan Notices*, 39(6):207–218, 2004.

[34] V. A. E. D. M. Kronrod and I. Faradzev. On economic construction of the transitive closure of a directred graph. In *Dokl. Acad. Nauk SSSR*, pages 487–88, 1970.

[35] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis amp; transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004.

[36] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*, pages 153–169, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[37] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1), Oct. 2008.

[38] O. Lhoták and L. 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.

[39] L. Li, C. Cifuentes, and N. Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 343–353, New York, NY, USA, 2011. Association for Computing Machinery.

[40] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis. Precision-guided context sensitivity for pointer analysis. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.

[41] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis. A principled approach to selective context sensitivity for pointer analysis. *ACM Transactions on Programming Languages and Systems*, 42(TOPLAS):1–40, 2020.

[42] Y. Li, Q. Zhang, and T. 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.

[43] B. Liu and J. Huang. D4: Fast concurrency debugging with parallel differential analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Program-*

ming Language Design and Implementation, page 359–373, New York, NY, USA, 2018. Association for Computing Machinery.

[44] B. Liu, J. Huang, and L. Rauchwerger. Rethinking incremental and parallel pointer analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 41(1):1–31, 2019.

[45] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[46] LLVM discussions on pointer analysis, 2016.

[47] J. Lu, D. He, and J. 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.

[48] J. Lu, D. He, and J. Xue. Selective context-sensitivity for k-cfa with cfl-reachability. In *International Static Analysis Symposium*, pages 261–285. Springer, 2021.

[49] J. Lu and J. 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.

[50] D. Melski and T. Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248(1-2):29–98, 2000.

[51] A. 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.

[52] A. Milanova, W. Huang, and Y. 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.

[53] A. Milanova, A. Rountev, and B. 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.

[54] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, 2005.

[55] N. A. Naeem, O. Lhoták, and J. Rodriguez. Practical extensions to the IFDS algorithm. In *International Conference on Compiler Construction*, pages 124–144, 2010.

[56] M. Naik, A. Aiken, and J. 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.

[57] Z. Porkoláb, T. Brunner, D. Krupp, and M. Csordás. Codecompass: An open software comprehension framework for industrial usage. In *Proceedings of the*

26th Conference on Program Comprehension, page 361–369, New York, NY, USA, 2018. Association for Computing Machinery.

[58] P. Pratikakis, J. S. Foster, and M. Hicks. Existential label flow inference via cfl reachability. In *International Static Analysis Symposium*, pages 88–106, Berlin, Heidelberg, 2006. Springer, Springer Berlin Heidelberg.

[59] M. Raghothaman, S. Kulkarni, K. Heo, and M. 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.

[60] J. Rehof and M. Fähndrich. Type-based flow analysis: from polymorphic subtyping to cfl-reachability. *ACM SIGPLAN Notices*, 36(3):54–66, 2001.

[61] M. Reif, M. Eichberg, B. Hermann, J. Lerch, and M. Mezini. Call graph construction for java libraries. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 474–486, New York, NY, USA, 2016. Association for Computing Machinery.

[62] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11):701–726, 1998.

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

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

[65] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT*

Symposium on Principles of Programming Languages, POPL '95, page 49–61, New York, NY, USA, 1995. Association for Computing Machinery.

[66] T. Reps, S. Horwitz, and M. 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, 1995.

[67] B. 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.

[68] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1-2):131–170, 1996.

[69] L. Shang, X. Xie, and J. 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.

[70] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences , New York, NY, USA, 1978.

[71] O. Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Citeseer, 1991.

[72] Y. Smaragdakis. Doop-framework for Java pointer and taint analysis (using p/taint), 2021.

[73] Y. Smaragdakis, M. Bravenboer, and O. 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.

[74] Y. Smaragdakis, G. Kastrinis, and G. 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.

[75] J. Späth, L. Nguyen Quang Do, K. Ali, and E. Bodden. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for Java. In *30th European Conference on Object-Oriented Programming*, pages 22:1–22:26, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

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

[77] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 112–122, New York, NY, USA, 2007. Association for Computing Machinery.

[78] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Confer-*

ence on Object-Oriented Programming, Systems, Languages, and Applications, page 59–76, New York, NY, USA, 2005. Association for Computing Machinery.

[79] Y. Sui, Y. Li, and J. Xue. Query-directed adaptive heap cloning for optimizing compilers. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11, New York, NY, USA, 2013. IEEE.

[80] Y. Sui and J. Xue. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 460–473, New York, NY, USA, 2016. Association for Computing Machinery.

[81] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. *ACM SIGPLAN Notices*, 35(10):264–280, 2000.

[82] T. Tan, Y. Li and J. 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.

[83] T. Tan, Y. Li, X. Ma, C. Xu, and Y. Smaragdakis. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–27, 2021.

[84] T. Tan, Y. Li, and J. Xue. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *International Static Analysis Symposium*, pages 489–510, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[85] H. Tang, X. Wang, L. Zhang, B. Xie, L. Zhang, and H. Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–95, New York, NY, USA, 2015. Association for Computing Machinery.

[86] R. Thiessen and O. Lhoták. Context transformations for pointer analysis. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 263–277, New York, NY, USA, 2017. Association for Computing Machinery.

[87] D. Trabish, T. Kapus, N. Rinetzky, and C. Cadar. Past-sensitive pointer analysis for symbolic execution. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 197–208, New York, NY, USA, 2020. Association for Computing Machinery.

[88] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering*, page 350–360, New York, NY, USA, 2018. Association for Computing Machinery.

[89] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., USA, 2010.

[90] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. Association for Computing Machinery.

[91] G. Xu, A. Rountev, and M. 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 Berlin Heidelberg.

[92] D. Yan, G. Xu, and A. 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.

[93] H. Yan, Y. Sui, S. Chen, and J. Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 327–337, New York, NY, USA, 2018. IEEE.

[94] Q. Zhang, M. R. Lyu, H. Yuan, and Z. 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.

[95] Q. Zhang and Z. Su. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 344–358, New York, NY, USA, 2017. Association for Computing Machinery.

[96] X. Zheng and R. 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.