Efficient Points-To Analysis based on CFL-Reachability Summarisation

Lei Shang

School of Computer Science and Engineering

University of New South Wales

A thesis submitted for the degree of

Doctor of Philosophy

2012

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.'

Signed .....

Date 3/12/2012

Acknowledgements

First of all, I would like to deliver sincere gratitude to Prof. Jingling Xue for supervising me during my PhD. I thank him for guiding me constantly through the road of my research and encouraging me to overcome the difficulties. I have been deeply impressed by his insight and breadth of knowledge in compilers and programming languages as well as his continuous dedication to research work. This work was not possible without his significant encouragements and supports.

I would like to acknowledge the continuous support from my senior colleague, Dr. Yi Lu, together with whom I conducted research on several research topics. I would also like to give my thanks to my friend, Xinwei Xie, for supporting the work presented in Chapter 3.

Besides, my thanks also go to all my friends and colleagues in Compiler Research Group (CORG) in UNSW. A number of colleagues, past and present, have helped greatly at many stages during my PhD study.

Finally, I thank my wonderful wife for her constant love, who has given me tremendous support and encouragement.

Publications

[1] **Lei Shang**, Xinwei Xie and Jingling Xue. On-demand Dynamic Summary-based Points-to Analysis. In *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'12)*, San Jose, California, 2012.

[2] **Lei Shang**, Yi Lu and Jingling Xue. Fast and Precise Points-to Analysis with Incremental CFL-Reachability Summarisation. In *27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*, Essen, Germany, 2012.

[3] **Lei Shang**, Yi Lu and Jingling Xue. Fast and Precise Points-to Analysis with Incremental CFL-Reachability Summarisation. In UNSW Computer Science and Engineering Technical Report no. UNSW-CSE-TR-201216, 2012.

Abstract

Points-to analysis plays a critical role in modern compilers and a wide range of program understanding and bug detection tools. Nevertheless, developing precise and scalable points-to analysis for large-scale object-oriented software remains a challenge, especially in the presence of different client requirements and frequent software modifications.

In this thesis, we present two new techniques for achieving more efficient points-to analysis based on Context-Free Language (CFL)-reachability. In general, our techniques significantly improve the state-of-the-art points-to analysis for Java applications when handling demand-driven queries and small code changes.

This thesis firstly presents an on-demand dynamic summary-based points-to analysis for Java, which provides a more scalable solution without affecting precision. Our second technique is an incremental summarisation framework designed for IDEs, which can efficiently handle frequent program edits, addressing a long-standing challenge in points-to analysis. For each technique, we describe the algorithms and evaluate the implementations with a set of Java applications and clients.

Contents

1 Introduction **10**

1.1 Problem 10

1.2 Challenges 11

1.3 Approaches 12

1.4 Contributions 14

1.5 Outline 15

2 Background **17**

2.1 Points-to Analysis 17

2.2 Program Representation 20

2.3 CFL-Reachability based Points-to Analysis 23

2.3.1 CFL-Reachability 24

2.3.2 Field Sensitivity 24

2.3.3 Context Sensitivity 27

2.3.4 Basic CFL-Reachability-based Points-to Analysis 27

2.3.5 The Benefits of CFL-Reachability Approach 30

2.4 State-of-the-Art 31

2.4.1 Refinement 31
2.4.2 Symbolic Graphs 33

3 Dynamic Summary-Based Demand-Driven Analysis 36
3.1 Overview 36
3.2 A Motivating Example 39
3.3 The DynSum Analysis 42
3.3.1 PPTA: Partial Points-To Analysis 43
3.3.2 Algorithms 45
3.3.3 An Example 48
3.3.4 Comparison 51
3.4 Evaluation 52
3.4.1 Implementation 53
3.4.2 Methodology 54
3.4.3 Results and Analysis 55
3.5 Related Work 60
3.6 Summary 62

4 Incremental Analysis 64
4.1 Overview 65
4.2 Background 68
4.2.1 Example 68
4.2.2 Challenges Facing Incremental Points-to Analysis 71
4.3 Points-to Analysis with Incremental Summarisation 72
4.3.1 Whole-Program Summarisation 75

4.3.2 Incremental Summary Update 80
4.3.3 On-demand Points-to Query 83
4.3.4 Handling Recursion and Call Graph 86
4.4 Evaluation 88
4.4.1 Methodology 88
4.4.2 Results and Analysis 90
4.5 Related Work 95
4.6 Chapter Summary 98

5 Conclusions and Future Work **99**
5.1 Conclusions 99
5.2 Future Work 100

A Formal Rules **103**

Bibliography **110**

List of Figures

1.1 The percentage of local statements and global statements across our benchmarks. More than 80% of statements only are involved only with local variables within their own methods. 14

2.1 An abstraction of Java programs. 21

2.2 A small code example and its graph representation (PAG) for points-to analysis. 23

2.3 Recursive State Machines (RSMs) for L_{FT} and R_{RP}. 25

3.1 A motivating example. 40

3.2 PAG for the example given in Figure 3.1. 41

3.3 Traversals of DynSum when answering the queries for $\mathtt{s1}$ and $\mathtt{s2}$ in our motivating example ($\mathtt{a}, \mathtt{e}$ and $\mathtt{v}$ stand for fields $\mathtt{arr}$, $\mathtt{elems}$ and $\mathtt{vector}$, respectively). 50

3.4 Normalised analysis times for each batch of queries normalised with respect to RefinePTS. 58

3.5 The cumulative number of summaries computed by DynSum normalised with respect to StaSum. 59

4.1 A Java example. 69

4.2 PAG for the Java example given in Figure 4.1. 70

4.3 The framework structure of Emu. 73

4.4 CFL-reachability summarisation for an example. 75

4.5 Abstract domains. 76

4.6 Summaries (with field stacks omitted) before and after deleting "`t2[index] = v`" in `put` in line 14 in Figure 4.1. 81

4.7 Precision of `SafeCast`. 94

A.1 Syntax of an abstraction of Java language. 104

A.2 Abstract domains. 105

A.3 Deduction rules for RefinePTS analysis 106

A.4 Summarisation by local reachability analysis. 107

A.5 Local reachability analysis. 107

A.6 Summary update for code changes. 108

A.7 Summary-based points-to analysis. 108

A.8 Global reachability analysis. 109

List of Tables

2.1 Dimensions that affect the cost/precision trade-offs of points-to analysis. 18

2.2 Representations of canonical statements for Java points-to analysis. For the method call statement, $\mathtt{f}_{\mathtt{m},\mathtt{i}}$ is the corresponding formal parameter of $\mathtt{p}_\mathtt{i}$ and $\mathtt{ret}_\mathtt{m}$ is the returned value. 21

3.1 Strengths and weaknesses of four demand-driven points-to analyses. 49

3.2 Benchmark statistics. Note that Column "O (objs)" is identical to Column "new". All of the numbers include the reachable parts of the Java library, determined using a call graph constructed on the fly with Andersen-style analysis [2] by Spark [31]. Column "*locality*" gives the ratio of local edges among all edges in a PAG. 53

3.3 Analysis times of NoRefine, RefinePTS and DynSum for the three clients: `SafeCast`, `NullDeref` and `FactoryM`. 56

4.1 Analysis times of `SafeCast` by RefinePTS and Emu. #M is the number of methods (in thousands) in Soot's context-insensitive call graph. #Q is the number of queries raised. 91

4.2 `NullDeref` and `FactoryM` in EMU. "Prec" denotes the precision at a budget of 75K against an exhaustive analysis. 95

List of Algorithms

1 Basic CFL-reachability-based analysis 28

2 RefinePTS's points-to analysis, SBPointsTo. 33

3 The RefinePTS analysis 34

4 PPTA-based summarisation 44

5 The DynSum analysis 46

6 Summarisation by local reachability analysis 77

7 Local reachability analysis 78

8 Summary update for code change 82

9 Summary-based points-to analysis 85

10 Global reachability analysis 87

Chapter 1

Introduction

1.1 Problem

Pointers are one of the most important features of imperative programming and have been widely supported in main-stream languages, such as C, C++, Java, Pascal and Fortran 90. The value of a pointer refers directly to (or *points to*) another value stored elsewhere in the memory using its address. Pointers are the key to effective memory handling, resource utilisation, and repetitive operations, such as traversing strings, tables and tree structures. It is often much efficient in time and space to copy and access the data using pointers.

The great benefits of pointers, however, are counterbalanced by the notorious difficulty of writing safe and effective programs with them. Because pointers can be directly manipulated, they are flexible and so can be difficult to control by developers. In order to understand program behaviours, program analysis is a key tool for diverse purposes such as optimisation [12, 31, 57], bug checking [18, 58], security detection [8,

15, 52] and many others. A fundamental and particularly useful program analysis is *points-to analysis* (or pointer analysis): a static program analysis to resolve the points-to set (denoted as $\texttt{Pts}(v)$) for a specific variable v, which contains all the possible memory locations that can be pointed by the pointer v. Points-to analysis is beneficial for many compiler optimisations and other analyses, since pointer information is a prerequisite for most program analyses. Without precise pointer information, most of these optimisations and analyses must be very conservative or unsound.

1.2 Challenges

While points-to analysis can be extremely useful in soundly checking the behaviour of pointers and in helping improve the quality of software systems, precise points-to analyses, like most static analyses, cannot scale to large software systems. In theory, points-to analysis is an undecidable problem. Even after making common approximations, e.g., modelling dynamic memory allocations conservatively and treating arrays monolithically, points-to analysis is still NP-hard [7]. Therefore, optimising analysis algorithms has become critically important in making points-to analysis practical in practice.

Over the past three decades or so, many research efforts have focussed on exploring the trade-off between precision and performance in developing practical points-to analysis. Points-to analysis is a well-studied research area with a large number of research papers published on this subject (e.g., an earlier survey [24] in 2001). To date, several research groups around the world have been actively working on the problem. Points-to analyses face two primary challenges:

- The first challenge is improving the scalability of points-to analysis without affecting precision. In all the approaches to points-to analysis, CFL-reachability has become a state-of-the-art technique in recent years, especially in handling on-demand queries. Some recent papers typically sacrifice some precision to obtain scalability. However, the software community is paying increasing attention to the precision of points-to analysis, especially for heap sensitivity and context sensitivity for object-oriented software systems. Therefore, exploring techniques that boost performance while maintaining precision is a demanding but difficult task.

- The second challenge is satisfying the needs of practical software, such as supporting client-driven, demand-driven analyses and handling program changes. Since points-to analysis is not a stand-alone task in practice, we need to tailor the analysis to satisfy the different needs of the clients. While software systems are evolving, it is also a challenging task to effectively compute points-to results in response to program changes. Each of these requirements is itself challenging and the separate literature in points-to analysis for some of them is sufficient. However, the studies on points-to analysis that effectively satisfies all the requirements are limited.

1.3 Approaches

This thesis presents two techniques to improve the state-of-the-art points-to analysis for Java based on CFL-reachability without affecting precision. Our analysis algorithms, which are scalable and precise, rely on some pervasive features observed from

practical software systems. The insight of our techniques is to exploit the *locality* in programs to enhance reusability. **Locality** is an important technique in computer science and a widely used term in many different research areas, such as cache optimisation and parallel computing. In this thesis, we exploit locality from two different aspects to show experimentally that our algorithms are effective in utilising locality of programs.

- **Locality of program statements**

  All the statements in a program can be categorised into two classes: local statements, which only involve local variables in a method, and global statements, which involve at least one variable outside the scope of the current method. We find that a large majority of statements are local in real-world applications. Figure 1.1 shows our statistics for the percentage of local and global statements in a number of Java programs. It can be seen that more than 80% of statements in these programs are local statements. As a result, the points-to analysis based on CFL-reachability can benefit from summarisation of local statements, which is the key insight for our first approach.

- **Locality of code changes**

  In response to program changes, especially incorporated into IDEs, most of current points-to analyses need to reanalyse a program from scratch. Several existing approaches are unbounded, i.e., the impact of a code change may propagate to the entire program in the worst case. The key insight is to utilise CFL-reachability-based summarisation to localise the impact of program changes made in a method.

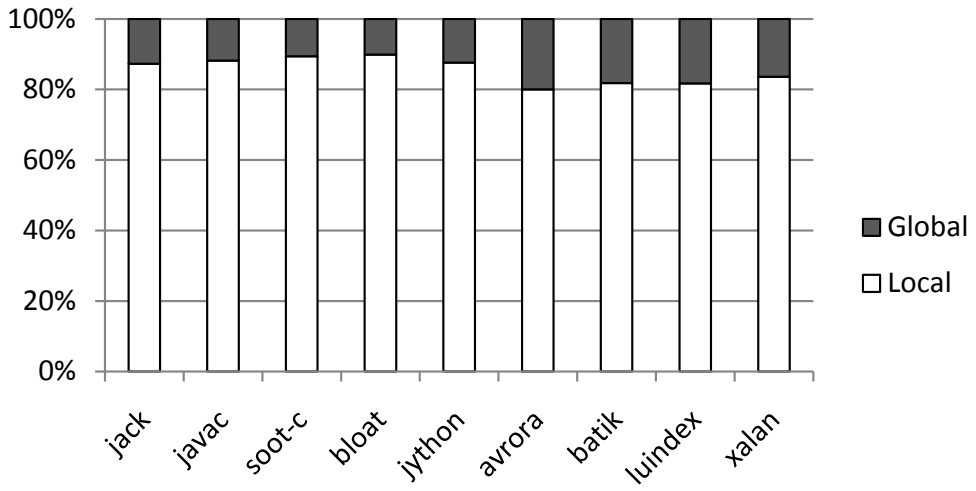


Figure 1.1: The percentage of local statements and global statements across our benchmarks. More than 80% of statements only are involved only with local variables within their own methods.

In order to tackle the challenges, we developed several techniques based on the insights above. Unlike recent studies, we firstly present an approach that exploits reuses by dynamic local reachability summaries, which improves the performance of points-to analysis without affecting precision. Furthermore, we extend the summarisation technique to limit the change impact in response to small code changes.

1.4 Contributions

This thesis makes the following contributions:

- **Dynamic summarisation for demand-driven points-to analysis**

  It presents a dynamic summarisation technique, called DynSum, to boost per-

formance of CFL-reachability based context-sensitive demand-driven points-to analysis by exploiting local reachability reuse. Our approach improves the performance of demand-driven points-to analysis without affecting precision and is *fully on-demand* without requiring any (costly) whole-program pre-analysis.

- **Incremental summarisation for supporting code changes in IDEs**

  It introduces a points-to analysis framework, called Emu, that enables developers to perform pointer-related queries while making changes to their programs in IDEs. Emu achieves its fast response times by adopting a modular approach to incrementally updating method summaries upon code changes: the points-to information in a method is summarised indirectly by CFL-reachability rather than directly by points-to sets, enabling the impact of code change made in a method to be localised within the method so that only the affected part of the method needs to be updated.

1.5 Outline

The rest of this thesis is organised as follows. Chapter 2 provides the general background and introduces CFL-reachability based points-to analysis for Java, including the program representation that serves to set up the foundation in which points-to analysis can be developed. We also present the basic and state-of-the-art context-sensitive points-to analysis based on CFL-reachability. Chapter 3 presents dynamic summarisation, a technique to boost the performance of state-of-the-art demand-driven points-to analysis without affecting precision. It successfully exploits the reuse from locality of statements and avoids the redundant traversals to local reachability

paths. Chapter 4 introduces a summary-based modular framework to handle small code changes in IDEs. The use of summarisation helps localise the impact of code changes and enables fast response time during program editing. Finally, Chapter 5 concludes the thesis with directions for future work.

Chapter 2

Background

The goal of this thesis is to develop techniques for points-to analysis to produce more efficient and practical solutions for Java programs. This chapter lays the foundations for our work and describes some preliminaries on points-to analysis via CFL-reachability. We present some background knowledges of points-to analysis in Section 2.1. Section 2.2 defines the notations and terminology that we use to represent Java program to facilitate CFL-reachability based points-to analysis. Then we present the basic CFL-reachability based points-to analysis algorithm in Section 2.3 and several important concepts are explained with examples whereas appropriate. Finally, we introduce some state-of-the-art approaches in optimisation and improvements of basic points-to analysis based on CFL-reachability in Section 2.4.

2.1 Points-to Analysis

Points-to analysis is widely used in static analysis tools [24]. It is especially important for object-oriented languages in which the extensive use of pointer-induced heap

Language Semantics	Precise Model	Imprecise Model
assignment	subset-based	equality-based
field accesses	field-sensitive	field-insensitive, field-based
control flows	flow-sensitive	flow-insensitive
method calls	context-sensitive	context-insensitive
heap abstractions	context-sensitive heap	context-insensitive heap

Table 2.1: Dimensions that affect the cost/precision trade-offs of points-to analysis.

accesses, subtyping and dynamic dispatch make it difficult to understand value-flow in a program. This section provides a brief introduction to some important points-to analysis terminology. A points-to analysis is to compute an over-approximation of the possible objects that a variable may point to during the execution of a program. Points-to targets are a finite set of *abstract locations*, represented as *points-to sets*, which are abstractions of runtime objects. For a variable x in a program P, we represent the points-to set of x as $\textsc{Pts}(\mathsf{x})$, and for each abstract location $\mathsf{o} \in \textsc{Pts}(\mathsf{x})$, x *may* point to some object represented by o in some particular execution of P.

In general, points-to analyses use various approximations of language semantics to balance precision and performance of the analysis. Table 2.1 summarises a number of orthogonal dimensions to model different language features, which introduces relevant terminology along the way. Note that the table only refers to limited but most important dimensions and approaches for points-to analysis.

In order to model assignment statements, there are *subset-based* [2] (or inclusion-based) analysis, which precisely handles an assignment $\mathsf{x} = \mathsf{y}$ by ensuring that $\textsc{Pts}(\mathsf{y}) \subseteq \textsc{Pts}(\mathsf{x})$ in the analysis result, and *equality-based* [56] (or unification-based)

analysis, which imprecisely ensures $\textsc{Pts}(\mathtt{x}) = \textsc{Pts}(\mathtt{y})$ given the same assignment on the contrary. While equality-based analysis can be efficiently computed at nearly linear time, subset-based analysis suffers from a worst-case cubic complexity. There exist some analysis, like one-level flow analysis [11], that makes trade-offs in between.

In order to model field accesses, there are *field-sensitive* analysis [41], which precisely distinguishes field access $\mathtt{x.f}$ by checking both base $\mathtt{x}$ and field $\mathtt{f}$, *field-insensitive* analysis, which only checks field base $\mathtt{x}$, and *field-based* analysis, which only checks field $\mathtt{f}$.

In order to model control flows, there are *flow-sensitive* analysis [21, 22, 26], which takes into account the order of statements and computes one points-to graph for each program point, and *flow-insensitive* analysis [4], which ignores control flow and produces a single result;

In order to model method calls, there are *context-sensitive* analysis [54, 61, 62], which precisely models method call semantics and computes only along realisable paths, and *context-insensitive* analysis [55], which approximately treats calls as goto instructions;

In order to model heap abstractions, there are context-sensitive heap abstraction, which distinguishes the contents of different data structure instances and represents objects of each data structure separately, and context-insensitive heap abstraction, which simply represents objects by its allocation statement ($\mathtt{malloc}$ in C or $\mathtt{new}$ in Java), causing merging of the contents of all instances in the analysis result. Moreover, some techniques can provide precision beyond context-sensitive heap abstraction. Iteration count abstraction (ICA) [35, 67] can prove properties of heap abstractions in loops. Shape analysis (e.g., [48]) is typically an expensive but more precise analysis

on heap and data structures.

Besides, there are also approximations on path sensitivity [13, 19, 59], array indexing [47] and other dimensions on precision.

Points-to analysis for Java is typically flow-insensitive, field-sensitive and context-sensitive (both method invocation and heap abstraction) to balance precision and efficiency [50, 54, 66]. That is also the setting and assumption used for all the points-to analyses for Java presented throughout this thesis.

2.2 Program Representation

In this section, we present the program representation of an abstraction of Java program that we use in subsequent sections to present points-to analysis. We adopt the similar program representation and notations from Sridharan's paper [54].

A *program* pro is defined as a set of *statements* stmt where each stmt is defined as a flow edge with a label. The syntax is given in Figure 2.1. In its canonical form, a Java program is represented by a *directed graph*, known as a *Pointer Assignment Graph* (PAG), which has threes types of nodes: objects, local variables and global variables. Since the analysis is flow-insensitive, control-flow statements are irrelevant. All edges are oriented in the direction of value flow, representing the statements in the program. Table 2.2 shows the simple transformation of canonical statements of Java into our graph representation. A method $\mathtt{m}$ is associated with the following seven types of edges:

(1) new, $\mathtt{n}_2 \xleftarrow{\mathsf{new}} \mathtt{n}_1$: $\mathtt{n}_1$ is an object created and $\mathtt{n}_2$ is a local variable, both in method $\mathtt{m}$, with $\leftarrow$ indicating the flow of $\mathtt{n}_1$ into $\mathtt{n}_2$. As a result, $\mathtt{n}_2$ points directly

Allocation sites	o	$\in$	O
Local variables	v	$\in$	V
Global variables	g	$\in$	G
Instance fields	f		
Call sites	i		
Nodes	n	$::=$	$o \mid v \mid g$
Labels	l	$::=$	$\mathsf{new} \mid \mathsf{assign} \mid \mathsf{ld}(f) \mid \mathsf{st}(f)$
			$\mid \mathsf{entry}_i \mid \mathsf{exit}_i \mid \mathsf{assignglobal}$
Statements	$stmt$	$::=$	$n \xleftarrow{l} n$
Programs	pro	$::=$	$\varnothing \mid pro \cup \{stmt\}$

Figure 2.1: An abstraction of Java programs.

Statement	Representation
s: $x = \texttt{new Obj}()$	$x \xleftarrow{\mathsf{new}} o_s$
$x = y$ $(x, y \in V)$	$x \xleftarrow{\mathsf{assign}} y$
$x = y.f$	$x \xleftarrow{\mathsf{ld}(f)} y$
$x.f = y$	$x \xleftarrow{\mathsf{st}(f)} y$
$x = y$ $(x \in G$ or $y \in G)$	$x \xleftarrow{\mathsf{assignglobal}} y$
s: $x = y.m(p_1, p_2, \ldots, p_k)$	$f_{m,i} \xleftarrow{\mathsf{entry}_s} p_i$ for $i \in 1..k$
	$\mathrm{this}_m \xleftarrow{\mathsf{entry}_s} y$, $x \xleftarrow{\mathsf{exit}_s} \mathrm{ret}_m$

Table 2.2: Representations of canonical statements for Java points-to analysis. For the method call statement, $f_{m,i}$ is the corresponding formal parameter of p_i and ret_m is the returned value.

to n_1.

(2) assign, $n_2 \xleftarrow{\text{assign}} n_1$: n_1 and n_2 are local variables in method m. So n_2 points to whatever n_1 points to.

(3) assignglobal, $n_2 \xleftarrow{\text{assignglobal}} n_1$: n_1 or n_2 or both are static variables. So n_2 points to whatever n_1 points to.

(4) ld(f), $n_2 \xleftarrow{\text{ld(f)}} n_1$: n_1 and n_2 are local variables in method m and f is an instance field for the load $n_2 = n_1.f$.

(5) st(f), $n_2 \xleftarrow{\text{st(f)}} n_1$: n_1 and n_2 are local variables in method m and f is an instance field for the store $n_2.f = n_1$.

(6) entry_i, $n_2 \xleftarrow{\text{entry}_i} n_1$: n_1 is a local variable in a caller that contains a call site at line i to method m, such that n_1 represents an actual parameter of the call and n_2 is its corresponding formal parameter of m. So n_2 points to whatever n_1 points to.

(7) exit_i, $n_2 \xleftarrow{\text{exit}_i} n_1$: n_1 is a local variable that contains a return value of method m and n_2 is a local variable assigned from n_1 at a call site i for m. So n_2 points to whatever n_1 points to.

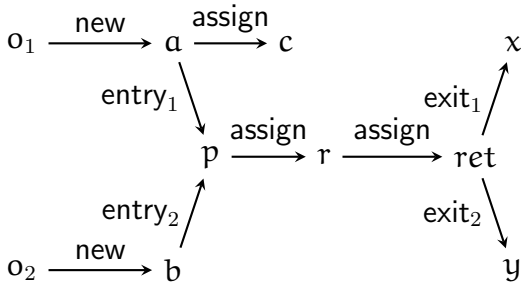
Loads and stores to array elements are modeled by collapsing all elements into a special field arr. By convention, it is assumed that no two classes (methods) contain the same identically named global (local) variable.

Figure 2.2 gives a simple Java example and its graph representation. On the left-hand side of Figure 2.2, in lines 1 – 2, two objects are created and then they are passed as parameters to method calls in lines 4 – 5. There is also a simple assignment in line 3. The definition of method id is shown in line 6.

For this example, its PAG is shown on the right-hand side in Figure 2.2. Some notations are in order: (1) o_i denotes the abstract object o created at the allocation

```
1  a = new A();   // o1
2  b = new A();   // o2
3  c = a;
4  x = id(a);  // call 1
5  y = id(b);  // call 2
6  id(p){r = p; return r;}
```

(a) program code



(b) graph representation

Figure 2.2: A small code example and its graph representation (PAG) for points-to analysis.

site in line i, and (2) ret is a temporary variable introduced in line 6 representing the return variable of the method. In next section, we would like to use this example to explain the basic points-to analysis based on CFL-reachability.

2.3 CFL-Reachability based Points-to Analysis

The points-to analyses presented in this thesis, which differ from most traditional points-to analyses, use *context-free language reachability* (CFL-reachability) as their underlying foundation. The program analysis using CFL-reachability is firstly presented by Reps et.al. [44, 45] more than ten years ago. The state-of-the-art CFL-reachability points-to analysis [54, 66, 67, 70] for Java is both context-sensitive (for both method calls and heap abstractions) and field-sensitive. In this section, we introduce the basic CFL-reachability based points-to analysis for Java. The recent

progress presented in this paper and some related work can be considered as the optimisation of the basic analysis. The basic analysis provides the baseline for us to enhance its performance using several new techniques.

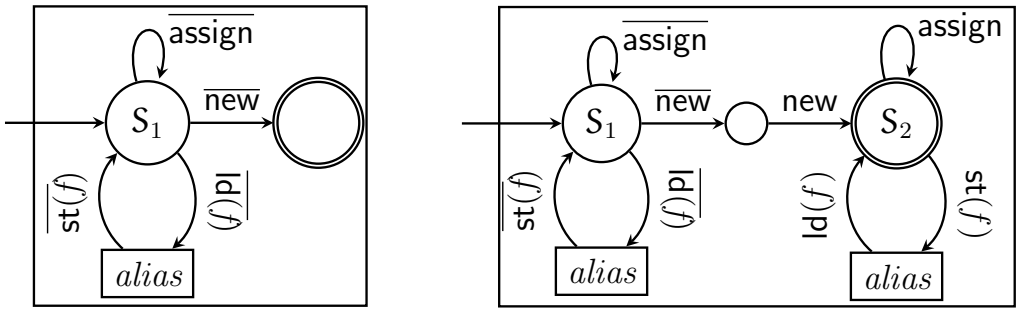
2.3.1 CFL-Reachability

Context-free language (CFL) reachability [44, 71] is an extension of graph reachability that is equivalent to the reachability problem formulated in terms of either recursive state machines (RSMs) [9] or set constraints [27]. Let G be a directed graph whose edges are labeled by symbols from an alphabet Σ. Let L be a CFL over Σ. Each path p in G has a string $w(\mathsf{p})$ in Σ^* formed by concatenating in order the labels of edges in p. A node u is L-*reachable* from a node v if there exists a path p from v to u, called an L-*path*, such that $w(\mathsf{p}) \in \mathsf{L}$.

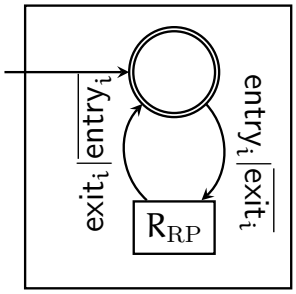
CFL-reachability is computationally more expensive to solve than the standard graph reachability. In the case of the single-source L-path problem, which requires finding all nodes L-reachable from a source node n in a graph G, the worst-case time complexity is $O(\Gamma^3 N^3)$, where Γ is the size of a normalised grammar for L and N is the number of nodes in G [44]. Therefore, we are motivated to exploit reachability reuse to lower its analysis overhead in this thesis.

2.3.2 Field Sensitivity

We discuss how to perform field-sensitive points-to analysis without considering context sensitivity in CFL-reachability. A context-insensitive analysis merges information from different calls of a method rather than reasoning about each call separately. As a result, global assignment, call entry or call exit edges are all treated as local



(a) $\mathsf{L_{FT}}$ (with $\overline{\mathit{flowsTo}}$ shown on the left and *alias* on the right)



(b) $\mathsf{R_{RP}}$

Figure 2.3: Recursive State Machines (RSMs) for $\mathsf{L_{FT}}$ and $\mathsf{R_{RP}}$.

assignment edges. Given a program, its PAG is thus simplified to possess only four types of local edges: new, assign, ld and st.

Let us first consider a PAG G with only new and assign. It suffices to develop a regular language, $\mathsf{L_{FT}}$ (FT for flows-to), such that if an object o can flow to a variable v during the execution of the program, then v will be $\mathsf{L_{FT}}$-reachable from o in G. Let *flowsTo* be the start symbol of $\mathsf{L_{FT}}$. Then we have the following (regular) grammar for $\mathsf{L_{FT}}$:

$$\mathit{flowsTo} \quad \rightarrow \quad \mathsf{new}\,(\,\mathsf{assign})^* \tag{2.1}$$

If o *flowsTo* v, then v is $\mathrm{L_{FT}}$-reachable from o. Thus, we know that o belongs to the points-to set of v.

For field accesses, precise handling of heap accesses is formulated with the updated $\mathrm{L_{FT}}$ being a CFL of *balanced parentheses* [55]. Two variables x and y may be aliases if an object o may flow to both x and y. Thus, v may point to o if there exists a pair of statements $\mathsf{p.f} = \mathsf{q}$ and $\mathsf{v} = \mathsf{u.f}$, such that the base variables p and u can be aliases. So o flows through the above two statements with a pair of parentheses (i.e., $\mathsf{st(f)}$ and $\mathsf{ld(f)}$), first into q and then into v. Therefore, the *flowsTo* production is extended into:

$$\mathit{flowsTo} \quad \rightarrow \quad \mathsf{new}\ (\ \mathsf{assign} \mid \mathsf{st(f)}\ \mathit{alias}\ \mathsf{ld(f)})^* \tag{2.2}$$

where x *alias* y means that x and y could be aliases. To allow *alias* paths in an *alias* language, $\overline{\mathit{flowsTo}}$ is introduced as the inverse of the *flowsTo* relation. A *flowsTo*-path p can be inverted to obtain its corresponding $\overline{\mathit{flowsTo}}$-path $\overline{\mathsf{p}}$ using inverse edges, and vice versa. For each edge $\mathsf{x} \xleftarrow{\ell} \mathsf{y}$ in p, its inverse edge is $\mathsf{y} \xleftarrow{\bar{\ell}} \mathsf{x}$ in $\overline{\mathsf{p}}$. (To avoid cluttering, the inverse edges in a PAG, such as the one given in Figure 3.2, are not shown explicitly.) Thus, o *flowsTo* x iff x $\overline{\mathit{flowsTo}}$ o. This means that $\overline{\mathit{flowsTo}}$ actually represents the standard points-to relation. As a result, x *alias* y iff x $\overline{\mathit{flowsTo}}$ o *flowsTo* y for some object o. Thus, the *alias* language is defined by:

$$\begin{aligned} \mathit{alias} \quad &\rightarrow \quad \overline{\mathit{flowsTo}}\ \mathit{flowsTo} \\ \overline{\mathit{flowsTo}} \quad &\rightarrow \quad (\ \overline{\mathsf{assign}} \mid \overline{\mathsf{ld(f)}}\ \mathit{alias}\ \overline{\mathsf{st(f)}})^*\ \overline{\mathsf{new}} \end{aligned}$$

Our final CFL $\mathrm{L_{FT}}$ for finding the points-to set of a variable consists of the productions given in (2.2) and (2.3) with $\overline{\mathit{flowsTo}}$ as its start symbol. For convenience, we often write *pointsTo* to mean $\overline{\mathit{flowsTo}}$.

The RSMs [9] for *pointsTo* and *alias* are shown in Figure 2.3(a); they will be referred later to facilitate the understanding of DynSum in Chapter 3.

2.3.3 Context Sensitivity

A call entry or exit edge is treated as an assign edge as before in L_{FT} to represent parameter passing and method return but assign and assignglobal edges are now distinguished.

A context-sensitive analysis requires call entries and exits to be matched, which is solved also as a balanced-parentheses problem [44, 46]. This is done by filtering out *flowsTo*- and $\overline{flowsTo}$-paths corresponding to unrealisable paths. The following CFL R_{RP} (RP for realisable paths) is used to describe all realisable paths in a PAG G; its RSM is given in Figure 2.3(b):

$$\begin{aligned} C &\rightarrow \mathsf{CallEntry_i}\ C\ \mathsf{CallExit_i} \mid C\ C \mid \epsilon \\ \mathsf{CallEntry_i} &\rightarrow \mathsf{entry_i} \mid \overline{\mathsf{exit_i}} \\ \mathsf{CallExit_i} &\rightarrow \mathsf{exit_i} \mid \overline{\mathsf{entry_i}} \end{aligned}$$

When traversing a *flowsTo*-path in G, entering a method via $\mathsf{entry_i}$ from call site i requires exiting from that method back to call site i via either (1) $\mathsf{exit_i}$ to continue its traversal along the same *flowsTo*-path or (2) $\overline{\mathsf{entry_i}}$ to start a new search for a $\overline{flowsTo}$-path. The situation for entering a method via $\overline{\mathsf{exit_i}}$ when traversing a $\overline{flowsTo}$-path is reversed.

2.3.4 Basic CFL-Reachability-based Points-to Analysis

With the formulations given above, we give the basic CFL-reachability based points-to analysis in Algorithm 1, which is to compute CFL-reachability for the CFL

Algorithm 1 Basic CFL-reachability-based analysis

PointsTo (v, c)

```
1: pts ← ∅
2: for each v ←new— o do
3:    pts ← pts ∪ {(o, c)}
4: for each v ←assign— x do
5:    pts ← pts ∪ PointsTo (x, c)
6: for each v ←assignglobal— x do
7:    pts ← pts ∪ PointsTo (x, ∅)
8: for each v ←exit_i— x do
9:    pts ← pts ∪ PointsTo (x, c.Push(i))
10: for each v ←entry_i— x do
11:    if c.Peek() = i or c = ∅ then
12:       pts ← pts ∪ PointsTo (x, c.Pop())
13: for each v ←ld(f)— u do
14:    for each q ←st(f)— p do
15:       CSalias ← ∅
16:       for (o, c′) ∈ PointsTo (u, c) do
17:          CSalias ← CSalias ∪ FlowsTo (o, c′)
18:       for (r, c″) ∈ CSalias do
19:          if r = q then
20:             pts ← pts ∪ PointsTo (p, c″)
21: return pts
```

$L_{pts} = L_{FT} \cap R_{RP}$. This is done by tracking the state of R_{RP} for each explored path while computing L_{FT} reachability. As we focus on computing *pointsTo*, i.e., $\overline{flowsTo}$ in the formulation, a state represents a calling context, which is typically a finite stack configuration corresponding to $\mathsf{CallEntry}_i$ edges. Note that the FlowsTo procedure, which is not given, is analogous to PointsTo but traverses edges in the opposite direction. By convention, standard stack operations, including Push(), Pop() and

Peek(), are used in the algorithm.

Given a variable v and a call stack c, PointsTo(v, c) computes $pointsTo(v, c)$, i.e., the points-to set of v in context c. It traverses edges in the reverse direction. Note that for each *flowsTo* edge $x \xleftarrow{\ell} y$, its inverse $\overline{flowsTo}$ edge is $y \xleftarrow{\bar{\ell}} x$. Therefore, traversing from x to y along $x \xleftarrow{\ell} y$ in reverse direction means traversing from x to y along $y \xleftarrow{\bar{\ell}} x$. The check for $c = \varnothing$, i.e, ϵ in line 11 allows for partially balanced parentheses (a prefix with unbalanced closed parentheses and a suffix with unbalanced open parentheses) since a realisable path may not start and end in the same method.

All the transitions strictly follow the formulation and RSM presented above. In the example given in Figure 2.2, we can conclude that c points to o_1 because there is a CFL-path from o_1 to c: $c \xleftarrow{\text{new}} a \xleftarrow{\text{assign}} o_1$.

PointsTo is context-sensitive for method invocation by matching call entries and exits and also for heap abstraction by distinguishing allocation sites with calling contexts propagating objects only along realisable paths. In our example in Figure 2.2, there are two abstract objects flowing to variable x: one is $x \xleftarrow{\text{exit}_1} ret \xleftarrow{\text{assign}} r \xleftarrow{\text{assign}} p \xleftarrow{\text{entry}_1} a \xleftarrow{\text{new}} o_1$, and the other is $x \xleftarrow{\text{exit}_1} ret \xleftarrow{\text{assign}} r \xleftarrow{\text{assign}} p \xleftarrow{\text{entry}_2} b \xleftarrow{\text{new}} o_2$. However, variable x can only point to abstract object o_1, since entry_1 and exit_1 matches on the first path, but entry_2 and exit_1 does not on the second.

Global variables are context-insensitive. As a result, the R_{RP} state is cleared across assignglobal edges (lines 6 and 7 in Algorithm 1). Thus, these edges "skip" the sequence of calls and returns between the reads and writes of a global variable.

2.3.5 The Benefits of CFL-Reachability Approach

CFL-reachability based analyses have been extensively studied [27, 55, 66, 67, 70, 74] in recent years. It has become a state-of-the-art technique in points-to analysis, especially in answering on-demand queries. There are several significant benefits compared to traditional points-to analysis based on constraint resolution.

- First, by formulating a points-to analysis problem as a CFL-reachability problem, we can discover some key insights behind the analysis. Therefore, CFL-reachability is powerful to model points-to relations with context sensitivity and field sensitivity. Moreover, it can also be applied to a broader context than points-to relations. For example, it is also used to formulate container semantics to detect memory problems [65] recently.

- Second, CFL-reachability is a good and natural solution for *demand-driven* analysis, by only computing results for specific variables (*queries*) issued by clients. Most traditional points-to analyses are *exhaustive*, analysing for all variables in the program when only answering a small number of queries. Existing demand-driven work, based on set constraints, e.g., the work of Heintze and Tardieu [23], is formulated through elaborate deductive rules and may not be easily applicable in dealing with context, field and heap sensitivity. In contrast, a CFL-reachability formulation of points-to relations automatically leads to demand algorithms for points-to analysis. The formulation enforces that only the statements along the CFL-path are necessary for queried variables. Note that demand-driven analysis alone does not yield a scalable analysis for Java [54]. The key to scalable and precise analysis is to exploit techniques to

improve the performance of CFL-reachability based analysis [50, 54, 66].

- Moreover, the analysis based on CFL-reachability can compute the most precise flow-insensitive points-to results for a Java program. CFL-reachability can effectively model the semantics of method calls and heap allocation with calling contexts. In general, context sensitivity can be formulated as a *balanced parenthesis* problem [45, 46]. Furthermore, CFL-reachability is especially suitable for Java. Unlike C, context sensitivity can provide a large precision benefit for Java [33] and balanced parentheses can also be used to model field sensitivity due to the type-safety in Java. In terms of the precision dimensions introduced in Section 2.1, the state-of-the-art CFL-reachability based analysis is the most precise in handling assignments, fields, method calls and heap allocation.

2.4 State-of-the-Art

While the basic context-sensitive points-to analysis based CFL-reachability cannot scale to some large programs, there are several recent approaches aiming to improve its performance. Some representative state-of-the-art approaches are simply described as below.

2.4.1 Refinement

Now, we introduce the work of Sridharan and Bodík [54] which is a context-sensitive demand-driven points-to analysis for Java formulated in terms of CFL-reachability. Their approach is denoted as RefinePTS in this thesis.

Sridharan et al. [54, 55] were the first to formulate the points-to analysis for

Java as computing CFL-reachability on a PAG. To make context-sensitive points-to analysis scalable, their work [54, 55] introduces a so-called refinement technique, an important optimisation for points-to analysis that approximates field sensitivity with a field-based approach. The authors propose the refinement technique to satisfy clients with imprecise resolutions without computing fully field-sensitive points-to sets, but the approach is artificially restricted answering some queries from a client and may incur pure overhead if the client cannot be satisfied in some sequence of refinement steps.

We give the algorithms of their work in Algorithm 2 and Algorithm 3. To support iterative refinements, RefinePTS operates with a refinement loop, which is simplified in Algorithm 3 to avoid the complications in dealing with points-to cycles. For more details, see [54, 55]. Given a points-to query, an initial approximation with a field-based analysis is adopted and then gradually refined until the client is satisfied. In lines 13 and 14, the base variables $\mathtt{u}$ and $\mathtt{q}$ are assumed to be aliases, if $\nu \xleftarrow{\mathsf{ld(f)}} \mathtt{u}$ is not in *fldsToRefine*, a set controlling the refinement. In this case, an artificial match edge $\nu \xleftarrow{\mathsf{match}} \mathtt{p}$ is considered to have been introduced. By moving directly from ν to $\mathtt{p}$, a sequence of calls and returns between the read and write of field $\mathtt{f}$ can be skipped. Hence, the state of R_{RP} is cleared (line 17). If satisfyClient(pts) returns false, then another refinement iteration is needed. All encountered match edges are removed, and the analysis becomes field-sensitive for each such match edge, $\nu \xleftarrow{\mathsf{match}} \mathtt{p}$, so that the paths between their endpoints are explored. This may lead to new match edges to be discovered and further refined until either a pre-set budget is exceeded or the query has been answered (lines 29 and 30).

Algorithm 2 RefinePTS's points-to analysis, SBPointsTo.

SBPointsTo (v, c)

1: pts $\leftarrow \varnothing$
2: **for** each $v \xleftarrow{\text{new}} o$ **do**
3: pts $\leftarrow$ pts $\cup \{(o, c)\}$
4: **for** each $v \xleftarrow{\text{assign}} x$ **do**
5: pts $\leftarrow$ pts $\cup$ SBPointsTo (x, c)
6: **for** each $v \xleftarrow{\text{assignglobal}} x$ **do**
7: pts $\leftarrow$ pts $\cup$ SBPointsTo (x, $\varnothing$)
8: **for** each $v \xleftarrow{\text{exit}_i} x$ **do**
9: pts $\leftarrow$ pts $\cup$ SBPointsTo (x, c.Push(i))
10: **for** each $v \xleftarrow{\text{entry}_i} x$ **do**
11: **if** c.Peek() = i or c = $\varnothing$ **then**
12: pts $\leftarrow$ pts $\cup$ SBPointsTo (x, c.Pop())
13: **for** each $e = v \xleftarrow{\text{ld}(f)} u$ **do**
14: **for** each $q \xleftarrow{\text{st}(f)} p$ **do**
15: **if** $e \notin$ *fldsToRefine* **then**
16: *fldsSeen* $\leftarrow$ *fldsSeen* $\cup \{e\}$
17: pts $\leftarrow$ pts $\cup$ SBPointsTo ($p, \varnothing$)
18: **else**
19: CSalias $\leftarrow \varnothing$
20: **for** $(o, c') \in$ SBPointsTo (u, c) **do**
21: CSalias $\leftarrow$ CSalias $\cup$ SBFlowsTo(o, c')
22: **for** (r, c'') $\in$ CSalias **do**
23: **if** $r = q$ **then**
24: pts $\leftarrow$ pts $\cup$ SBPointsTo (p, c'')
25: **return** pts

2.4.2 Symbolic Graphs

A different approach that primarily targets the scalability challenge analyses a program using a pre-computed symbolic graph, which has been used with a great success

Algorithm 3 The RefinePTS analysis

RefinePTS(v)

```
1: while true do
2:   fldsSeen ← ∅
3:   pts ← SBPointsTo (v, ∅)
4:   if satisfyClient(pts) then
5:     return true
6:   else
7:     if fldsSeen = ∅ then
8:       return false
9:     else
10:      fldsToRefine ← fldsToRefine ∪ fldsSeen
```

in model checking [3] and memory bloat detection [65]. This approach replaces a standard PAG with an approximate symbolic graph, which is generally smaller than the original PAG. A number of papers [65, 66, 67, 70] have shown that the analysis using symbolic graphs can greatly improve the performance of alias analysis for Java.

We briefly discuss the pros and cons of using symbolic graphs for points-to analysis. Xu et al. [66] proposed a program representation, which is referred to as the interprocedural symbolic points-to graph (ISPG). The construction of a symbolic points-to graph is carried out by pre-computing the relationship between object nodes. The symbolic graph abstracts away variable nodes, and partitions the heap using symbolic and allocation nodes. Using this representation, it is possible to disambiguate alias relations with a lower cost as this representation contains fewer nodes and edges than a PAG used by CFL-reachability points-to analysis. The complexity of the CFL-reachability computation depends only on the size of the symbolic graph, not the number of variables in the program. Therefore, the analysis using symbolic graphs

takes advantage of these features to perform analysis that would be less expensive both in time and memory.

The symbolic graph approach benefits from its smaller graph size, however it may introduce some imprecision to the points-to analysis for some clients. As a result, it may not be well-suited to distinguishing individual variables or memory locations of a symbolic node. This problem may not be that serious when used to compute alias relations, but it may directly impact the results of points-to analysis. The points-to information computed during the analysis depends on the individual characteristics of the points-to sets and it may make a difference whether a symbolic node represents a single memory location or multiple memory locations. Furthermore, symbolic graphs differ from PAGs not only in terms of symbolic nodes, but also in how load and store edges are merged. Therefore, the analysis based on symbolic graphs conservatively approximates the original PAGs and consequently lead to some precision loss.

Chapter 3

Dynamic Summary-Based Demand-Driven Analysis

This chapter presents our on-demand dynamic summary-based points-to analysis. The analysis dynamically exploits local reachability reuse to improve the performance of CFL-reachability based demand-driven points-to analysis. We have implemented the algorithm in the Soot compiler and evaluated it with a suite of Java programs.

3.1 Overview

Many static analyses can be accelerated if some redundant computations can be avoided. Considerable progress has been made, resulting in, for example, cycle elimination [14, 20] for Andersen-style points-to analysis [2] and sparse analysis [21, 22, 72] for flow-sensitive points-to analysis. In the case of context-sensitive points-to analysis, computing a points-to summary for a method [40, 62, 72] avoids re-summarising it unnecessarily for the same and different calling contexts. Despite many earlier efforts,

it remains unclear how to craft points-to analyses that can efficiently answer demand queries (e.g., non-aliasing) for a specific client.

The majority of the current solutions perform a whole-program points-to analysis to improve precision at the expense of efficiency, by computing points-to information for all variables in the program. Such exhaustive algorithms are too resource-intensive to be useful in environments with small time budgets, such as just-in-time (JIT) compilers and IDEs. One widely acceptable observation is that points-to analysis is not a stand-alone task since it needs to be tailored to suit the specific needs of a client application. As a result, much recent work [54, 55, 66, 74] has focussed on *demand-driven* points-to analysis, which mostly relies on CFL-reachability [44] to perform only the necessary work for a set of variables specified by a client rather than a whole-program analysis to find all its points-to information.

To perform points-to analysis with CFL-reachability, a program is represented as a PAG as we discussed in Chapter 2, with nodes denoting variables/objects and edges pointer-manipulating statements. Determining if a variable v points to an object o requires finding a path p between the nodes v and o in the graph such that p's label is in a CFL that ensures the corresponding statements can cause v to point to o. To balance precision and efficiency for on-demand queries, a points-to analysis is typically flow-insensitive, field-sensitive and context-sensitive [54]. Context sensitivity is realised as a balanced-parentheses problem along two axes: method invocation (by matching call entries and exits so that only realisable paths are considered) and heap abstraction (by distinguishing the same abstract object from different paths).

While CFL-reachability formulation introduced in Section 2.3 automatically leads to a demand-driven approach, performing basic CFL-reachability based points-to

analysis, which is introduced in Algorithm 1, for large, complex software can still be costly, especially when a client issues a large number of queries. Existing state-of-the-art solutions [54, 66, 74] discussed in Section 2.4 have addressed the performance issue by using some kind of approximations. However, redundant traversals along the same path are still repeatedly made, unless they are identified by a time-consuming whole-program pre-analysis.

In this chapter, we introduce a novel technique, called DynSum, to perform context-sensitive demand-driven points-to analysis fully on-demand. Unlike existing techniques [54, 66, 74], our approach exploits local reachability reuse by performing a *Partial Points-To Analysis* (PPTA) within a method dynamically. PPTA is field-sensitive but context-independent, thereby enabling the summarised points-to relations in a method to be reused in its different calling contexts without any precision loss. We identify such reuse as a practical basis for developing an effective optimisation for demand-driven points-to analysis.

In this chapter, we make the following contributions:

- Our dynamic approach improves the performance of demand-driven points-to analysis without affecting precision and is *fully on-demand* without requiring any (costly) whole-program pre-analysis. This appears to be the first points-to analysis that computes dynamic method summaries to answer demand queries.

- We have implemented DynSum in the Soot compiler framework for Java. We have used three representative clients (safe casting, null dereferencing and factory methods) to evaluate the performance improvements against RefinePTS, the state-of-the-art demand-driven points-to analysis introduced in [54]. The average speedups achieved by DynSum for the three clients over a suite of nine

Java benchmarks are 1.95×, 2.28× and 1.37×, respectively.

- We show that DynSum computes only a small percentage of the summaries computed by StaSum, a static whole-program analysis [70]. This makes DynSum more scalable and better-suited for answering demand queries in low budget environments.

3.2 A Motivating Example

In this section, we illustrate the problem using an example. Figure 3.1 gives a Java program, which represents the most common operations of a `Vector` container. Its PAG representation is shown in Figure 3.2.

To avoid cluttering, the labels "assign" and "assignglobal" for assignment edges are omitted. Note that $\mathtt{o}_\mathtt{i}$ denotes the object created at the allocation site in line $\mathtt{i}$ and $\mathtt{v}_\mathtt{m}$ (with a subscript) denotes variable $\mathtt{v}$ declared in method $\mathtt{m}$.

We explain how RefinePTS works by using it to compute the points-to sets for s1 and s2 in Figure 3.2. We motivate the need for local reachability reuse in DynSum in Section 3.3.

Consider RefinePTS(s1) first. To fully resolve its points-to set, the following four iterations are performed:

1. Initially, RefinePTS starts being field-based since *fldsSeen* = *fldsToRefine* = $\varnothing$. In this first iteration, due to the existence of the match edge, $\mathtt{p} \xrightarrow{\mathsf{match}} \mathtt{ret}_{\mathrm{get}}$, we find that $\mathrm{SBPointsTo}(\mathtt{s1}, \varnothing) = \{\mathtt{o}_{26}, \mathtt{o}_{29}\}$ since there are two *flowsTo*-paths: (1) $\mathtt{o}_{26} \xrightarrow{\mathsf{new}} \mathtt{tmp1} \xrightarrow{\mathsf{entry}_{26}} \mathtt{p} \xrightarrow{\mathsf{match}} \mathtt{ret}_{\mathrm{get}} \xrightarrow{\mathsf{exit}_{22}} \mathtt{ret}_{\mathrm{retrieve}} \xrightarrow{\mathsf{exit}_{32}} \mathtt{s1}$ and (2) $\mathtt{o}_{29} \xrightarrow{\mathsf{new}} \mathtt{tmp1} \xrightarrow{\mathsf{entry}_{29}} \mathtt{p} \xrightarrow{\mathsf{match}} \mathtt{ret}_{\mathrm{get}} \xrightarrow{\mathsf{exit}_{22}} \mathtt{ret}_{\mathrm{retrieve}} \xrightarrow{\mathsf{exit}_{32}}$

```
 1 class Vector{                    18 void set(Vector v)
 2   Object[] elems;                19   { this.vec=v; }
 3   int count;                     20   Object retrieve()
 4   Vector(){                      21   { t=this.vec;
 5     t=new Object[8];             22     return t.get(0); }}
 6     this.elems=t;}               23 class Main{
 7   void add(Object p){            24 static void main(...){
 8     t=this.elems;                25   Vector v1=new Vector();
 9     t[count++]=p;}               26   v1.add(new Integer(1));
10   Object get(int i){             27   Client c1=new Client(v1);
11     t=this.elems;                28   Vector v2=new Vector();
12     return t[i]; }}              29   v2.add(new String());
13 class Client{                    30   Client c2=new Client();
14   Vector vec;                    31   c2.set(v2);
15   Client() {}                    32   s1=c1.retrieve();
16   Client(Vector v)               33   s2=c2.retrieve();}
17   { this.vec=v; }                34 }
```

Figure 3.1: A motivating example.

s1.

2. In the second iteration, RefinePTS starts with *fldsToRefine* = $\{t_{get} \xrightarrow{\mathsf{ld}(arr)} ret_{get}\}$. There are two new match edges found: $t_{Vector} \xrightarrow{\mathsf{match}} t_{get}$ and $t_{Vector} \xrightarrow{\mathsf{match}} t_{add}$. As $t_{add} \xleftarrow{\mathsf{match}} t_{Vector} \xleftarrow{\mathsf{new}} o_5 \xrightarrow{\mathsf{new}} t_{Vector} \xrightarrow{\mathsf{match}} t_{get}$, t_{add} and t_{get} are found to be aliases. Thus, SBPointsTo(s1, $\varnothing$) = $\{o_{26}, o_{29}\}$ remains unchanged.

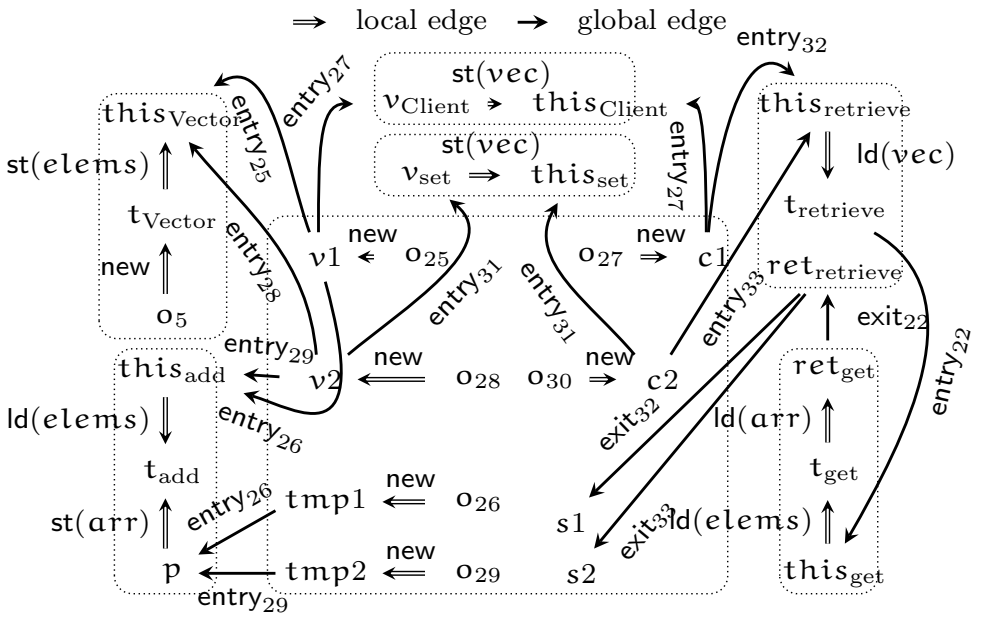


Figure 3.2: PAG for the example given in Figure 3.1.

3. In the third iteration, RefinePTS continues to refine the two new match edges discovered in the second iteration. SBPointsTo starts its traversal from s1 along the right part of the graph. Initially, $\mathsf{R}_{\mathrm{RP}} = [\![\ ]\!]$. On encountering $\overline{\mathsf{exit}_{32}}$ and $\overline{\mathsf{exit}_{22}}$, the analysis pushes their call sites into the context stack at node $\mathsf{ret}_{\mathrm{get}}$: $\mathsf{R}_{\mathrm{RP}} = [\![32, 22]\!]$. Then it arrives at $\mathsf{t}_{\mathrm{retrieve}}$ after having popped the stack once so that $\mathsf{R}_{\mathrm{RP}} = [\![32]\!]$. Traversing along another two new match edges, $\mathsf{t}_{\mathrm{retrieve}} \xleftarrow{\mathsf{match}} v_{\mathrm{Client}}$ and $\mathsf{t}_{\mathrm{retrieve}} \xleftarrow{\mathsf{match}} v_{\mathrm{set}}$, RefinePTS will next explore from v_{Client} and v_{set}, one by one. As both o_{25} and o_{28} can flow to $\mathsf{this}_{\mathrm{Vector}}$ and $\mathsf{this}_{\mathrm{add}}$, so $\mathsf{this}_{\mathrm{Vector}}$ and $\mathsf{this}_{\mathrm{add}}$ are aliases. So once again $\mathrm{SBPointsTo}(\mathsf{s1}, \varnothing) = \{\mathsf{o}_{26}, \mathsf{o}_{29}\}$ is the same as before.

4. In the last iteration, RefinePTS continues to refine the two new match edges

discovered in the third iteration. Due to context sensitivity, only the edge $\mathsf{this}_{\text{retrieve}} \xleftarrow{\mathsf{entry}_{32}} \mathsf{c}1$ is realisable because $\overline{\mathsf{entry}_{32}}$ matches the top of context stack $[\![32]\!]$ but $\mathsf{this}_{\text{retrieve}} \xleftarrow{\mathsf{entry}_{33}} \mathsf{c}2$ does not. Therefore, $\mathsf{this}_{\text{Client}}$ and $\mathsf{this}_{\text{retrieve}}$ may be aliases. So SBPointsTo will eventually visit o_{26} and obtain the final solution: SBPointsTo $(\mathsf{s}1, \varnothing) = \{\mathsf{o}_{26}\}$.

Similarly, s_2 is resolved. However, RefinePTS will traverse redundantly a few paths that it did before in resolving s_1 in order to conclude that SBPointsTo $(\mathsf{s}2, \varnothing) = \{\mathsf{o}_{29}\}$.

3.3 The DynSum Analysis

While RefinePTS may bring benefits for some clients, our motivating example has exposed several of its limitations:

- The same paths can be traversed multiple times for a set of queries under the same or different calling contexts. This problem becomes more severe as modern software relies heavily on common libraries (e.g., Java JDK).

- Ad hoc caching techniques [54, 66, 74] are ineffective for three reasons. First, SBPointsTo(ν, c) cannot be cached unless it is fully resolved within a pre-set budget. Second, the cached SBPointsTo(ν, c) can only be reused in the same context c. When resolving SBPointsTo$(\mathsf{s}1, \varnothing)$ and SBPointsTo$(\mathsf{s}2, \varnothing)$ previously, the points-to set of $\mathsf{ret}_{\text{get}}$ is computed twice, once for $[\![32, 22]\!]$ and once for $[\![33, 22]\!]$. As a result, the same path from $\mathsf{ret}_{\text{get}}$ to $\mathsf{this}_{\text{get}}$ is still redundantly traversed for such different contexts. Finally, caching and refinement

may be incompatible as a cached points-to set may depend on the match edges encountered when the points-to set was computed.

- All field-based refinement iterations are pure overhead before a client can be satisfied with a particular query. This "lazy" strategy is not well-suited for clients that require precise points-to or aliasing information.

In this chapter, we propose to overcome these limitations by giving up refinement and relying on exploiting local reachability reuse to efficiently answer demand queries. As shown in Figure 3.2, we categorise the two types of edges in a PAG: *local edges* (new, assign, ld and st) and *global edges* (assignglobal, entry_i and exit_i). The key observation is that local edges have no effects on the context of a query while global edges have no effects on its field sensitivity.

Therefore, our DynSum analysis is broken down into two parts. DSPointsTo given in Algorithm 4 performs a *partial points-to analysis* (PPTA) on-the-fly for a queried variable to summarise its points-to relations along the local edges within a method field-sensitively but context-independently. DynSum in Algorithm 5 handles the context-dependent global edges while collaborating with PPTA to compute new summaries if they are unavailable for reuse.

3.3.1 PPTA: Partial Points-To Analysis

It is easy to understand what PPTA is in terms of the RSMs given in Figure 2.3, as the two RSMs (for *pointsTo* and *alias*) in Figure 2.3(a), which are together equivalent to L_{FT}, handle field sensitivity, and the RSM for R_{RP} shown in Figure 2.3(b) handles context sensitivity.

Algorithm 4 PPTA-based summarisation

DSPointsTo (v, f, s)

```
1: ppts ← ∅
2: if s = S1 then
3:   for each v ←new o do
4:     if f = ∅ then
5:       ppts ← ppts ∪ { o }
6:     else
7:       ppts ← ppts ∪ DSPointsTo (v, f, S2)
8:   for each v ←assign x do
9:     ppts ← ppts ∪ DSPointsTo (x, f, S1)
10:  for each v ←ld(g) x do
11:    ppts ← ppts ∪ DSPointsTo (x, f.Push(ḡ), S1)
12:  if v ←exit/entry/assignglobal n then
13:    ppts ← ppts ∪ { (v, f, S1) }
14: if s = S2 then
15:  for each x ←ld(g) v do
16:    if f.Peek() = g then
17:      ppts ← ppts ∪ DSPointsTo (x, f.Pop(), S2)
18:  for each x ←assign v do
19:    ppts ← ppts ∪ DSPointsTo (x, f, S2)
20:  for each x ←st(g) v do
21:    ppts ← ppts ∪ DSPointsTo (x, f.Push(g), S1)
22:  for each v ←st(g) x do
23:    if f.Peek() = ḡ then
24:      ppts ← ppts ∪ DSPointsTo (x, f.Pop(), S1)
25:  if n ←exit/entry/assignglobal v then
26:    ppts ← ppts ∪ { (v, f, S2) }
27: return ppts
```

PPTA aims to summarise all state transitions field-sensitively but context-insensitively made along the local edges of a method according to the *pointsTo* and *alias* RSMs given in Figure 2.3(a). Starting with a points-to query for a variable

ν in context c, we will eventually arrive at the two RSMs with a new query $(\mathsf{u}, \mathsf{f}, \mathsf{s})$, where u is a node in some method m, f is a *field stack* containing the field edge labels encountered but not yet matched, and s is a state indicating the *direction* in which the analysis traverses—along a $\overline{\mathit{flowsTo}}$ path if $\mathsf{s} = \mathsf{S}_1$ and a *flowsTo* path if $\mathsf{s} = \mathsf{S}_2$. The objective of performing PPTA for $(\mathsf{u}, \mathsf{f}, \mathsf{s})$ is to compute a so-called *partial points-to set* for u, denoted $\mathsf{ppta}(\mathsf{u}, \mathsf{f}, \mathsf{s})$, so that (1) $\mathsf{ppta}(\mathsf{u}, \mathsf{f}, \mathsf{s})$ contains all objects o in method m that flow to u, and (2) all tuples $(\mathsf{u}', \mathsf{f}', \mathsf{s}')$ eventually reached by the *pointsTo* and *alias* RSMs given in Figure 2.3(a) along only the local edges in method m. Each such tuple represents a state reached this way and will be cached for later reuse just before a global edge is about to be traversed.

Consider our example given in Figure 3.1 again. We have $\mathsf{ppta}(\mathsf{ret}_{\mathrm{get}}, \varnothing, \mathsf{S}_1)$ $= \{(\mathsf{this}_{\mathrm{get}},\ [\![\overline{\mathsf{arr}}, \overline{\mathsf{elems}}]\!],\ \mathsf{S}_1)\}$, which shows intuitively that the points-to set of $\mathsf{this}_{\mathrm{get}}.\mathsf{elems}.\mathsf{arr}$ must be included in the points-to set of $\mathsf{ret}_{\mathrm{get}}$. Note that this PPTA information is computed when answering the points-to query for s1 and will be reused later when the points-to query s2 is answered.

For another example, suppose we want to compute the points-to set for s2 with an empty context. By traversing the right part of the PAG in Figure 3.2, we will eventually need to compute a query for $(\mathsf{this}_{\mathrm{set}}, [\![\overline{\mathsf{arr}}, \overline{\mathsf{elems}}, \overline{\mathsf{vec}}]\!], \mathsf{S}_2)$ (as later illustrated in Steps 6 – 7 for s2 in Figure 3.3). By performing a PPTA, we find that $\mathsf{ppta}(\mathsf{this}_{\mathrm{set}}, [\![\overline{\mathsf{arr}}, \overline{\mathsf{elems}}, \overline{\mathsf{vec}}]\!], \mathsf{S}_2) = \{(\nu_{\mathrm{set}}, [\![\overline{\mathsf{arr}}, \overline{\mathsf{elems}}]\!], \mathsf{S}_1)\}$.

3.3.2 Algorithms

Algorithm 4 This is a recursive algorithm that propagates the context-independent CFL-reachability information across a given PAG.

Algorithm 5 The DynSum analysis

DynSum (v, c)

```
 1: pts ← ∅
 2: w ← { (v, ∅, S1, c) }
 3: while w ≠ ∅ do
 4:    remove (u, f, s, c′) from w
 5:    if ((u, f, s), l) ∈ Cache then
 6:       ppta ← l
 7:    else
 8:       ppta ← DSPointsTo (u, f, s, ∅)
 9:       Cache ← Cache ∪ ((u, f, s), ppta)
10:    for each o ∈ ppta do
11:       pts ← pts ∪ { (o, c′) }
12:    for each (x, f′, s′) ∈ ppta do
13:       if s′ = S1 then
14:          for each x ←exit_i— y do
15:             Propagate(w, y, f′, S1, c′.Push(i))
16:          for each x ←entry_i— y do
17:             if c′ = ∅ or c′.Peek()= i then
18:                Propagate(w, y, f′, S1, c′.Pop())
19:          for each x ←assignglobal— y do
20:             Propagate(w, y, f′, S1, ∅)
21:       if s′ = S2 then
22:          for each y ←exit_i— x do
23:             if c′ = ∅ or c′.Peek()= i then
24:                Propagate(w, y, f′, S2, c′.Pop())
25:          for each y ←entry_i— x do
26:             Propagate(w, y, f′, S2, c′.Push(i))
27:          for each y ←assignglobal— x do
28:             Propagate(w, y, f′, S2, ∅)
29: return pts
```

Propagate(w, n, f, s, c)

```
1: w ← w ∪ { (n, f, s, c) }
```

The analysis strictly follows the *pointsTo* and *alias* RSMs for L_{FT} given in Figure 2.3(a), which has two states, S_1 and S_2. All transitions on S_1 are handled in lines 2 – 13 and those on S_2 in lines 15 – 26. Let us consider S_1 first. On encountering an edge $v \xleftarrow{new} o$ (lines 3 – 7), the analysis will insert the object o into pts only when the field stack f is empty. Otherwise, it will traverse a *flowsTo* path to find an *alias* relation between v and some x such that v *alias* x holds. An alias relation is discovered by following the *alias* RSM given in Figure 2.3(a). In lines 8 – 11, the $\overline{assign}$ and $\overline{ld}$ edges are handled. In lines 12 – 13, on encountering a global edge, PPTA stores the current state in pts. Lines 15 – 26 for dealing with state S_2 are similar. The only interesting part happens in lines 22 – 24, which accepts a $\overline{st}$ edge when the top of the field stack f matches the label of the store edge, $\overline{g}$.

Note that the two states S_1 and S_2 are handled asymmetrically since the *alias* RSM in Figure 2.3(a) is "asymmetric", or precisely, is recursive. There are four cases involved in handling field accesses: $ld(g)$, $st(g)$, $\overline{ld(g)}$ and $\overline{st(g)}$. In the PPTA algorithm, the $\overline{ld(g)}$ edges are handled in S_1 while the other three in S_2. In S_1, the *alias* RSM will process a $\overline{ld(g)}$ edge, $v \xleftarrow{ld(g)} x$, and stay in S_1. In S_2, the *alias* RSM will process (1) a $ld(g)$ edge, $x \xleftarrow{ld(g)} v$, and stay in S_2, (2) a $st(g)$ edge, $x \xleftarrow{st(g)} v$, and then transit to S_1 to look for aliases for the base variable x of the store, and (3) a $\overline{st(g)}$ edge, $v \xleftarrow{st(g)} x$, and transit to S_1 if the base variable v is an alias of the base variable of the most recent load processed earlier in lines 13 – 14. Note that the *alias* RSM can only move from S_1 to S_2 at an allocation site on $\overline{new}$ new, i.e., by first traversing the corresponding $\overline{new}$ edge and then the same edge in the opposite direction, which is the new edge.

Note that there can be points-to cycles in a PAG. Therefore, in our implementation

a set of visited nodes is used to avoid re-traversing a cycle more than once, similarly as in [54].

Algorithm 5 This is where our DynSum analysis starts. When called, DynSum (v, c) will return the points-to set of a queried variable v in context c. This is a worklist algorithm that propagates the CFL-reachability facts through a given PAG. Because the local edges are handled as a PPTA by Algorithm 4, Algorithm 5 deals with only the context-dependent global edges according to the RSM R_{RP} in Figure 2.3(b) while calling Algorithm 4 to perform all required PPTA steps.

Each worklist element is a tuple of the form (u, f, s, c), indicating that the computation for v has reached node u, where u is a new queried variable generated, with the current field stack f, the current "direction" state $s \in \{S1, S2\}$ of the RSM given in Figure 2.3(a) and the current context stack c. In lines 5 – 9, the summary ppta for the query (u, f, s) is reused if it is available in Cache and computed otherwise by calling Algorithm 4. As ppta returned from PPTA contains both objects and tuples, DynSum handles objects in lines 10 – 11 and tuples in lines 12 – 28. The assignglobal, exit_i and entry_i edges are handled according to the RSM for R_{RP} given in Figure 2.3(b), similarly as in RefinePTS.

3.3.3 An Example

We highlight the advantages of DynSum using the example given in Figure 3.1. In our implementation of Algorithm 5, DSPointsTo is not called in line 8 to perform the PPTA if u has no local edges.

Suppose we want to answer the same two points-to queries s1 and s2 as before.

Algorithm	Full Precision	Memorisation	Reuse	On-Demandness
NoRefine	Yes	No	No	Yes
RefinePTS	Yes	Dynamic (within queries)	Context Dependent	Yes
StaSum	No	Static (across queries)	Context Independent	Partly
DynSum	Yes	Dynamic (across queries)	Context Independent	Yes

Table 3.1: Strengths and weaknesses of four demand-driven points-to analyses.

Figure 3.3 illustrates how local reachability reuse is exploited in our analysis by showing only the traversed edges that lead directly to their points-to targets: o_{26} for s1 and o_{29} for s2.

Suppose s1 is issued first and then followed by s2. DynSum starts from s1 with the initial state being $(\mathsf{s1}, \varnothing, \mathsf{S}_1, \varnothing)$. The analysis encounters the incoming $\overline{\mathsf{exit}_{32}}$ edge, staying at S_1 and pushing 32 into the context stack. The new state is $(\mathtt{ret}_{\mathrm{retrieve}}, \varnothing, \mathsf{S}_1, [\![32]\!])$.

Next, DynSum processes edges according to the RSMs given in Figures 2.3(a) and (b). On encountering a node with some local edges, the analysis first performs a PPTA on the node and then uses its summarised partial points-to set to continue its exploration. If the summarised partial points-to set is available in the cache, then it is reused straightaway to speed up the exploration.

When a node is visited, there can be several paths to be explored from the node. For example, ν1 is visited at Step 11 when the query s1 is processed. In this case, there are three entry edges $\nu \xrightarrow{\mathsf{entry}_{25}} \mathtt{this}_{\mathrm{Vector}}$, $\nu \xrightarrow{\mathsf{entry}_{26}} \mathtt{this}_{\mathrm{add}}$ and $\nu \xrightarrow{\mathsf{entry}_{27}} \nu_{\mathrm{Client}}$ to deal with. The latter two do not lead to more paths to be explored and entry_{25} is the only acceptable path to keep traversing.

Finally, DynSum reaches $\mathtt{tmp1} \xleftarrow{\mathsf{new}} \mathsf{o}_{26}$, by completing its analysis in 23 steps.

Step	v	f	s	c	Edge
0	$s1$	$[\![\,]\!]$	S_1	$[\![\,]\!]$	$\overline{\mathsf{exit}_{32}}$
1	ret_{retrieve}	$[\![\,]\!]$	S_1	$[\![32]\!]$	$\overline{\mathsf{exit}_{22}}$
2	ret_{get}	$[\![\,]\!]$	S_1	$[\![32,22]\!]$	$\overline{\mathsf{load}(a)}$
3	t_{get}	$[\![\overline{a}]\!]$	S_1	$[\![32,22]\!]$	$\overline{\mathsf{load}(e)}$
4	$this_{\text{get}}$	$[\![\overline{a},\overline{e}]\!]$	S_1	$[\![32,22]\!]$	$\overline{\mathsf{entry}_{22}}$
5	t_{retrieve}	$[\![\overline{a},\overline{e}]\!]$	S_1	$[\![32]\!]$	$\overline{\mathsf{load}(v)}$
6	$this_{\text{retrieve}}$	$[\![\overline{a},\overline{e},\overline{v}]\!]$	S_1	$[\![32]\!]$	$\overline{\mathsf{entry}_{32}}$
7	$c1$	$[\![\overline{a},\overline{e},\overline{v}]\!]$	S_1	$[\![\,]\!]$	$\overline{\mathsf{new}}$ new
8	$c1$	$[\![\overline{a},\overline{e},\overline{v}]\!]$	S_2	$[\![\,]\!]$	$\overline{\mathsf{entry}_{27}}$
9	$this_{\text{Client}}$	$[\![\overline{a},\overline{e},\overline{v}]\!]$	S_2	$[\![27]\!]$	$\overline{\mathsf{store}(v)}$
10	v_{Client}	$[\![\overline{a},\overline{e}]\!]$	S_1	$[\![27]\!]$	$\overline{\mathsf{entry}_{27}}$
11	$v1$	$[\![\overline{a},\overline{e}]\!]$	S_1	$[\![\,]\!]$	$\overline{\mathsf{new}}$ new
12	$v1$	$[\![\overline{a},\overline{e}]\!]$	S_2	$[\![\,]\!]$	$\overline{\mathsf{entry}_{25}}$
13	$this_{\text{Vector}}$	$[\![\overline{a},\overline{e}]\!]$	S_2	$[\![25]\!]$	$\overline{\mathsf{store}(e)}$
14	t_{Vector}	$[\![\overline{a}]\!]$	S_1	$[\![25]\!]$	$\overline{\mathsf{new}}$ new
15	t_{Vector}	$[\![\overline{a}]\!]$	S_2	$[\![25]\!]$	$\mathsf{store}(e)$
16	$this_{\text{Vector}}$	$[\![\overline{a},e]\!]$	S_1	$[\![25]\!]$	$\overline{\mathsf{entry}_{25}}$
17	$v1$	$[\![\overline{a},e]\!]$	S_1	$[\![\,]\!]$	$\overline{\mathsf{new}}$ new
18	$v1$	$[\![\overline{a},e]\!]$	S_2	$[\![\,]\!]$	entry_{26}
19	$this_{\text{add}}$	$[\![\overline{a},e]\!]$	S_2	$[\![26]\!]$	$\mathsf{load}(e)$
20	t_{add}	$[\![\overline{a}]\!]$	S_2	$[\![26]\!]$	$\overline{\mathsf{store}(a)}$
21	p	$[\![\,]\!]$	S_1	$[\![26]\!]$	$\overline{\mathsf{entry}_{26}}$
22	$tmp1$	$[\![\,]\!]$	S_1	$[\![\,]\!]$	$\overline{\mathsf{new}}$
23	o_{26}	$[\![\,]\!]$	S_1	$[\![\,]\!]$	
0	$s2$	$[\![\,]\!]$	S_1	$[\![\,]\!]$	$\overline{\mathsf{exit}_{33}}$
1	ret_{retrieve}	$[\![\,]\!]$	S_1	$[\![33]\!]$	$\overline{\mathsf{exit}_{22}}$
2	ret_{get}	$[\![\,]\!]$	S_1	$[\![33,22]\!]$	= = **reuse**
	$\overline{\mathsf{load}(a)}\ \overline{\mathsf{load}(e)}$				
	$this_{\text{get}}$	$[\![\overline{a},\overline{e}]\!]$	S_1	$[\![33,22]\!]$	$\overline{\mathsf{entry}_{22}}$
3	t_{retrieve}	$[\![\overline{a},\overline{e}]\!]$	S_1	$[\![33]\!]$	= = **reuse**
	$\overline{\mathsf{load}(v)}$				
	$this_{\text{retrieve}}$	$[\![\overline{a},\overline{e},\overline{v}]\!]$	S_1	$[\![33]\!]$	$\overline{\mathsf{entry}_{33}}$
4	$c2$	$[\![\overline{a},\overline{e},\overline{v}]\!]$	S_1	$[\![\,]\!]$	$\overline{\mathsf{new}}$ new
5	$c2$	$[\![\overline{a},\overline{e},\overline{v}]\!]$	S_2	$[\![\,]\!]$	$\overline{\mathsf{entry}_{31}}$
6	$this_{\text{set}}$	$[\![\overline{a},\overline{e},\overline{v}]\!]$	S_2	$[\![31]\!]$	$\overline{\mathsf{store}(v)}$
7	v_{set}	$[\![\overline{a},\overline{e}]\!]$	S_1	$[\![31]\!]$	$\overline{\mathsf{entry}_{31}}$
8	$v2$	$[\![\overline{a},\overline{e}]\!]$	S_1	$[\![\,]\!]$	$\overline{\mathsf{new}}$ new
9	$v2$	$[\![\overline{a},\overline{e}]\!]$	S_2	$[\![\,]\!]$	entry_{28}
10	$this_{\text{Vector}}$	$[\![\overline{a},\overline{e}]\!]$	S_2	$[\![28]\!]$	= = **reuse**
	$\overline{\mathsf{store}(e)}\ \overline{\mathsf{new}}\ \mathsf{new}\ \mathsf{store}(e)$				
	$this_{\text{Vector}}$	$[\![\overline{a},e]\!]$	S_1	$[\![28]\!]$	$\overline{\mathsf{entry}_{28}}$
11	$v2$	$[\![\overline{a},e]\!]$	S_1	$[\![\,]\!]$	$\overline{\mathsf{new}}$ new
12	$v2$	$[\![\overline{a},e]\!]$	S_2	$[\![\,]\!]$	entry_{29}
13	$this_{\text{add}}$	$[\![\overline{a},e]\!]$	S_2	$[\![29]\!]$	= = **reuse**
	$\mathsf{load}(e)\ \overline{\mathsf{store}(a)}$				
	p	$[\![\,]\!]$	S_1	$[\![29]\!]$	$\overline{\mathsf{entry}_{29}}$
14	$tmp2$	$[\![\,]\!]$	S_1	$[\![\,]\!]$	$\overline{\mathsf{new}}$
15	o_{29}	$[\![\,]\!]$	S_1	$[\![\,]\!]$	

Figure 3.3: Traversals of DynSum when answering the queries for s1 and s2 in our motivating example (a, e and v stand for fields arr, elems and vector, respectively).

The points-to set of s1 is $\{o_{26}\}$.

When s2 is issued, the summaries computed earlier can be reused. As shown in the bottom part of Figure 3.3, DynSum takes only 15 steps to find $\{o_{29}\}$ as its points-to set. Ad hoc caching techniques [54, 66, 74] are not helpful since both queries require different calling contexts to be traversed, as explained earlier.

For this example, the summaries computed when query s1 is processed are not reused within the same query. In general, however, reuse can happen both within a query and during subsequent queries.

3.3.4 Comparison

We compare four context- and field-sensitive demand-driven points-to or alias analyses in Table 3.1 now and in our evaluation later:

- RefinePTS. This is the implementation from [54] with an open-source release. As reviewed earlier, RefinePTS uses a refinement policy to satisfy a client's queries. All queries are handled independently. Ad hoc caching is used to avoid unnecessary traversals within a query.

- NoRefine. This is the version of RefinePTS with neither refinement nor ad hoc caching.

- StaSum. This is the implementation with similar concepts introduced in [66, 70], which computes all-pair reachability summaries for each method off-line and then reuses the summaries to accelerate demand queries. In our experiments, such summaries are computed for all methods on the PAG instead of a symbolic

graph of the program. No efforts are made to avoid some summaries based on some user-supplied heuristics.

- DynSum. This is the one introduced in this thesis. DynSum can deliver the same precision as RefinePTS with enough budgets and is fully on-demand without performing any unnecessary computations to achieve great reuse.

3.4 Evaluation

We evaluate the efficiency of DynSum by comparing it with RefinePTS using nine Java benchmarks, selected from the Dacapo and SPECjvm98 benchmark suites. For reference purposes, the performance of NoRefine is also given. As StaSum is not available to us, we will compare it with DynSum in terms of the number of summaries computed. Our evaluation has validated the following two experimental hypotheses about the proposed DynSum approach:

- **DynSum is more scalable than RefinePTS.** DynSum outperforms RefinePTS by 1.95×, 2.28× and 1.37× on average for the three clients discussed below.

  DynSum avoids a great number of unnecessary computations and thus represents a good optimisation for context-sensitive demand-driven analysis.

- **DynSum is more scalable than StaSum.** DynSum computes significantly fewer summaries than StaSum for the same three clients, making it better-suited for low-budget environments like JIT compilers and IDEs.

Benchmark	#Methods (K)	#Nodes (K)		#Edges (K)							Locality
		O	(V∪G)	new	assign	ld	st	entry	exit	assignglobal	
jack	0.5	16.6	207.9	16.6	328.1	25.1	8.8	39.9	12.8	2.4	87.3%
javac	1.1	17.2	216.1	17.2	367.4	26.8	9.1	42.4	13.3	0.5	88.2%
soot-c	3.4	9.4	104.8	9.4	195.1	13.3	4.2	19.3	6.4	0.7	89.4%
bloat	2.2	10.3	115.2	10.3	217.2	14.5	4.6	20.6	6.1	1.0	89.9%
jython	3.2	9.5	109.0	9.5	168.4	14.4	4.2	19.5	7.1	1.3	87.6%
avrora	1.6	4.5	45.1	4.5	38.1	6.0	2.9	9.7	2.9	0.3	80.0%
batik	2.3	10.8	118.1	10.8	119.7	13.4	5.3	24.8	7.8	0.6	81.8%
luindex	1.0	4.4	48.2	4.4	42.6	6.9	2.3	9.1	3.0	0.5	81.7%
xalan	2.5	6.6	75.8	6.6	76.4	14.1	4.4	15.7	4.0	0.2	83.6%

Table 3.2: Benchmark statistics. Note that Column "O (objs)" is identical to Column "new". All of the numbers include the reachable parts of the Java library, determined using a call graph constructed on the fly with Andersen-style analysis [2] by Spark [31]. Column "*locality*" gives the ratio of local edges among all edges in a PAG.

3.4.1 Implementation

RefinePTS is publicly available in the Soot 2.4.0 [60] and Spark [31] frameworks. We have implemented DynSum and NoRefine in the same frameworks and conducted our experiments using the Sun JDK 1.6.0_16 libraries. Unmodeled native methods and reflection calls [37, 38] are handled conservatively and Tamiflex [5] is used. As all three analyses are context-sensitive, the call graph of the program is constructed on-the-fly so that a *context-sensitive* call graph is always maintained during the CFL-reachability exploration.

When introducing all three algorithms earlier, we have assumed cycle-free PAGs to make them easy to understand. However, recursion is handled as described in [54] by computing the call graph on-the-fly with recursion cycles collapsed. Points-to cycles are also handled using visited flags in Algorithm 4 as described in [54] by ensuring that a node is not cyclically visited.

3.4.2 Methodology

We have conducted our experiments on a machine consisting of four AMD Opteron 2.2GHz processors (12 cores each) with 32 GB memory, running RedHat Enterprise Linux 5 (kernel version 2.6.18). Although the system has multi-cores, each analysis algorithm is single-threaded.

We have selected the following three representative clients:

- `SafeCast`. This client checks the safety of downcasts in a program as also discussed in [54].
- `NullDeref`. This client detects null pointer violations, demanding high precision from points-to analysis.
- `FactoryM`. This client checks that a factory method returns a newly-allocated object for each call as in [54].

Each client continuously issues points-to queries to an analysis. A query is either positively answered by the analysis or terminated once a pre-set budget is exceeded. In our experiments, we have also carefully divided the queries from a client into batches to demonstrate the scalability of DynSum compared to RefinePTS and StaSum as the number of queries increases.

The benchmarks we used for evaluation are nine Java programs selected from the SPECjvm98 and Dacapo benchmark suites. Table 3.2 shows the number of different kinds of nodes and edges in the context-sensitive PAG of a program. The *locality* of a PAG is measured as the percentage of local (*flowsTo*) edges (including new, assign, ld and st) among all (*flowsTo*) edges. This metric is used to demonstrate the scope of our optimisation. As can be seen from Table 3.2, the majority of the edges in a PAG are local edges. This implies that a large number of paths with only local edges can be summarised in context-independent manner and reused later.

We repeated each experiment three times and reported the average time of the three runs, which includes the time elapsed on points-to analysis and client analysis. All the experiments have low variance in performance. For all analysis algorithms compared, the budget limitation is 75,000, indicating the maximum number of edges that can be traversed in a PAG in order to answer one points-to query.

3.4.3 Results and Analysis

Analysis Times Table 3.3 compares the analysis times of DynSum with RefinePTS and NoRefine for the three clients. NoRefine is the slowest in most cases but can be faster than RefinePTS in some benchmarks for clients `SafeCast` and `NullDeref`. In contrast, DynSum is always faster than NoRefine in all benchmarks for all three clients.

Let us compare DynSum and RefinePTS. DynSum is only slightly slower in `avrora` for `SafeCast` and `luindex` for `FactoryM`. DynSum attains its best performance in `soot-c` for `NullDeref`, outperforming RefinePTS by 4.19×. The average speedups achieved by DynSum for the three clients `SafeCast`, `NullDeref` and

		jack	javac	soot-c	bloat	jython	avrora	batik	luindex	xalan
SafeCast	NoRefine	31.0	68.1	134.7	68.2	61.8	39.1	43.4	47.6	459.1
	RefinePTS	28.4	77.9	127.9	76.3	50.9	30.2	29.8	44.9	457.5
	DynSum	15.2	41.3	37.5	32.8	32.2	35.1	19.7	25.3	194.5
NullDeref	NoRefine	121.0	174.4	212.3	72.8	160.0	84.4	95.0	57.1	797.9
	RefinePTS	145.6	163.9	221.0	73.5	150.2	20.6	80.7	60.1	575.7
	DynSum	52.6	87.5	52.8	42.6	72.3	13.6	46.4	41.3	194.1
FactoryM	NoRefine	26.3	85.1	22.8	147.1	15.7	30.1	41.2	20.7	139.1
	RefinePTS	25.4	60.5	9.5	104.6	15.4	27.9	33.9	13.1	117.8
	DynSum	23.4	47.2	6.7	75.1	6.3	24.4	24.3	13.4	99.5

Table 3.3: Analysis times of NoRefine, RefinePTS and DynSum for the three clients: SafeCast, NullDeref and FactoryM.

FactoryM are 1.95×, 2.28× and 1.37×, respectively.

The client that benefits the most from DynSum is NullDeref, which requires more precision than the other two clients. Given such high-precision requirements, RefinePTS can hardly terminate early, effectively rendering its repeated refinement steps as pure overhead. This fact is also reflected by the similar analysis times taken by both RefinePTS and NoRefine for this client.

As garbage collection is enabled, it is difficult to monitor memory usage precisely. In our all experiments, DynSum never exceeds 20% more than RefinePTS in terms of the peak memory usage.

Scalability in Answering Demand Queries We have selected soot-c, bloat and jython to demonstrate that DynSum is more scalable than RefinePTS and StaSum. These applications are selected because they have large code bases, i.e.,

large PAGs as shown in Table 3.2. For each program, we divide the sequence of queries issued by a client into 10 batches. If a client has n_q queries, then each of the first nine batches contains $\lfloor n_q/10 \rfloor$ queries and the last one gets the rest.

- **Comparing with RefinePTS** Figure 3.4 compares the times taken by DynSum for handling each batch of queries normalised with respect to RefinePTS. As more batches are processed, more points-to relations will have been summarised dynamically and recorded for later reuse, and consequently, the less time that DynSum takes to process each subsequent batch.

- **Comparing with StaSum** We collect the number of summaries computed by DynSum at the end of each batch and compare it with StaSum for the three selected benchmarks. For DynSum, the number of summaries computed is available as the size of `Cache` given in Algorithm 5. For StaSum, all possible summaries for each call entry or exit in a PAG are computed. While StaSum can reduce its number of such summaries based on a user-supplied threshold [70], it is unclear how this can be done effectively by the user, particularly when its optimal value varies from program to program.

  Figure 3.5 compares the (cumulative) size of summaries computed by DynSum normalised with respect to StaSum. DynSum only needs to compute 41.3%, 47.7% and 37.3% of the summaries computed by StaSum on average in order to handle all the queries issued by the three clients. Furthermore, the number of summaries increases dynamically as the number of queries increases, highlighting the dynamic nature of DynSum.

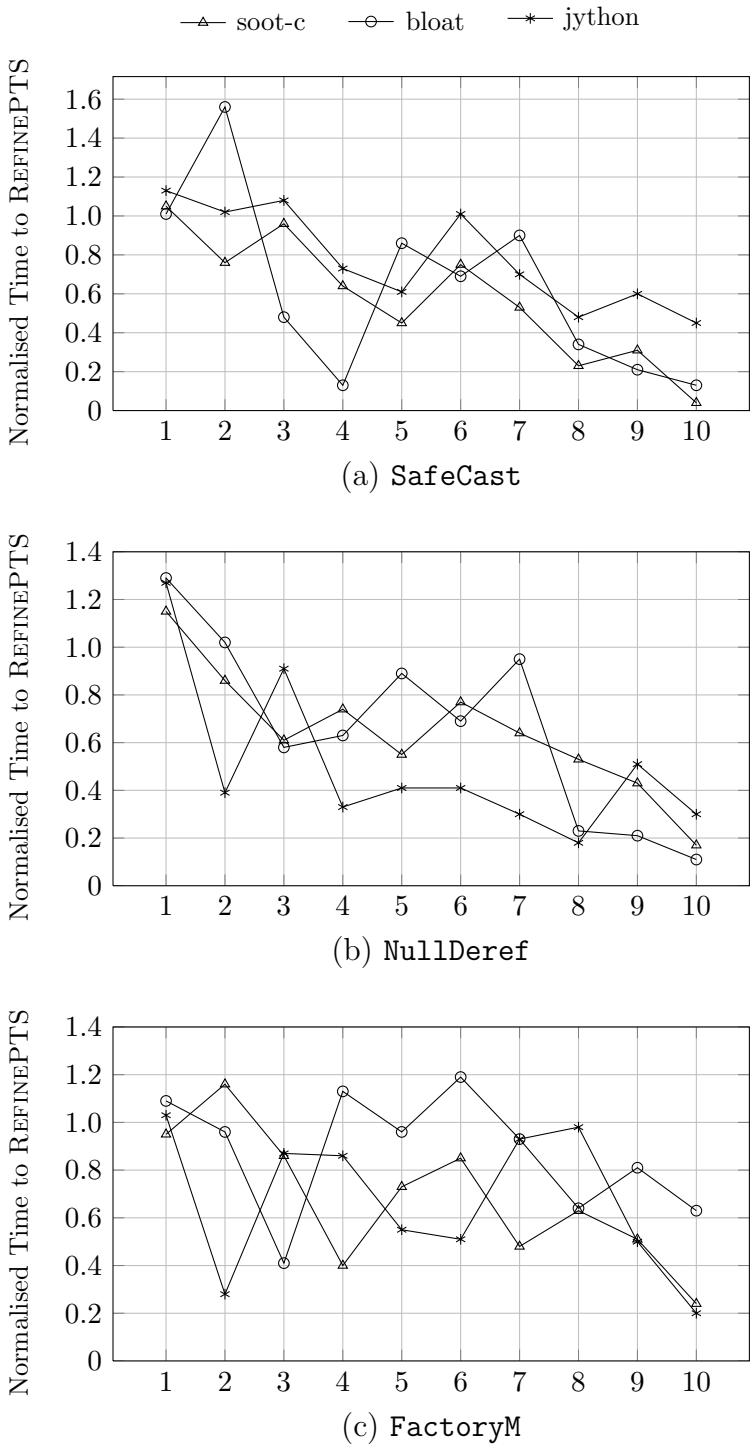


Figure 3.4: Normalised analysis times for each batch of queries normalised with respect to RefinePTS.

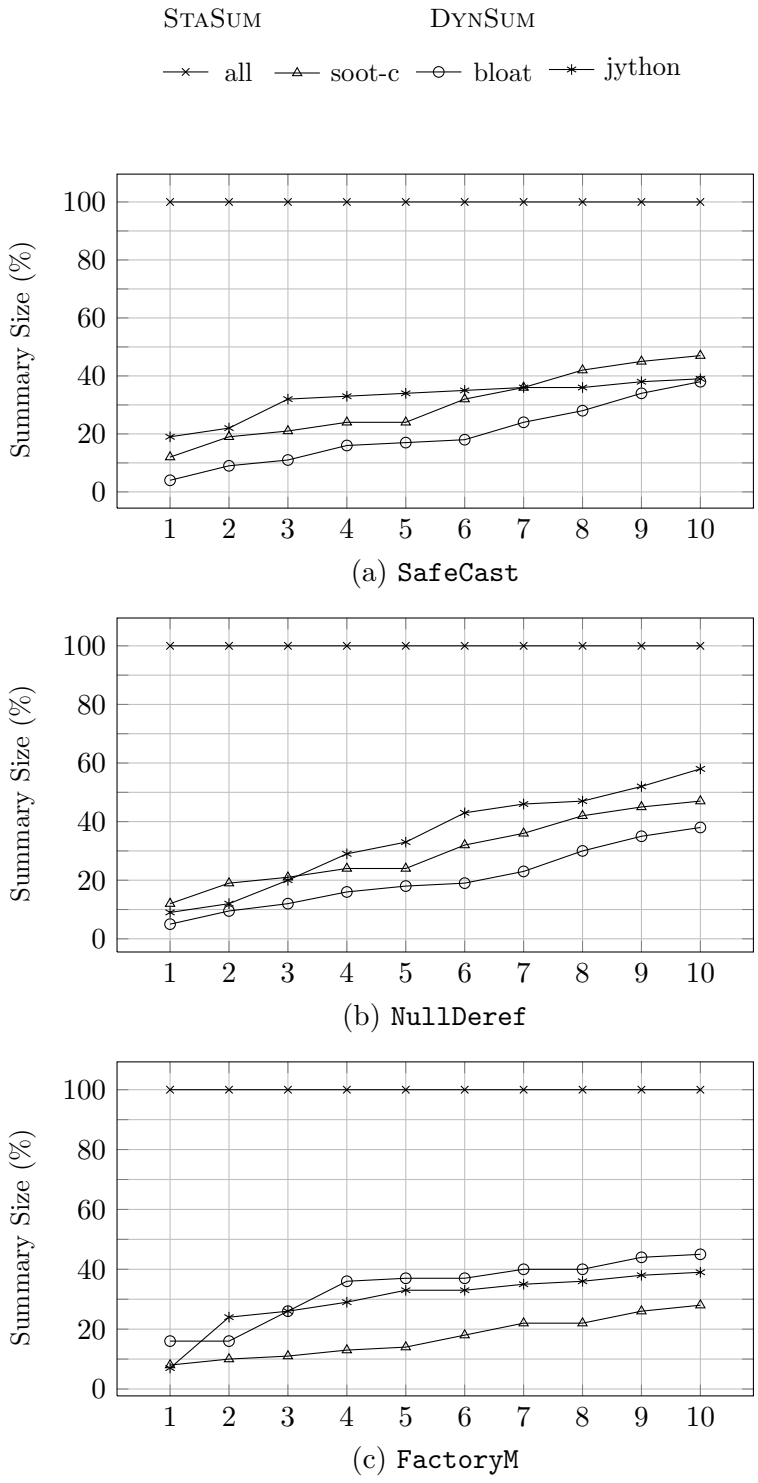


Figure 3.5: The cumulative number of summaries computed by DynSum normalised with respect to StaSum.

Through these studies, we find that DynSum is effective in avoiding unnecessary traversals made as in RefinePTS and unnecessary summaries computed as in StaSum. The increased scalability makes DynSum better-suited to low-budget environments such as JIT compilers and IDEs in which software may undergo frequent changes.

3.5 Related Work

In recent years, there has been a large body of research devoted to points-to analysis, with the summary-based approach to be the most popular and general for achieving context sensitivity. However, existing summary-based algorithms [40, 62, 72] are mostly whole-program-based. How to compute summaries efficiently for demand-driven analysis is less well-understood. Below we focus only on the work directly related to demand-driven points-to analysis.

To accelerate demand queries, some techniques to speed up demand-driven points-to analysis have been explored. In the refinement-based approach introduced in [54], the analysis starts to be field-based for all heap accesses and gradually introduces field sensitivity into those heap accesses where a better precision may be obtained. In [66], a (whole-program) pre-analysis is presented to improve the performance of demand-driven points-to analysis in Java. In demand-driven analysis techniques [54, 66, 74], budget limitation is commonly used to give a conservative answer for a query once a pre-set budget has been exceeded.

Reps et al. [34, 44] pioneered the research on program analysis via graph reachability. They formulate a number of static analysis programs in terms of CFL-reachability,

leading to a natural solution to demand-driven points-to analysis.

Heintze and Tardieu [23] introduced a deduction-based demand-driven points-to analysis for C to determine the points-to sets based on demand queries from a client. They adopted an Andersen-style inclusion-based pointer analysis by reasoning about dependent constraints.

Sridharan et al. [54, 55] have proposed two approaches to solving CFL-reachability-based demand-driven points-to analysis for Java. They initially presented a CFL-reachability formulation to model heap accesses as a balanced-parentheses problem in a context-insensitive manner [55]. Later, they extended this earlier work to obtain a context-sensitive points-to analysis [54]. The starting point of our PPTA-based solution, DynSum, is Sridharan and Bodik's refinement-based analysis [54], using Spark's PAG [31] as our program representation. DynSum improves the performance of this state-of-the-art work significantly without affecting precision.

Zheng and Rugina [74] described a demand-driven alias analysis for C. Unlike Heintze and Tardieu's analysis [23], Zheng and Rugina's analysis relies a memory alias CFL-reachability formulation. Their analysis is context-insensitive with indirect function calls being conservatively handled. As a result, realisable and unrealisable paths are not distinguished, resulting in both precision and performance loss for some queries.

Xu et al. [66] proposed a pre-analysis to speed up the context-sensitive points-to analysis introduced in [54]. The analysis builds a symbolic graph to reduce the size of a program's PAG but it is whole-program-based.

Yan et al. [70] have recently extended the work of [66] to perform a demand-driven alias analysis without having to compute points-to sets. The proposed approach,

denoted StaSum, is compared with DynSum in Table 3.1 and Figure 3.5.

Some existing techniques [54, 66, 74] on memorisation are ad hoc, limiting their scope and effectiveness. The points-to set $pts(\nu, \mathbf{c})$ of a variable ν in a calling context $\mathbf{c}$ is cached only after all ν's pointed-to objects have been fully resolved, which does not happen once a pre-set budget has been exceeded. Due to such full reachability reuse, $pts(\nu, \mathbf{c})$ can only be reused for ν in exactly the same (full) context $\mathbf{c}$. In addition, these existing memorisation techniques do not directly apply to the state-of-the-art refinement-based approach [54] since the underlying PAG may change due to the iterative refinement used. To the best of our knowledge, this work represents the first systematic investigation on how to exploit local reachability reuse dynamically in order to improve the performance of context-sensitive demand-driven points-to analysis in CFL-reachability.

3.6 Summary

In this chapter, we present a new technique on how to dynamically exploit local reachability reuse to improve the performance of CFL-reachability based demand-driven points-to analysis without sacrificing precision and ondemandness. Evaluation and validation using three client applications over a range of nine Java benchmarks show that our summarisation approach can significantly boost the performance of state-of-the-art work without affecting precision by exploiting the locality of statements. Our approach is not only an improvement of demand-driven points-to analysis, but also provides a potential for it to be used in low-budget environments such as JIT and IDEs, especially when a program undergoes constantly many changes as further

investigated in the next chapter.

Chapter 4

Incremental Analysis

Incremental analysis is a technique used to analyse the program by assessing the impact of small changes based on previous analysis results, while traditional program analysis completely starts over from the beginning. The incremental approach may result in considerable savings in analysis time, however, studies on points-to analysis for handling program changes are limited. Previous state-of-the-art incremental points-to analyses suffer from use of complicated rules or limited precision [10, 25, 28, 42, 49, 73]. This chapter presents an approach to incrementally handle program changes based on CFL-reachability. The goal is to reduce the unnecessary analysis time of recomputing from scratch by limiting the change impact to those affected parts in a program. Section 4.1 provides an overview of our incremental approach. Section 4.2 shows an example used to describe our framework. Section 4.3 presents our points-to framework based on incremental summarisation. Section 4.4 evaluates our approach with three clients on a set of Java benchmarks. Section 4.5 discusses the previous work most closely related to our work. Section 4.6 summaries the chapter.

4.1 Overview

Software evolution is an important feature of modern software. In practice, larger software applications are never complete and will continue to evolve once deployed.

Points-to analysis researches over several decades have focused on obtaining precise points-to information efficiently, with impressive progress being made in points-to analysis for Java [32, 50, 54, 61, 70]. However, existing algorithms are generally not formulated to be used in environments like IDEs where the software is still being developed. Precise whole-program analyses [32, 61] are expensive because they reanalyse a program from scratch in response to code changes. Others [50, 54, 70] reduce analysis times by performing points-to analysis on demand for a set of specified variables.

Points-to information is increasingly used in a variety of static analysis tools incorporated into an IDE to assist developers with program understanding and debugging. In such environments, a points-to analysis must satisfy the following three constraints:

Frequent Code Changes The analysis must be engineered to work well in response to small but frequent edits.

Non-Intrusiveness The analysis must handle each code change quickly without disrupting developer productivity.

Demand Queries with Limited Budgets The analysis must answer queries as precisely as possible within a small time budget.

For example, the developer may wish to know if `(T)v` is a safe downcast or not while still making code changes to the program. The analysis is required to answer the query quickly within a given budget while handling the impact of the code changes being made.

Traditional Method Summarisation In the case of whole-program analysis, context sensitivity has been traditionally achieved via summarisation [26, 62, 72]. The interprocedural modification side-effects of a method are summarised directly in terms of the points-to sets established by the method itself and the others called directly or indirectly in the method. The summary is then instantiated context-sensitively at its different call sites. These traditional summary-based approaches are not suitable for IDEs. When a code change occurs in a method, the summaries for the method and all its direct and indirect callers may have to be updated, resulting in unbounded propagation of the changed points-to information.

Change impact analysis Change impact analysis [1, 6, 17, 43] identifies software artifacts being affected by a change. However, the underlying whole-program points-to analyses currently employed do not even scale for small programs if flow and context sensitivity are considered [1]. Existing incremental points-to analyses [28, 49, 73] were not formulated for IDEs and thus cannot suitably be used due to their lack of efficiency or precision, as discussed in Section 4.5.

In this chapter, we introduce a new approach, called Emu, to points-to analysis for Java that simultaneously satisfies all the three constraints above for IDEs [51]. Leveraging recent advances on demand-driven points-to analysis [50, 54, 70], Emu adopts a CFL-reachability formulation to facilitate processing demand queries raised in IDEs after code changes are made. Emu achieves its efficiency by adopting a novel modular approach to allowing method summaries to be incrementally updated upon code changes. The points-to information in a method is summarised indirectly by CFL-reachability so that the impact of a code change made in a method is localised, i.e., bounded, requiring only the affected part of the method to be re-summarised just

to reflect the change. In contrast, existing summary-based techniques for points-to analysis [26, 62, 72] are unbounded, because the amount of re-summarisation triggered by code changes is proportional to the size of the program rather than the size of the impact. EMU achieves its precision by being (fully) context-sensitive (for both method invocation and heap abstraction) and field-sensitive (by distinguishing different fields of an object). These two axes are known to be crucial to achieving analysis precision for Java [32, 61].

This chapter makes the following contributions:

- In this chapter, we present EMU, the first points-to analysis formulated directly for IDEs by CFL-reachability to answer pointer-related queries on demand upon code changes.

- We introduce a modular incremental summarisation to achieving both efficiency and precision. Method summaries are described by CFL-reachability rather than points-to sets as is done traditionally, enabling the impact of a code change made in a method to be localised within the method so that only the affected part of the method needs to be updated.

- We have implemented EMU in Soot, a Java optimisation and analysis framework integrated into the Eclipse IDE. We have evaluated EMU with three representative clients, safe casting, null dereferencing and factory methods, using seven Java programs. For small code changes, such as adding/deleting statements (assignments or calls), EMU can answer each query under 0.054 secs on average and under 0.87 secs in the worst case, at nearly the same precision achieved by EMU without any time budget (or a whole-program analysis). Our results

are encouraging, suggesting that EMU can be promisingly deployed in an IDE where the changes are small.

4.2 Background

In this section, we show the points-to analysis problem with program changes from an example. We also illustrate the major challenges facing incremental algorithm.

4.2.1 Example

First we introduce the problem using a Java example given in Figure 4.1, providing an abstraction for the Java container pattern. In this example, we consider a simple code change during development of the program. Lines 1 – 18 defines a `Hashtable` class which contain two fields `key` and `val`. In lines 22 – 26, a `Hashtable`, `h1`, is created and populated with an object of `Element`, $e1$, with its field `f` being initialised with a string. In lines 27 – 31, the same thing happens for another `Hashtable` `h2` with a different object $e2$ of `Element`. In lines 32 and 33, calling `get` results in $e3 = e1$ and $e4 = e2$. In line 34, `m` retrieves the string stored in $e3$.f.

To obtain the points-to set of variable `m` in line 34, it is necessary to distinguish the different calling contexts to obtain precise solution. We consider a statement deletion in this example: the statement `t2[index] = v` in line 14. We need to resolve the points-to solution after this code change.

The PAG of this example is shown in Figure 4.2. We use the PAG of our example to show how to resolve some pointers on demand via CFL-reachability. Let us first see how to discover o_5 as a pointer target for `t1` through L_{FT}. In Figure 4.1, o_{22} flows

```
1  class Hashtable {
2    Object[] key;
3    Object[] val;
4    Hashtable() {
5      Object[] s1 = new Object[MAXSIZE];
6      Object[] s2 = new Object[MAXSIZE];
7      this.key = s1;
8      this.val = s2;}
9    void put(Object k, Object v) {
10     int index = k.hashCode();
11     Object[] t1 = this.key;
12     t1[index] = k;
13     Object[] t2 = this.val;
14     t2[index] = v;}
15   Object get(Object k) {
16     int index = k.hashCode();
17     Object[] t = this.val;
18     return t[index]; }}
19 class Element{ Object f; }
20 class Main {
21 static void main(...) {
22   Hashtable h1 = new Hashtable();
23   Element e1 = new Element();
24   e1.f = new String("hello");
25   String k1 = "first";
26   h1.put(k1, e1);
27   Hashtable h2 = new Hashtable();
28   Element e2 = new Element();
29   e2.f = new String("world");
30   String k2 = "second";
31   h2.put(k2, e2);
32   Element e3 = (Element) h1.get(k1);
33   Element e4 = (Element) h2.get(k2);
34   Object m = e3.f; }}
```

Figure 4.1: A Java example.

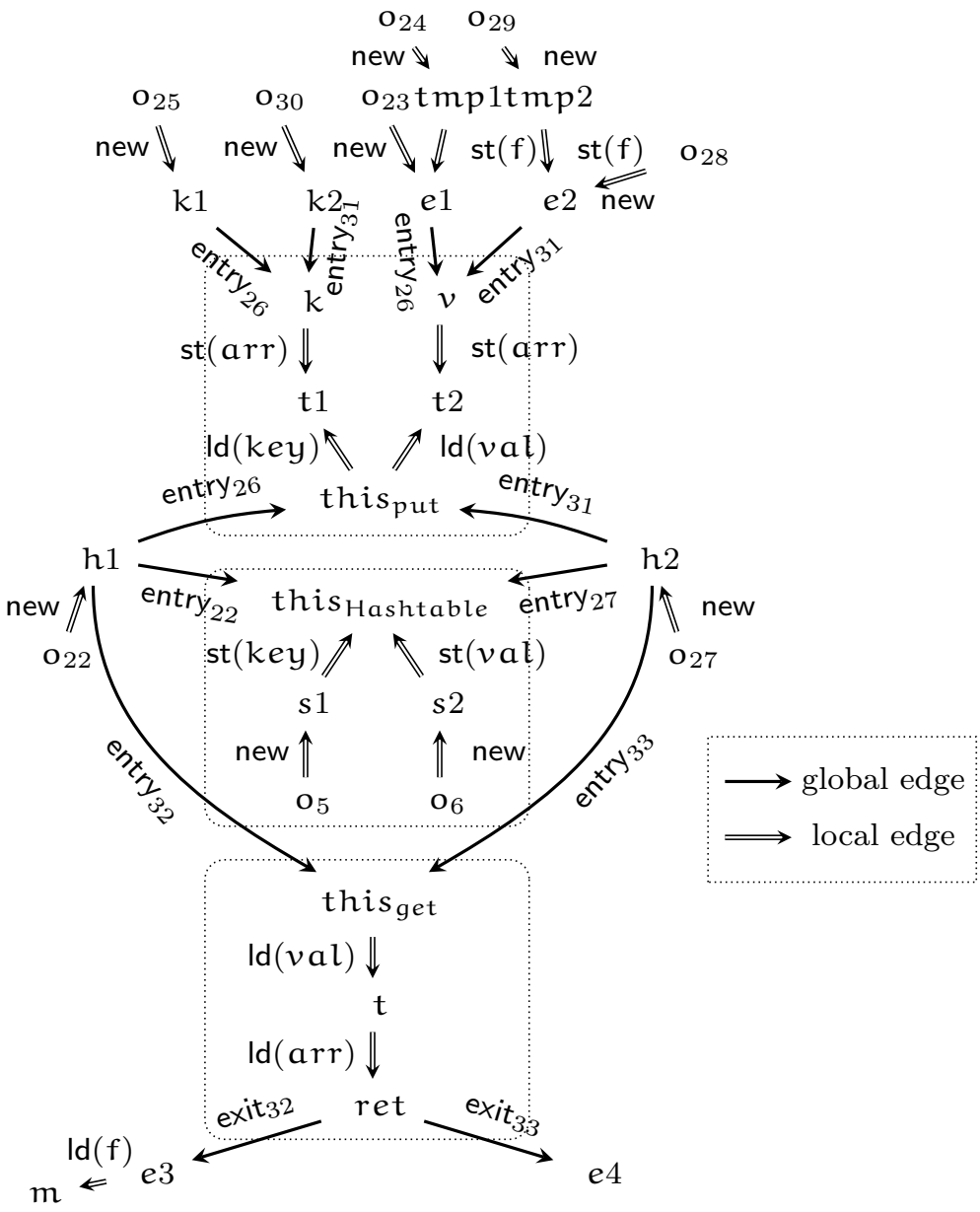


Figure 4.2: PAG for the Java example given in Figure 4.1.

to h1, which is the actual parameter passed to the formal parameter $\texttt{this}_{\texttt{Hashtable}}$ of constructor `Hashtable` and $\texttt{this}_{\texttt{put}}$ of `put`. So $\texttt{this}_{\texttt{Hashtable}}$ *alias* $\texttt{this}_{\texttt{put}}$. This

fact is found in $\mathrm{L_{FT}}$ because

$$\mathsf{this}_{\mathtt{Hashtable}}\ \overline{\mathsf{assign}}\ \ \mathtt{h1}\ \ \overline{\mathsf{new}}\ \mathsf{o}_{22}\ \ \mathsf{new}\ \ \mathtt{h1}\ \ \mathsf{assign}\ \ \mathsf{this}_{\mathtt{put}}$$

We then know that o_5 *flowsTo* $\mathtt{t1}$ since $\mathrm{L_{FT}}$ has the *flowsTo*-path:

$$\mathsf{o}_5\ \ \mathsf{new}\ \ \mathtt{s1}\ \ \mathsf{st}(\mathsf{key})\ \ \mathsf{this}_{\mathtt{Hashtable}}\ \ \textit{alias}\ \ \mathsf{this}_{\mathtt{put}}\ \ \mathsf{ld}(\mathsf{key})\ \ \mathtt{t1}$$

This *flowsTo*-path is also in $\mathrm{R_{RP}}$. So $\mathtt{t1}$ points to o_5.

Similarly, we find that o_{24} *flowsTo* $\mathtt{m}$ and o_{29} *flowsTo* $\mathtt{m}$. However, when context sensitivity is considered, only the former *flowsTo*-path is realisable. In Figure 4.1, o_{24} is initially inserted by calling `put` in line 26 into `Hashtable` $\mathtt{h1}$, which is created in line 22, and later retrieved by calling `get` in line 32 and saved into $\mathtt{m}$. This *flowsTo*-path is realisable because $\mathsf{entry}_{26} \rightarrow \overline{\mathsf{entry}_{26}} \rightarrow \mathsf{entry}_{22} \rightarrow \overline{\mathsf{entry}_{22}} \rightarrow \mathsf{entry}_{32} \rightarrow \mathsf{exit}_{32}$ is in $\mathrm{R_{RP}}$. In contrast, the *flowsTo*-path corresponding to o_{29} *flowsTo* $\mathtt{m}$ is not because $\mathsf{entry}_{31} \rightarrow \overline{\mathsf{entry}_{31}} \rightarrow \mathsf{entry}_{27} \rightarrow \overline{\mathsf{entry}_{27}} \rightarrow \mathsf{entry}_{33} \rightarrow \mathsf{exit}_{32}$ is not in $\mathrm{R_{RP}}$, i.e., o_{29} does not flow to $\mathtt{m}$ context-sensitively.

After the code change made in line 14, the local edge from $\mathtt{v}$ to $\mathtt{t2}$ will disappear and thus the existing solutions may not be valid any more. For example, o_{24} cannot flow to $\mathtt{m}$ any more due to the change. Existing CFL-reachability based points-to analyses can only recompute the points-to information for the changed program from scratch, which is costly. Therefore, an incremental analysis that can effectively handle the change is critical to tackling this problem.

4.2.2 Challenges Facing Incremental Points-to Analysis

The research on points-to analysis that takes program changes into consideration is limited. There are several major challenges facing incremental analysis:

- **Dependency maintenance.** Incremental analysis needs to find out the part of existing computed points-to solutions influenced by the program changes. Without dependency information between the points-to results and the program structures, it is in general not possible to determine the impact of program changes. Therefore, the analysis must keep the relationship during the points-to analysis. Typically, however, only a small percentage of the program codes actually may change. The maintenance of this dependency is costly for most incremental analyses.

- **Expensive recomputation.** While affected points-to results are found, the incremental analysis must recompute their results in the modified program. For most exhaustive points-to analyses, the points-to sets of all variables are computed together. Therefore, the recomputation is often expensive or complicated.

- **High memory requirements.** Each variable in the points-to queries must maintain its relevant accumulated program results during analysis. These extra data structures tend to be large, with potentially hundreds to thousands of nodes. This problem is exacerbated for large programs with hundreds of thousands of nodes, which may consume a significant amount of memory.

4.3 Points-to Analysis with Incremental Summarisation

To overcome the challenges, our incremental approach relies on a *local CFL-reachability analysis* to compute context-independent CFL-reachability summaries for individ-

ual methods. As a result, these summaries are amenable to fast incremental re-summarisation upon code changes, leading to fast and precise on-demand points-to queries for IDEs.

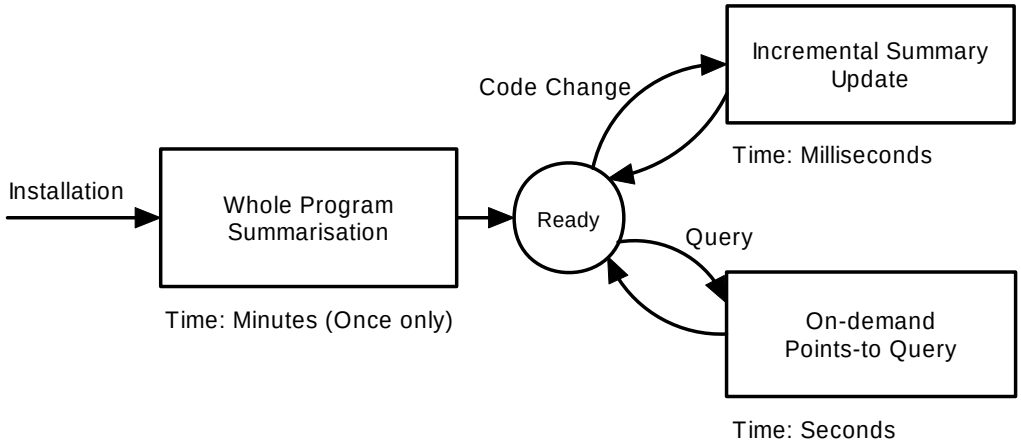


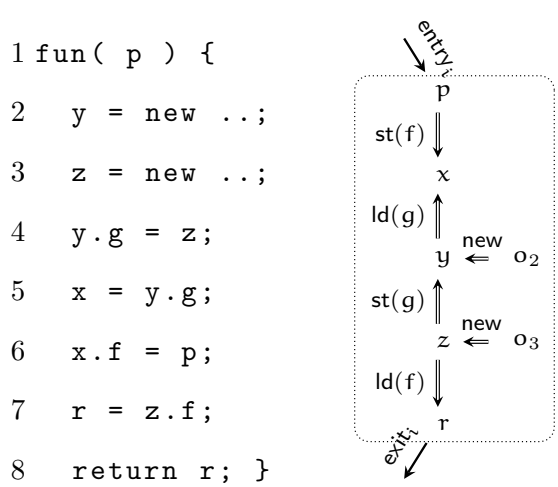
Figure 4.3: The framework structure of Emu.

As discussed in Chapter 3, we can categorise two types of edges in a PAG: local edges (new, assign, ld or st) and global edges (assignglobal, entry or exit). The key observation is that the impact of local change can be isolated by summarisation. Local edges are encapsulated within methods while global edges connect methods and global variables. Our local reachability analysis finds CFL-reachability relations between nodes in a PAG along only the local edges within a method field-sensitively but context-independently. For each parameter (including the implicit this) or return variable in a method, the summarisation process uses the local reachability analysis to construct and maintain a *CFL-reachability summary*. The summary in this chapter is similar to the technique introduced in Chapter 3 but more general and suitable

for handling of changes. A summary consists of all reachable local objects and local variables (parameters, returns and other locally defined variables) that are connected to a global edge. The summary for a variable is not computed for a specific on-demand field stack like Chapter 3, but constructed as a general summary including all unmatched field edges along the paths in the method. Such summaries enables fast on-demand points-to queries to be answered, as they can be recomputed quickly and independently in response to code changes.

The EMU framework comprises the three phases as shown in Figure 4.3. Initially, we compute and cache the local CFL-reachability summaries for all methods. The summaries may be updated individually for each given code change, and are always kept up-to-date. On-demand points-to queries are answered by using summaries and traversing context-dependent global edges to find field- and context-sensitive points-to set for each given query.

We specify our analysis by presenting our analysis algorithms here as well as a set of formal deductive formulations (similar to [23]) in Appendix A. Unlike the stack operations (Push(), Pop() and Peek()) used in Chapter 3, we present our algorithms in this chapter using syntactical equivalence, formally defined in Figure 4.5. We will explain each phase in detail using the motivating example in Figure 4.1 and others to demonstrate how it works. In particular, we will check the potential null dereference error in `e3.f` at line 34, before and after deleting the statement `t2[index] = v` in method `put` in line 14 for the motivating example. Our goal is to show how to update summaries and answer queries quickly and precisely.



step	state
1	$(r, S_1, \varnothing)$
2	$(z, S_1, \overline{f})$
3	$(z, S_2, \overline{f})$
4	$(y, S_1, \overline{f}{:}g)$
5	$(y, S_2, \overline{f}{:}g)$
6	$(x, S_2, \overline{f})$
7	$(p, S_1, \varnothing)$

Figure 4.4: CFL-reachability summarisation for an example.

4.3.1 Whole-Program Summarisation

To answer queries as precisely as possible in a small time budget, we use a once-off initialisation to compute and store local CFL-reachability summaries for formal parameters and return variables of all methods. The summarisation is similar to our dynamic summary introduced in Chapter 3, but there are some differences to be suitable for our incremental framework. Let us first consider a method with only local assignments, such as `fun(p){ ret`$_{\texttt{fun}}$ `= p; return ret`$_{\texttt{fun}}$`;}`. The summary for $\texttt{ret}_{\texttt{fun}}$ can be easily obtained as $\texttt{p} \Rightarrow \texttt{ret}_{\texttt{fun}}$, indicating that $\texttt{ret}_{\texttt{fun}}$ points to whatever p points to. Note that a summary for a method is always independent of its calling contexts so that it can be universally used.

In the presence of field accesses, as illustrated for `fun` in Figure 4.4, summarisation involves handling the balanced parentheses of ld and st. The process strictly follows the $\overline{\textit{flowsTo}}$ and *alias* RSMs for L_{FT} given in Figure 2.3(a), which has two states, S_1,

Directions	s	::=	$\mathsf{S}_1 \mid \mathsf{S}_2$
Call stacks	c	::=	$\varnothing \mid \mathsf{c}{:}\mathfrak{i}$
Field stacks	k	::=	$\varnothing \mid \mathsf{d} \mid -\mathsf{d} \mid \mathsf{k}{:}\mathsf{k}$
Directed fields	d	::=	$\mathsf{f} \mid \bar{\mathsf{f}}$
Caches	Γ	::=	$\varnothing \mid \Gamma \cup \{(\nu, \mathsf{s}) \mapsto \sigma\}$
Summaries	σ	::=	$\varnothing \mid \sigma \cup \{(\mathfrak{n}, \mathsf{s}, \mathsf{k})\}$
States	ζ	::=	$(\mathfrak{n}, \mathsf{s}, \mathsf{k}, \mathsf{c})$

Figure 4.5: Abstract domains.

representing $\overline{\mathit{flowsTo}}$, and S_2, representing $\mathit{flowsTo}$. Let us compute the summary for $\mathtt{r}$ with the steps taken shown in the right side of Figure 4.4. Because $\mathtt{r}$ is a return node, which may flow to a caller, we need to summarise its local $\overline{\mathit{flowsTo}}$ information, starting from S_1. The analysis first accepts a $\overline{\mathsf{ld}(\mathsf{f})}$ and then encounters $\mathtt{z}$. After having visited $\mathtt{o}_3$ and returning to $\mathtt{z}$, state S_2 is entered. Then the analysis can accept a $\mathsf{st}(\mathsf{g})$ edge to find an `alias` relation to match the $\mathsf{ld}(\mathsf{g})$ to reach $\mathtt{x}$. Now, $\mathtt{x}$ and $\mathtt{z}$ are known to be aliases. According to the leftmost transition in the `alias` RSM in Figure 2.3(a), the analysis can accept a $\overline{\mathsf{st}(\mathsf{f})}$ to reach $\mathtt{p}$. So we find that $\mathtt{p} \Rightarrow \mathtt{r}$, representing that the points-to set of $\mathtt{p}$ is in the points-to set of $\mathtt{r}$.

If we replace `x.f = p` by `t = p.h` and `x.f = t`, then the summary for $\mathtt{r}$ will be $\mathtt{p.h} \Rightarrow \mathtt{r}$. So the summary for a parameter/return consists of not only locally reachable objects and variables but also the fields to be matched later when the summary is applied.

Algorithm 6 Summarisation by local reachability analysis

ComputeSummaries()

1: $\Gamma \leftarrow \varnothing$
2: **for** each $v \xleftarrow{\mathsf{entry}} n$ **do**
3: $\quad \Gamma \leftarrow \Gamma \cup \{ v, \mathsf{S}_2 \mapsto \text{LocalReachable}(v, \mathsf{S}_2, \varnothing) \}$
4: **for** each $n \xleftarrow{\mathsf{exit}} v$ **do**
5: $\quad \Gamma \leftarrow \Gamma \cup \{ v, \mathsf{S}_1 \mapsto \text{LocalReachable}(v, \mathsf{S}_1, \varnothing) \}$

Abstract Domains

We now present the incremental analysis based on the CFL-reachability summarisation. In the presentation, we construct analysis algorithms from a program specified in Figure 2.1 with additional abstract domains defined in Figure 4.5.

In Figure 4.5, we provide the formal definitions for these semantic domains. The state s indicates the direction in which the analysis traverses: along a $\overline{\mathit{flowsTo}}$ path if $\mathsf{s} = \mathsf{S}_1$ or a *flowsTo* path if $\mathsf{s} = \mathsf{S}_2$. S_1 and S_2 correspond to the two states in the RSMs given in Figure 2.3. A *context (call stack)* c is defined as an ordered sequence of call sites to achieve context sensitivity by matching CallEntry and CallExit. Similarly, a *field stack* k is an ordered sequence of directed fields to achieve field sensitivity by matching field stores/loads. Directed fields are used to distinguish the traversal directions of ld and st edges, where $\bar{\mathsf{f}}$ is the inverse of f. We use a special annotation $(-\mathsf{d})$ to indicate a field that is not matched on the local field stack of a summary but may be matched later on the global field stack used during the global points-to analysis described in Section 4.3.3. A cache Γ maps a variable and a direction of the analysis to its summary, which contains its local reachability relations for the variable. A summary σ is a set of context-insensitive local states, representing the reachable states within a method by traversing its local edges only. Global states ζ

Algorithm 7 Local reachability analysis

LocalReachable($\mathfrak{n}, \mathsf{s}, \mathsf{k}$)

1: $\sigma \leftarrow \varnothing$
2: **if** $\mathsf{s} = \mathsf{S}_1$ **then**
3: **for** each $\mathfrak{n} \xleftarrow{\mathsf{new}} \mathsf{o}$ **do**
4: $\sigma \leftarrow \sigma \cup \{(\mathsf{o}, \mathsf{S}_1, \mathsf{k})\}$
5: $\sigma \leftarrow \sigma \cup$ LocalReachable($\mathfrak{n}, \mathsf{S}_2, \mathsf{k}$)
6: **for** each $\mathfrak{n} \xleftarrow{\mathsf{assign}} \mathfrak{n}'$ **do**
7: $\sigma \leftarrow \sigma \cup$ LocalReachable($\mathfrak{n}', \mathsf{S}_1, \mathsf{k}$)
8: **for** each $\mathfrak{n} \xleftarrow{\mathsf{ld}(\mathsf{f})} \mathfrak{n}'$ **do**
9: $\sigma \leftarrow \sigma \cup$ LocalReachable($\mathfrak{n}', \mathsf{S}_1, \mathsf{k}{:}\overline{\mathsf{f}}$)
10: **if** $\mathfrak{n} \xleftarrow{\mathsf{exit/entry/assignglobal}} \mathfrak{n}'$ **then**
11: $\sigma \leftarrow \sigma \cup \{(\mathfrak{n}, \mathsf{S}_1, \mathsf{k})\}$
12: **if** $\mathsf{s} = \mathsf{S}_2$ **then**
13: **for** each $\mathfrak{n}' \xleftarrow{\mathsf{assign}} \mathfrak{n}$ **do**
14: $\sigma \leftarrow \sigma \cup$ LocalReachable($\mathfrak{n}', \mathsf{S}_2, \mathsf{k}$)
15: **for** each $\mathfrak{n}' \xleftarrow{\mathsf{st}(\mathsf{f})} \mathfrak{n}$ **do**
16: $\sigma \leftarrow \sigma \cup$ LocalReachable($\mathfrak{n}', \mathsf{S}_2, \mathsf{k}{:}\mathsf{f}$)
17: **for** each $\mathfrak{n}' \xleftarrow{\mathsf{ld}(\mathsf{f})} \mathfrak{n}$ **do**
18: **if** $\mathsf{k} = \mathsf{k}'{:}\mathsf{f}$ **then**
19: $\sigma \leftarrow \sigma \cup$ LocalReachable($\mathfrak{n}', \mathsf{S}_2, \mathsf{k}'$)
20: **else if** $\mathsf{k} \neq \mathsf{k}'{:}\mathsf{d}$ **then**
21: $\sigma \leftarrow \sigma \cup$ LocalReachable($\mathfrak{n}', \mathsf{S}_2, \mathsf{k}{:}{-}\mathsf{f}$)
22: **for** each $\mathfrak{n} \xleftarrow{\mathsf{st}(\mathsf{f})} \mathfrak{n}'$ **do**
23: **if** $\mathsf{k} = \mathsf{k}'{:}\overline{\mathsf{f}}$ **then**
24: $\sigma \leftarrow \sigma \cup$ LocalReachable($\mathfrak{n}', \mathsf{S}_1, \mathsf{k}'$)
25: **else if** $\mathsf{k} \neq \mathsf{k}'{:}\mathsf{d}$ **then**
26: $\sigma \leftarrow \sigma \cup$ LocalReachable($\mathfrak{n}', \mathsf{S}_1, \mathsf{k}{:}{-}\overline{\mathsf{f}}$)
27: **if** $\mathfrak{n}' \xleftarrow{\mathsf{exit/entry/assignglobal}} \mathfrak{n}$ **then**
28: $\sigma \leftarrow \sigma \cup \{(\mathfrak{n}, \mathsf{S}_2, \mathsf{k})\}$
29: **return** σ

are used by the field- and context-sensitive global points-to analysis discussed later.

CFL-reachability-based Summarisation

The algorithms of the first phase are given in Algorithm 6 and Algorithm 7. Algorithm 6 shows the construction of summaries, which is the first stage of our incremental framework, by using the local reachability analysis defined in Algorithm 7.

Let us consider Algorithm 7 firstly. The algorithm in this part is similar to the dynamic summarisation algorithm (Algorithm 4) in Section 3.3. The difference is also obvious: Algorithm 4 computes summarisation for a specific field stack, while Algorithm 7 solves a general summary for all possible field stacks. The trick is to introduce the annotation $-\mathsf{d}$ to represent the unmatched field on the local reachability path in current method. In such a way, the field stack k, containing both fields to be matched in and out of the current method, can represent full information along the local reachable path. Since the summarisation is more general, we can maintain fewer summaries, which is suitable for incremental updates.

The description of algorithm details is as follows. In Algorithm 7, lines 2 – 11 handle state S_1, and lines 12 – 28 handle state S_2. The local transition computation is transitive for local reachability. The handling of each kind of statements in the algorithm corresponds to a transition in the $\overline{\mathit{flowsTo}}$ and *alias* RSMs for $\mathrm{L_{FT}}$ given in Figure 2.3(a). On encountering an edge $\mathsf{n} \xleftarrow{\mathsf{new}} \mathsf{o}$, the analysis finds the object o. Different from Algorithm 4, it will keep traversing a *flowsTo* path (by transiting to S_2) to find an *alias* relation between n and some n' such that n *alias* n' holds. The interesting part happens when a matching field is not found on the local field stack (note that unmatchable fields are excluded by $\mathsf{k} \neq \mathsf{k}'{:}\mathsf{d}$), we push the field onto the

local field stack for it to be recorded in the summary. We resolve inter-procedural field matching when the summary is used by the on-demand points-to analysis described in Section 4.3.3.

During the initialisation phase shown in Algorithm 6, we compute a *flowsTo* summary in S_2 for each parameter and a $\overline{\textit{flowsTo}}$ summary in S_1 for the return, with the field stack being empty initially. The process exhaustively searches for and includes (1) all reachable local objects (always in state S_1) and (2) all reachable local variables connected to an incoming global edge in S_1 or an outgoing global edge in S_2.

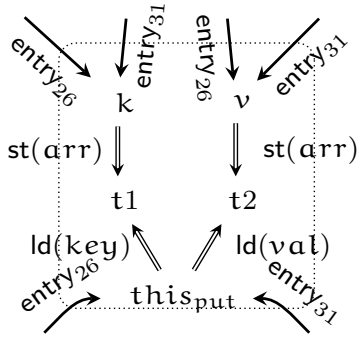
Now let us consider `put` in the `Hashtable` class in our motivating example. According to our algorithms, the summarisation will produce the following:

$$\begin{aligned}
\Gamma((\mathtt{this}_{\mathtt{put}}, \mathsf{S}_2)) &= \{(\mathsf{k}, \mathsf{S}_1, -\mathtt{key}{:}-\overline{\mathtt{arr}}), (\mathsf{v}, \mathsf{S}_1, -\mathtt{val}{:}-\overline{\mathtt{arr}})\} \\
\Gamma((\mathsf{v}, \mathsf{S}_2)) &= \{(\mathtt{this}_{\mathtt{put}}, \mathsf{S}_1, \mathtt{arr}{:}\overline{\mathtt{val}})\} \\
\Gamma((\mathsf{k}, \mathsf{S}_2)) &= \{(\mathtt{this}_{\mathtt{put}}, \mathsf{S}_1, \mathtt{arr}{:}\overline{\mathtt{key}})\}
\end{aligned}$$

Note that the unmatched fields, e.g., $-\mathtt{key}$, $-\mathtt{val}$, $-\overline{\mathtt{arr}}$ in the summary of $\mathtt{this}_{\mathtt{put}}$, are added in lines 21 and 26 in Algorithm 7, which is a significant difference from the dynamic summary in Chapter 3.

4.3.2 Incremental Summary Update

After its installation, EMU enters into a ready state, as shown in Figure 4.3. If a code change is made in a method, then its local reachability summaries are updated. Existing incremental points-to analyses [28, 49, 73] typically cache the points-to sets of the variables queried earlier and update (often approximately) all affected points-to sets after a code change, leading to unbounded (slow) and imprecise propagation of

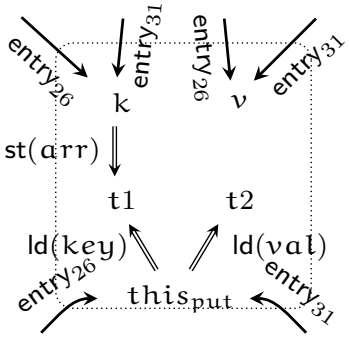


k reaches $\mathsf{this}_{\mathrm{put}}$

v reaches $\mathsf{this}_{\mathrm{put}}$

$\mathsf{this}_{\mathrm{put}}$ reaches k, v

(a) Before



k reaches $\mathsf{this}_{\mathrm{put}}$ (unchanged)

v reaches null (changed)

$\mathsf{this}_{\mathrm{put}}$ reaches k (changed)

(b) After

Figure 4.6: Summaries (with field stacks omitted) before and after deleting "`t2[index] = v`" in `put` in line 14 in Figure 4.1.

changed points-to information. Since our summaries store local reachability relations, we only need to update the affected summaries for the modified methods. Such modularity leads to a fast and precise analysis suitable for IDEs.

We do not consider the modifications on software design that may trigger class hierarchy and/or interface changes. In the case of such major design changes, EMU can be re-installed as suggested in Figure 4.3 to re-produce a program-wise summarisation for all methods (in parallel if needed due to the modularity of our approach). Therefore, we consider only atomic changes made in a program, i.e., additions or deletions of edges in the PAG. Note that adding/deleting a call statement consists of adding/deleting the entry and exit edges for all possible methods invoked.

Figure 4.6 shows how the summaries for `put` in our motivating example are updated incrementally after its line 14 is deleted. Before the change, the summaries are given in (4.1). After the change, the summaries should be updated as illustrated.

The summaries for all modified methods are updated independently. To update a method, the algorithms given in Algorithm 8 are applied. The Update method below performs the actual re-summarisation while SumUpdate method determines what summaries are affected and need to be recomputed for different types of edges. We do not distinguish edge insertions and deletions, by treating both as edge changes.

Algorithm 8 Summary update for code change

SumUpdate($\Gamma, \mathtt{stmt}$)

1: **if** $\mathtt{stmt} = n \xleftarrow{\mathsf{new/assign/ld/st}} n'$ **then**
2: Update(Γ, n, S_2)
3: Update(Γ, n', S_1)
4: **if** $\mathtt{stmt} = n \xleftarrow{\mathsf{entry}} n'$ **then**
5: Update(Γ, n', S_1)
6: **if** $\mathtt{stmt} = n' \xleftarrow{\mathsf{exit}} n$ **then**
7: Update(Γ, n', S_2)
8: **if** $\mathtt{stmt} = g \xleftarrow{\mathsf{assignglobal}} n$ **then**
9: Update(Γ, n, S_1)
10: **if** $\mathtt{stmt} = n \xleftarrow{\mathsf{assignglobal}} g$ **then**
11: Update(Γ, n, S_2)

Update(Γ, n, s)

1: **for** each $(n', s', k') \in$ LocalReachable$(n, s, \varnothing)$ **do**
2: remove $(n', \overline{s'}) \mapsto \sigma$ from Γ
3: $\Gamma = \Gamma \cup \{ (n', \overline{s'}) \mapsto$ LocalReachable$(n', \overline{s'}, \varnothing) \}$

SumUpdate function in Algorithm 8 performs updates for various statements. Changes of all local edges are treated uniformly in lines 1 – 3, where we update summaries of variables that $\mathtt{n}$ may flow to (in S_2) and summaries of variables that $\mathtt{n}'$ may points to (in S_1). Then in Update function, we first search for the affected variables in the given direction, then recompute their summaries in the reversed direction (note that $\overline{\mathsf{S}_1} = \mathsf{S}_2$ and $\overline{\mathsf{S}_2} = \mathsf{S}_1$), and finally, update the cache.

Changes of global edges are treated individually. In lines 4 – 5, we recompute the summaries of local variables that $\mathtt{n}'$ may point to. We do not consider $\mathtt{n}$ that is a parameter of another method whose summaries are in the cache and not affected. Similarly, in lines 6 – 7, we only need to update summaries of local variables that $\mathtt{n}'$ may flow to, because $\mathtt{n}$ is a return variable in another method.

For changes made on a global assignment in lines 8 – 11, we consider only the local variables (if any) involved in the assignment. Otherwise, then temporary local variables can be introduced.

To re-summarise `put` in Figure 4.6, we apply Algorithm 8 to find that the summaries of $\mathtt{v}$ and $\mathtt{this}_{\mathtt{put}}$ are affected but not $\mathtt{k}$:

$$\begin{aligned}\Gamma((\mathtt{this}_{\mathtt{put}}, \mathsf{S}_2)) &= \{(\mathtt{k}, \mathsf{S}_1, -\mathtt{key}{:}-\overline{\mathtt{arr}})\}\\ \Gamma((\mathtt{v}, \mathsf{S}_2)) &= \varnothing\\ \Gamma((\mathtt{k}, \mathsf{S}_2)) &= \{(\mathtt{this}_{\mathtt{put}}, \mathsf{S}_1, \mathtt{arr}{:}\overline{\mathtt{key}})\}\end{aligned}$$

4.3.3 On-demand Points-to Query

With up-to-date summaries, EMU answers on-demand points-to queries from developers or other analyses by using local summaries and traversing global edges. EMU

performs on-demand points-to analysis as described in Section 4.2 except it uses reachability-based method summaries to speed up the analysis. We will first present our algorithm and then illustrate it with our motivating example.

Consider Figure 4.6(a). Suppose when a particular query is processed, EMU needs to determine where the value of $\mathtt{v}$ flows to in `put`. This can be found directly from the local summary $\Gamma((\mathtt{v}, \mathtt{S}_2))$ given in (4.1), which means that $\mathtt{v}$ flows into $\mathtt{this}_{\mathtt{put}}.\mathtt{val}.\mathtt{arr}$, so that redundant traversals along the path are avoided.

Algorithm 9 shows the algorithm for our summary-based on-demand points-to analysis with Γ, which contains all up-to-date summaries. The outer while loop picks up a tuple $(\mathtt{n}, \mathtt{s}, \mathtt{k}, \mathtt{c}')$ from the worklist, which includes both field stack $\mathtt{k}$ and context stack $\mathtt{c}'$. Note that we distinguish two types of field stacks: *local field stack* used in method summaries and *global field stack* used here for on-demand points-to analysis. The only place adding element to the points-to set `pts` is in lines 5 – 6, when the final global field stack is empty—all fields introduced by ld and st edges must have been matched up. The local field stack that appears in a summary may contain specially annotated fields (i.e., $-\mathtt{d}$) to be matched on the global field stack. We may remove a field from the global field stack by defining a syntactical equivalence on two types of field stacks:

$$\mathtt{k}{:}\mathtt{d}{:}-\mathtt{d}{:}\mathtt{k}' \equiv \mathtt{k}{:}\mathtt{k}' \qquad \varnothing{:}\mathtt{k} \equiv \mathtt{k} \qquad \mathtt{k}{:}\varnothing \equiv \mathtt{k}$$

which is used in lines 13 – 14 to merge both.

In Algorithm 9, lines 8 – 10 are used for queried local variables that have no summaries (local variables other than method parameters and returns), in which case, we need to perform a local reachability analysis. Lines 11 – 12 deal with context-dependent global edges by a global reachability analysis specified in Algorithm 10.

Algorithm 9 Summary-based points-to analysis

PointsTo(Γ, v, c)

```
 1: pts ← ∅
 2: w ← {(v, S1, ∅, c)}
 3: while w ≠ ∅ do
 4:    remove (n, s, k, c′) from w
 5:    if n ∈ O AND k = ∅ then
 6:       pts ← pts ∪ {(n, c′)}
 7:    else
 8:       if (n, s) ∉ dom(Γ) then
 9:          for each (n′, s′, k′) ∈ LocalReachable(n, s, k) do
10:             Propagate(w, n′, s′, k′, c′)
11:       for each (n′, s′, c′) ∈ GlobalReachable(n, s, c) do
12:          Propagate(w, n′, s′, k, c′)
13:       for each (n′, s′, k′) ∈ Γ((n, s)) AND −d ∉ k:k′ do
14:          Propagate(w, n′, s′, k:k′, c′)
15: return pts
```

Propagate(w, n, s, k, c)

```
1: w ← w ∪ {(n, s, k, c)}
```

Lines 13 – 14 look up and reuse the existing summaries available in the cache and requires that any to-be-matched fields in the local field stack of a summary to be matched on the global field stack.

Algorithm 10 is concerned with global reachability analysis for handling context-dependent global edges according to the RSM for R_{RP} in Figure 2.3(b). As discussed in Section 2.3.3, the analysis is context-sensitive for method invocation and heap

abstraction.

Let us compute $\textsc{Pts}(e3)$ for our motivating example given in Figure 4.1 before and after `t2[index] = v` in line 14 is deleted. Before the deletion, we can find a points-to path as follows:

$$
\begin{array}{lll}
 & (e3, \mathsf{S}_1, \varnothing, \varnothing) & \text{initial state} \\
\longrightarrow & (\mathtt{this}_{\mathtt{put}}, \mathsf{S}_2, \overline{\mathtt{arr}}{:}\mathtt{val}, 26) & \\
\longrightarrow & (v, \mathsf{S}_1, \overline{\mathtt{arr}}{:}\mathtt{val}{:}-\mathtt{val}{:}-\overline{\mathtt{arr}}, 26) & \\
\equiv & (v, \mathsf{S}_1, \varnothing, 26) & \text{by syntactical equivalence} \\
\longrightarrow & (\mathsf{o}_{23}, \mathsf{S}_1, \varnothing, \varnothing) &
\end{array}
$$

Each tuple is a worklist element, which shows the reachable states from initial query $(e3, \mathsf{S}_1, \varnothing, \varnothing)$ to the final allocation $(\mathsf{o}_{23}, \mathsf{S}_1, \varnothing, \varnothing)$. The null dereferencing checker will not report an error for accessing $e3.\mathtt{f}$. Note that syntactical equivalence of unmatched field $(-\mathtt{d})$ is used during the analysis on the global field stack, which corresponds to lines 13 – 14 in Algorithm 9.

After the deletion, we cannot find any points-to path to an object:

$$
\begin{array}{lll}
 & (e3, \mathsf{S}_1, \varnothing, \varnothing) & \text{initial state} \\
\longrightarrow & (\mathtt{this}_{\mathtt{put}}, \mathsf{S}_2, \overline{\mathtt{arr}}{:}\mathtt{val}, 26) & \\
\longrightarrow & (v, \mathsf{S}_1, \overline{\mathtt{arr}}{:}\mathtt{val}{:}-\mathtt{key}{:}-\overline{\mathtt{arr}}, 26) & \\
\equiv & ? & \textbf{fields cannot be matched!}
\end{array}
$$

Thus, a null dereferencing error on $e3.\mathtt{f}$ is reported.

4.3.4 Handling Recursion and Call Graph

So far we have not considered recursive calls and changes on the call graph in a program, since their treatments are rather standard. For recursive calls, we simply

Algorithm 10 Global reachability analysis

GlobalReachable($\mathtt{n}, \mathtt{s}, \mathtt{c}$)

1: $\mathtt{sta} \leftarrow \varnothing$
2: **if** $\mathtt{s} = \mathtt{S}_1$ **then**
3: **for** each $\mathtt{n} \xleftarrow{\mathsf{exit}_\mathtt{i}} \mathtt{n}'$ **do**
4: $\mathtt{sta} \leftarrow \mathtt{sta} \cup \{ (\mathtt{n}', \mathtt{S}_1, \mathtt{c}{:}\mathtt{i}) \}$
5: **for** each $\mathtt{n} \xleftarrow{\mathsf{entry}_\mathtt{i}} \mathtt{n}'$ **do**
6: **if** $\mathtt{c} = \varnothing$ **then**
7: $\mathtt{sta} \leftarrow \mathtt{sta} \cup \{ (\mathtt{n}', \mathtt{S}_1, \varnothing) \}$
8: **else if** $\mathtt{c} = \mathtt{c}'{:}\mathtt{i}$ **then**
9: $\mathtt{sta} \leftarrow \mathtt{sta} \cup \{ (\mathtt{n}', \mathtt{S}_1, \mathtt{c}') \}$
10: **for** each $\mathtt{n} \xleftarrow{\mathsf{assignglobal}} \mathtt{n}'$ **do**
11: $\mathtt{sta} \leftarrow \mathtt{sta} \cup \{ (\mathtt{n}', \mathtt{S}_1, \varnothing) \}$
12: **if** $\mathtt{s} = \mathtt{S}_2$ **then**
13: **for** each $\mathtt{n}' \xleftarrow{\mathsf{exit}_\mathtt{i}} \mathtt{n}$ **do**
14: **if** $\mathtt{c} = \varnothing$ **then**
15: $\mathtt{sta} \leftarrow \mathtt{sta} \cup \{ (\mathtt{n}', \mathtt{S}_2, \varnothing) \}$
16: **else if** $\mathtt{c} = \mathtt{c}'{:}\mathtt{i}$ **then**
17: $\mathtt{sta} \leftarrow \mathtt{sta} \cup \{ (\mathtt{n}', \mathtt{S}_2, \mathtt{c}') \}$
18: **for** each $\mathtt{n}' \xleftarrow{\mathsf{entry}_\mathtt{i}} \mathtt{n}$ **do**
19: $\mathtt{sta} \leftarrow \mathtt{sta} \cup \{ (\mathtt{n}', \mathtt{S}_2, \mathtt{c}{:}\mathtt{i}) \}$
20: **for** each $\mathtt{n}' \xleftarrow{\mathsf{assignglobal}} \mathtt{n}$ **do**
21: $\mathtt{sta} \leftarrow \mathtt{sta} \cup \{ (\mathtt{n}', \mathtt{S}_2, \varnothing) \}$
22: **return** sta

approximate the calling edges in the strongly-connected components (SCCs) of the call graph as gotos. To maintain a precise context-sensitive call graph, we construct the call graph on-the-fly while performing on-demand points-to analysis. Whenever we encounter a virtual method call $v.fun()$, we initiate a points-to query for v to find the possible callees (target objects of $fun()$) and include them in the call graph. For virtual calls whose targets cyclically depend on each other, we track the pending queries and re-propagating them as new call targets, a standard technique used in demand-driven analyses [54].

4.4 Evaluation

Our evaluation has validated the following two hypotheses:

- **Our modular approach is scalable.** The update cost for small code changes does not grow with the program size as it is bounded, i.e., localised to where the change is made.
- **Our query processing is fast and precise.** Given a pre-set time budget, Emu takes at most 0.054 secs on average to correctly answer at least 97.4% of all queries raised compared to an exhaustive exploration for all three clients used.

4.4.1 Methodology

We evaluate the suitability of Emu for deployment in an IDE in terms of its efficiency and precision in response to small code changes, such as adding/deleting statements (assignments or calls). We have selected three representative clients, safe

casting, null dereferencing and factory methods, using seven Java programs from the Dacapo benchmark suite. We compare EMU with REFINEPTS, a state-of-the-art demand-driven points-to analysis [54] also formulated via CFL reachability. Due to the incremental CFL-reachability-based summarisation used, EMU is faster and more precise and can be promisingly used in IDEs where the changes are small.

Implementation We have implemented EMU in Soot-2.4.0 [60], which is integrated into the Eclipse IDE.

All experiments were conducted on a machine with an Intel Xeon 3.0GHz processor (4 cores) with 16 GB memory, running RedHat Enterprise Linux 5 (kernel 2.6.18) and Sun JDK 1.6.0_23.

Clients We evaluate the incremental analysis with three clients: `SafeCast`, `FactoryM` and `NullDeref`, which have been introduced in Section 3.4. Each query is answered in a pre-set time budget. Failing to complete its exploration within the budget, an analysis will terminate with a conservative answer (e.g., the downcast tested may not be safe). In our experiments, the default budget is 75K steps, indicating the maximum number of edges that can be traversed in a PAG per query. This is a tunable parameter discussed further below.

Code Changes As discussed in Section 4.3.2, we consider only small changes at the level of statements. For a program, we randomly select four kinds of changes corresponding to adding/deleting assignments and calls. We exclude statements that do not impact our queries. We present and analyse our results for four tests (or four changes) with each test consisting of adding or deleting 10 statements in the program:

del (deleting 10 assignments) and add (adding the 10 deleted assignments back), delC (deleting 10 call statements) and addC (adding the 10 deleted calls back). As EMU is modular, the results for many other changes tested by us are similar in both analysis times and precision reported.

Efficiency and Precision For each code change, we report the times taken by EMU and REFINEPTS and also compare their precision achieved. For a given client, the precision achieved by an analysis is compared to that achieved by an exhaustive analysis without any time budget constraint (equivalent to a whole-program analysis).

4.4.2 Results and Analysis

We present and analyse our results for `SafeCast` in detail in Section 4.4.2 and discuss the other two, `FactoryM` and `NullDeref`, briefly in Section 4.4.2.

Our results show that EMU is fast enough to be used in IDEs and also more precise than REFINEPTS for small code changes tested.

`SafeCast` We examine first analysis times and then precision.

Analysis Times Table 4.1 gives the analysis times from both REFINEPTS and EMU for the Java programs used. The "#M (K)" column gives the number of reachable methods in the PAG in thousands for each program. Note that the numbers here are larger than those in [54] because we used a different Java JDK library (1.6.0). The "#Q" column gives the number of queries raised for each program. A large number of queries are issued in order to accurately measure the average and worst-case times taken in answering each query.

Benchmark	#M (K)	#Q	Change	RefinePTS (secs)		Emu (secs)		
				BT	CT	IT	UT	DT
soot-c	10.4	973	del	43.3	131.5	41.2	0.015	47.4
			add	43.0	125.5		0.013	44.6
			delC	43.0	125.4		0.010	44.8
			addC	44.7	135.7		0.011	48.7
sablecc-j	21.4	358	del	127.3	76.6	63.6	0.016	31.8
			add	126.7	77.0		0.013	32.0
			delC	126.9	71.7		0.012	31.5
			addC	127.9	79.3		0.013	31.7
antlr	12.9	281	del	35.9	15.4	12.5	0.015	4.8
			add	35.9	15.5		0.014	4.8
			delC	36.6	15.1		0.020	4.8
			addC	36.0	15.4		0.013	4.8
bloat	10.8	1221	del	43.8	143.9	24.2	0.009	57.7
			add	43.2	145.8		0.015	57.7
			delC	43.7	144.9		0.011	57.9
			addC	46.0	148.1		0.012	58.9
chart	17.4	682	del	143.0	191.8	74.5	0.011	68.9
			add	140.1	192.4		0.016	69.0
			delC	141.3	191.9		0.007	68.9
			addC	142.3	194.1		0.005	69.6
jython	27.5	524	del	38.3	27.5	28.0	0.015	5.2
			add	38.6	27.5		0.016	5.2
			delC	38.8	26.9		0.015	5.1
			addC	38.7	27.6		0.014	5.2
ps	13.5	660	del	128.4	96.1	63.4	0.011	42.3
			add	127.7	97.5		0.011	44.3
			delC	124.8	104.7		0.007	46.8
			addC	129.7	94.7		0.007	42.3

Table 4.1: Analysis times of `SafeCast` by RefinePTS and Emu. #M is the number of methods (in thousands) in Soot's context-insensitive call graph. #Q is the number of queries raised.

In the "Change" column, the four tests are performed in that order for each program. The analysis times consumed by RefinePTS and Emu, given in the last five columns in Table 4.1, are discussed below.

RefinePTS After a code change is made in a program, the time taken per query is given by BT + CT/#Q, where BT represents the time spent on rebuilding the PAG for the program by using a context-insensitive pointer analysis provided in Soot and CT represents the time spent by RefinePTS to answer all queries raised for the program. For RefinePTS [54] and other state-of-the-art demand-driven points-to analyses [50, 70, 74], the cost for rebuilding the PAG after each code change is too high to make it suitable for them to be directly deployed in IDEs.

Emu For each program, Emu only needs to build its PAG initially once by taking the time indicated by BT for RefinePTS. Subsequently, Emu proceeds according to the three phases, Whole-Program Summarisation, Incremental Summary Update, and On-demand Query, illustrated in Figure 4.3, consuming the times as given in the last three columns in Table 4.1. Whole-Program Summarisation happens only once for a program. IT gives the time for computing the local CFL-reachability summaries for all methods in its PAG. As our approach is modular, IT can be significantly reduced (in parallel) if needed.

After each code change is made in a program, the (amortised) time per query taken by Emu is given by DT/#Q if Incremental Summary Update can complete before On-demand Query starts and is bounded by (UT+DT)/#Q otherwise. Here, UT gives the time for updating the summaries affected by the code

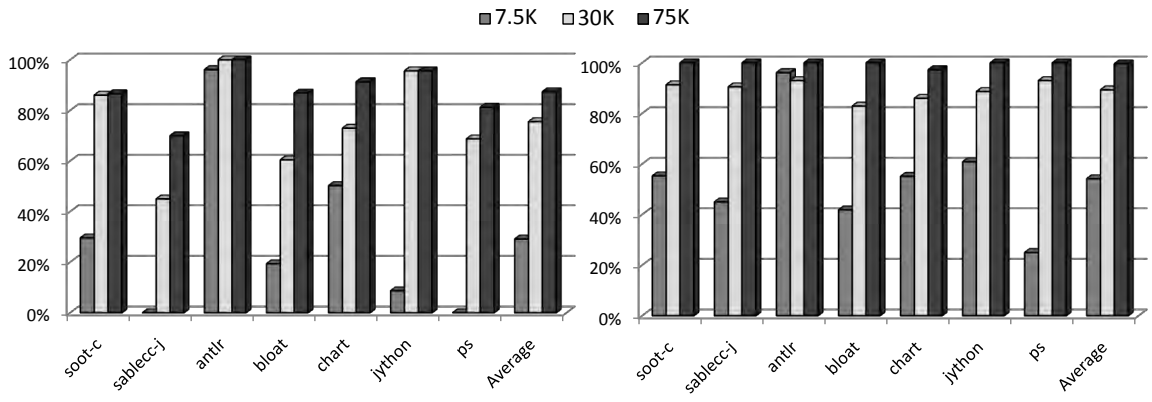
change and DT gives the time taken by Emu in answering the queries issued. As Emu takes a modular approach to updating summaries, UT is under 20ms for all the queries combined. In addition, Emu is about 3× faster in answering queries than RefinePTS (by comparing the CT and DT columns).

For all the three clients tested, Emu can answer each query under 0.054 secs on average and under 0.87 secs in the worst case. There are no significant performance variations among the four tests due to the modularity of our approach.

Memory Usage Because garbage collection is enabled, it is difficult to monitor memory usage precisely (a well-known fact for JVMs). We measured the memory usage of the whole JVM heap with and without summaries. We find that summarisation consumes only slightly more memory, ranging from 21MB to 70MB across the seven benchmarks with an average of 46.25MB. This translates into a percentage increase across the benchmarks, ranging 3.27% to 16.24% with an average of 6.94%.

Precision We measure the precision of RefinePTS and Emu with different budgets against an exhaustive analysis without any budget constraint. As described in [54], `SafeCast` is an exacting test of points-to analysis precision, especially of the ability to distinguish the contents of different data structures. Figure 4.7 compares RefinePTS and Emu with the precision of `SafeCast` being defined as the percentage of queries that gives the same answers as the exhaustive analysis under three budgets, 7.5K, 30K and 75K. Being summary-based, Emu is no less precise than RefinePTS as Emu avoids making redundant traversals across the queries on the PAG. For RefinePTS, the average precision percentages for 7.5K, 30K and 75K are 29.16%, 75.55% and 87.38%, respectively. For Emu, better ones, 54.19%, 89.34% and

99.61%, are achieved, respectively. In the case of 75K, the default budget used in Table 4.1, EMU can answer nearly all the queries (99.6% on average) positively compared to an exhaustive analysis, proving 13.2% more safe downcasts than REFINEPTS.



(a) REFINEPTS (b) EMU

Figure 4.7: Precision of `SafeCast`.

`FactoryM` and `NullDeref` As in the case of `SafeCast`, REFINEPTS must be charged for the cost of rebuilding the PAG after each code change and is no more precise than EMU. So we will no longer compare it directly with EMU.

The results given in Table 4.2 for these two clients show the same trend as `SafeCast`. UT never exceeds 20ms. In addition, each query can be answered under 0.035s and 0.025s on average for `FactoryM` client and `NullDeref`, respectively. No more than 0.87 secs are needed for each query in the worst case for both clients. Finally, under the default budget of 75K, `FactoryM` client and `NullDeref` achieve a precision of 98.5% and 97.4%, respectively, compared against an exhaustive analysis.

4.5 Related Work

Our discussion is restricted to four related areas: context-sensitive points-to analysis (depending on whether if it is whole-program, demand-driven or incremental) and change impact analysis. As shown for `SafeCast`, context sensitivity is needed in

Benchmark	`NullDeref`			
	#Q	UT (secs)	DT (secs)	Prec (%)
soot-c	2047	0.012	38.7	99.7
sablecc-j	1084	0.008	23.2	97.2
antlr	650	0.016	19.5	100.0
bloat	3007	0.011	77.3	95.7
chart	1281	0.011	97.4	93.0
jython	2411	0.008	68.8	97.5
ps	1426	0.012	69.0	98.4
Benchmark	`FactoryM`			
	#Q	UT (secs)	DT (secs)	Prec (%)
soot-c	619	0.013	6.0	100.0
sablecc-j	331	0.007	6.5	96.1
antlr	146	0.009	3.3	100.0
bloat	813	0.009	18.9	97.8
chart	195	0.014	9.7	99.8
jython	214	0.008	4.5	100.0
ps	326	0.010	8.6	97.4

Table 4.2: `NullDeref` and `FactoryM` in Emu. "Prec" denotes the precision at a budget of 75K against an exhaustive analysis.

IDEs for Java as all queries issued cannot be positively answered otherwise.

Whole-Program Points-to Analysis Context sensitivity is achieved by cloning [61] or summarisation [26, 62, 72]. Cloning is trivially context-sensitive for method invocation and heap abstraction since distinct calls to a method are diverted to its distinct clones. Unfortunately, this does not scale to large programs and is also unsuitable for IDEs. Summarisation suffers from the unbounded propagation of changed points-to information as discussed in Section 4.1. In contrast, the summarisation approach described in this chapter is modular since the impact of a code change is bounded, i.e., localised to the method where the code change is made. In addition, full context sensitivity is usually approximated by using aliases holding at call sites [62, 72] for method calls and by recognising allocation wrappers for heap abstraction.

Demand-Driven Points-to Analysis A demand-driven points-to analysis computes points-to information for a set of specified variables rather than all the variables in a program. The state-of-the-art algorithms for Java [50, 54, 70] and C [74] are all formulated in terms of CFL-reachability initially introduced in [44, 45]. Given a CFL-reachability formulation, a demand-driven analysis resolves pointers and aliases as described in Section 4.2.

However, none of these algorithms are directly formulated to deal with code changes efficiently, as validated for a representative solution in our experiments. The novelty of this work lies in using context-independent CFL-reachability summaries and adopting a modular approach to incrementally updating the affected summaries to avoid redundant updates after code changes are made.

Incremental Points-to Analysis Existing techniques are not formulated directly for the IDEs targeted by this work. Earlier, Yur et al. [73] introduced an incremental approximation of their previous flow- and context-sensitive points-to analysis [29] for C, achieving a 6-fold speedup with a precision loss (solution agreement on 75% of tests on average) for programs with 1 – 25KLOC. Kodumal and Aiken [28] discussed a timestamp-based analysis in their Banshee toolkit. For a code change, their coarse-grained analysis backtracks to the first affected constraint and reanalyses the program from that point forward. Their analysis is fast but imprecise for IDEs due to its lack of support for context sensitivity. Saha and Ramakrishnan [49] presented a solution also for C based on logic deduction rules. When context sensitivity is considered, their analysis is slow, by consuming 50 – 73% of the from-scratch time. These approaches typically falsify all invalid points-to sets greedily rather than on demand and then rederive them. In contrast, Emu provides instant feedback to developers for their queries by performing incremental CFL-reachability-based summarisation while achieving nearly the same precision of a whole-program analysis.

Change Impact Analysis Change impact analysis determines the effects of code changes to support the planning, implementation and validation of code changes in software evolution and maintenance. A taxonomy for impact analysis can be found in [30]. Recent approaches [1, 6, 17, 43] rely on slicing, dependence analysis, dynamic tracing and history mining. In general, impact analysis requires fast and precise points-to information to be effective. According to [1], change impact analysis needs to be performed in nightly build environments as developers need fast access to the impact information for risk analysis, effort estimation and regression testing. Unfortunately, precise flow- and context-sensitive points-to analyses still do not scale

even for small programs to enable them to be used during a nightly build. Unlike such whole-program points-to analyses, the points-to analysis proposed in this chapter is fast for IDEs, achieving nearly the same precision as a whole-program field- and context-sensitive points-to analysis.

4.6 Chapter Summary

In this chapter, we have introduced a new points-to analysis that enables software developers to perform pointer-related queries on demand while making code changes to the programs being developed. Our modular approach, which is formulated in terms of CFL-reachability, allows method summaries to be incrementally updated after some small code changes are made. By bounding the propagation of changed points-to information, pointer-related queries can be answered quickly, with high precision. Our approach can be promisingly deployed in IDEs to facilitate program understanding and debugging when changes are small and frequent.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Modern software is becoming more complex in size and functionality, and it is more difficult to understand or optimise a large program without automated program analysis techniques. Points-to analysis, which is to resolve points-to information, is a fundamental technology for static program analysis to manage the complexity. The precision of points-to solutions is significant to subsequent program analysis and other clients, especially for object-oriented software systems. Therefore, developing highly efficient and precise points-to analysis becomes vitally important.

In this thesis, we investigate new approaches aimed at improving points-to analysis scalability without affecting precision. The key observation on which our work relies is identifying and exploiting different types of *locality*: the locality when traversing local statements in Chapter 3 and the locality when handling local program changes in Chapter 4 to avoid redundant computations. We believe that understanding such

software features is necessary to develop effective solutions for improving the performance of points-to analysis based on CFL-reachability.

Based on these observations, this thesis first presents an approach that dynamically exploits the local reachability reuse to improve the performance of CFL-reachability based demand-driven points-to analysis (Chapter 3). The second approach proposed by the thesis addresses the points-to analysis for a program that undergoes a lot of small changes by providing the CFL-reachability summarisation to localise the change impact (Chapter 4). Central to these two approaches is to exploit reusability and modularity when analysing local statements based on CFL-reachability. Evaluation and validation using several clients over a range of Java benchmarks show that our approaches can significantly boost the performance of the state-of-the-art points-to analysis without affecting precision.

We expect the techniques developed in this thesis to be useful in program analysis for modern object-oriented software applications.

5.2 Future Work

Points-to analysis based on CFL-reachability has been extensively studied recently and novel techniques have been proposed to answer demand queries efficiently. However, there are still a number of interesting problems to be investigated in the future.

Generalisation Many clients other than those we tested could benefit from the scalability and precision of our CFL-reachability-based summarisation techniques. By using our points-to analysis, a number of optimisations and bug detections may be able to scale to much larger programs. Furthermore, we believe that the principle

of locality goes beyond context-sensitive points-to analysis; it can be exploited for other types of program analysis as well. Our summarisation techniques are developed for a specific points-to formulation. However, it remains an open question as to whether the general CFL-reachability problems can be accelerated using summarisation. Exploiting locality properties in new program analysis is an interesting direction for future work.

Practical incremental points-to analysis Our incremental summarisation technique is promising for IDEs. A practical IDE-based implementation of our points-to analysis would pose interesting engineering challenges. One key issue would be how to maintain the program representations while making various changes in IDEs. Our current research is still limited to some simple and small changes in Java programs. How to handle more complex code changes, including method and class level changes, will be a future topic. Moreover, it is more challenging to incrementally analyse large-scale software systems.

Annotated CFL-reachability While CFL-reachability formulations have been extremely successful in modelling points-to or alias relations, they can be potentially used for other purposes as well. In the current formulation, the edges in a PAG are either fields or calling contexts. Looking a bit far into the future, we may consider to annotate the edges with more information, such as loop counts and execution frequencies, which could be expected to produce some powerful analyses.

Dynamic Class Loading The analysis introduced in this thesis assumes a closed world in the sense that only classes reachable from the main method of a program

at analysis time can be used by the program at run time. It will be interesting to generalise this work in the presence of dynamic class loading [39, 53, 68, 69].

Demand-Driven Analysis for Concurrency Bugs Static analysis [36] for detecting data races, deadlocks and atomicity violations suffers from high false positive rates. On the other hand, dynamic analysis has low coverage [16, 63, 64]. One possible future research is to investigate how to combine CFL-reachability-based static analysis with dynamic instrumentation in detecting concurrency bugs more efficiently and effectively.

Appendix A

Formal Rules

The analyses introduced in our thesis are presented using algorithms. In this part, we state the alternative formal deduction formulations for some of the analyses .

We construct the deduction rules (similar to [23]) from a program as specified in Figure A.1 (the syntax) with additional abstract domains in Figure A.2.

Firstly, we give the deduction formulations of our baseline analysis RefinePTS in Figure A.3. Note that if we omit the refinement rule, it just represents the basic CFL-reachability-based points-to analysis.

In the following, we present the deduction rules of our points-to analysis with incremental summarisation introduced in Chapter 4. We specify our local reachability analysis using deductive reachability formulations. The reachability analysis is described by a set of deduction rules in the form of:

$$(n, s, k) \Longrightarrow (n', s', k')$$

which follow only local edges and are context-independent. Each local edge in a PAG is translated into one or more deduction rules. For example, given a points-to query

for variable v in state S_1 with field stack k, we may conclude that v reaches o with field stack k' if we can derive a reachable path: $(v, S_1, k) \Longrightarrow (o, s, k')$. Here is a derivable local reachability relation found in Figure 4.2:

$$\frac{\dfrac{t1 \xleftarrow{\mathsf{ld(key)}} this_{put}}{(this_{put}, S_2, \varnothing) \Longrightarrow (t1, S_2, -key)} \quad \dfrac{t1 \xleftarrow{\mathsf{st(arr)}} k}{(t1, S_2, -key) \Longrightarrow (k, S_1, -key{:}-\overline{arr})}}{(this_{put}, S_2, \varnothing) \Longrightarrow (k, S_1, -key{:}-\overline{arr})}$$

The deduction rules of summarisation analysis, local reachability, summary update, points-to query and global reachability are given in Figure A.4-A.8 respectively.

Allocation sites	o		
Local variables	v		
Global variables	g		
Instance fields	f		
Call sites	i		
Nodes	n	::=	$o \mid v \mid g$
Labels	l	::=	$\mathsf{new} \mid \mathsf{assign} \mid \mathsf{ld}(f) \mid \mathsf{st}(f)$
			$\mid \mathsf{entry}_i \mid \mathsf{exit}_i \mid \mathsf{assignglobal}$
Statements	$stmt$	::=	$n \xleftarrow{l} n$
Programs	pro	::=	$\varnothing \mid pro \cup \{stmt\}$

Figure A.1: Syntax of an abstraction of Java language.

Directions	s	::=	$S_1 \mid S_2$
Call stacks	c	::=	$\varnothing \mid c{:}i$
Field stacks	k	::=	$\varnothing \mid d \mid -d \mid k{:}k$
Directed fields	d	::=	$f \mid \overline{f}$
Caches	Γ	::=	$\varnothing \mid \Gamma \cup \{(\nu, s) \mapsto \sigma\}$
Summaries	σ	::=	$\varnothing \mid \sigma \cup \{(n, s, k)\}$
States	ζ	::=	(n, s, k, c)

Figure A.2: Abstract domains.

$$\text{(transitivity)}\quad \frac{(n, c) \longrightarrow (n'', c'') \qquad (n'', c'') \longrightarrow (n', c')}{(n, c) \longrightarrow (n', c')}$$

$$\text{(new)}\quad \frac{n \xleftarrow{\mathsf{new}} n'}{(n, c) \longrightarrow (n', c)}$$

$$\text{(assign)}\quad \frac{n \xleftarrow{\mathsf{assign}} n'}{(n, c) \longrightarrow (n', c)}$$

$$\text{(assignglobal)}\quad \frac{n \xleftarrow{\mathsf{assignglobal}} n'}{(n, c) \longrightarrow (n', \varnothing)}$$

$$\text{(refinement)}\quad \frac{n \xleftarrow{\mathsf{ld(f)}} n'' \qquad n \xleftarrow{\mathsf{match}} n' \qquad n''' \xleftarrow{\mathsf{st(f)}} n'}{(n, c) \longrightarrow (n', \varnothing)}$$

$$\text{(field)}\quad \frac{n \xleftarrow{\mathsf{ld(f)}} n'' \qquad (n'', c) \approx (n''', c') \qquad n''' \xleftarrow{\mathsf{st(f)}} n'}{(n, c) \longrightarrow (n', c')}$$

$$\text{(entry)}\quad \frac{n \xleftarrow{\mathsf{entry(i)}} n'}{(n, c{:}i) \longrightarrow (n', c)}$$

$$\text{(entry-}\varnothing\text{)}\quad \frac{n \xleftarrow{\mathsf{entry(i)}} n'}{(n, \varnothing) \longrightarrow (n', \varnothing)}$$

$$\text{(exit)}\quad \frac{n \xleftarrow{\mathsf{exit(i)}} n'}{(n, c) \longrightarrow (n', c{:}i)}$$

$$\text{(alias)}\quad \frac{(n, c) \longrightarrow (o, c'') \qquad (n', c') \longrightarrow (o, c'')}{(n, c) \approx (n', c')}$$

Figure A.3: Deduction rules for RefinePTS analysis

$$\frac{\begin{array}{c}\sigma = \{(o, S_1, k) \mid (\nu, s, \varnothing) \Longrightarrow (o, S_1, k)\} \cup \\ \{(n, S_1, k) \mid (\nu, s, \varnothing) \Longrightarrow (n, S_1, k) \ \wedge \ n \xleftarrow{\text{exit/entry/assignglobal}} n'\} \cup \\ \{(n', S_2, k) \mid (\nu, s, \varnothing) \Longrightarrow (n, S_2, k) \ \wedge \ n' \xleftarrow{\text{exit/entry/assignglobal}} n\}\end{array}}{\textsc{Sum}(\nu, s) = \sigma}$$

Figure A.4: Summarisation by local reachability analysis.

$$\frac{n \xleftarrow{\text{new}} o}{(n, S_1, k) \Longrightarrow (o, S_1, k)}$$

$$\frac{n \xleftarrow{\text{new}} o}{(n, S_1, k) \Longrightarrow (n, S_2, k)}$$

$$\frac{n \xleftarrow{\text{assign}} n'}{(n, S_1, k) \Longrightarrow (n', S_1, k)}$$

$$\frac{n \xleftarrow{\text{ld}(f)} n'}{(n, S_1, k) \Longrightarrow (n', S_1, k{:}\bar{f})}$$

$$\frac{(n, s, k) \Longrightarrow (n'', s'', k'') \qquad (n'', s'', k'') \Longrightarrow (n', s', k')}{(n, s, k) \Longrightarrow (n', s', k')}$$

$$\frac{n' \xleftarrow{\text{assign}} n}{(n, S_2, k) \Longrightarrow (n', S_2, k)}$$

$$\frac{n' \xleftarrow{\text{st}(f)} n}{(n, S_2, k) \Longrightarrow (n', S_1, k{:}f)}$$

$$\frac{n' \xleftarrow{\text{ld}(f)} n}{(n, S_2, k{:}f) \Longrightarrow (n', S_2, k)}$$

$$\frac{n' \xleftarrow{\text{ld}(f)} n \qquad k \neq k'{:}d}{(n, S_2, k) \Longrightarrow (n', S_2, k{:}{-}f)}$$

$$\frac{n \xleftarrow{\text{st}(f)} n'}{(n, S_2, k{:}\bar{f}) \Longrightarrow (n', S_1, k)}$$

$$\frac{n \xleftarrow{\text{st}(f)} n' \qquad k \neq k'{:}d}{(n, S_2, k) \Longrightarrow (n', S_1, k{:}{-}\bar{f})}$$

Figure A.5: Local reachability analysis.

$$\mathtt{stmt} = n \xleftarrow{\mathsf{new/assign/ld/st}} n'$$

$$\text{(local-edge)}\ \frac{\textsc{Update}(\Gamma, n, S_2) = \Gamma' \qquad \textsc{Update}(\Gamma', n', S_1) = \Gamma''}{\textsc{Update}(\Gamma, \mathtt{stmt}) = \Gamma''}$$

$$\text{(entry)}\ \frac{\mathtt{stmt} = n \xleftarrow{\mathsf{entry}} v \qquad \textsc{Update}(\Gamma, v, S_1) = \Gamma'}{\textsc{Update}(\Gamma, \mathtt{stmt}) = \Gamma'}$$

$$\text{(exit)}\ \frac{\mathtt{stmt} = v \xleftarrow{\mathsf{exit}} n \qquad \textsc{Update}(\Gamma, v, S_2) = \Gamma'}{\textsc{Update}(\Gamma, \mathtt{stmt}) = \Gamma'}$$

$$\text{(global-from)}\ \frac{\mathtt{stmt} = g \xleftarrow{\mathsf{assignglobal}} v \qquad \textsc{Update}(\Gamma, v, S_1) = \Gamma'}{\textsc{Update}(\Gamma, \mathtt{stmt}) = \Gamma'}$$

$$\text{(global-to)}\ \frac{\mathtt{stmt} = v \xleftarrow{\mathsf{assignglobal}} g \qquad \textsc{Update}(\Gamma, v, S_2) = \Gamma'}{\textsc{Update}(\Gamma, \mathtt{stmt}) = \Gamma'}$$

$$\text{(update)}\ \frac{\textsc{Sum}(n, s) = (n_1, s_1, k_1), .., (n_j, s_j, k_j) \qquad \Gamma' = \Gamma[(n_1, \overline{s_1}) \mapsto \textsc{Sum}(n_1, \overline{s_1})]..[(n_j, \overline{s_j}) \mapsto \textsc{Sum}(n_j, \overline{s_j})]}{\textsc{Update}(\Gamma, n, s) = \Gamma'}$$

Figure A.6: Summary update for code changes.

$$\text{(local)}\ \frac{(n, s) \notin \mathtt{dom}(\Gamma) \qquad (n, s, k) \Longrightarrow (n', s', k')}{\Gamma, (n, s, k, c) \longrightarrow \Gamma, (n', s', k', c)}$$

$$\text{(global)}\ \frac{(n, s, c) \longrightarrow (n', s', c')}{\Gamma, (n, s, k, c) \longrightarrow \Gamma, (n', s', k, c')}$$

$$\text{(summary)}\ \frac{(n', s', k') \in \Gamma((n, s)) \qquad -\mathtt{d} \notin k{:}k'}{\Gamma, (n, s, k, c) \longrightarrow \Gamma, (n', s', k{:}k', c)}$$

$$\text{(transitivity)}\ \frac{\Gamma, \zeta \longrightarrow \Gamma, \zeta'' \qquad \Gamma, \zeta'' \longrightarrow \Gamma, \zeta'}{\Gamma, \zeta \longrightarrow \Gamma, \zeta'}$$

Figure A.7: Summary-based points-to analysis.

$$\frac{n \xleftarrow{\text{assignglobal}} n'}{(n, S_1, c) \longrightarrow (n', S_1, \varnothing)} \qquad \frac{n' \xleftarrow{\text{assignglobal}} n}{(n, S_2, c) \longrightarrow (n', S_2, \varnothing)}$$

$$\frac{n \xleftarrow{\text{exit}_i} n'}{(n, S_1, c) \longrightarrow (n', S_1, c{:}i)} \qquad \frac{n' \xleftarrow{\text{exit}_i} n}{(n, S_2, c{:}i) \longrightarrow (n', S_2, c)}$$

$$\frac{n \xleftarrow{\text{entry}_i} n'}{(n, S_1, c{:}i) \longrightarrow (n', S_1, c)} \qquad \frac{n' \xleftarrow{\text{entry}_i} n}{(n, S_2, \varnothing) \longrightarrow (n', S_2, \varnothing)}$$

$$\frac{n \xleftarrow{\text{entry}_i} n'}{(n, S_1, \varnothing) \longrightarrow (n', S_1, \varnothing)} \qquad \frac{n' \xleftarrow{\text{entry}_i} n}{(n, S_2, c) \longrightarrow (n', S_2, c{:}i)}$$

Figure A.8: Global reachability analysis.

Bibliography

[1] Mithun Acharya and Brian Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 746–755, New York, NY, USA, 2011. ACM.

[2] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[3] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 203–213, New York, NY, USA, 2001. ACM.

[4] Sam Blackshear, Bor-Yuh Evan Chang, Sriram Sankaranarayanan, and Manu Sridharan. The flow-insensitive precision of andersen's analysis in practice. In *Proceedings of the 18th international Symposium on Static analysis*, SAS '11, pages 60–76, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 241–250, New York, NY, USA, 2011. ACM.

[6] Michele Ceccarelli, Luigi Cerulo, Gerardo Canfora, and Massimiliano Di Penta. An eclectic approach for change impact analysis. In *Proceedings of the 32nd International Conference on Software Engineering*, ICSE '10, pages 163–166, New York, NY, USA, 2010.

[7] Venkatesan T. Chakaravarthy. New results on the computability and complexity of points–to analysis. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 115–125, New York, NY, USA, 2003. ACM.

[8] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 39–50, New York, NY, USA, 2008. ACM.

[9] Swarat Chaudhuri. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 159–169, New York, NY, USA, 2008. ACM.

[10] C.L. Conway, K.S. Namjoshi, Dennis Dams, and S.A. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In *Proceedings of the*

17th international conference on Computer Aided Verification, CAV '05, 2005.

[11] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 35–46, New York, NY, USA, 2000. ACM.

[12] Manuvir Das, Ben Liblit, Manuel Fahndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. In Patrick Cousot, editor, *Static Analysis*, volume 2126 of *Lecture Notes in Computer Science*, pages 260–278. 2001.

[13] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 270–280, New York, NY, USA, 2008. ACM.

[14] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 85–96, New York, NY, USA, 1998. ACM.

[15] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *Proceedings of the 2006 international symposium on Software testing and analysis*, ISSTA '06, pages 133–144, New York, NY, USA, 2006. ACM.

[16] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[17] Robert Goeritzer. Using impact analysis in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1155–1157, New York, NY, USA, 2011.

[18] Samuel Z. Guyer and Calvin Lin. Error checking with client-driven pointer analysis. *Science of Computer Programming*, 58:83 – 114, 2005.

[19] Brian Hackett and Alex Aiken. How is aliasing used in systems software? In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 69–80, New York, NY, USA, 2006. ACM.

[20] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 290–299, New York, NY, USA, 2007. ACM.

[21] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 226–238, New York, NY, USA, 2009. ACM.

[22] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th International Symposium on Code Generation and Optimization*, CGO '11, pages 289–298, 2011.

[23] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 24–34, New York, NY, USA, 2001. ACM.

[24] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[25] Martin Hirzel, Daniel Von Dincklage, Amer Diwan, and Michael Hind. Fast online pointer analysis. *ACM Trans. Program. Lang. Syst.*, 29(2), April 2007.

[26] Vineet Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 249–259, New York, NY, USA, 2008. ACM.

[27] John Kodumal and Alex Aiken. The set constraint/CFL reachability connection in practice. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 207–218, New York, NY, USA, 2004. ACM.

[28] John Kodumal and Alex Aiken. Banshee: A scalable constraint-based analysis toolkit. In *Proceedings of the 12nd international Symposium on static analysis*, SAS '05, pages 218–234, 2005.

[29] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 235–248, New York, NY, USA, 1992. ACM.

[30] Steffen Lehnert. A taxonomy for software change impact analysis. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, pages 41–50, 2011.

[31] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using SPARK. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 153–169, Berlin, Heidelberg, 2003. Springer-Verlag.

[32] Ondřej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: is it worth it? In *CC '06*, pages 47–64, 2006.

[33] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Evaluating the precision of static reference analysis using profiling. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 22–32, New York, NY, USA, 2002. ACM.

[34] David Melski and Thomas Reps. Interconvertbility of set constraints and context-free language reachability. In *Proceedings of the 1997 ACM SIGPLAN symposium*

on Partial evaluation and semantics-based program manipulation, PEPM '97, pages 74–89, New York, NY, USA, 1997. ACM.

[35] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 327–338, New York, NY, USA, 2007. ACM.

[36] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM.

[37] Phung Hua Nguyen and Jingling Xue. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. In *ACSC '05*, pages 9–18, 2005.

[38] Phung Hua Nguyen and Jingling Xue. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. In *Proceedings of the Twenty-eighth Australasian conference on Computer Science - Volume 38*, ACSC '05, pages 9–18, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[39] Phung Hua Nguyen and Jingling Xue. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. In *Proceedings of the Twenty-eighth Australasian conference on Computer Science - Volume 38*, ACSC

'05, pages 9–18, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[40] Erik M. Nystrom, Hong seok Kim, and Wen mei W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proceedings of the 11th international symposium on static analysis*, SAS '04, pages 165–180, 2004.

[41] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis for C. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '04, pages 37–42, New York, NY, USA, 2004. ACM.

[42] G. Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 502–510, New York, NY, USA, 1993. ACM.

[43] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of Java programs. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, pages 432–448, New York, NY, USA, 2004.

[44] Thomas Reps. Program analysis via graph reachability. In *In Proceedings of 1997 International Logic Programming Symposium*, ILPS '97, pages 5–19, 1997.

[45] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-*

SIGACT symposium on Principles of programming languages, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[46] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '94, pages 11–20, New York, NY, USA, 1994. ACM.

[47] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 182–195, New York, NY, USA, 2000. ACM.

[48] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 105–118, New York, NY, USA, 1999. ACM.

[49] Diptikalyan Saha and C.R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '05, 2005.

[50] Lei Shang, Xinwei Xie, and Jingling Xue. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 264–274, 2012.

[51] Lei Shang, Lu Yi, and Jingling Xue. Fast and precise points-to analysis with incremental cfl-reachability summarisation. In *Proceedings of the 2012 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE '12, 2012.

[52] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 16–16, Berkeley, CA, USA, 2001. USENIX Association.

[53] Vugranam C. Sreedhar, Michael Burke, and Jong-Deok Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 196–207, New York, NY, USA, 2000. ACM.

[54] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 387–400, New York, NY, USA, 2006. ACM.

[55] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for Java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 59–76, New York, NY, USA, 2005. ACM.

[56] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[57] Yulei Sui, Yue Li, and Jingling Xue. Query-directed adaptive heap cloning for optimizing compilers. In *Proceedings of the 11th International Symposium on Code Generation and Optimization*, CGO '13, 2013.

[58] Yulei Sui, Ding Ye, and Jingling Xue. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA '12, pages 254–264, New York, NY, USA, 2012. ACM.

[59] Yulei Sui, Sen Ye, Jingling Xue, and Pen-Chung Yew. SPAS: scalable path-sensitive pointer analysis on full-sparse ssa. In *Proceedings of the 9th Asian conference on Programming Languages and Systems*, APLAS'11, pages 155–171, Berlin, Heidelberg, 2011. Springer-Verlag.

[60] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: a java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, CASCON '10, pages 214–224, Riverton, NJ, USA, 2010. IBM Corp.

[61] John Whaley and Monica 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*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM.

[62] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming language design and implementation*, PLDI '95, pages 1–12, 1995.

[63] Xinwei Xie and Jingling Xue. Acculock: Accurate and efficient detection of data races. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 201–212, Washington, DC, USA, 2011. IEEE Computer Society.

[64] Xinwei Xie, Jingling Xue, and Jie Zhang. Acculock: accurate and efficient detection of data races. *Software: Practice and Experience*, pages n/a–n/a, 2012.

[65] Guoqing Xu and Atanas Rountev. Detecting inefficiently-used containers to avoid bloat. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 160–173, 2010.

[66] Guoqing Xu, Atanas Rountev, and Manu Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings of the 23rd European conference on Object-oriented programming*, ECOOP '09, pages 98–122, 2009.

[67] Guoqing Xu, Dacong Yan, and Atanas Rountev. Static detection of loop-invariant data structures. In *Proceedings of the 26th European conference on Object-oriented programming*, ECOOP '12, pages 738–763, 2012.

[68] Jingling Xue and Phung Hua Nguyen. Completeness analysis for incomplete object-oriented programs. In *Proceedings of the 14th international conference*

on Compiler Construction, CC '05, pages 271–286, Berlin, Heidelberg, 2005. Springer-Verlag.

[69] Jingling Xue, Phung Hua Nguyen, and John Potter. Interprocedural side-effect analysis for incomplete object-oriented software modules. *J. Syst. Softw.*, 80(1):92–105, January 2007.

[70] Dacong Yan, Guoqing Xu, and Atanas Rountev. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 155–165, 2011.

[71] Mihalis Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '90, pages 230–242, New York, NY, USA, 1990. ACM.

[72] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th International Symposium on Code Generation and Optimization*, CGO '10, pages 218–229, 2010.

[73] Jyh-Shiarn Yur, Barbara G. Ryder, and William Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, 1999.

[74] Xin Zheng and Radu Rugina. Demand-driven alias analysis for C. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 197–208, New York, NY, USA, 2008. ACM.