Refinement-Based Program Analysis Tools

Manu Sridharan

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-125
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-125.html

October 23, 2007

Refinement-Based Program Analysis Tools

by

Manu Sridharan

B.S. (Massachusetts Institute of Technology) 2001
M.Eng. (Massachusetts Institute of Technology) 2002

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Rastislav Bodík, Chair
Professor George Necula
Professor Leo Harrington

Fall 2007

The dissertation of Manu Sridharan is approved:

Professor Rastislav Bodík, Chair Date

Professor George Necula Date

Professor Leo Harrington Date

University of California, Berkeley

Fall 2007

Refinement-Based Program Analysis Tools

by

Manu Sridharan

Abstract

Refinement-Based Program Analysis Tools

by

Manu Sridharan

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Rastislav Bodík, Chair

Program analysis tools are starting to change how software is developed. Verifiers can now eliminate certain complex bugs in large code bases, and automatic refactoring tools can greatly simplify code cleanup. Nevertheless, writing robust large-scale software remains a challenge, as greater use of component frameworks complicates debugging and program understanding. Developers need more powerful programming tools to combat this complexity and produce reliable code.

This dissertation presents two techniques for more powerful debugging and program understanding tools based on *refinement*. In general, refinement-based techniques aim to discover interesting properties of a large program by only reasoning about the most important parts of the program (typically a small amount of code) precisely, abstracting away the behavior of much of the program. Our key contribution is the first framework for effective refinement-based handling of object-oriented data structures; pervasive use of such data structures thwarts the effectiveness of most existing analyses and tools.

Our two refinement-based techniques significantly advance the state-of-the-art in program analyses and tools for object-oriented languages. The first technique is a *refinement-based points-to analysis* that can compute precise answers in interactive

time. This analysis enables precise reasoning about data structure behavior in clients that require interactive performance, *e.g.*, just-in-time compilers and interactive development environment (IDE) tools. Our second technique is *thin slicing*, which gives usable answers to code relevance questions—*e.g.*, "What code might have caused this crash?"—addressing a long-standing challenge for analysis tools. In this dissertation, we describe the design and implementation of these techniques and present experimental evaluations that validate their key properties.

Professor Rastislav Bodík, Chair Date

Contents

List of Figures

List of Tables

Acknowledgements

I owe deep thanks to my advisor Ras Bodík, who has been an outstanding mentor and collaborator. I cannot hope to list all the ways that Ras has helped me improve as a researcher, but perhaps the most important thing he taught me was how to recognize and communicate the broader importance of my research to others. His tireless feedback on ideas, paper drafts, and talks vastly increased the quality of my work and its impact. I am certain that his influence and example will continue to inspire me in my future pursuits.

Much of the work in this thesis benefited enormously from other collaborators. Denis Gopan did some initial work on points-to analysis via CFL-reachability and continued to give essential feedback after I took over the project. Lexin Shan helped greatly with the experimental evaluation of the context-insensitive points-to analysis. Mooly Sagiv contributed significantly to the formalization of the context-sensitive points-to analysis and participated in many other fun brainstorming sessions. Finally, Stephen Fink helped come up with the idea of thin slicing and implemented both the traditional and thin slicers for our experimental evaluation, a heroic effort in the weeks before the PLDI deadline that made the paper possible.

I have received excellent advice from other mentors that helped me greatly at many stages of graduate school. Daniel Jackson has continued to provide valuable guidance in spite of my traitorous decision to leave MIT for Berkeley, for which I'm very grateful. Zhe Yang was my mentor during a summer internship at Microsoft Research, an exciting experience that helped inspire ideas in my other work. Manuvir Das provided much good advice to me, both during that internship and afterward. George Necula and Leo Harrington graciously agreed to be on my thesis committee and provided valuable feedback during my qualifying exam. Susan Graham also gave

useful feedback during my qualifying exam, including the suggestion of the name 'thin slicing'. I'd also like to thank Robert O'Callahan and Jim Larus for their occasional advice and support.

Graduate school would not have been nearly as enjoyable without my fantastic officemates and friends. Brian Fields, Dave Mandelin, AJ Shankar, and Bill McCloskey made 517 Soda a really fun place to be. Though having such interesting people in the office didn't always improve my research productivity, I definitely learned a lot and enjoyed school a lot more because of them. David Ratajczak was a brilliant research collaborator and a good friend during his time at Berkeley. Naveen Sastry helped me stay in shape and gave lots of useful advice, research-related and otherwise. Nick Eriksson helped smooth my transition to Berkeley and cooked up many a tasty dessert in our apartment. Anand Sarwate was a fellow Indian tenor EECS graduate student from MIT who always had insightful things to say about research, music, or just about any other topic.

My parents and my brother Vishnu have always provided me with tireless love and support. From letting me play with our VIC-20 when I was six to tolerating my hours tying up the phone line with the modem in high school, my family constantly nurtured my technical interests. They have always encouraged me to work hard and persevere, and they have been incredibly supportive anytime I needed help. This dissertation would not have been possible without them.

Finally, I thank my wonderful wife Padma, who has given me tremendous support and encouragement. She both kept me sane during the weeks of craziness before paper deadlines and kept me motivated when I was feeling unproductive and frustrated. I am very lucky to be leaving Berkeley with such an amazing life companion.

Chapter 1

Introduction

Nearly 40 years since the popularization of the term "software engineering" [NR69], building robust, large-scale software remains an enormous engineering challenge. Stories abound of failed software projects with disastrous consequences, from the well-known Therac-25 incident [LT93] to more recent troubles at the IRS [McC06].[1] A recent NIST study estimates the costs of failed software projects to be in the billions of dollars per year [oST]. Hence, techniques for reducing the cost and difficulty of building software could have enormous economic and societal impact.

Software components, long heralded as a possible solution to software complexity, have significantly eased the process of developing large-scale software. As documented by Lampson [Lam04], large "platform" components like operating systems, databases, and web browsers have greatly reduced the effort required to develop many applications. At a smaller scale, platforms based on the Java language like J2SE [J2S], J2EE [J2E], and Eclipse [Ecl] have proved to be enormously popular with developers. As of July 2007, 904 Eclipse plugins are available at one popular repository [EcP],

[1] In the IRS case, an attempt to build and deploy new fraud detection software failed. By the time it was known that the new system would not work, it was too late to re-start the old system, leading to an estimated $200 million in fraudulent refunds.

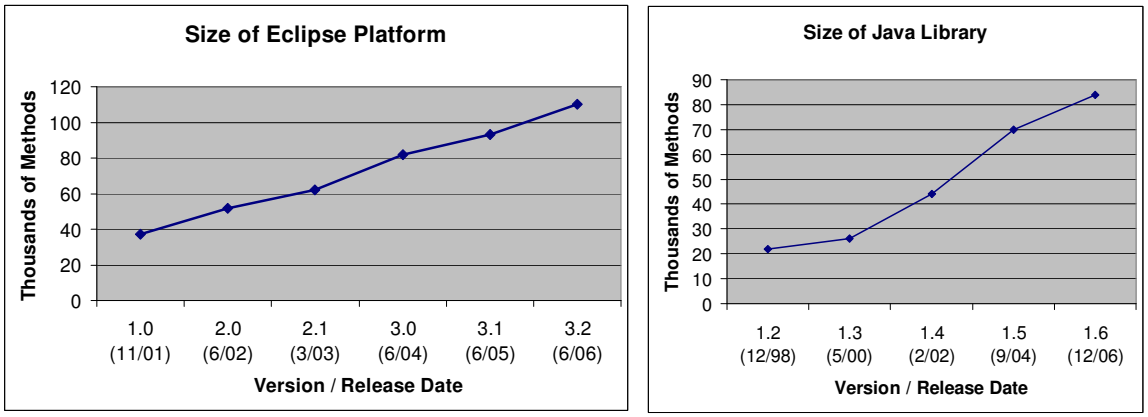


Figure 1.1: Graphs showing the number of public methods for the Eclipse platform [Ecl] and the Java standard library [J2S] in successive releases. The size of Eclipse has increased by a factor of 3 since its initial release, while the Java library size has increased by a factor of 3.8.

reflecting the platform's popularity and extensibility.

Though they have eased software development to some degree, components have *not* solved software complexity, and in some ways they have exacerbated the problem. Component platforms are often a key source of software complexity due to their enormous size, ballooning with each release as new features are added. For example, Figure 1.1 shows the dramatic size increase of J2SE [J2S] and Eclipse [Ecl] over their last few releases. Increasing platform size often translates to greater complexity, as discovering how to use the few platform features needed for a particular task can be non-trivial. Good documentation can partially alleviate this problem, but in practice, documentation is often incomplete and, when it exists, difficult to locate. The complexity of platforms can turn even the simplest tasks into a challenge: work on the Prospector system [MXBK05] describes how it took hours for experienced programmers to learn the *three lines of code* needed to re-use Eclipse's Java parser.

How can the challenge of programming to large platforms be eased? We believe that *better development tools* are the most promising approach for attacking this

problem, as they can provide significant benefits for both existing and future software projects. Recent tools like ESP [DLS02] and xgcc [HCXE02] are able to find many serious bugs in large code bases, with no effort beyond running the tool required from the user. Similarly, automatic refactoring tools [Fow99] can be used in an "off-the-shelf" fashion to improve maintainability of existing code. Due to their usefulness and ease of adoption, such tools are already having a significant impact on large-scale software projects, *e.g.*, Microsoft Windows [HDWY06]. While some problems addressed by tools could also be solved by other techniques (*e.g.*, new programming language constructs), the integration of tools into an existing development process is often much easier, and hence preferable.

Unfortunately, many current development tools are too resource intensive to provide immediate feedback to developers as they write and debug their code. Many questions that arise during development can be answered by existing static analyses, *e.g.*, "What methods can be invoked by this virtual call?" or "What code writes into this object field?" However, applying these analyses to large component-based software can require up to hours of time and gigabytes of memory. Hence, today these questions are usually answered by hand—an error-prone process that can require digging through many layers of component abstractions. Also, the output of bug-finding tools would be even more useful if it was provided immediately after a developer introduces a bug, since at that point her understanding of the code is still fresh.

Just as large-scale software can overwhelm current tools, the voluminous output of those tools can sometimes overwhelm developers, greatly lessening the tools' usefulness. When presenting some program behavior to the user, a standard approach in tools is to show all code that is possibly relevant to that behavior. This approach is highly problematic with large, component-based software for two reasons:

1. The amount of possibly relevant code can be enormous, often a significant fraction of the whole program.

2. Code from libraries is often relevant, but developers are typically insulated from details of library implementations.

As understandable results are key to the usefulness of development tools, existing tools sometimes suppress reports of difficult-to-explain behavior. For example, bugs involving aliased pointers are not reported by ESP since the tool has no mechanism for properly explaining the aliasing [MSA+04]. This choice is unfortunate for tool users, as such suppressed results often correspond to issues that are also very difficult to understand without tool support, *e.g.*, aliasing bugs.

The problem statement of this thesis is to develop techniques that address the scalability and understandability drawbacks of current development tools. In this work, we focus specifically on treatment of pointers and heap-allocated data structures, since understanding heap behavior is critical to understanding most other behaviors of large object-oriented software (further discussion in §1.2.1 and §1.3.1). First, we aim to develop a pointer analysis that can both match or exceed the precision of existing analyses while providing results *in interactive time*, enabling immediate tool feedback for developers. Second, we pursue a technique for explaining heap behaviors to users in a more concise and useful manner than simply showing all possibly relevant program statements. Together, these techniques could enable a new class of development tools that significantly ease the debugging and understanding of component-based software.

1.1 Our Approach: Refinement

This dissertation presents techniques based on *refinement* that address drawbacks of current development tools. A refinement-based technique assumes that in practice, precise reasoning about a small amount of *important code* is sufficient for understanding many interesting program properties. Refinement is the iterative process by which an analysis tries to discover this important code. Our use of refinement enables our techniques to overcome the aforementioned drawbacks of current tools: performance is dramatically improved since only important code is analyzed precisely, and understandability is improved since programmer attention is focused on important code.

A key open problem addressed by our work is how to use refinement effectively for reasoning about heap behavior in large object-oriented programs. Refinement has previously been employed with great success in other program analysis domains (*e.g.*, model checking [BR01]). However, previous refinement techniques for heap behavior [PC94, GL03] proved ineffective at identifying a small amount of important code in large Java programs, and hence were insufficient for our needs.

Our refinement framework rests on our identification of *hierarchical structure* in heap-based value flow for Java-like languages. Questions about value flow concern how some object o flows to some variable x,[2] *i.e.*, what statements cause the flow to occur. In Java, an object flows through a heap memory location through *field writes and reads*, respectively of the form `w.f = z` and `x = y.f`. When an object is written and read from the heap through such field accesses, a secondary question may arise: how do the field accesses come to operate on the same object, *i.e.*, what causes base pointers `w` and `y` to be pointer aliases? Explaining this condition requires recursively tracing the flow of some object o' to both `w` and `y`. The object o' may itself flow

[2]When discussing the flow of an object to a variable, we mean the flow of a pointer to the object, consistent with Java semantics.

```
1 ...; z = p.g;
2 x = q.g;
3 w = new Obj();
4 z.f = w;
5 y = x.f;
```

(a) Initial example.

```
1 w = new Obj();
2 o1.f = w;
3 y = o1.f;
```

(b) No important field accesses.

```
1 ...; z = o2.g;
2 x = o2.g;
3 w = new Obj();
4 z.f = w;
5 y = x.f;
```

(c) Treating the `f` accesses as important.

Figure 1.2: Code examples to informally illustrate the effects of our refinement technique.

through the heap, yielding the aforementioned hierarchy in heap-based flow.

We exploit hierarchical heap accesses in our refinement technique by only investigating aliasing of base pointers for a small number of field accesses. In practice, many interesting program properties can be discovered without considering the statements that cause base pointers of field accesses to be aliased, instead simply assuming that such statements exist. This treatment of field accesses can dramatically decrease the amount of code considered by the enclosing analysis or the user, increasing both scalability and understandability. For some field accesses, considering the statements causing base pointer aliasing is important, *e.g.*, because further inspection may prove that no aliasing is possible. Our refinement technique succeeds by enabling fast identification of these important field accesses.

Figure 1.2 informally illustrates the effects of our refinement technique with a small example. Say that for Figure 1.2(a), one is interested in how objects flow to `y`. Ignoring aliasing explanations for all field accesses essentially abstracts the program to that of Figure 1.2(b). The base pointers `x` and `z` from Figure 1.2(a) have been replaced with a new variable `o1`, reflecting the assumption that some object flows to both base pointers.[3] Note that attention is now focused on the value flow from `w` to

[3]Note that this use of new scalar variables is intended only to provide the intuition behind our refinement technique; in general, it may not be possible to model our treatment of the heap with

`y`, as code related to base pointers `x` and `z` can be elided.

Figure 1.2(c) illustrates what happens when refinement exposes the statements causing base pointer aliasing for the accesses of field `f` (line 4 and line 5). Here, the base pointers `x` and `z` from Figure 1.2(a) return, but the base pointers for the `g` accesses at line 1 and line 2 have been abstracted. Our refinement technique heuristically chooses where to inspect statements causing base pointer aliasing, aiming to focus effort on those statements most relevant to the property of interest for the user.

We have instantiated our refinement framework in two different ways. The first instantiation is a refinement-based points-to analysis, the first to compute precise answers in "interactive time." This analysis dramatically improves on the scalability of previous approaches, enabling precise reasoning about heap behavior in interactive development tools. §1.2 discusses the importance of points-to analysis and further details how our analysis advances the state-of-the-art.

Thin slicing, our second technique, uses refinement to give usable answers to code relevance questions, *e.g.*, "What code might have caused this crash?" Providing such answers has been a long-standing challenge for analysis tools. To improve on program slicing [Wei79], the state-of-the-art technique, we devised a definition of code relevance whose key novelty is its exclusion of statements only relevant to base pointer aliasing. In practice, this notion of relevance captures nearly exactly the desired statements for many programmer tasks. In cases when more statements are needed, the user can refine by selectively exposing statements relevant to base pointer aliasing—again exploiting the hierarchy of Java heap accesses. §1.3 describes why classical program slicing is often not useful in practice and further details the new notion of relevance used by thin slicing.

such a translation.

1.2 Refinement-Based Points-To Analysis

Here we give an overview of our refinement-based points-to analysis technique. Via refinement, our analysis is able to match or exceed the precision of state-of-the-art analyses while requiring orders-of-magnitude less memory and providing "interactive" running times. We first detail the increasing need for scalable and precise points-to analysis in §1.2.1. Then, we give a brief introduction to points-to analyses of the last 15 years to place our work in context (§1.2.2). Finally, we describe in more detail the key contributions of our points-to analysis (§1.2.3).

1.2.1 Motivation

Effective analysis of pointer variables in programs is becoming increasingly critical, as more software is being written in object-oriented programming languages that encourage pervasive use of heap-allocated data. For the widely-adopted Java and C# languages, data must be stored on the heap to make effective use of many language features and library classes. Increasingly popular scripting languages like Python, Javascript, and Ruby have essentially pure object models, in which all values are objects, making programs even more heap intensive. Finally, use of platforms like Eclipse [Ecl] often requires greater use of the heap; for example, data in the often-used XML format are typically represented with a tree of objects.

To aid reasoning about pointer behavior in large object-oriented programs, we have developed a scalable and precise *points-to analysis* (first defined in [EGH94]). A points-to analysis typically computes a *points-to relation* that conservatively maps each pointer variable to the heap objects it can point to at runtime. This points-to relation is often represented with a *points-to set* for each variable, with each points-to set containing *abstract locations* from some finite model of all possible runtime heaps (further discussion in §2.1).

The results of a points-to analysis have many uses in program analysis. One common use of points-to sets is in reasoning about pointer aliasing, which can greatly complicate reasoning about a program's behavior (breaking Hoare's elegant axiomatic treatment of assignments [Hoa69]). A precise points-to analysis can alleviate the problem of reasoning about pointer aliases by showing that most pointer variable pairs cannot be aliased, *i.e.*, they can never point to the same object. The points-to set of a single variable is also useful to other analyses, for example to refine the variable's declared type. If variable x of declared type `Object` can only point to objects of type `A`, other analyses can use the more precise type `A` for x when performing optimization or verification.

An increasing number of state-of-the-art Java development tools are limited by the need for a *context-sensitive* points-to analysis. Intuitively, a context-sensitive analysis treats a program as if all method calls were inlined, thereby gaining precision by computing results separately for each invocation of a method (see § 2.2.4 for a more detailed definition). Studies have shown the importance of context sensitivity in precisely analyzing Java pointer behavior [LPH02, LH06]. Also, context-sensitive points-to analysis is critical to the usefulness of recent tools like the static race detector of of Naik *et al.* [NAW06] and the type-state verifier of Fink *et al.* [FYD+06].[4] Unfortunately, existing context-sensitive points-to analyses do not scale well: although the aforementioned tools are effective on medium-sized programs, points-to analysis remains a scalability bottleneck for handling even larger programs.

Beyond context sensitivity, important modern clients of program analysis also require *interactive performance*, *i.e.*, time and space overheads suitable for an interactive application. Just-in-time (JIT) compilers require interactive performance

[4] Regarding context-sensitivity in the points-to analysis, [FYD+06] states: "We ran many of the analyses with a context-*insensitive* Andersen-style pointer analysis [...] Many of the benchmarks timed out on several rules; we conclude that adequate precision in the preceding pointer analysis is vital."

Language \ CI/CS	Context Insensitive	Context Sensitive
C	subset based [And94], equality based [Ste96], mix of eq and subset [Das00], optimized subsets [FFSA98, SFA00, HT01b]	partial summaries [WL95], equality based [OJ97, LLA07], BDD based [ZC04]
Java	adapted from C [RMR01, WL02, LH03], BDD based [BLQ+03]	equality based [O'C00], BDD based [WL04], object sensitive [MRR05]

Table 1.1: A summary of selected work in points-to analysis, categorized by what language was analyzed (C or Java) and whether the analysis was context sensitive. A key attribute of each analysis is briefly described.

from analyses, since analysis time and space add directly to the requirements of the executing application. Interactive performance is also required for tools running in interactive development environments (IDEs), since they run as the developer edits code. Note that running a points-to analysis once up-front and re-using its result is not sufficient for interactive performance since the input program can change over time, due to dynamic class loading for a JIT compiler or code editing in an IDE. Before our work, no points-to analysis could provide context-sensitive results for large programs in interactive time. As shown in Table 1.2, a state-of-the-art context-sensitive points-to analysis took 90 minutes and 2GB of memory to analyze a large Java program in our experiments, clearly not acceptable for IDE usage.

1.2.2 A Brief History of Points-to Analysis

Hind's 2001 discussion of pointer analysis [Hin01] begins:

> During the past twenty-one years, over seventy-five papers and nine Ph.D. theses have been published on pointer analysis. Given the tomes of work on this topic, one may wonder, "Haven't we solved this problem yet?"

Hind then continued to show that the points-to analysis problem had not been solved, and today the topic remains an active research area. Here we provide a very brief overview of points-to analyses for both C and Java developed over the last 15 years, with the aim of emphasizing two key points:

1. Context sensitive precision is more important for Java points-to analysis than for C; therefore, the large body of work on C points-to analysis does *not* immediately yield an effective Java analysis.

2. No previous analysis was able to provide scalable context-sensitive results for Java with the interactive performance needed for development tools.

Table 1.1 briefly summarizes the analyses discussed here. For a more detailed technical discussion of related points-to analyses, see §7.1.

Points-To Analysis for C Scalable, precise points-to analysis for C was a long-standing open problem in program analysis. Steensgaard's equality-based analysis [Ste96] ran in almost linear time, but lost precision due to its imprecise handling of the semantics of assignments. Andersen's subset-based analysis [And94] handled assignments precisely, but required a worst-case cubic time algorithm and initially did not scale to large programs. Andersen's analysis was first made scalable through optimizations devised by Aiken and collaborators [SFA00, FFSA98], and further scalability improvements were obtained by Heintze and Tardieu [HT01b]. Also, Das's one-level flow algorithm [Das00] found a "sweet spot" in the precision / scalability spectrum for C points-to analysis, obtaining nearly the precision of Andersen's analysis with a very scalable, worst-case quadratic algorithm.

While several context-sensitive points-to analyses for C have been developed, the precision gain from context sensitivity is typically not worth the added computational expense. In C, many functions with pointer parameters do not mutate pointer values, instead only mutating primitive values like integers reached through pointer

dereferences (this phenomenon often arises when passing values by reference). Since such functions do not change what pointers point to, a context-sensitive treatment of the functions would not yield much benefit for a points-to analysis. In the most comprehensive study of the topic [FFA00], Foster *et al.* show that context sensitivity adds significant precision to the results of an equality-based analysis, but has little effect on the precision of a subset-based analysis. Similarly, an evaluation by Das *et al.* [DLFR01] showed that context sensitivity adds little precision to the one-level flow algorithm [Das00].

Nevertheless, some of the work on context-sensitive points-to analysis for C was influential on later work for Java. O'Callahan and Jackson's Lackwit system [OJ97] performs a context-sensitive equality-based analysis based on polymorphic type inference. Wilson and Lam [WL95] developed a summary-based flow- and context-sensitive analysis which worked well for C programs with up to 20,000 lines of code; however, further research has been unable to make this algorithm scale to larger programs. Recently, Lattner and Adve have successfully used a context-sensitive points-to analysis [LLA07] for improving performance through automatic pool allocation of heap-allocated data structures [LA05].

Points-To Analysis for Java Early work on Java points-to analysis focused on adapting existing work like Andersen's analysis to handle Java's language constructs [RMR01, WL02, LH03], and today, these early analyses scale well to large Java programs. However, it was soon discovered that unlike C, context sensitivity could provide a large precision benefit for Java points-to analysis [O'C00, LPH02]. Java pointers are often used to manipulate data structures provided by the standard library, *e.g.*, `java.util.Vector`. Since the methods associated with these data structures (*e.g.*, `Vector.add()`) are called many times on many different objects, context sensitivity is needed separately reason about distinct instances of these data structures.

The key challenge of context-sensitive points-to analysis of large Java programs is handling the exponentially larger analysis result stemming from keeping the result for each call to a method separately (see § 2.2.4 for more discussion). Recent analyses that are relatively scalable and precise rely on two key recent insights. The first is that the computed points-to relation often has a lot of redundancy (*i.e.*, many variables have mostly identical points-to sets), and hence can be represented compactly by binary decision diagrams (BDDs), as shown by Berndl *et al.* [BLQ+03] and Zhu and Calman [ZC04]. Whaley and Lam used BDDs for context-sensitive points-to analysis [WL04], yielding large scalability gains over previous work. The second insight is that sufficiently precise results can often be obtained by only analyzing methods once per receiver object instead of once per call, termed an *object-sensitive analysis* by Milanova *et al.* [MRR05]. For example, an object-sensitive analysis would analyze the `Vector.add()` method only once per `Vector` abstract location. The benefits of object sensitivity as compared to standard context sensitivity were shown in the work of Lhoták and Hendren [LH06].

In spite of many impressive advances, no points-to analysis could provide scalable context sensitivity with interactive performance before our work. BDD-based analyses are only able to analyze large programs through various approximations to full context sensitivity (discussed in detail in §2.2), causing some precision loss. Even with this approximation, such analyses can require several minutes of time and more than a gigabyte of memory to run on large Java programs, and they have not successfully scaled to huge frameworks like Eclipse. Finally, time and memory requirements make existing context-sensitive analyses unsuitable when interactive performance is required.

1.2.3 Our Approach

The main contribution of our work is a novel points-to analysis technique that simultaneously (1) scales to large programs, (2) yields more precise results than most existing analyses, and (3) can provide interactive performance for demanding clients like JIT compilers and IDEs. Our analysis currently provides the most precise information about heap behavior in large Java programs. Obtaining significantly more precise information (*e.g.*, with shape analysis [SRW02]) drastically reduces scalability; hence, our analysis sits in a sweet spot in the performance-precision space of static heap analyses for Java. We achieve this precision and scalability through a fundamentally different approach than other state-of-the-art Java points-to analyses: rather than finding clever ways to represent a the exponentially large result of a context-sensitive analysis, we use refinement to only apply precise analysis when needed, greatly increasing scalability while maintaining sufficiently precise results.

Our refinement technique leverages the needs of the analysis client (a *client-driven* approach) to limit analysis effort. The *client* of a points-to analysis is the enclosing program analysis (a verifier, optimizer, etc.) that requires points-to information. Typically, a client uses points-to analysis to prove some other property P about the program, *e.g.*, that some variable only points to objects of a specific type. Given a variable x and a property P, our technique first computes a result for x using less precise but cheaper techniques. If this result is sufficient for proving P, the loop terminates, as the analysis client will be satisfied with the result. If not, a new result for x is recomputed with certain parts of the program analyzed with greater precision. This loop continues until either (1) P can be proved, (2) no more refinement is possible, or (3) some time budget for the query is exceeded.

Integrating client needs into our refinement technique yields scalability benefits over previous work. Unlike work by Plevyak and Chien [PC94] and Guyer and

Lin [GL03], which performs all refinement that could possibly improve the result, our technique only refines for certain key parts of the program, using P to gauge when refinement is sufficient. This difference has a large impact on scalability, since we have observed that in practice a fully refined result cannot be tractably computed for many variables.

To further aid scalability, our analysis is *demand driven*, only computing results for variables relevant to the client. Clients typically interact with a points-to analysis by issuing *queries* for points-to information for specific variables. Most points-to analyses are *exhaustive*, doing most of the analysis work up front and then answering queries with a constant-time lookup operation. A demand-driven analysis takes the opposite approach: after minimal up-front parsing of the program, the analysis only computes required points-to information once a variable is queried. The demand-driven approach allows for time savings when only a small percentage of variables in the program are queried, the behavior of most analysis clients. Furthermore, demand-driven analysis is critical for effective refinement, as detailed analysis state can easily be maintained to help determine where to refine; maintaining such state efficiently in an exhaustive analysis would be difficult.

Note that demand-driven analysis alone (*e.g.*, the work of Heintze and Tardieu [HT01a]) does not yield a scalable and precise analysis for Java. While demand-driven analysis only does the work necessary to compute results for queried variables, its worst-case behavior is identical to that of an exhaustive analysis, and we have observed this behavior in practice. The key to scalable and precise analysis is to combine demand-driven analysis with the right approximation and refinement strategy.

Our initial work on refinement-based points-to analysis focused on devising an analysis suitable for JIT compilers [SGSB05]. We began by formulating Andersen's analysis [And94] for Java, a flow-insensitive, context-insensitive points-to analysis, as a context-free-language reachability (CFL-reachability) problem [Rep98]. In con-

trast, existing work on Java points-to analysis had primarily used the set constraints formalism. Our CFL-reachability formulation showed that Andersen's analysis for Java is a *balanced-parentheses problem*, an insight that enabled our new techniques. We exploited the balanced parentheses structure to approximate Andersen's analysis by *regularizing* the CFL-reachability problem, yielding an asymptotically cheaper algorithm. We also showed how refinement could regain most of the precision lost through regularization. Our evaluation showed that our regularization and refinement approach achieves nearly the precision of field-sensitive Andersen's analysis in time budgets as small as 2 ms per query. Our technique yielded speedups of up to 16-fold over computing Andersen's analysis exhaustively for some clients, with little to no precision loss.

We next developed a scalable and precise context-sensitive points-to analysis [SB06] based on a novel refinement strategy. The analysis has three types of context sensitivity: (1) filtering out of *unrealizable paths*, (2) a *context-sensitive heap abstraction*, and (3) a *context-sensitive call graph* (detailed definitions in Chapter 2. Previous work [LH06] has shown that all three properties are important for precisely analyzing large programs, *e.g.*, to show safety of downcasts. Existing analyses typically give up one or more of the properties for scalability. The key insight behind the analysis is that to maintain all three types of context sensitivity, handling of method calls and heap accesses should be refined *simultaneously*. This technique allows the analysis to precisely analyze important code while entirely skipping irrelevant code. In our experimental evaluation, our analysis proved the safety of 61% more casts than one of the most precise existing analyses [LH06] across a suite of large benchmarks. The analysis checked the casts in under 13 minutes per benchmark (taking less than 1 second per cast) and required only 35MB of memory, compared to 1GB or more for previous approaches [Ste, LH06].

Table 1.2 summarizes the results of our two points-to analysis configurations.

CI/CS	Baseline	Relative Prec.	Relative Time	Relative Mem.
CI	Andersen for Java [LH03]	90% of virtual calls	0.46s (2ms/var) vs. 16s	50KB vs. 30MB
CS	1-ObjSens [LH06]	60% more casts	13min (1s/var) vs. 90min	35MB vs. 2GB

Table 1.2: A summary of the relative precision and scalability of our points-to analysis, in both its context insensitive (CI) and context sensitive (CS) configuration. We give the state-of-the-art analysis used as a baseline, and the relative precision, time, and memory for a representative experiment.

In both cases, the precision of a state-of-the-art baseline analysis was matched or improved upon, memory usage was reduced by orders-of-magnitude, and interactive performance was provided for the first time.

1.3 Thin Slicing

Ideas from our work on refinement-based points-to analysis led us to develop *thin slicing*, a refinement-based technique for answering code relevance questions (*i.e.*, questions of the form "What code affects the behavior of this statement?"). A key result of our points-to analysis work is that, through refinement, sufficiently precise results can be computed while completely ignoring much of the code that a traditional algorithm would analyze. With thin slicing, we applied this same refinement technique to make code more understandable to humans: our hope was that if the points-to analysis could produce good results while ignoring some code, a human could probably also ignore the code when trying to understand the program. This section first describes why better program understanding tools are needed §1.3.1. Then in §1.3.2, we explain how we use a new notion of relevance and refinement to improve significantly on the state of the art.

1.3.1 Motivation

As discussed at the beginning of the chapter, large-scale object-oriented programs can be very hard to understand and debug. Pervasive use of heap-allocated data and complex data structures in these programs cause much of this difficulty. Specifically, multiple levels of pointer indirection in common data structures can make manually tracing the flow of data through the heap prohibitively difficult. When using frameworks like Eclipse or J2EE, the details of such data structures are often unfamiliar to the programmer, making debugging even more difficult for the programmer. In these situations, programmers could benefit from a tool that abstracts away details of heap behavior irrelevant to code inspection and debugging.

Program slicing is a well-known technique that identifies a program subset containing all code that is possibly relevant to a statement or value of interest, called a *seed*[5]. Slicing applies to a variety of program understanding tasks, ranging from testing and debugging to reverse engineering [Tip95]. Weiser [Wei79] originally defined a slice as an *executable* program subset in which the seed statement performs the same computation as in the original program. Weiser's definition is elegant and intuitive, but imposes a rather broad definition of relevance: any statement t that could possibly affect the computation at the seed statement s must appear in the slice. This definition pollutes slices with many statements that indirectly affect a seed but are not pertinent to typical program understanding tasks.

Data structures are a key source of slice pollution. Slices often include internal implementation details of these data structures, which are almost always irrelevant to programmer tasks. Consider a value stored in a deeply nested data structure, *e.g.*, a hash table which holds trees with lists at each tree node. A backwards slice for a statement reading from one such list must include the statements that construct

[5]The seed is often termed the *slicing criterion* in the literature [Tip95]; we use 'seed' for brevity.

and manipulate all levels of this complex data structure, since they affect the read. For many program understanding tasks, however, the programmer needs information about the values stored in the list, but does not care about other details of nested data structures containing the value. Furthermore, modern programs typically rely heavily on well-tested data structures provided by standard libraries, whose internal details rarely concern the end-user programmer. For these common cases, an ideal slicing tool would abstract away internal data structure details in results presented to the user.

1.3.2 Our Approach

Thin slicing is a program understanding technique that redefines relevance in a manner aimed at only including statements useful for developer tasks. For thin slicing, statements that compute or copy a value to the seed are relevant; we call such statements *producers*. Statements that explain *why* producers affect the seed are excluded from a thin slice. In practice, producer statements alone are sufficient for many debugging and program understanding tasks.

We demonstrate the relevance notion used by‘ thin slicing on the Java program fragment of Figure 1.3, which manipulates `String`s stored in a container. Given a stream of full names as input, the example extracts the first names and stores them in a `Vector` (the `readNames()` function), and then later prints out the first names (the `printNames()` function). The `main()` method illustrates a use of the code in a web application, storing and retrieving the names from a `SessionState` object. The example contains a bug: when the program receives as input full name “John Doe”, line 24 erroneously prints “FIRST NAME: Joh”.

Traditional slicing does not help in diagnosing this bug, as a slice seeded with line 24 includes all the code in the example. The entire example is included because

```
1  class Vector {
2    Object[] elems; int count;
3    Vector() { elems = new Object[10]; }
4    void add(Object p) {
5      this.elems[count++] = p;
6    }
7    Object get(int ind) {
8      return this.elems[ind];
9    } ...
10 }
11 Vector readNames(InputStream input) {
12   Vector firstNames = new Vector();
13   while (!eof(input)) {
14     String fullName = readFullName(input);
15     int spaceInd = fullName.indexOf(' ');
16     String firstName =
         fullName.substring(0,spaceInd-1);
17     names.add(firstName);
18   }
19   return firstNames;
20 }
21 void printNames(Vector firstNames) {
22   for (int i = 0; i < firstNames.size(); i++) {
23     String firstName = (String)firstNames.get(i);
24     print("FIRST NAME: " + firstName);
25   }
26 }
27 void main(String[] args) {
28   Vector firstNames =
       readNames(new InputStream(args[0]));
29   SessionState s = getState();
30   s.setNames(firstNames);
31   ...;
32   SessionState t = getState();
33   printNames(t.getNames());
34 }
```

Figure 1.3: Example showing the advantages of thin slicing. The six statements with underlined expressions are in the thin slice seeded with line 24, while the traditional slice for line 24 contains all displayed code. The bodies of functions with inessential behavior (*e.g.*, `print()`) have been elided for clarity.

the slice must include all the code to construct and populate the `Vector` passed to `printNames()` and the code in `main()` retrieving the `Vector` from the `SessionState` object (all of which affects line 24). As in this example, slices for Java programs typically include most of the program.

What lines of code would be shown by an ideal debugging tool for this example? The bug lies at line 16, which incorrectly passes `spaceInd-1` (rather than `spaceInd`) to `String.substring()`. Seeing how this erroneous `String` flows to where it is printed would almost immediately lead the user to the problematic line. In this case, the flow traverses a `Vector`: line 17 adds the `String`, and line 23 retrieves it.

A thin slice only includes *producer statements* for the seed. We say statement s is a producer for statement t if s is part of a chain of assignments that computes and copies a value to t. In terms of our refinement technique (see §1.1), only including producers in a thin slice abstracts away the question of why base pointers are aliased for all field accesses. In Figure 1.3, the producer statements for the seed, highlighted with underlining, are a superset of the statements most relevant to the bug in question. We are interested in the pointer value in `firstName` at line 24, and the thin slice allows us to easily trace its flow (relevant expressions are underlined):

- Line 23 copies the value returned by `Vector.get()`.
- `Vector.get()` obtains the value from an array read (line 8).
- The value is copied into the array in `Vector.add()` (line 5).
- `Vector.add()` gets the value from the actual parameter at line 17.
- Line 17 passes the value returned at line 16, the buggy statement.

Unlike a traditional slice, the thin slice does not provide an executable program; for example, statements initializing the `Vector` containing the erroneous `String` are excluded. However, the thin slice more directly leads the user to the bug.

Advantages of Thin Slicing One reason that thin slicing works well is that it ignores value flow into base pointers of heap accesses, focusing just on the value read from or written to the heap. For example, line 8 reads `this.elems[ind]`. A thin slicer ignores the values of the two dereferenced base pointers (`this` and `this.elems`), focusing solely on statements that can write into the array (*i.e.*, the statement `this.elems[count++] = p;` at line 5). In contrast, a traditional slicer includes statements that influence both the base pointers `this` and `this.elems`, contributing to the blowup in slice size. For many program understanding tasks, base pointer manipulation matters less than actual copying of the value through the heap.

In addition to being better focused on statements of interest for programmer tasks, thin slices have an intuitive semantic definition that makes them understandable in isolation. Simply stated, a thin slice contains all statements flowing values to the seed, ignoring uses of base pointers. These well-defined semantics allow a user to reason about thin slices in a self-contained manner, since she knows that all producer statements are included in a thin slice. If slices were shrunk using some ad-hoc method, such as setting a constant limit on slice size, the user could not easily characterize what is in the presented slice subset and what is missing.

In cases where a thin slice alone is insufficient for some programmer task, it can be *refined through expansion* with additional thin slices to ease the understanding of other relevant statements. One case in which statements outside the thin slice may be needed is to explain the cause of base pointer aliasing for heap accesses, as previously discussed in §1.1. Given a field read `x := y.f` and a field write `w.f := z` in a thin slice, a user may ask how aliasing between `y` and `w` arises, causing heap-based flow from `z` to `x`. This question can be answered via two more thin slices, respectively showing how some object o flows to both `y` and `w`. Note that is observation is simply an instantiation of our general refinement technique (see §1.1): the requested base pointer aliasing is explained, while the behavior of other field accesses remains

abstracted. This refinement of the thin slice can be repeated recursively for further aliasing questions, exploiting the hierarchical nature of Java heap accesses as was done with our pointer analysis.

A similar refinement technique applies to explaining *why* statements in the thin slice can execute. Explaining why a statement can execute requires showing its *control dependences*. Since traditional slices include all possibly relevant statements, they must include all transitive control dependences. Unfortunately, Java's semantics make many statements a type of conditional branch, often yielding a huge number of control dependences. For example, if a statement might throw an exception, many statements will be control dependent on its successful execution. Similarly, each virtual call `x.m()` is a conditional expression because it branches on the runtime type of `x`.

In practice, we found that when control dependences are relevant, they usually appear *lexically near* a thin slice statement, and hence can usually be identified from browsing the surrounding source code. Thin slices seeded with these conditionals can be computed to further understand their behavior. In the limit, hierarchically expanding a thin slice to show control and aliasing explanations yields a traditional slice; hence, any possibly relevant statement can eventually be discovered.

Of course, thin slicing does not provide a panacea: in certain cases, thin slices with expansion grow too large to effectively identify statements of interest. In our experiments, thin slicing was useful for 13 of 19 debugging tasks and 22 of 29 program understanding tasks, an encouraging result. Furthermore, of those tasks where thin slicing was useful, the thin slice alone was sufficient for half of them, and for the others typically one or two additional statements were needed. Hence, we believe thin slicing holds great promise as a basis for a practical program understanding tool.

Our work focuses on static thin slicing for Java, but the technique is more broadly applicable. Thin slicing itself relies on standard data dependence concepts [HPR89] and hence should apply to many programming languages. Our hierarchical expansion

technique relies on properties of Java pointer accesses, however, and may not work as effectively for languages like C (see §6.3). Also note that dynamic thin slices can be defined in a straightforward manner using dynamic data dependences.

Our initial evaluation of thin slicing yielded highly promising results. Our experiments compared thin slicing and traditional slicing for several debugging and program understanding tasks, using a methodology that simulates realistic use of a slicing tool (details in §6.5). Our results showed that thin slicing had the following properties:

- Thin slices usually included the desired statements for the tasks (*e.g.*, the buggy statement for a debugging task).
- Thin slices revealed desired statements after inspecting 3.3 times fewer statements than traditional slicing for debugging tasks and 9.4 times fewer statements for program understanding tasks.
- Our thin slicing algorithm (a simple modification of well-known techniques [HRB88, RHSR94]) scales to relatively large Java benchmarks.

1.4 Dissertation Overview

The bulk of this dissertation is devoted to describing our work on refinement-based points-to analysis. Chapter 2 gives definitions and explanations of various terminology relevant to points-to analysis and context-free-language reachability (CFL-reachability). Then, Chapter 3 presents formulations of several points-to analysis variants in CFL-reachability, highlighting the structure exposed by the formalism. Chapter 4 discusses our refinement-based context-insensitive points-to analysis, and finally, Chapter 5 presents our refinement-based context-sensitive points-to analysis. Both techniques provide high precision and dramatic scalability improvements, as summarized in Table 1.2.

Chapter 6 describes thin slicing in detail, including its new notion of relevance, algorithms for computing thin slices, and results from our experimental evaluation. Related work for both our refinement-based points-to analysis and thin slicing is discussed in Chapter 7. Finally, Chapter 8 has concluding thoughts and possible directions for future work.

Chapter 2

Points-To Analysis Background

This chapter provides a brief introduction to points-to analysis terminology. Our aim is to provide sufficient background to make the presentation of our points-to analysis understandable, but not to give a complete, formal treatment of all discussed terminology; we provide references to formalizations in the literature where possible. We begin in §2.1 by defining what program properties are computed by a points-to analysis. Then, in §2.2 we discuss terminology related to points-to analysis precision.

2.1 Points-To Analysis Definition

A *points-to analysis* computes an over-approximation of the possible objects that each pointer variable may point to during any execution of a program. Runtime objects are represented with a finite set of *abstract locations*. Abstract locations are necessary because non-terminating programs may allocate an unbounded number of objects (impossible to represent directly), and distinguishing such programs from those that do terminate is undecidable. The granularity with which a points-to analysis models dynamic objects using abstract locations is termed its *heap abstraction*. We discuss heap abstractions in greater detail in §2.2.5.

Traditionally, points-to analysis results are represented with *points-to sets*; we write $pt(x)$ for the points-to set of variable x. If $o \notin pt(x)$ for abstract location o and variable x from program P, then x can *never* point to an object represented by o in any execution of P. If $o \in pt(x)$, then x *may* point to some object represented by o in some execution of P; since points-to sets are over-approximations, it is possible that x never points to such an object.

We illustrate the input and output of a points-to analysis with a simple example. Consider a program containing a statement "`x = new Obj();`" and no other statements defining the variable `x`. The heap abstraction used by Andersen's points-to analysis [And94] and many others (details in §2.2.5) represents the objects allocated by all executions of this statement with a single abstract location o. This heap abstraction leads to the analysis result $pt(x) = \{o\}$.

2.2 Points-To Analysis Precision

Points-to analyses use various approximations of the language semantics to balance precision of the analysis results with performance of the analysis (both in time and space). In this section, we will describe the types of precision that can be used to model various language features, introducing relevant terminology along the way. We will also indicate how the points-to analyses in our work handle these language features.

In §2.2.1 through §2.2.5, discussions of precision assume that the points-to analysis is *flow insensitive*. In brief, a flow-insensitive analysis treats each intraprocedural control-flow graph as if it has all possible edges, preventing reasoning about the order in which statements execute or how many times they execute. For scalability reasons, most points-to analyses in the literature, including ours, are flow insensitive. We discuss flow sensitivity in detail in §2.2.6.

```
1  y = new Obj();
2  z = new Obj();
3  x = y;
4  x = z;
```

Figure 2.1: A simple example to illustrate different treatments of assignments.

Before we begin, a quick note on heap abstractions: unless stated otherwise, in the rest of this dissertation we assume a heap abstraction with one abstract location per allocation site (a **new** expression for Java) when giving examples of points-to sets. (We discuss other possible heap abstractions in §2.2.5.) Furthermore, by convention we name an abstract location o_k when the corresponding allocation site occurs on line k of an example program.

2.2.1 Assignments

A points-to analysis can model assignment statements in either a precise *subset-based* manner or in an approximate *equality-based* manner. A subset-based analysis handles an assignment `x = y` by ensuring that $pt(y) \subseteq pt(x)$ in the analysis result. This constraint precisely models the assignment, since it updates `x` to point to whatever value `y` points to. In contrast, an equality-based analysis ensures $pt(y) = pt(x)$ given the same assignment, an approximation since the statement does not cause a flow of values from `x` to `y`. (In essence, an equality-based analysis effectively adds the inverse assignment `y = x` to the analyzed program.) While algorithms exist for equality-based analysis that are nearly linear in the size of the program [Ste96], subset-based analysis requires worst-case cubic time. Some analyses use a mix of equality- and subset-based techniques to find a sweet spot in scalability and precision, *e.g.*, Das's one-level flow analysis [Das00].

To see how modeling of assignments affects the analysis result, consider the simple

example in Figure 2.1. A subset-based analysis models line 3 with the constraint $pt(y) \subseteq pt(x)$ and line 4 with the constraint $pt(z) \subseteq pt(x)$. Solving these constraints yields the precise result $pt(y) = \{o_1\}, pt(z) = \{o_2\}, pt(x) = \{o_1, o_2\}$. On the other hand, an equality-based analysis models line 3 with the constraint $pt(x) = pt(y)$ and line 4 with $pt(x) = pt(z)$. Solving these constraints yields identical points-to sets for all the variables, *i.e.*, $pt(x) = pt(y) = pt(z) = \{o_1, o_2\}$. Our refinement-based points-to analysis always treats assignments in a subset-based manner.

2.2.2 Field accesses

Points-to analyses use various techniques to model the semantics of reads and writes to fields of structures or objects. Handling of fields in a points-to analysis boils down to answering the following question: Given some field write statement `w.f = z` and some field read statement `x = y.f`, can executing the statements cause values to flow from `z` to `x`? A precise answer to this question requires checking the following two conditions:[1]

1. **Base pointer equality** In order to cause heap-based flow, the field read and write statements must be accessing the same object / structure. For the above statements, this condition corresponds to checking that the base pointers `w` and `y` may be aliased, *i.e.*, $pt(w) \cap pt(y) \neq \emptyset$.

2. **Offset equality** The field accesses must also be reading and writing the same offset in the object in order to cause a flow. Strongly-typed languages like Java guarantee that distinct object fields cannot name the same memory location, in which case this condition reduces to ensuring the statements access the same object field. For languages like C that allow unchecked casting and pointer

[1] Again, this discussion assumes that the points-to analysis is flow insensitive (details in §2.2.6).

```
1  class A {
2    A foo() {
3      return new A();
4    }
5  }
6  class B extends A {
7    A foo() {
8      return new B();
9    }
10 }
11 main() {
12   A x = new A();
13   A y = new B();
14   A z = y.foo();
15 }
```

Figure 2.2: A simple code example to illustrate virtual calls.

arithmetic, more complex checking may be necessary (see [YHR99] for a detailed discussion).

There are three well-known treatments of fields in the points-to analysis literature. A *field-sensitive* analysis checks both of the above conditions. A *field-insensitive* analysis (also known as field independent [HT01b]) only checks the first condition, while a *field-based* analysis only checks the second condition. Chapter 4 presents a configuration of our analysis in which refinement adds targeted field sensitivity to an initial field-based analysis pass.

2.2.3 Call Graph

For languages with any form of indirect function calls (*e.g.*, calls through function pointers in C, virtual calls in Java, higher-order functions in Scheme, etc.), performing a points-to analysis and computing a call graph for a program are mutually dependent: points-to information is needed to determine possible targets of indirect function

calls, and the possible targets of those calls must be known to compute points-to information for variables involved in the call (*e.g.*, return values). Consider the code example in Figure 2.2. In order to determine $pt(z)$, a points-to analysis must consult a call graph to determine the possible targets of the virtual call `y.foo()` at line 14, as `z` is assigned the return value of the call. Computing the possible targets of this virtual call requires determining the possible runtime types of its receiver argument `y`. But, finding `y`'s possible runtime types requires computing what it can point to, illustrating the circular dependence between call graph construction and points-to analysis.

Call graphs for points-to analysis are either constructed *ahead of time* or *on the fly*. With ahead-of-time call graph construction, first some less expensive analysis, *e.g.*, rapid type analysis (RTA) [BS96], is used to construct a conservative call graph. This pre-computed call graph is used to find virtual call targets during points-to analysis. With on-the-fly call graph construction, the points-to analysis uses its current points-to information to determine call graph information, iterating between points-to analysis and call graph construction until a fixed point is reached. On-the-fly call graph construction is the most precise technique for points-to analysis, since all virtual call targets are determined using the results of the points-to analysis itself.

Building a call graph ahead of time may yield less precise points-to analysis results than building it on the fly. Consider again the example of Figure 2.2. The `y.foo()` call at line 14 invokes `B.foo()`, since `y` points to a `B` object. However, a call graph constructed ahead of time with rapid type analysis will conclude imprecisely that the call could invoke either `A.foo()` or `B.foo()`.[2] This imprecise call graph will then lead the points-to analysis to conclude that `z` can either point to o_3 or o_8. In contrast, a points-to analysis with on-the-fly call graph construction would find only the `B.foo()`

[2]The creation of an `A` object at line 12 of Figure 2.2 causes this imprecise result, as RTA inspects which classes have been instantiated when determining call targets.

target for the virtual call, thereby concluding that `z` can only point to o_8.

Though it may decrease precision, in some cases ahead-of-time call graph construction performs better than on-the-fly call graph building. Consider the case where ahead-of-time and on-the-fly techniques yield similarly sized call graphs for a particular program. In this case, ahead-of-time call graph construction often yields a faster running time, since on-the-fly call graph construction increases the number of iterations required for the points-to analysis to reach a fixed point. However, an on-the-fly call graph may include many fewer methods and call graph edges than an ahead-of-time call graph, since it is constructed more precisely. In this case, the time required to analyze the additional methods in the ahead-of-time call graph may dwarf the extra iterations required for on-the-fly call graph construction.

Our points-to analyses can refine an ahead-of-time call graph on-demand. This refinement technique yields the precision benefits of on-the-fly call graph construction at lesser cost, since many unimportant call sites will not have their targets refined. Our call graph handling is discussed more in §3.4 and §5.3.2.2.

2.2.4 Method Calls

Points-to analyses vary in how precisely they model the semantics of method calls and returns. A *context-insensitive* analysis does not precisely model the target address of return statements, instead treating calls as goto instructions. A return from a call to some method `f` is conservatively treated as if it could branch to the point after *any* call to `f`, not just the actual caller. Hence, context-insensitive analyses include results from impossible control-flow paths with mismatched calls and returns, termed *unrealizable paths* [RHS95]. A *context-sensitive* analysis does not have this imprecision; method call semantics are precisely modeled, and only realizable paths

affect analysis results.[3]

Let us define context-sensitive points-to analysis more formally. For the moment, we assume that the program P to be analyzed is recursion-free; we will discuss relaxing this assumption shortly. We also temporarily assume a the existence of a pre-computed call graph CG for P, for simplicity. Let P' be the program constructed by inlining all method calls in P, according to CG. The *context-sensitive points-to analysis problem* is defined as computing the result of a context-insensitive points-to analysis on the (call-free) program P'. Note that the actual algorithm need not work over P'; it must only compute the same points-to relation.

With this definition, a context-sensitive points-to analysis computes a separate points-to set for each inlined copy of a variable. We denote a (local) variable in P' as $\langle v, c\rangle$, where v is the corresponding variable in P and c is the complete sequence of call sites (or *call string*) that was inlined to create the copied variable, from the root of the call graph CG for P to the method declaring v. The (copied) allocation sites of P' have analogously named abstract locations. Thus, pt' maps "context-refined" variables to "context-refined" abstract locations: $\langle o, c'\rangle \in pt'(\langle x, c\rangle)$. For reasoning about variables in the original program P, the context-sensitive result is projected back in the natural manner:

$$pt(x) \equiv \{\ o \mid \exists\ c, c'\ .\ \langle o, c'\rangle \in pt'(\langle x, c\rangle)\}$$

We illustrate the potential precision benefits of context-sensitive analysis with the example of Figure 2.3. A context-insensitive points-to analysis would treat the two calls to `id` in the example as if they could return their result to either call site, yielding the imprecise result $pt(c) = pt(d) = \{o_2, o_3\}$. In contrast, a context-sensitive analysis

[3]Note that in type-inference-based formulations of points-to analysis, context-insensitive and context-sensitive analyses are respectively referred to as *monomorphic* and *polymorphic* analyses [OJ97, FFA00, O'C00, RF01, KA07], standard type inference terminology.

```
1  Obj id(Obj p) { return p; }
2  a = new Obj();
3  b = new Obj();
4  c = id(a);
5  d = id(b);
```

Figure 2.3: An example illustrating the precision benefit of context-sensitive points-to analysis.

keeps the effects of the two calls separate, yielding the projected result $pt(c) = \{o_2\}$ and $pt(d) = \{o_3\}$.

A second precision benefit of context-sensitive analysis stems from direct use of the context-sensitive points-to relation. For the example of Figure 2.3, $pt(p) = \{o_2, o_3\}$, since both o_2 and o_3 are passed as a parameter to `id`. However, the context-sensitive points-to relation pt' maintains two different points-to sets for `p`, one for each call: $pt'(p, [4]) = o_2$ and $pt'(p, [5]) = o_3$. (Here and in the rest of the dissertation, we name call sites in example programs by their line number.) Certain tools can benefit from directly querying the context-sensitive points-to relation, for example static race detection [NAW06].

Context-sensitive analysis combined with on-the-fly call graph construction yields a *context-sensitive call graph*.[4] Such an analysis can be defined as performing inlining on-the-fly: starting from the entry methods of a recursion-free program P (*e.g.*, `main()`), a context-insensitive points-to analysis is performed, but as call targets are discovered on-the-fly, the callees are inlined and the analysis continues.

A context-sensitive call graph yields additional precision by computing virtual call targets separately for each calling context. For example, consider the call to `Object.equals()` inside the `Vector.contains()` method in the Java standard library,

[4]While it is theoretically possible for an ahead-of-time call graph to be context sensitive, this configuration makes little practical sense due to the high cost of computing a context-sensitive call graph. Hence, we assume that any context-sensitive call graph is computed on-the-fly.

used to check if the `Vector` contains some object. Say the program has a call "`v.contains(a)`", where `v` only contains `A` objects. A context-sensitive call graph has a distinct call site for the `equals()` call within this `contains()` call, and hence can prove that it invokes `A.equals()`. In contrast, a context-insensitive call graph has only one copy of `contains()`, and therefore one copy of its nested `equals()` call site. With this representation, the call graph will indicate that the nested `equals()` call could invoke the `equals()` method of objects stored in *any* `Vector`.

Handling recursive calls in a context-sensitive points-to analysis requires approximation, due to interactions with handling of fields. Reps proved that context-sensitive, field-sensitive analysis is undecidable [Rep00]. Therefore, either fields or method calls must be modeled approximately by a points-to analysis to ensure termination. Our refinement-based points-to analysis approximates handling of recursive method calls for decidability, as discussed in §5.2.

Even with state-of-the-art algorithms, full context sensitivity (approximated for decidability) is still intractable for large programs. The exhaustive inlining approach to context sensitivity suggested by our analysis definition can cause a worst-case exponential blowup in the size of the program, as there are a worst-case exponential number of paths in a program's call graph. In practice, exhaustive inlining has been shown to create up to 10^{23} copies of a method for large Java programs [WL04]. To this date, no technique for context-sensitive points-to analysis has been devised that has better worst-case complexity than the exhaustive inlining technique. Hence, fully context-sensitive points-to analysis for large programs remains intractable.

To overcome the intractability of context-sensitive points-to analysis, numerous approximations have been devised. One of the earliest approximation techniques was *k-limiting* [Shi88], which limits the modeling of the call stack to depth k. Typically, points-to analyses have only scaled to large programs with $k \leq 3$. Cartesian product analysis [Age95] and its variants analyze each method once for each set of

concrete argument types, rather than once per call. A similar recent approximation is *object sensitivity* [MRR05], which analyzes methods separately for each (abstract) receiver object rather than for each call site. Recent work [LH06, NAW06] has shown that a k-limited object-sensitive analysis is empirically more scalable and precise than the comparable k-limited context-sensitive analysis. Finally, another approximation of full context sensitivity is to use a context-insensitive heap abstraction (*e.g.*, in [WL04]), discussed further in the next section.

2.2.5 Heap Abstraction

As discussed in §2.1, a key parameter of a points-to analysis is its *heap abstraction*, *i.e.*, its finite model of a program's possible runtime objects. Here, we first describe some typical heap abstractions used with context-insensitive analyses. Then, we discuss the interaction between context sensitivity and the heap abstraction, showing why a context-sensitive heap abstraction can yield greatly improved precision.

Context-insensitive heap abstractions can vary widely in coarseness, depending on the desired trade-off between scalability and precision. Analyses using equality-based techniques [Ste96, Das00] unify abstract locations deemed equivalent during analysis, sometimes resulting in a single abstract location representing a large portion of the heap. While this representation may be coarse, such analyses scale very well to large programs. Other analyses utilize a more precise type-based heap abstraction, where all objects of a particular type are represented with a single abstract location. This abstraction is useful for tasks like performing type inference for object-oriented programs, but can be too imprecise for more demanding tasks like disambiguating aliases [DMM98]. Andersen's analysis [And94] and its variants use an even more precise heap abstraction, representing the objects created by each allocating statement (`malloc` in C, **`new`** in Java) with a separate abstract location. This heap abstraction is

```
1 Obj[] makeArr() { return new Obj[10]; }
2 Obj[] a = makeArr();
3 Obj[] b = makeArr();
4 a[0] = new Obj();
5 b[0] = new Obj();
6 Obj x = a[0];
7 Obj y = b[0];
```

Figure 2.4: A simple example to illustrate the benefits of a context-sensitive heap abstraction.

the most difficult to scale, though modern BDD-based techniques can represent the resulting points-to sets quite compactly [BLQ+03, ZC04].

Some context-sensitive points-to analyses also use a context-sensitive heap abstraction, which especially improves treatment of heap-based data structures. As defined in §2.2.4, a context-sensitive analysis has a separate abstract location for each inlined copy of an allocation site. In practice, the key benefit of this heap abstraction is the ability to distinguish the contents of different data structure instances [LH06, LLA07]. Internal objects of such data structures are often allocated in standard methods (*e.g.*, the constructor in Java), and a context-insensitive heap abstraction uses a single abstract location to represent all objects allocated in such methods, causing merging of the contents of all data structure instances in the analysis results. In contrast, a context-sensitive heap abstraction can represent the internal objects of each data structure separately, allowing for separate reasoning about the contents of each data structure instance [LLA07].

Figure 2.4 provides an example illustrating how a context-sensitive heap abstraction can better distinguish the contents of arrays, a simple heap-based data structure. The heap abstraction of Andersen's analysis [And94]—using one abstract location per **new** expression—models the arrays created in all calls to `makeArr()` with a single abstract location o_1. This coarse model would lead a points-to analysis to conclude

that `a` and `b` point to the same array, and therefore that line 4 through line 7 all access the same array. The points-to analysis would then conclude, for example, that $pt(x) = \{o_4, o_5\}$, an imprecise result. A context-sensitive heap abstraction uses distinct abstract locations for the arrays allocated by the two calls to `makeArr()`, removing this imprecision. Standard data structures like Java's `Vector` benefit from a context-sensitive heap in a similar way, as the internal array implementing each `Vector` is allocated inside its constructor method (shown in greater detail in §5.1.3).

Our refinement-based points-to analysis is the first to provide a context-sensitive heap abstraction sufficient for distinguishing data structure contents while scaling to large programs. The BDD-based points-to analysis of Whaley and Lam [WL04] uses a context-insensitive heap abstraction, which aids scalability but dramatically decreases precision for demanding clients [LH06]. Equality-based analyses with a fully context-sensitive heap abstraction have been developed, but in our experience do not scale to large Java programs (discussed in more detail in §5.4). Subset-based analyses have scaled with a k-limited context-sensitive heap abstraction [LH06, NAW06], but k-limiting decreases precision for some clients. Our analysis is able to both provide a context-sensitive heap and scale because it is able to *refine the heap abstraction* during analysis, thereby only utilizing the context-sensitive heap where needed; see §5.2.1 for further discussion.

Shape analyses (*e.g.*, [SRW02]) can provide precision beyond that available with a context-sensitive heap abstraction. A context-sensitive heap abstraction is often insufficiently precise to prove interesting properties about recursive data structures (*e.g.*, that a linked list is acyclic). The objects comprising such data structures are often allocated in a loop, and hence even a context-sensitive heap abstraction will represent the objects with a single abstract location. In contrast, shape analyses are often able to reason separately about objects allocated and manipulated in distinct loop iterations (at great cost in scalability). The more precise heap abstraction pro-

```
1  x = new Obj();
2  x.f = new Obj();
3  y = x.f;
4  x.f = new Obj();
5  z = x.f;
6  y = z;
```

Figure 2.5: A simple example to illustrate the benefits of flow-sensitive points-to analysis.

vided by shape analysis often only makes sense if the analysis is sensitive to control flow, as discussed in the next section.

2.2.6 Control Flow

More precise modeling of a program's control flow can lead to various precision benefits for points-to analysis. Most points-to analyses are *flow insensitive*, *i.e.*, they assume that statements within a procedure may execute (1) in any order and (2) any number of times. A *flow-sensitive* analysis aims to reduce imprecision by eliminating consideration of impossible statement execution orderings (much like context-sensitive analysis eliminates consideration of unrealizable paths).[5] Our points-to analyses are not flow sensitive to remain scalable. Here, for completeness, we discuss some of the potential benefits of flow sensitivity.

Strong Updates Flow-sensitive analyses can benefit from performing *strong updates* of memory locations. An analysis performs a strong update when in reasoning about a write to some memory location l, it models the fact that the old value in l is overwritten. In the example of Figure 2.5, consider computing the points-to set of `z`. The value in `z` is copied from `x.f`, which is assigned different values at line 2 and line 4. A flow-insensitive analysis assumes that those assignments can execute

[5] In this work, flow sensitivity has its typical meaning of precise modeling of control flow. In other work (for example [O'C00]), flow sensitivity is used to refer to data-flow sensitivity, *i.e.*, subset-based modeling of assignments (see §2.2.1).

in any order, and hence concludes that $pt(z) = \{o_2, o_4\}$. In contrast, a flow-sensitive analysis knows that line 4 must execute after line 2. Hence, the analysis can use a strong update to treat the statement at line 4 as overwriting the o_2 value in `x.f` with o_4, yielding the more precise result $pt(z) = \{o_4\}$.

The key challenge of strong updates is in identifying the single heap location updated by a statement. Strong updates on local variables can be performed transparently by first converting the input program into static single assignment (SSA) form [CFR+91].[6] Doing a strong update for a write to a heap location is more difficult because the analysis must identify *a single runtime object* being updated by the write. A strong update cannot be performed if a statement updates an abstract location that can represent multiple runtime objects. For Figure 2.5, say that line 1 were replaced by the following:

```
while (...) {
  ...; x = new Obj();
}
```

Most points-to analyses would model the allocation in the loop with a single abstract location o_l, representing the objects allocated in all loop iterations. However, at runtime line 2 and line 4 only update the `f` field of *one* of those dynamic objects. So, performing a strong update on $o_l.f$ for those statements (representing the objects pointed to by the `f` field of *all* the loop-allocated objects) would be unsound. Doing strong updates for cases like this one typically requires a very precise heap model, *e.g.*, that used in shape analysis [SRW02], though in certain cases less expensive analysis can be used [DADY04, FYD+06].

Separate Result per Program Point Flow-sensitive analysis may also enable greater precision by computing a distinct result for each program point separate. In

[6] In fact, SSA can be used to make otherwise flow-insensitive points-to analyses like ours partially flow sensitive.

```
1 if (...)
2   a = y;
3 else
4   b = y;
```

Figure 2.6: A simple example to illustrate the benefits of path-sensitive points-to analysis.

Figure 2.5, `y` is assigned at both line 3 and line 6. A flow-insensitive analysis can only provide one points-to set for `y` that includes the effects of both of these assignments. However, a flow-sensitive analysis can give more precise answers when asked about a points-to set at a specific program point. In this case, the flow-sensitive analysis would compute a separate points-to set for `y` for each assignment, $\{o_2\}$ after line 3 and $\{o_4\}$ after line 6. Note that conversion to SSA form provides much of this benefit, since in SSA form each local variable is assigned exactly once.

Path Sensitivity Further precision gains may be possible through *path sensitivity*, *i.e.*, reasoning separately about different control-flow paths rather than merging their effects. In Figure 2.6, a path-insensitive analysis would imprecisely conclude that `a` and `b` may be aliased at the end of the example, as it would merge the effects of line 2 and line 4, essentially treating the statements as if they could both execute in one run of the program. A path-sensitive analysis would keep the effects of the 'then' and 'else' branches of the conditional separate, and hence could prove that `a` and `b` cannot be aliased. A path-sensitive analysis may also use conditional expressions to prove that certain sequences of branch outcomes are infeasible. Recent work has made impressive progress on building a scalable a points-to analysis with intraprocedural path sensitivity [HA06], but scalable interprocedural path sensitivity for points-to analysis is still an open problem.

Chapter 3

Points-To Analysis Formulations

Chapter 2 introduced the terminology of points-to analysis. Here, we build on that background to formally specify Java points-to analysis using the *context-free language reachability* (CFL-reachability) framework. Our formulations will show how *balanced parentheses* can be used to model a variety of Java language features. Apart from the standard use of balanced parentheses in modeling method calls and returns [RHSR94, RHS95], we also use them to handle heap accesses and virtual calls precisely. In Chapter 4 and Chapter 5, we give refinement-based points-to analysis algorithms that exploit this balanced parentheses structure.

We define our analysis problem as computing the *best possible* (*i.e.*, most precise) flow-insensitive points-to information for a program in a slightly restricted subset of Java. (In terms of the precision axes introduced in §2.2, our formulation employs the most precise handling for assignments, fields, method calls, virtual calls, and heap allocation, but it is flow insensitive.) The analysis is precise for programs that (1) have no arrays and (2) do not make use of native methods or reflection; our approximate handling of these program features is discussed in §3.2.3. Furthermore, not that our analysis algorithm (described in Chapter 5) does not compute the fully precise result

formulated here, as recursive method call semantics are approximated for decidability (see §5.2).

We present a series of increasingly precise points-to analysis formulations, culminating in the aforementioned best possible flow-insensitive points-to analysis. We first present some background on CFL-reachability in §3.1. Then, §3.2 presents a points-to analysis that is context insensitive (defined in §2.2.4) and uses ahead-of-time call graph construction (defined in §2.2.3). In §3.3, we show how to add context sensitivity (defined in §2.2.4) to this formulation. Finally, §3.4 shows how to enhance the formulation with on-the-fly call graph construction (defined in §2.2.3).

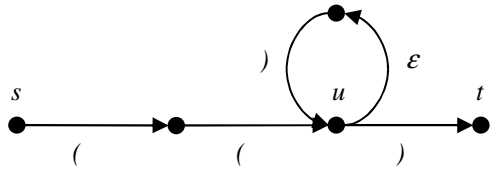
3.1 Context-Free Language Reachability

Our points-to analysis work differs from most previous work in its use of *context-free language reachability* (CFL-reachability) as an underlying formalism. Our formulation of Java points-to analysis as a CFL-reachability problem (based on a previous formulation for C [Rep98]) exposed structure that enabled many of the key insights behind our analysis algorithms. In this section, we briefly define CFL-reachability and discuss some of its key properties; see the excellent overview by Reps [Rep98] for a more detailed discussion.

CFL-reachability is an extension of traditional graph reachability that allows for filtering of uninteresting paths. The input for a CFL-reachability program is a directed graph G with edge labels taken from some alphabet Σ. Each path p in G has a corresponding string $s(p)$ in Σ^*, constructed by concatenating in order the labels of edges in p. Let L be a context-free language over Σ. We say p is an *L-path* iff $s(p) \in L$. The (all-pairs) CFL-reachability problem requires determining for all nodes s and t whether G contains an L-path from s to t. The language L characterizes interesting paths for CFL-reachability, as non-L-paths are filtered from consideration. (For pro-

gram analysis, uninteresting paths are typically those that correspond to infeasible executions of the analyzed program.) When an L-path exists from s to t, we say t is *L-reachable* from s, or simply, $s\ L\ t$; we similarly refer to S-paths and use notation $s\ S\ t$ for any non-terminal S in L's grammar. Finally, the notation $s\ S\ t\ T\ u$ means that there is an S-path from s to t and a T-path from t to u.

As an example, let Σ be the letters '(' and ')', and L be the set of strings with balanced parentheses generated by the grammar $S \rightarrow SS \mid (\ S\ ) \mid \epsilon$. Consider the following input graph (adapted from an example in [Rep98]):



There is exactly one L-path p from s to t, with $s(p) =$ "(())", so t is L-reachable from s. However, there is no L-path from u to t.

For points-to analysis, G represents the program: its nodes model variables and abstract locations, and its edges model different types of assignments. L describes paths in G corresponding to program executions that *might* cause a variable to point to some abstract location; other paths are guaranteed not to affect the points-to relation. In Chapter 3, We define L such that if x may point to o (*i.e.*, $o \in pt(x)$), then $o\ L\ x$. Perhaps counter-intuitively, the L-path goes from o to x rather than from x to o, since our edges are oriented in the direction of value flow, *i.e.*, from the right-hand side of an assignment to the left-hand side (see §3.2.1 for details). Hence, computing the points-to set of a variable x is a single-target L-path problem [Rep98], requiring backwards reachability.

In most existing CFL-reachability formulations of program analyses, the filter language is a balanced parentheses language (also known as a Dyck language [Har78]).

The best-known work on using CFL-reachability for program analysis [RHSR94, RHS95] uses a balanced parentheses language for *context sensitivity*, *i.e.*, precise handling of method call semantics (discussed further in §3.3). In addition to using balanced parentheses to achieve context sensitivity, our work contributes two novel uses of balanced parentheses, modeling both Java heap accesses (see §3.2) and on-the-fly determination of virtual call targets (see §3.4) with the technique. Note that not all analyses formulated in CFL-reachability use a balanced parentheses language, for example the formulation of Andersen's analysis for C in [Rep98].[1]

Determining CFL-reachability is in general computationally more expensive than standard graph reachability. For many years, the best known algorithm for CFL-reachability required worst-case $O(\Gamma^3 N^3)$ time [Rep98], where N is the number of nodes in G and Γ is the size of a normalized grammar for L. Recently, Chaudhuri has recently devised an more efficient algorithm which runs in worst-case $O(N^3/\log N)$ time [Cha06] (this bound assumes that Γ is a constant). However, standard transitive closure can be solved more efficiently in worst-case $O(NE)$ time.[2]

Note that for our points-to analysis, we do not use the generic CYK-based CFL-reachability algorithm [Rep98] due to space overhead. The CYK-based algorithm requires saturating the graph with closure edges corresponding to each non-terminal in a normalized version of the language grammar [Rep98]. In our experience, these closure edges add an enormous space overhead for points-to analysis, without yielding commensurate time savings. Hence, we develop our own algorithms for reachability specific to the context-free languages we formulate for points-to analysis. We are not aware of a scalable program analysis based on the generic CFL-reachability algorithm.

In certain cases, the structure of a CFL-reachability problem can be exploited

[1] Note that Zheng and Rugina [ZR07] have recently presented an alternate formulation of alias analysis for C with balanced parentheses.

[2] The greater efficiency assumes a sparse graph, with the number of edges E being $O(N)$. In practice, the graphs we construct to model programs are always sparse.

to yield better worst-case complexity bounds. When L is a regular language, L-reachability can be solved in $O(SNE)$ time, where S is the number of states in a deterministic finite automaton for L [Yan90]. Also, for context-sensitive dataflow analysis and slicing, algorithms with better worst-case complexity for real-world programs have been devised [RHSR94, RHS95], exploiting the typical structure of procedure calls. §4.7 shows that a similar algorithm for context-insensitive Java points-to analysis has better worst-case complexity than existing algorithms.

3.2 Context-Insensitive Formulation

Our first Java points-to analysis formulation is for context-insensitive points-to analysis with ahead-of-time call graph construction. The key distinguishing feature of this formulation is its use of balanced parentheses to model the semantics of Java heap accesses, *i.e.*, reads and writes of object fields. We first describe how, given a program P, a graph G representing pointer behavior in P is constructed (§3.2.1). We then formulate points-to analysis for P as a CFL-reachability problem over G (§3.2.2). Finally, we discuss our handling of Java arrays and reflection in §3.2.3.

3.2.1 Graph Representation

Our graph-reachability formulation rests on representing the pointer-manipulating statements in a program P with a graph G. Nodes in G represent variables and abstract locations, with one abstract location node per `new` statement in the program.[3] Within a procedure, edges represent four canonical assignment forms: (1) allocation statements `x = new T()`, (2) copy statements `x = y`, (3) heap reads `x = y.f`, and (4)

[3]Note that in spite of representing each `new` statement with one abstract location node, we are still able to formulate an analysis with a context-sensitive heap abstraction over this graph; see §3.3 for details.

Statement	Graph Edge(s)
`s: x = new T()`	$o_s \xrightarrow{\text{new}} x$
`x = y`	$y \xrightarrow{\text{assignglobal}} x$ if x or y is a global; $y \xrightarrow{\text{assign}} x$ otherwise
`x = y.f`	$y \xrightarrow{\text{getfield}[f]} x$
`x.f = y`	$y \xrightarrow{\text{putfield}[f]} x$
`s: x = m(`$a_1, a_2, \ldots, a_k$`)`	$a_i \xrightarrow{\text{param}[s]} f_{m,i}$ for $i \in [1..k]$; $ret_m \xrightarrow{\text{return}[s]} x$

Table 3.1: Canonical statements for Java points-to analysis, and the edge(s) for each statement in our graph representation.

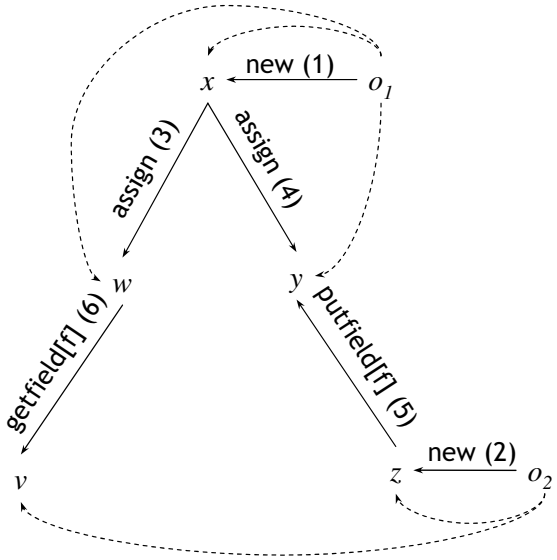
heap writes `x.f = y`. Table 3.1 shows the edges and edge labels used to represent these statement types; more complex statements can be modeled by suitable introduction of temporary variables.

For intraprocedural statements, the edges in our graph represent the value flow from the right-hand side to the left-hand side of the assignment. Hence, all edges in Table 3.1 are oriented in the direction of value flow. We use the assignglobal label for copy assignments where either `x` or `y` is a global variable (*i.e.*, a static field), and the assign label otherwise; distinguishing such assignments is important for context-sensitive analysis (see §3.3). The source of a new edge is the corresponding abstract location node. For getfield$[f]$ and putfield$[f]$ edges, the field name f is part of the edge label. These field access terminals will serve as the parentheses in the points-to analysis formulation of §3.2.2. Figure 3.1(b) gives G for the program of Figure 3.1(a).

We model method calls in our graph with specially labeled assignment edges, as shown by the final entry in Table 3.1. For each method m, G has nodes $f_{m,i}$ for m's formal parameters and a special ret_m node for m's **return** statements. At call site s of m, we add param$[s]$ edges from each actual parameter to the corresponding formal parameter and a return$[s]$ edge from the ret_m node to the appropriate caller's

```
1  x = new Obj(); // o1
2  z = new Obj(); // o2
3  w = x;
4  y = x;
5  y.f = z;
6  v = w.f;
```

(a) Code example



(b) Graph representation

Figure 3.1: A small code example and its graph representation for CFL-reachability-based points-to analysis. Line numbers from (a) are given on corresponding edges in (b). Dashed edges in (b) indicate the existence of a *flowsTo*-path from the source to the sink.

variable, as shown in Table 3.1. This is a standard technique for modeling method call semantics for points-to analyses. Note that in code examples, our convention is to identify call sites by their line number, so a call at line j will have $\mathsf{param}[j]$ and $\mathsf{return}[j]$ edges. For this section, we assume the existence of some pre-computed conservative call graph to determine possible targets of virtual calls. §3.4 shows how on-the-fly call graph construction can be formulated in CFL-reachability.

3.2.2 Analysis Grammar

We now define the language L_{F} used to compute context-insensitive points-to analysis with CFL-reachability. Recall from §3.1 that we want to define L_{F} such that x is L_{F}-reachable from o iff $o \in pt(x)$, *i.e.*, the address of o may flow to x. We first consider programs without field accesses, corresponding to graphs restricted to new and assign

edges. For such graphs, L_{F} is defined by the regular expression below (*flowsTo* is the start non-terminal):

$$\textit{flowsTo} \rightarrow \mathsf{new}\ (\ \mathsf{assign}\ )^*$$

Informally, an object can flow from an allocation site to a variable only through a new edge followed by a (possibly empty) sequence of assign statements. For example, in Figure 3.1(b), the path $o_1 \xrightarrow{\mathsf{new}} x \xrightarrow{\mathsf{assign}} w$ is a *flowsTo*-path witnessing $o_1 \in pt(w)$.

We now extend L_{F} to track value flow through the heap via $\mathsf{putfield}[f]$ and $\mathsf{getfield}[f]$ statements. We seek a precise handling of field accesses, defined as a *field-sensitive* handling in §2.2.2. Recall from §2.2.2 that field sensitivity requires that given a heap read r and write w, the analysis must check that that (1) r and w access the same field and (2) the base pointers of r and w are may aliased to establish a heap-based flow from w to r. To achieve this precision, we extend the *flowsTo* production:

$$\textit{flowsTo} \rightarrow \mathsf{new}\ \ (\ \mathsf{assign}\ \mid\ \mathsf{putfield}[f]\ \ \textit{alias}\ \ \mathsf{getfield}[f]\ )^*$$

This *flowsTo* production assumes the existence of an *alias* language (to be defined shortly) that captures may-aliasing, *i.e.*, the condition when two variables may point to the same object: $x\ \textit{alias}\ y \Leftrightarrow pt(x) \cap pt(y) \neq \emptyset$. The *alias* path connects the base variables of the field accesses, as desired by rule (2) of the field sensitivity definition.

Our remaining task is to define the *alias* language. Observe that we can check for aliasing of x and y by means of *flowsTo*-paths: x and y are may-aliased iff there is an object o such that o *flowsTo* x and o *flowsTo* y, *i.e.*, $o \in pt(x) \cap pt(y)$. Unfortunately, reasoning about may-aliasing in terms of two *flowsTo*-paths is unsuitable for CFL-reachability, which can only check language membership of a *single* path connecting

x and y. The two aforementioned *flowsTo*-paths cannot be concatenated to form a single path from x to y, since x and y are both sinks of the paths. Hence, we must extend our graph representation to allow for *inverse paths*, as in Melski's formulation of Andersen's analysis for C [Rep98].

Inverse paths enable *alias* paths by introducing reversed *flowsTo* paths (from variables to abstract locations), thereby allowing for a single path connecting two may-aliased variables. More concretely, the desired (x *alias* y)-path can be constructed by concatenating the inverse of the (o *flowsTo* x)-path with the (o *flowsTo* y)-path. We invert the *flowsTo*-path using *inverse edges*: for each edge $x \rightarrow y$ in G labeled t, we add an inverse edge $y \rightarrow x$ in G labeled with $\bar{\mathsf{t}}$, following the notation of [Rep98]. Given a path p, the inverse path $\overline{p}$ is then constructed using inverse edges in the obvious way. So, an (x *alias* y)-path can be now defined as a path $x\ \overline{\mathit{flowsTo}}\ o\ \mathit{flowsTo}\ y$, for some node o. The *alias* language is defined by the following grammar:

$$\begin{aligned} \mathit{alias} &\rightarrow \overline{\mathit{flowsTo}}\ \ \mathit{flowsTo} \\ \overline{\mathit{flowsTo}} &\rightarrow (\ \overline{\mathsf{assign}}\ \mid\ \overline{\mathsf{getfield}}[f]\ \ \mathit{alias}\ \ \overline{\mathsf{putfield}}[f]\ )^*\ \ \overline{\mathsf{new}} \end{aligned}$$

Note the absence of the $\overline{\mathit{alias}}$ non-terminal symbol; we use *alias* instead because the two generate the same language: $\overline{\mathit{alias}} \rightarrow \overline{\overline{\mathit{flowsTo}}\ \mathit{flowsTo}} = \overline{\mathit{flowsTo}}\ \overline{\overline{\mathit{flowsTo}}} = \overline{\mathit{flowsTo}}\ \mathit{flowsTo} = \mathit{alias}$.

Figure 3.2 gives the complete context-free grammar for L_{F}. Each $\overline{\mathit{flowsTo}}$ production reverses the preceding *flowsTo* production and inverts its edges. The *ciAssign* and $\overline{\mathit{ciAssign}}$ non-terminals treat edges corresponding to assignments to globals, parameter passing, and return values as if they were assignments between locals. This treatment yields a sound but context-insensitive analysis, as no filtering of unrealizable paths (defined in §2.2.4) is performed. (§3.3 gives a context-sensitive formulation that uses $\mathsf{param}[i]$ and $\mathsf{return}[i]$ edges to filter unrealizable paths.)

$$
\begin{array}{rl}
\mathit{flowsTo} & \rightarrow \mathsf{new} \\
\overline{\mathit{flowsTo}} & \rightarrow \overline{\mathsf{new}} \\
\mathit{flowsTo} & \rightarrow \mathit{flowsTo}\ \ \mathit{ciAssign} \\
\overline{\mathit{flowsTo}} & \rightarrow \overline{\mathit{ciAssign}}\ \ \overline{\mathit{flowsTo}} \\
\mathit{flowsTo} & \rightarrow \mathit{flowsTo}\ \ \mathsf{putfield}[f]\ \ \mathit{alias}\ \ \mathsf{getfield}[f] \\
\overline{\mathit{flowsTo}} & \rightarrow \overline{\mathsf{getfield}}[f]\ \ \mathit{alias}\ \ \overline{\mathsf{putfield}}[f]\ \ \overline{\mathit{flowsTo}} \\
\mathit{alias} & \rightarrow \overline{\mathit{flowsTo}}\ \ \mathit{flowsTo} \\
\mathit{ciAssign} & \rightarrow \mathsf{assign} \mid \mathsf{assignglobal} \mid \mathsf{param}[i] \mid \mathsf{return}[i] \\
\overline{\mathit{ciAssign}} & \rightarrow \overline{\mathsf{assign}} \mid \overline{\mathsf{assignglobal}} \mid \overline{\mathsf{param}}[i] \mid \overline{\mathsf{return}}[i] \\
\mathit{pointsTo} & \rightarrow \overline{\mathit{flowsTo}}
\end{array}
$$

Figure 3.2: A context-free grammar for L_{F}.

As discussed in §3.1, determining a points-to set for a variable x requires solving a backwards L_{F}-reachability problem from x, *i.e.*, finding those abstract locations that can flow to x. Figure 3.2 gives a production $\mathit{pointsTo} \rightarrow \overline{\mathit{flowsTo}}$ that makes this backwards-reachability correspondence explicit: $o \in pt(x)$ iff x $\mathit{pointsTo}$ o, *i.e.*, x $\overline{\mathit{flowsTo}}$ o. In the remainder of this thesis, we sometimes refer to $\mathit{pointsTo}$-paths rather than $\overline{\mathit{flowsTo}}$-paths to make the discussion more intuitive.

The balanced-parentheses structure of L_{F} will be exploited by our points-to analysis algorithms. Note that because of inverse edges, we have two pairs of matched parentheses for each field f, ($\mathsf{putfield}[f]$, $\mathsf{getfield}[f]$) and ($\overline{\mathsf{getfield}}[f]$, $\overline{\mathsf{putfield}}[f]$). We will use these parentheses to guide our approximation and refinement, as discussed in Chapter 4 and Chapter 5.

Example Let us derive a $\mathit{flowsTo}$-path from o_2 to v in Figure 3.1(b). First, we

derive y *alias* w using statements 1, 3, and 4.

$$\begin{aligned} & y\ \overline{\mathsf{assign}}\ x\ \overline{\mathsf{new}}\ o_1\ \mathsf{new}\ x\ \mathsf{assign}\ w \\ \rightarrow\ & y\ \overline{ciAssign}\ x\ \overline{\mathsf{new}}\ o_1\ \mathsf{new}\ x\ ciAssign\ w \\ \rightarrow\ & y\ \overline{flowsTo}\ o_1\ flowsTo\ w \\ \rightarrow\ & y\ alias\ w \end{aligned}$$

With this *alias* path, we can derive o_2 *flowsTo* v using statements 2, 5 and 6:

$$\begin{aligned} & o_2\ \mathsf{new}\ z\ \mathsf{putfield}[f]\ y\ alias\ w\ \mathsf{getfield}[f]\ v \\ \rightarrow\ & o_2\ flowsTo\ v \end{aligned}$$

In contrast, there is no *flowsTo*-path from o_2 to y, and hence $o_2 \notin pt(y)$. □

3.2.3 Other Java Language Features

Here, we briefly detail our handling of Java language constructs relevant to points-to analysis that were not discussed in §3.2.1.

Arrays Loads and stores to array elements are modeled by collapsing all array elements into a single element, modeled with a field *arr*. For example, `x.a[i]=y` is translated to `tmp=x.a; tmp.arr=y;`. For the programs and points-to analysis clients we have tested, a more faithful modeling of array indices would not significantly improve precision. However, the ability to refine handling of array indices may be useful for more demanding clients, especially if combined with greater flow sensitivity (in handling of both array indices and array contents). More precise modeling of arrays is a topic for future work.

Reflection and Native Methods Our formulation does not include handling

of reflection or native code, and hence it is unsound for programs using those constructs. Our implementation makes a best effort to conservatively model the most commonly used reflective constructs and native methods in the standard library, as in other work using Java points-to analysis [O'C00, LH03, FYD+06, NAW06]. Common techniques that we employ include modeling of native methods in the library and manual annotation where possible of which classes can be loaded reflectively at runtime. Nevertheless, the implementation can return unsound results if native methods or certain other reflective constructs are used in the application, like nearly all other Java points-to analyses.[4]

3.3 Context-Sensitive Formulation

Our second use of balanced parentheses in CFL-reachability will be for context sensitivity, defined in §2.2.4. In this section, we show how to add context sensitivity to the context-insensitive points-to analysis formulated in §3.2 as L_{F}-reachability. The key idea is to formulate a language L_{C} that filters unrealizable paths [Rep98], but does not precisely model other language features like fields. Then, a field-sensitive and context-sensitive points-to analysis can be formulated as reachability over the *intersection* of L_{F} and L_{C}. This specification strategy simplifies L_{C} greatly, as it need not model any program semantics beyond method calls and returns.

Note that computing $(L_{\mathsf{F}} \cap L_{\mathsf{C}})$-reachability has been proved undecidable [Rep00], a result we discuss in greater detail later in the section. Since fully field- and context-sensitive points-to analysis is undecidable, our algorithm computes an approximation of $(L_{\mathsf{F}} \cap L_{\mathsf{C}})$-reachability, as described in §5.2.

The formulation in this section both filters unrealizable paths *and* yields a context-

[4] The points-to analysis of Hirzel *et al.* [HDDH07] is the only sound Java points-to analysis we know of; it achieves soundness for a particular execution of a program by analyzing the behavior of reflection and native code at runtime.

sensitive heap abstraction (see §2.2.5). However, as in §3.2, the formulation in this section assumes an ahead-of-time call graph (see §2.2.3). In §3.4, we show how to adapt this formulation for on-the-fly call graph construction, thereby yielding a context-sensitive call graph (defined in §2.2.4).

Specifying L_C The language L_C only contains strings where letters representing entries and exits of method calls are appropriately balanced. Because of inverse paths (needed to specify may-aliasing; see §3.2.2), defining which terminals correspond to call entries and exits is slightly tricky. In the absence of inverse paths, a realizable *flowsTo*-path traverses a method in the direction of value flow, entering through a $\mathsf{param}[i]$ edge and exiting through a matching $\mathsf{return}[i]$ edge. However, when a realizable *flowsTo*-path p contains inverse $\overline{\textit{flowsTo}}$ sub-paths, p might enter a method through a $\overline{\mathsf{return}}[i]$ edge and/or exit through a $\overline{\mathsf{param}}[i]$ edge. Hence, we define call entries and exits through the following non-terminals $\textit{callEntry}[i]$ and $\textit{callExit}[i]$:

$$\begin{aligned} \textit{callEntry}[i] &\rightarrow \mathsf{param}[i] \mid \overline{\mathsf{return}}[i] \\ \textit{callExit}[i] &\rightarrow \mathsf{return}[i] \mid \overline{\mathsf{param}}[i] \end{aligned}$$

Unlike L_F, L_C includes some strings with *partially balanced parentheses*. In the context of interprocedural dataflow analysis, Reps defined *realizable paths* to include paths ending with unmatched call entry parentheses, as those paths correspond to control-flow paths where some method calls have not yet returned [Rep98]. For points-to analysis, call parentheses may be partially balanced on valid paths in our graph since, for example, values may flow from callee to caller or vice versa.[5] Consider the following simple example:

```
1 Foo makeFoo() { return new Foo(); }
2 ...
```

[5] In general, values may flow from ancestor methods in the call graph to descendant methods or vice versa.

$$
\begin{aligned}
\mathit{csStart} &\rightarrow \mathit{unbalExits}\ \ \mathit{unbalEntries} \\
\mathit{unbalExits} &\rightarrow \mathit{balanced}\ \ \mathit{unbalExits} \mid \mathit{callExit}[i]\ \ \mathit{unbalExits} \mid \epsilon \\
\mathit{unbalEntries} &\rightarrow \mathit{balanced}\ \ \mathit{unbalEntries} \mid \mathit{callEntry}[i]\ \ \mathit{unbalEntries} \mid \epsilon \\
\mathit{balanced} &\rightarrow \mathit{callEntry}[i]\ \ \mathit{balanced}\ \ \mathit{callExit}[i] \\
&\quad \mid \mathit{balanced}\ \ \mathit{balanced} \mid \mathit{nonCallTerm} \mid \epsilon \\
\mathit{callEntry}[i] &\rightarrow \mathsf{param}[i] \mid \overline{\mathsf{return}}[i] \\
\mathit{callExit}[i] &\rightarrow \mathsf{return}[i] \mid \overline{\mathsf{param}}[i] \\
\mathit{nonCallTerm} &\rightarrow \mathsf{new} \mid \overline{\mathsf{new}} \mid \mathsf{assign} \mid \overline{\mathsf{assign}} \mid \mathsf{assignglobal} \mid \overline{\mathsf{assignglobal}} \\
&\quad \mid \mathsf{getfield}[f] \mid \overline{\mathsf{getfield}}[f] \mid \mathsf{putfield}[f] \mid \overline{\mathsf{putfield}}[f]
\end{aligned}
$$

Figure 3.3: A context-free grammar for L_{C} that only includes strings corresponding to realizable paths, adapted from previous work [Rep98, FRD99, KA04].

```
3  x = makeFoo();
```

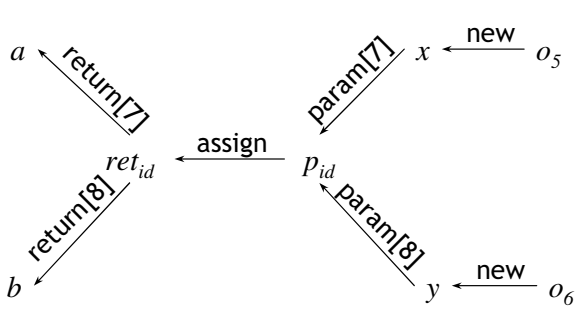
Here, an object allocated in `makeFoo()` can flow to `x`. The path in our graph from the abstract location to `x` is labeled "new return[1]", and hence is a valid flow path with an unmatched call exit edge. Similarly, a *flowsTo*-path whose sink is a formal parameter will end with an unmatched call entry edge. To avoid filtering out valid flows, L_{C} allows a prefix of unmatched call exit edges and a suffix of unmatched call entry edges, as in previous work [Rep98, FRD99, KA04].[6]

Figure 3.3 gives a complete grammar for L_{C}, adopted from previous work on context-sensitive analysis [Rep98, FRD99, KA04]. The *unbalExits* and *unbalEntries* non-terminals respectively allow for unmatched call exit and call entry edges, while the *balanced* non-terminal only allows balanced parentheses. Notice that since the *nonCallTerm* non-terminal reduces to any terminal not related to calls, those non-call terminals can appear anywhere in a L_{C} string, *i.e.*, they are essentially ignored. Hence, L_{C} solely filters out unrealizable paths, yielding context sensitivity as desired.

[6] As originally defined, realizable paths do not allow a prefix of unmatched call exit edges, though *slice paths* do allow such a prefix [Rep98]. In this work, we use the term "realizable paths" to refer to paths that may include both the unmatched prefix and the unmatched suffix.

```
1  static Object id(Object o) {
2    return o;
3  }
4  main() {
5    x = new Object();
6    y = new Object();
7    a = id(x);
8    b = id(y);
9  }
```

(a) Code example



(b) Graph representation

Figure 3.4: A small example program and graph to illustrate context-sensitive analysis.

Computing a context- and field-sensitive points-to analysis requires reachability over the language $L_{\mathsf{CF}} = L_{\mathsf{F}} \cap L_{\mathsf{C}}$.

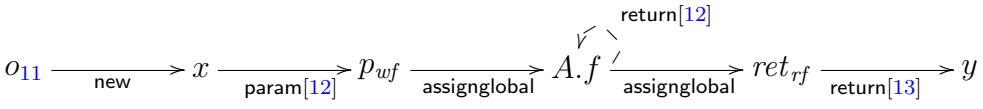
Example Here we give an example illustrating how L_{C}-reachability—and hence $(L_{\mathsf{F}} \cap L_{\mathsf{C}})$-reachability—filters out unrealizable paths. Figure 3.4 presents an example program that makes two calls to the identity function `id`, along with its graph representation. The graph contains a L_{C}-path from o_5 to a:

$$
\begin{aligned}
& o_5 \ \mathsf{new} \ x \ \mathsf{param}[7] \ p_{id} \ \mathsf{assign} \ ret_{id} \ \mathsf{return}[7] \ a \\
\rightarrow \ & o_5 \ \mathit{nonCallTerm} \ x \ \mathit{callEntry}[7] \ p_{id} \ \mathit{nonCallTerm} \ ret_{id} \ \mathit{callExit}[7] \ a \\
\rightarrow \ & o_5 \ \mathit{balanced} \ x \ \mathit{balanced} \ a \\
\rightarrow \ & o_5 \ \mathit{balanced} \ a
\end{aligned}
$$

Since this path is also an L_{F}-path, a is L_{CF}-reachable from o_5, *i.e.*, $o_5 \in pt(a)$. On the other hand, there is no L_{C}-path from o_5 to b, since the $x \xrightarrow{\mathsf{param}[7]} p_{id}$ edge is not matched by the $ret_{id} \xrightarrow{\mathsf{return}[8]} b$ edge, so $o_5 \notin pt(b)$. Hence, L_{CF}-reachability successfully keeps the effects of the two calls to `id` separate by filtering out unrealizable paths. □

```
1  class A {
2    static Obj f;
3  }
4  Obj rf() {
5    return A.f;
6  }
7  void wf(Obj p) {
8    A.f = p;
9  }
10 main() {
11   Obj x = new Obj();
12   wf(x);
13   Obj y = rf();
14 }
```

(a) Code example



(b) Graph representation

Figure 3.5: A small example to illustrate handling of globals in our context-sensitive formulation. The code is given in (a), and (b) shows a $(L_{\mathsf{F}} \cap L_{\mathsf{C}})$-path from o_{11} to y in the corresponding graph. The dashed return[12] self edge on $A.f$ added to ensure sound handling of the global.

Heap Abstraction By treating new and $\overline{\text{new}}$ edges identically to other intraprocedural edges, L_{C}-reachability yields a context-sensitive heap abstraction. We defined a context-sensitive heap abstraction in §2.2.5 as using a separate abstract location for each inlined copy of an allocation site, assuming that context sensitivity is achieved via exhaustive inlining. L_{C}-reachability tracks the unmatched *callEntry*$[i]$ edges on paths through abstract location nodes, just as it does for local variables. This tracking is equivalent to creating a copy of the abstract location particular to that sequence of call entries, yielding a context-sensitive heap abstraction.

Handling Globals Global variables (*i.e.*, Java static fields) require additional attention for context-sensitive analysis, since the graph edges representing global accesses can connect locals in different methods. As discussed in §3.2.1, we model accesses to globals in our graph with direct edges to a node representing the global, labeling the edges assignglobal and $\overline{\text{assignglobal}}$. These edges allow paths to connect local variables in distinct methods *without* including the call entry and exit edges showing how those methods were invoked. For example, consider the path in Figure 3.5(b) (temporarily ignoring the dashed edge) taken from the graph for the code in Figure 3.5(a). The path connects the `p` parameter of `wf()` to the return value of `rf()` with assignglobal edges to and from `A.f`, skipping edges that reflect the call exit from `wf()` and call entry to `rf()` necessary for the global accesses to execute.

Without modifying our graph representation, L_{C}-reachability would unsoundly filter some paths with assignglobal edges. Such unsound filtering can occur when a call entry edge entering method m precedes assignglobal edges on a path, but those assignglobal edges lead to a local in a distinct method m'. For example, consider again the path in Figure 3.5(b), ignoring the dashed edge. The $x \xrightarrow{\mathsf{param}[12]} p_{wf}$ edge enters `wf()`, while the subsequent assignglobal edges lead to method `rf()`. The param[12] edge should not be used for filtering after the assignglobal edges, as the path has exited the `wf()` method. However, as written L_{C} will unsoundly filter the path, treating the param[12] edge and the later return[12] edge as mismatched.

A simple technique to restore the soundness of L_{C}-reachability is to add self-edges on all global variable nodes labeled with all possible *callExit*[i] edges [KA04]. These edges will balance any *callEntry*[i] edges on paths leading to the global, essentially "consuming" the calling context accumulated on incoming paths. In Figure 3.5(b), the dashed $A.f \xrightarrow{\mathsf{return}[12]} A.f$ self-edge matches the previous param[12] edge, and hence L_{C} will not filter the path that includes the dashed edge, as desired. Our analysis implementation does not explicitly add these self edges to the graph, but its handling

of globals is equivalent, as described in Chapter 5.

Undecidability L_{CF} is not a context-free language, and hence L_{CF}-reachability is in fact not a CFL-reachability problem. In general, context-free languages are not closed under intersection. L_{CF} is not context free since the parentheses of L_{F} and L_{C} need not be properly nested on a valid graph path (unlike the parentheses of L_{F} and L_{OTF}, as discussed in §3.4.5=). As an example, consider a call `y = x.getFoo()`, where `getFoo()` is a "getter" method for a field `foo`. There is a valid partial L_{CF}-path from x to y labeled $\mathsf{param}[i]$ $\mathsf{getfield}[foo]$ $\mathsf{return}[i]$. On this path, the $\mathsf{param}[i]$ parenthesis is balanced before the later $\mathsf{getfield}[foo]$ parenthesis, showing non-nesting of call and field parentheses.

Reps proved that L_{CF}-reachability is in fact undecidable through a reduction of Post's correspondence problem [Rep00]. Hence, any terminating algorithm aiming to compute L_{CF}-reachability must approximate in some way. Our algorithm achieves decidability by using a regular language approximation of L_{C}, presented fully in Chapter 5.

3.4 On-The-Fly Call Graph Formulation

The analysis formulations in both §3.2 and §3.3 assumed that virtual call targets were determined via a pre-computed (*i.e.*, "ahead-of-time") call graph. Here, we formulate an analysis that computes virtual call targets *on-the-fly*, *i.e.*, by computing points-to sets for receivers of virtual calls during the analysis (previously discussed in §2.2.3). First, we formulate a context-insensitive version of the analysis as L_{OTF}-reachability over a modified graph representation of the program; L_{OTF} captures both field access semantics (like L_{F} of §3.2.2) and on-the-fly discovery of virtual call targets. Then, similar to §3.3, we formulate a context-sensitive variant of the analysis as reachability over the language $L_{\mathsf{OTF}} \cap L_{\mathsf{C}'}$, where $L_{\mathsf{C}'}$ is a slightly modified version of L_{C} from §3.3.

The section is organized as follows. First, §3.4.1 provides the intuition behind our formulation by relating L_{OTF} sub-paths for discovering virtual call targets to the dynamic semantics of virtual calls. Then, §3.4.2 presents the necessary modifications to our graph representation for on-the-fly call graph reasoning. §3.4.3 describes the context-free grammar of L_{OTF}, and §3.4.4 shows how context sensitivity can be added through language intersection. Finally, §3.4.5 further discusses some aspects of our formulation and how it relates to other formulations in the literature.

3.4.1 Intuition

Our CFL-reachability formulation of on-the-fly call graph construction is best understood as an abstraction of the dynamic semantics of virtual calls. Executing a virtual call "`r.m(x)`" requires the following three key steps:

1. Determining the object o that `r` points to.
2. Determining the concrete type `T` of o.
3. Determining the method `C.m()` that should be invoked when the receiver object is of type `T`, based on the class hierarchy.

Our formulation determines possible value flow at virtual calls by abstractly computing the above steps through CFL-reachability: step 1 is computed by finding *pointsTo*-paths from the receiver, and steps 2 and 3 are computed by encoding concrete types and virtual method dispatch tables in our graph representation.

Our on-the-fly call graph formulation replaces direct $\mathsf{param}[i]$ and $\mathsf{return}[i]$ edges at virtual call sites with *virtParam*$[i]$ and *virtReturn*$[i]$ *paths*, whose discovery requires on-the-fly computation of virtual call targets. With an ahead-of-time call graph CG, parameters and return values at virtual call sites are connected with direct $\mathsf{param}[i]$ and $\mathsf{return}[i]$ edges added according to CG (see §3.2.1). For on-the-fly call

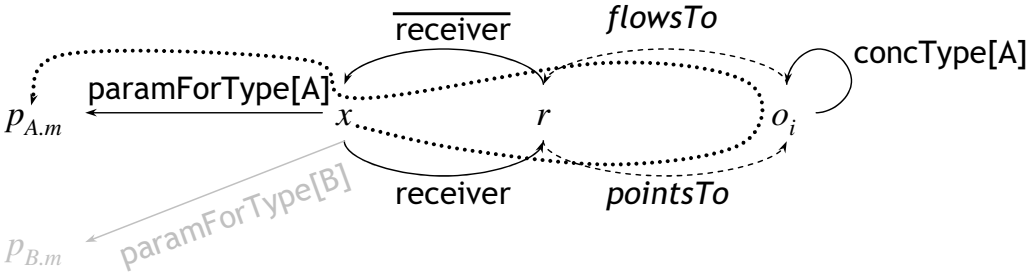


Figure 3.6: An abstracted view of a path in our graph for computing virtual call targets on-the-fly (some edge parameters have been removed). The path is for some virtual call "`r.m(x)`," and it connects actual parameter x to formal parameter $p_{A.m}$ by going through o_i and back, as indicated by the dotted edge. Dashed edges indicate sub-paths, and the gray edge is filtered out by our context-free language.

graph construction, an $x \xrightarrow{\mathsf{param}[i]} p_{A.m}$ edge for a virtual call `r.m(x)` is replaced by a *virtParam*$[i]$ path, shown abstractly in Figure 3.6. This *virtParam*$[i]$ path has several sub-paths (*virtReturn*$[i]$ paths are similarly constructed):

- First, a receiver edge connects x to the receiver r of the virtual call.
- Then, a *pointsTo*-path connects r to some abstract location o_i it may point to. This *pointsTo*-path corresponds to abstractly computing step 1 of virtual call execution, *i.e.*, determining what the receiver points to. Computing this information using the points-to analysis itself indicates on-the-fly call graph construction.
- Next, a concType$[A]$ *edge* indicates that o_i has concrete type `A`, corresponding to step 2 of virtual call execution (determining the concrete type of the receiver).
- A *flowsTo*-path and a $\overline{\mathsf{receiver}}$ edge connect the path back to the actual parameter `x`.
- Finally, a paramForType$[A]$ edge connects x to $p_{A.m}$, indicating that when the receiver of the call is of type `A`, x is copied to $p_{A.m}$, the formal parameter of

Statement	Graph Edge(s)
`s: x = new T()`	$o_s \xrightarrow{\text{new}} x$ $o_s \xrightarrow{\text{concType}[T]} o_s$
`s: x =` a_1`.m(`$a_2, a_3, \ldots, a_k$`)`	$a_i \xrightarrow{\text{receiver}[i][s]} a_1$ for $i \in [1..k]$ $x \xrightarrow{\text{receiver}[ret][s]} a_1$ for each possible concrete type `T` of a_1, and for each instance method `C.m()` of `T`: $a_i \xrightarrow{\text{paramForType}[T][s]} f_{C.m,i}$ $ret_{C.m} \xrightarrow{\text{returnForType}[T][s]} x$

Table 3.2: The changes in graph representation required for on-the-fly call graph construction. Remaining assignments are modeled as in Table 3.1.

`A.m()`. Note that the A type in the paramForType$[A]$ and concType$[A]$ edge labels must be matched. This final edge corresponds to step 3 of virtual call execution (determining which method is invoked for the concrete receiver type).

The paramForType$[B]$ edge from x to $p_{A.m}$ in Figure 3.6 is filtered from consideration, as there is no matching concType$[B]$ edge indicating the receiver can point to a `B` object. This filtering shows the potentially increased precision from on-the-fly call graph construction, as a less precise ahead-of-time call graph may have caused the inclusion of param$[i]$ edges from x to both $p_{A.m}$ and $p_{B.m}$.

Several details were abstracted from the edge labels in Figure 3.6: further edge label parameters and matching are required to handle multi-parameter functions and multiple calls to the same function precisely. We flesh out these remaining details in §3.4.2 and §3.4.3.

3.4.2 Graph Representation

To facilitate on-the-fly call graph construction, we add two new types of edges to our graph representation:

- A receiver$[i][s]$ edge connects the i^{th} actual parameter at virtual call site s to the receiver argument at the call site. Similarly, a receiver$[ret][s]$ edge connects the caller variable holding the return value to the receiver. The i or ret argument is used to ensure that the receiver edge source and the $\overline{\text{receiver}}$ edge sink on a $virtParam[s]$ path (see Figure 3.6) are the same actual parameter or return value. The call site s is used to keep targets for distinct virtual calls separate, described in more detail in §3.4.3.

- A concType$[T]$ edge connects an abstract location of concrete type T to itself. This concrete type information for abstract locations is needed to determine virtual call targets, as discussed in §3.4.1.

In addition, we modify the param$[s]$ and return$[s]$ edges previously used to connect caller and callee at virtual calls (see §3.2.1) to be paramForType$[T][s]$ and returnForType$[T][s]$ edges. A paramForType$[T][s]$ edge connects an actual parameter at call site s to the corresponding formal parameter in method `C.m()`, such that `C.m()` is the method invoked at s when the receiver has concrete type T. (The meaning of returnForType$[T][s]$ edges is similar.) As discussed in §3.4.1, the type T on a paramForType$[T][s]$ or returnForType$[T][s]$ edge is matched with a previous concType$[T]$ edge on a $virtParam[s]$ path to ensure feasibility of the corresponding virtual call targets. As with param$[s]$ and return$[s]$ edges, the call site s is used to facilitate context sensitivity, discussed further in §3.4.4. Note that param$[s]$ and return$[s]$ edges are still used in the graph for non-virtual calls, *i.e.*, calls to static methods, constructors, and private methods. Table 3.2 gives a complete description of the changes to our graph representation for on-the-fly call graph construction.

Since we aim to compute a program's call graph on-the-fly, it is not immediately clear how paramForType$[T][s]$ and returnForType$[T][s]$ edges can be added to the representation, since their addition requires some knowledge of possible targets for

virtual calls. For our demand-driven analyses, we add these edges using a separate, pre-computed call graph. We discuss this choice further in §3.4.5.

Figure 3.7 gives a small code example and a relevant subset of our modified graph representation for the code. The graph in Figure 3.7(b) changes the param edges of §3.2.1 to paramForType edges, additionally parametrized with either `A` or `B` depending on the containing method of the sink node. The receiver and concType edges are also added, with appropriate parameters. In §3.4.3, we give a grammar that shows how the edge label parameters are matched against each other on valid L_{OTF} paths.

3.4.3 The L_{OTF} Language

Here we present the full L_{OTF} language for on-the-fly call graph construction via CFL-reachability. First, we present the production for $\mathit{virtParam}[s]$ paths in detail, fleshing out the abstract example given in Figure 3.6. Once $\mathit{virtParam}[s]$ paths are understood, reasoning about the rest of the L_{OTF} grammar becomes more straightforward. We then discuss the meaning of different edge label matchings that must occur on $\mathit{virtParam}[s]$ paths, analogous to the matching of fields on $\mathsf{getfield}[f]$ and $\mathsf{putfield}[f]$ edges for L_{F} (see §3.2) or of call sites on $\mathsf{param}[i]$ and $\mathsf{return}[i]$ edges for L_{C} (see §3.3). Finally, we present the full grammar for L_{OTF}.

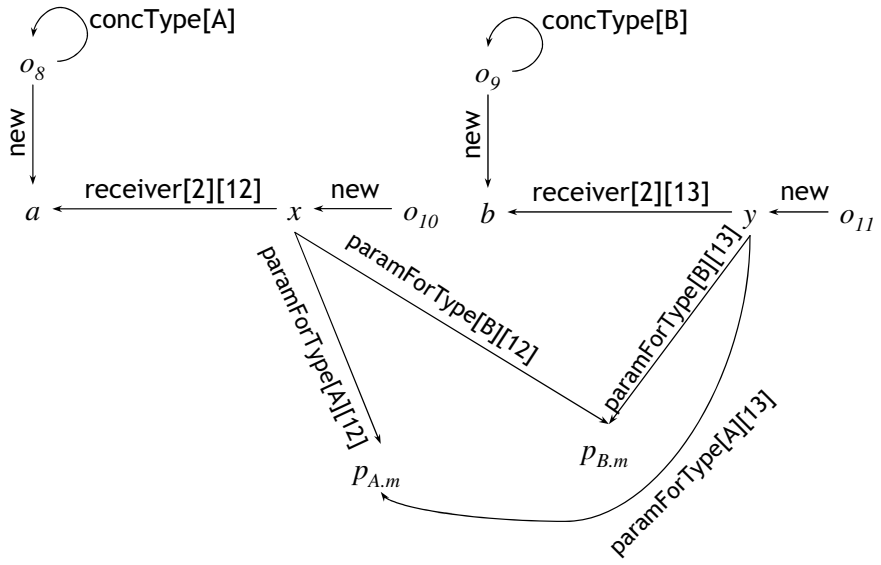
The virtParam[s] production Following the example of Figure 3.6, one could write the following production for $\mathit{virtParam}[s]$:

$$\begin{aligned}\mathit{virtParam}[s] \rightarrow\ & \mathsf{receiver}[i][s]\ \ \mathit{pointsTo}\ \ \mathsf{concType}[T]\ \ \mathit{flowsTo} \\ & \overline{\mathsf{receiver}}[i][s]\ \ \mathsf{paramForType}[T][s] \\ |\ & \mathsf{param}[s]\end{aligned}$$

The $\mathsf{param}[s]$ case of the production handles non-virtual calls, which are still rep-

```
1  class A {
2    void m(Object p) { ... }
3  }
4  class B extends A {
5    void m(Object p) { ... }
6  }
7  main() {
8    A a = new A();
9    A b = new B();
10   Object x = new Object();
11   Object y = new Object();
12   a.m(x);
13   b.m(y);
14 }
```

(a) Code example



(b) Graph representation

Figure 3.7: A small example illustrating our graph representation for on-the-fly call graph construction. The graph in (b) contains a relevant subset of the edges for representing the program in (a).

resented with $\mathsf{param}[s]$ and $\mathsf{return}[s]$ edges. The other case corresponds directly to the abstract $virtParam[s]$ path shown in Figure 3.6, except that the extra edge label parameters introduced in §3.4.2 have been added.

To simplify the $virtParam[s]$ production, we introduce a $dispatch[i][s]$ non-terminal and re-write the production as follows:

$$virtParam[s] \rightarrow \mathsf{receiver}[i][s] \;\; pointsTo \;\; dispatch[i][s] \;\mid\; \mathsf{param}[s]$$
$$dispatch[i][s] \rightarrow \mathsf{concType}[T] \;\; flowsTo \;\; \overline{\mathsf{receiver}}[i][s] \;\; \mathsf{paramForType}[T][s]$$

A $dispatch[i][s]$ path includes the sub-path of the $virtParam[s]$ path from the abstract location o that the receiver may point to to the path's end at some formal parameter f. A dispatch path models the value flow that occurs at a virtual call site when dispatched for a particular receiver object. More precisely, a $dispatch[i][s]$ path indicates which method's formal parameter f gets the value of the actual parameter in position i at call site s when o is the receiver object at s.

Meaning of matchings Several edge label parameters must be matched on a $virtParam[s]$ path, ensuring precise modeling of corresponding aspects of the program semantics. Here, we discuss the meaning of each of the necessary matchings in turn. Given the $virtParam[s]$ production,

$$virtParam[s] \rightarrow \mathsf{receiver}[i][s] \;\; pointsTo \;\; dispatch[i][s]$$

there are two edge label parameters that must be matched:

- **The i in $\mathsf{receiver}[i][s]$ and $dispatch[i][s]$:** This matching ensures that flows between distinct method parameters are kept separate. Without the matching, given a virtual call "`r.m(x,y)`" and target "`A.m(p,q)`", a $virtParam[s]$ path could connect x to q or y to p, a clear violation of program semantics.

- **The s in $\mathsf{receiver}[i][s]$ and $\mathit{dispatch}[i][s]$:** This matching filters out unrealizable paths between two invocations of the same virtual method. Without the matching, given call site s_1 "`q.m(x)`" and site s_2 "`r.m(y)`", a $\mathit{virtParam}[s_1]$ path may begin at actual parameter y from site s_2, indicating that that y may be passed at s_1, an unrealizable result.

Within the $\mathit{dispatch}[i][s]$ production,

$$\mathit{dispatch}[i][s] \rightarrow \mathsf{concType}[T] \;\; \mathit{flowsTo} \;\; \overline{\mathsf{receiver}}[i][s] \;\; \mathsf{paramForType}[T][s]$$

we have two more required matchings in the edge labels:

- **The T in $\mathsf{concType}[T]$ and $\mathsf{paramForType}[T][s]$:** As discussed in §3.4.1, this matching filters out virtual call targets that do not correspond to some abstract location in the receiver's points-to set.

- **The s in $\overline{\mathsf{receiver}}[i][s]$ and $\mathsf{paramForType}[T][s]$:** This matching ensures that virtual call targets are computed separately in the case where two different virtual calls pass the same parameter. Consider the following example program (assume classes `A` and `B` are as in Figure 3.7(a)):

```
x = new A(); //  o1
z = new B(); //  o2
x.m(y); //  s1
z.m(y); //  s2
```

Note that y is passed at both call sites s_1 and s_2. Without this matching, the following $\mathit{dispatch}[2][s_2]$ path could arise:

$$o_1 \;\; \mathsf{concType}[A] \;\; o_1 \;\; \mathit{flowsTo} \;\; x \;\; \overline{\mathsf{receiver}}[2][s_1] \;\; y \;\; \mathsf{paramForType}[A][s_2] \;\; p_{A.m}$$

$$
\begin{aligned}
\mathit{flowsTo} &\rightarrow \mathsf{new} \\
\overline{\mathit{flowsTo}} &\rightarrow \overline{\mathsf{new}} \\
\mathit{flowsTo} &\rightarrow \mathit{flowsTo}\ \ \mathit{ciAssign} \\
\overline{\mathit{flowsTo}} &\rightarrow \overline{\mathit{ciAssign}}\ \ \overline{\mathit{flowsTo}} \\
\mathit{flowsTo} &\rightarrow \mathit{flowsTo}\ \ \mathsf{putfield}[f]\ \ \mathit{alias}\ \ \mathsf{getfield}[f] \\
\overline{\mathit{flowsTo}} &\rightarrow \overline{\mathsf{getfield}}[f]\ \ \mathit{alias}\ \ \overline{\mathsf{putfield}}[f]\ \ \overline{\mathit{flowsTo}} \\
\mathit{alias} &\rightarrow \overline{\mathit{flowsTo}}\ \ \mathit{flowsTo} \\
\mathit{ciAssign} &\rightarrow \mathsf{assign} \mid \mathsf{assignglobal} \mid \mathit{virtParam}[s] \mid \mathit{virtReturn}[s] \\
\overline{\mathit{ciAssign}} &\rightarrow \overline{\mathsf{assign}} \mid \overline{\mathsf{assignglobal}} \mid \overline{\mathit{virtParam}}[s] \mid \overline{\mathit{virtReturn}}[s] \\
\mathit{pointsTo} &\rightarrow \overline{\mathit{flowsTo}} \\
\mathit{virtParam}[s] &\rightarrow \mathsf{receiver}[i][s]\ \ \mathit{pointsTo}\ \ \mathit{dispatch}[i][s] \mid \mathsf{param}[s] \\
\overline{\mathit{virtParam}}[s] &\rightarrow \overline{\mathit{dispatch}}[i][s]\ \ \mathit{flowsTo}\ \ \overline{\mathsf{receiver}}[i][s] \mid \overline{\mathsf{param}}[s] \\
\mathit{virtReturn}[s] &\rightarrow \mathit{dispatch}[\mathit{ret}][s]\ \ \mathit{flowsTo}\ \ \overline{\mathsf{receiver}}[\mathit{ret}][s] \mid \mathsf{return}[s] \\
\overline{\mathit{virtReturn}}[s] &\rightarrow \mathsf{receiver}[\mathit{ret}][s]\ \ \mathit{pointsTo}\ \ \overline{\mathit{dispatch}}[\mathit{ret}][s] \mid \overline{\mathsf{return}}[s] \\
\mathit{dispatch}[i][s] &\rightarrow \mathsf{concType}[T]\ \ \mathit{flowsTo}\ \ \overline{\mathsf{receiver}}[i][s]\ \ \mathsf{paramForType}[T][s] \\
\overline{\mathit{dispatch}}[i][s] &\rightarrow \overline{\mathsf{paramForType}}[T][s]\ \ \mathsf{receiver}[i][s]\ \ \mathit{pointsTo}\ \ \mathsf{concType}[T] \\
\mathit{dispatch}[\mathit{ret}][s] &\rightarrow \mathsf{returnForType}[T][s]\ \ \mathsf{receiver}[\mathit{ret}][s]\ \ \mathit{pointsTo}\ \ \mathsf{concType}[T] \\
\overline{\mathit{dispatch}}[\mathit{ret}][s] &\rightarrow \mathsf{concType}[T]\ \ \mathit{flowsTo}\ \ \overline{\mathsf{receiver}}[\mathit{ret}][s]\ \ \overline{\mathsf{returnForType}}[T][s]
\end{aligned}
$$

Figure 3.8: A context-free grammar for L_{OTF}, an extension of L_{F} that includes on-the-fly call graph construction. The rules for *flowsTo*, $\overline{\textit{flowsTo}}$, *pointsTo*, and *alias* are unchanged from those in Figure 3.2.

> Through o_1 flows to the receiver at call site s_1, the path ends with the paramForType$[A][s_2]$ edge, imprecisely indicating that "`A.m()`" can be invoked at call site s_2.

In addition to the matchings enforced on $\mathit{virtParam}[s]$ paths, the call site parameters on paramForType$[T][s]$ and returnForType$[T][s]$ edges can be matched for context sensitivity, as we shall discuss in §3.4.4.

The L_{OTF} grammar Figure 3.8 presents a context-free grammar for L_{OTF}. Much of the grammar is identical to that of L_{F} (see Figure 3.2), as L_{OTF} includes precise handling of field accesses. The key difference with L_{F} is that the $\mathsf{param}[s]$ and $\mathsf{return}[s]$ cases of the *ciAssign* production have been replaced with $\mathit{virtParam}[s]$ and $\mathit{virtReturn}[s]$ (and similarly for the barred equivalents).

The new productions in the L_{OTF} grammar include the already-discussed $\mathit{virtParam}[s]$ production and similar productions for $\mathit{virtReturn}[s]$, $\overline{\mathit{virtParam}}[s]$, and $\overline{\mathit{virtReturn}}[s]$. The construction of the latter paths and the reasoning behind their matchings can be understood analogously to $\mathit{virtParam}[s]$ paths.

Example Here we show a derivation of a *flowsTo* path in L_{OTF} for the example of Figure 3.7 and illustrate some of the filtering done by the language. In particular, we will show how there is an o_{10} *flowsTo* $p_{A.m}$ path in Figure 3.7(a), but there is no o_{10} *flowsTo* $p_{B.m}$ path, showing how L_{OTF}-reachability can filter out consideration of virtual call targets inconsistent with the receiver points-to set.

The graph in Figure 3.7(b) contains a $\mathit{virtParam}[12]$ path from x to $p_{A.m}$, showing that `A.m()` is a feasible target of the virtual call at line 12 of Figure 3.7(a) according to the on-the-fly call graph. To derive this path, we first derive a $\mathit{dispatch}[2][12]$ path from o_8 to $p_{A.m}$:

$$
\begin{aligned}
& o_8\ \mathsf{concType}[A]\ o_8\ \mathsf{new}\ a\ \overline{\mathsf{receiver}}[2][12]\ x\ \mathsf{paramForType}[A][12]\ p_{A.m} \\
\rightarrow\ & o_8\ \mathsf{concType}[A]\ o_8\ \mathit{flowsTo}\ a\ \overline{\mathsf{receiver}}[2][12]\ x\ \mathsf{paramForType}[A][12]\ p_{A.m} \\
\rightarrow\ & o_8\ \mathit{dispatch}[2][12]\ p_{A.m}
\end{aligned}
$$

This $\mathit{dispatch}[2][12]$ path shows that the receiver a can point to o_8, an object of type `A`, in which case $p_{A.m}$ will get the value of x at the call. Given this path, deriving the

virtParam[12] path is straightforward:

$$\begin{aligned} & x \ \mathsf{receiver}[2][12] \ a \ \overline{\mathsf{new}} \ o_8 \ \mathit{dispatch}[2][12] \ p_{A.m} \\ \rightarrow \ & x \ \mathsf{receiver}[2][12] \ a \ \mathit{pointsTo} \ o_8 \ \mathit{dispatch}[2][12] \ p_{A.m} \\ \rightarrow \ & x \ \mathit{virtParam}[12] \ p_{A.m} \end{aligned}$$

Once we have the *virtParam*[12] path, we can easily derive the desired *flowsTo*-path from o_{10} to $p_{A.m}$:

$$\begin{aligned} & o_{10} \ \mathsf{new} \ x \ \mathit{virtParam}[12] \ p_{A.m} \\ \rightarrow \ & o_{10} \ \mathit{flowsTo} \ x \ \mathit{ciAssign} \ p_{A.m} \\ \rightarrow \ & o_{10} \ \mathit{flowsTo} \ p_{A.m} \end{aligned}$$

Note that we cannot derive a o_{10} *flowsTo* $p_{B.m}$ path (*i.e.*, $o_{10} \notin pt(p_{B.m})$), since on-the-fly call graph construction shows that `B.m()` cannot be called at line 12 of Figure 3.7(a). With L_{OTF}-reachability, deriving an o_8 *dispatch*[2][12] $p_{B.m}$ path fails, as the $o_8 \xrightarrow{\mathsf{concType}[A]} o_8$ and $x \xrightarrow{\mathsf{paramForType}[B][12]} p_{B.m}$ edge labels are mismatched. Since the receiver a can only point to o_8, there is no way to derive an x *virtParam*[12] $p_{B.m}$ path, *i.e.*, `B.m()` cannot be a target of the virtual call at line 12. □

3.4.4 Adding Context Sensitivity

Here, we show how to combine context sensitivity with on-the-fly call graph discovery, yielding a context-sensitive call graph (defined in §2.2.4). Formulating this combination requires minor modifications to the L_{C} language of §3.3.

We formulate field-sensitive, context-sensitive points-to analysis with on-the-fly call graph construction as $(L_{\mathsf{OTF}} \cap L_{\mathsf{C}'})$-reachability, with the grammar for $L_{\mathsf{C}'}$ ap-

$$
\begin{aligned}
\mathit{csStart} &\rightarrow \mathit{unbalExits}\ \ \mathit{unbalEntries} \\
\mathit{unbalExits} &\rightarrow \mathit{balanced}\ \ \mathit{unbalExits} \mid \mathit{callExit}[i]\ \ \mathit{unbalExits} \mid \epsilon \\
\mathit{unbalEntries} &\rightarrow \mathit{balanced}\ \ \mathit{unbalEntries} \mid \mathit{callEntry}[i]\ \ \mathit{unbalEntries} \mid \epsilon \\
\mathit{balanced} &\rightarrow \mathit{callEntry}[i]\ \ \mathit{balanced}\ \ \mathit{callExit}[i] \\
&\quad \mid \mathit{balanced}\ \ \mathit{balanced} \mid \mathit{nonCallTerm} \mid \epsilon \\
\mathit{callEntry}[i] &\rightarrow \mathsf{param}[i] \mid \mathsf{paramForType}[T][i] \mid \overline{\mathsf{return}}[i] \mid \overline{\mathsf{returnForType}}[T][i] \\
\mathit{callExit}[i] &\rightarrow \mathsf{return}[i] \mid \mathsf{returnForType}[T][i] \mid \overline{\mathsf{param}}[i] \mid \overline{\mathsf{paramForType}}[T][i] \\
\mathit{nonCallTerm} &\rightarrow \mathsf{new} \mid \overline{\mathsf{new}} \mid \mathsf{assign} \mid \overline{\mathsf{assign}} \mid \mathsf{assignglobal} \mid \overline{\mathsf{assignglobal}} \\
&\quad \mid \mathsf{getfield}[f] \mid \overline{\mathsf{getfield}}[f] \mid \mathsf{putfield}[f] \mid \overline{\mathsf{putfield}}[f] \\
&\quad \mid \mathsf{receiver}[i][s] \mid \mathsf{receiver}[\mathit{ret}][s] \\
&\quad \mid \overline{\mathsf{receiver}}[i][s] \mid \overline{\mathsf{receiver}}[\mathit{ret}][s] \mid \mathsf{concType}[T]
\end{aligned}
$$

Figure 3.9: A grammar for $L_{\mathsf{C}'}$, a modification of L_{C} that works on the representation of Table 3.2. The productions for *csStart*, *unbalExits*, *unbalEntries*, and *balanced* are unmodified from those for L_{C} in Figure 3.3.

pearing in Figure 3.9. Compared to L_{C}, $L_{\mathsf{C}'}$ has two key changes:

- The *callEntry*$[i]$ and *callExit*$[i]$ productions have been extended to include $\mathsf{paramForType}[T][i]$ and $\mathsf{returnForType}[T][i]$ edges (and their barred equivalents). This extension handles the interprocedural edges for virtual calls introduced in §3.4.2 by treating them like $\mathsf{param}[i]$ and $\mathsf{return}[i]$ edges, *i.e.*, using their call site annotations to filter out unrealizable paths. Note that the type T annotations on $\mathsf{paramForType}[T][i]$ and $\mathsf{returnForType}[T][i]$ edges are ignored, as $L_{\mathsf{C}'}$ only filters unrealizable paths and does not reason about virtual call targets.

- The *nonCallTerm* production now includes all $\mathsf{receiver}$ and $\overline{\mathsf{receiver}}$ labels and $\mathsf{concType}[T]$ labels. As these edges are intraprocedural, they should be ignored like any other intraprocedural edge by $L_{\mathsf{C}'}$.

As in the case with context- and field-sensitive analysis with ahead-of-time call graphs (see §3.3), $(L_{\mathsf{OTF}} \cap L_{\mathsf{C}'})$-reachability is undecidable, requiring approximation in the

implementation (described in Chapter 5).

3.4.5 Discussion

Here, we discuss two more aspects of our formulation of on-the-fly call graph construction. We first show that L_{OTF}-reachability is in fact a type of balanced parentheses problem. Then, we discuss issues related to how our graph representation is constructed.

Balanced parentheses L_{OTF}-reachability can be viewed as a type of balanced parentheses problem, just like L_{F}-reachability (§3.2) and L_{C}-reachability (§3.3). The "parentheses" in L_{OTF} would be $\mathsf{receiver}[i][s]$ edges and $dispatch[i][s]$ paths (and their barred equivalents), which must be fully balanced on $virtParam[s]$ and $virtReturn[s]$ paths as shown in Figure 3.8. Having paths serve as parentheses makes L_{OTF}-reachability slightly different from the other formulations, where only terminal edges served as parentheses. However, this difference is not fundamental: a pre-processing step could discover all $dispatch[i][s]$ paths in a graph and mark them with $dispatch[i][s]$ edges, after which any technique applicable to balanced-parentheses problems could also be applied to L_{OTF}-reachability.

Interestingly, the $\mathsf{receiver}[i][s]$ and $dispatch[i][s]$ parentheses of L_{OTF} can be viewed as reads and writes of fields in an object's dispatch table. Suppose that Java supported function pointers invoked with a C-like syntax, *i.e.*, "`(*func_ptr)(...)`." Given a class `A` with an implementation of virtual method `m()`, one could translate the code "`x = new A(); ...; x.m();`" into the following equivalent code with function pointers:[7]

```
x = new A(); x.m_func = A.m; ...; (*x.m_func)();
```

[7]Standard implementations of virtual calls are similar to this translation except that a single dispatch table per class is used, with a pointer in each object to the appropriate dispatch table.

This translation shows how $\mathsf{receiver}[i][s]$ and $\mathit{dispatch}[i][s]$ parentheses correspond to field accesses: the $\mathsf{receiver}[i][s]$ parenthesis corresponds to reading the appropriate function field from the receiver, and the $\mathit{dispatch}[i][s]$ parenthesis corresponds to the initial write of the field to point to the appropriate method.

This analogy between virtual method calls and field accesses provides the intuition behind the fact that field access and virtual call parentheses are *properly nested* in L_{OTF}. By properly nested, we mean that unmatched field access parentheses cannot appear between balanced virtual call parentheses and vice versa. For example, a path with labels '$\overline{\mathsf{getfield}}[f]$ $\mathsf{receiver}[ret][s]$ $\overline{\mathsf{putfield}}[f]$' cannot be part of a *flowsTo*-path, since the field access parenthesis cannot be matched before the virtual call parenthesis. This proper nesting is sensible since the virtual call parentheses can be viewed as field accesses, and hence they should of course be nested with other field accesses. Proper nesting of these parentheses allows for performing field-sensitive analysis with on-the-fly call graph construction via reachability over a single context-free language L_{OTF}. In contrast, method call parentheses for context sensitivity are *not* always properly nested with field access parentheses, making simultaneous precise treatment of method calls and field accesses undecidable, as discussed in §3.3.

Graph construction In §3.4.2, we briefly raised the issue of how $\mathsf{paramForType}[T][s]$ and $\mathsf{returnForType}[T][s]$ edges should be added to the graph representation, given that we aim to compute a call graph for the program on-the-fly. There are two possible ways to add these edges to the graph:

- **Using a pre-computed call graph:** One solution is to use some pre-computed call graph to add the edges. L_{OTF}-reachability will ignore edges from the pre-computed call graph if the on-the-fly call graph shows the edges to be infeasible.

- **Simultaneous reachability computation:** The second solution is to compute what code is reachable during on-the-fly call graph construction, adding

```
1  static void doSomething(Object o) { ... }
2  class A {
3    void foo() { // reachable
4      doSomething(new Integer());
5    }
6  }
7  class B {
8    void foo() { // unreachable
9      doSomething(new String());
10   }
11 }
```

Figure 3.10: A small example to illustrate possible reduced precision with demand-driven analysis and on-the-fly call graph handling.

relevant edges for reachable code as it is discovered. This technique begins by analyzing only the entrypoints of a program (typically `main()`, the static initializer for the main class, and other known entrypoints in the standard library). Then, whenever the analysis discovers a method `m()` to be reachable (*e.g.*, if it is called from `main()`), edges for `m()` and calls to `m()` are added, and then analysis continues.

There are several advantages to using simultaneous reachability computation to add graph edges instead of a pre-computed call graph. The on-the-fly call graph is likely to have fewer methods than the pre-computed call graph, *i.e.*, less code is treated as reachable. Apart from the obvious space and time benefits of analyzing fewer methods, treating less code as reachable can possibly improve the precision of analysis results.[8] The reason for this benefit is that the behavior of unreachable code may affect points-to results for reachable code, and hence completely ignoring the unreachable code can improve precision.

As an example of the potential negative effects of unreachable code on points-to

[8]We thank Barbara Ryder for discussions that led to this insight.

analysis precision, consider the example in Figure 3.10. Say that a pre-computed call graph determines that both `A.foo()` and `B.foo()` are reachable, but that simultaneous reachability computation with on-the-fly call graph construction determines that `B.foo()` is not reachable. If the points-to analysis graph representation is constructed with the pre-computed call graph, the analysis will conclude that $pt(o) = \{o_4, o_9\}$, since both `A.foo()` and `B.foo()` are assumed to be reachable. In contrast, simultaneous reachability computation will yield the result $pt(o) = \{o_4\}$, since the behavior of code in the unreachable method `B.foo()` is never considered. The precision improvement with simultaneous reachability computation has been seen in other analyses, *e.g.*, sparse conditional constant propagation [WZ91].

Though our demand-driven points-to analysis uses a pre-computed call graph to construct its graph representation, we have not observed the above precision for the benchmarks and clients we tested. The choice between using a pre-computed call graph and simultaneous reachability computation applies independent of how a points-to analysis is formulated, and hence it arises for previous analyses based on set constraints or other formalisms. All exhaustive Java points-to analyses with on-the-fly call graph construction we are aware of [RMR01, LH03, WL04, LH04] use simultaneous reachability computation. We use a pre-computed call graph with our demand-driven analysis since simultaneous reachability computation could dramatically increase the time required to compute a result for a single variable (the reachability of all analyzed methods and their transitive callers would have to be determined) and we have not observed a precision decrease in our experiments.

Chapter 4

Context-Insensitive Points-To Analysis

In this chapter, we present a context-insensitive demand-driven points-to analysis for Java, suitable for use in environments with extreme resource constraints like just-in-time (JIT) compilers. Just-in-time compiler optimizations could benefit greatly from precise points-to information, for example through more inlining of virtual calls or more aggressive optimization in the presence of aliasing. Our refinement-based analysis is able to achieve nearly the precision of field-sensitive Andersen's analysis with up to a 16-fold speedup over a state-of-the-art algorithm for the analysis, and is hence the first points-to analysis with performance suitable for a JIT compiler.

The chapter is organized as follows. We begin in §4.1 by giving a high-level overview of our refinement algorithm, illustrated on a simplified version of the points-to analysis problem. Then, in §4.2, we describe our approximation technique, which *regularizes* the original CFL-reachability problem to obtain asymptotic speedups. We show a refinement algorithm in §4.3 that can, when needed, recover most of the precision lost by this initial approximation. In §4.4, we present an experimental evaluation

of our approximation and refinement algorithms. §4.5 discusses language issues that make our refinement-based technique for suitable for Java points-to analysis than C. Finally, §4.6 gives a formulation of the Heintze and Tardieu demand-driven algorithm [HT01a] adapted to Java, used as a baseline in our evaluation.

4.1 Algorithm Overview

This section provides an overview of our refinement-based algorithm. First, we introduce the ideas of demand-driven points-to analysis (§4.1.1) and client-driven refinement (§4.1.2). Then, we present a simplified formulation of the context-insensitive points-to analysis problem in §4.1.3, focusing attention on its balanced-parentheses nature. We present our refinement algorithm in §4.1.4 formulated for an even further simplified problem, aiming to make the key properties of the algorithm clear. Finally, §4.1.5 proves termination and soundness for the algorithm presented in §4.1.4.

4.1.1 Demand-Driven Points-To Analysis

Points-to analyses differ in how eagerly they compute points-to sets for the client (the user or enclosing analysis). Here, we discuss the potential benefits of computing points-to information in a lazy, *demand-driven* manner (the strategy used by our points-to analysis), rather than using the typical *exhaustive* strategy of eagerly computing all results.

Before discussing the performance trade-offs of demand-driven vs. exhaustive points-to analysis, we must first describe how a points-to analysis is typically invoked. The results of a points-to analysis are often consumed by some enclosing analysis that essentially invokes the points-to analysis as a subroutine. For example, a program optimizer may use points-to analysis results to prove that two pointers cannot be

aliased, enabling more powerful optimization. We term the enclosing analysis making use of a points-to analysis the *client* analysis. Each request for a variable's points-to set from the client is called a *query*.

If a client raises only a small number of points-to analysis queries, computing points-to information lazily can yield improved performance. Typically, points-to analyses are *exhaustive*, *i.e.*, they compute and store points-to sets for all variables before answering any client queries. With the exhaustive strategy, queries can be answered in constant time, since all points-to sets have already been computed. In contrast, a demand-driven analysis [Rep94] only computes the points-to information necessary to answer client queries.[1] Many points-to analysis clients only query a small number of program variables. For example, an optimizer may only care about aliasing of variables in frequently executed methods, often a small subset of all variables in the program. For such clients, a demand-driven points-to analysis may provide better performance than an exhaustive analysis.

Program analyses formulated in CFL-reachability can naturally be made demand driven, a particularly useful feature for our points-to analysis work. A demand-driven version of a CFL-reachability problem corresponds to solving the *single-source L-path problem*, *i.e.*, only determining which nodes are L-reachable from some specific source node. Also, an all-pairs CFL-reachability algorithm can be employed to solve a single-source L-path problem by performing the magic-sets transformation on the grammar for L, a technique from deductive databases [Rep94].

In practice, demand-driven points-to analysis alone often does not provide a performance improvement over exhaustive analysis. In the worst case, answering a single query with demand-driven analysis can require as much computation as an exhaus-

[1] Note that some points-to analyses do a mix of pre-processing work and on-demand computation [SFA00, Das00]. In this work, we consider an analysis to be demand driven if it *only* examines statements that may change the analysis result (as in the work of Duesterwald *et al.* [DGS97]), and hence we do not consider these "mixed" points-to analyses to be truly demand driven.

tive analysis [HRS95]. Though previous work has shown significant performance gains with demand-driven points-to analysis for C [HT01a], we have found that for Java points-to analysis, behavior close to the worst case occurs often (see §5.4.2). The key to the success of our points-to analyses is our combination of demand-driven analysis and refinement, which together lead to much improved precision and performance.

4.1.2 Client-Driven Refinement

As described previously in §1.2.3, we aim to develop an analysis based on *client-driven refinement*. A client of a points-to analysis typically aims to perform some transformation or verification of a program that relies on some pointer-related program property. We say that a points-to query is *positively answered* when the points-to information computed by the analysis is sufficiently precise for the client to prove its property of interest. The refinement loop of our algorithm iterates until the given query is (i) positively answered, (ii) no further refinement is possible, or (iii) some time budget for the query is exceeded.

For example, consider a points-to analysis client that tries to resolve virtual calls for the purpose of inlining. Given a virtual call `x.foo()`, this client tries to use points-to information for x to show that only one implementation of `foo()` can be invoked by this call at runtime, allowing for inlining of that implementation at the call site. The client issues a query for the receiver x of the call to `foo`, and considers the query positively answered when the points-to information for x shows that the call to `foo` has only one possible target.

We refer to the use of a time budget to prematurely end long-running queries as *early termination*. When early termination is employed, a sound result must still be returned to the client. For points-to analysis, an early-terminated query could simply return a result stating that the queried variable can point to any abstract location.

Note that early termination does *not* necessarily imply a precision loss for the client. Consider the case where even when run to completion, a points-to analysis cannot positively answer a query. (This can be caused by analysis imprecision or by feasible program behaviors.) In this case, early termination of the query makes no difference to the client: with our without early termination, the query is not positively answered. If early termination is employed primarily in cases where the points-to analysis cannot provide a positive answer, performance is improved without negative effects on precision for the client. In §4.4, we show that early termination has precisely this effect for our points-to analysis.

4.1.3 Simplified Formulation

In this section, we present a simplified CFL-reachability formulation of context-insensitive, flow-insensitive, field-sensitive Java points-to analysis (previously formulated in §3.2). This simplified formulation retains the essential properties of the sound formulation while allowing us to more clearly explain the key properties of our refinement algorithm. The present formulation simplifies the full formulation of §3.2 in two (unsound) ways: (1) it ignores simple copy assignments (*e.g.*, `x = y`), and (2) it does not include the inverse edges required to soundly handle paths between field parentheses (cf. §3.2.2).

Our simplifications yield a formulation of points-to analysis a straightforward balanced-parentheses problem. Let Σ_P be the alphabet of open and close brackets, respectively representing heap writes and reads:

$$\Sigma_P = \left\{ [_f \ , \ ]_f \mid f \text{ is a field} \right\}$$

We formulate our analysis as CFL-reachability with language L_{sf} (`s` for simplified)

over Σ_P:

$$L_{\mathsf{sf}} : F \rightarrow [_f \ F \ ]_f \mid [_g \ F \ ]_g \mid \ \ldots \ \mid F \ F \mid \epsilon$$

L_{sf} captures the key balanced parentheses property of L_{F} (see §3.2.2).

4.1.4 Refinement Algorithm

Here we present our refinement-based algorithm for solving the *single-path problem*, a further simplification of the L_{sf}-reachability problem presented in §4.1.3. We focus on techniques for showing that a node x is *not* L_{sf}-reachable from a node o; our refinement algorithm is designed to quickly prove such unreachability properties. The key idea of the algorithm is to focus effort on parts of the graph likely to have unbalanced parentheses in practice, handling the rest of the graph approximately (in fact by skipping over it entirely). Refinement allows for checking more and more of the graph for balanced parentheses, in the limit yielding the same answer as computing full L_{sf}-reachability.

Through our refinement algorithm does not provide any asymptotic improvement over computing CFL-reachability directly, in practice we have found its benefits to be dramatic. For the context-insensitive analysis of this chapter, even our initial approximation (with no refinement) often yields positive answers for queries from the tested clients. Though it provides some benefit here, refinement becomes more critical for the context-sensitive analysis of Chapter 5.

Single-Path Problem To focus on the key ideas of our technique, we consider the following additional simplification of the single-source/single-target L_{sf}-reachability problem. We constrain the input graph to contain a single acyclic path p from some

node o to some node x, with edge labels chosen from Σ_P (defined in §4.1.3). The analysis problem is then to determine if p is a L_{sf}-path, *i.e.*, of $o\ L_{\mathsf{sf}}\ x$.

To model how refinement improves performance, we add the following optimality constraint to the problem: if p is not an L_{sf}-path, the algorithm should determine this fact while minimizing the total number of edges inspected (we consider an edge inspected if it is processed by the algorithm in any way, *e.g.*, to check its label). As a trivial example, if the first edge e of p is a closed parenthesis, the algorithm should conclude that p is not an L_{sf}-path by inspecting only e. Note that our algorithm does not necessarily visit the minimal number of edges. However, we will show that for certain inputs, the refinement algorithm inspects fewer edges than if L_{sf}-reachability were directly computed by traversing p.

Our refinement technique as applied to the single-path problem generalizes to points-to analysis for arbitrary programs. In the general reachability problem, we are given a variable x, and then must find all o such that $o\ L_{\mathsf{sf}}\ x$. Furthermore, for each o, we must consider all paths from o to x, not just one. For an acyclic graph, both of these generalizations can be viewed as solving multiple instances of the single-path problem (though of course our generalized algorithm aggregates information from different paths for efficiency). We discuss cyclic graphs when presenting the full refinement algorithm in §4.3.

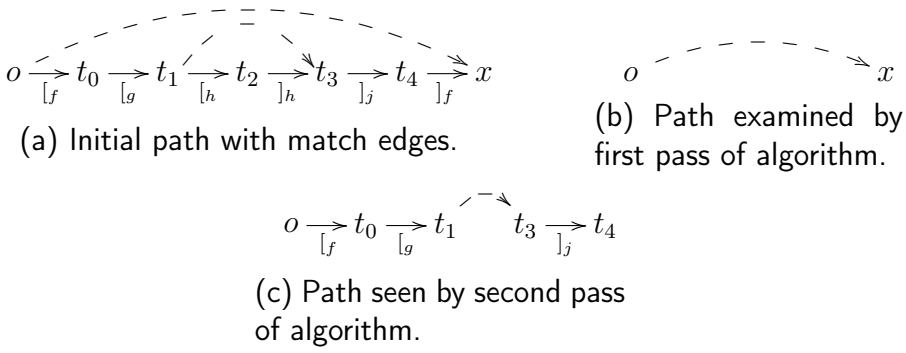
Also note that while refinement may seem unnecessary for the single-path problem, its empirical benefit becomes significant when computing full Java points-to analysis. Since L_{sf}-reachability for a single path can be computed in linear time, refinement may not be necessary to obtain good performance for such cases. However, as discussed in §3.1, full CFL-reachability is more expensive, requiring $O(N^3/\log N)$ time in the worst case. While refinement does not improve on this worst-case behavior, it yields significantly improved performance in practice for Java points-to analysis, as shown in §4.4.

Figure 4.1(a) gives an example input path for our simplified problem (the dashed edges will be explained shortly). The path is not an L_{sf}-path, as the $[_g$ and $]_j$ parentheses are mismatched. We will illustrate how our analysis can discover this fact without inspecting the entire path.

Key Ideas Our technique heuristically improves performance through *approximation* and *refinement*. For the simplified problem, an *approximate* analysis must answer correctly when p is an L_{sf}-path, but can answer incorrectly when it is not. *Refinement* gradually removes the imprecision of this analysis, eventually yielding the same answer as directly computing L_{sf}-reachability.

Our analysis approximates by only checking selected parts of p for balanced parentheses, *entirely skipping* inspection of other edges on p. Proving the existence of just one unbalanced parenthesis on p (*i.e.*, an open parenthesis without a balancing close parenthesis or vice versa) is sufficient to show that p is not an L_{sf}-path. Hence, if our analysis chooses the right parts of p to check for balanced parentheses, it may be able to show p is unbalanced without inspecting the entire path. For example, let α be any string from L_{sf} (*i.e.*, a string with balanced parentheses). Then, the string $s = [_a\ [_b\ \alpha\ ]_c\ ]_a$ cannot be balanced, since the $[_b$ and $]_c$ parentheses are mismatched. Our refinement algorithm is able to only check the beginning and end of a path with label s for balanced parentheses, discovering the mismatched $[_b$ and $]_c$ parentheses while skipping inspection of α.

For correctness, our algorithm cannot skip inspection of arbitrary sub-paths of p. In particular, the algorithm must avoid the case where skipped parentheses can possibly balance inspected parentheses. Consider again the string $s = [_a\ [_b\ \alpha\ ]_c\ ]_a$, which must be unbalanced if α is balanced. If α need *not* be balanced, then skipping inspection of α and concluding that $[_b$ and $]_c$ are mismatched may be incorrect—it is possible that $\alpha =]_b\ [_c$, making $s = [_a\ [_b\ ]_b\ [_c\ ]_c\ ]_a$ balanced. To maintain correctness, our algorithm only skips sub-paths beginning and ending with *matched parentheses*



(a) Initial path with match edges.

(b) Path examined by first pass of algorithm.

(c) Path seen by second pass of algorithm.

Figure 4.1: Paths to illustrate the behavior of our refinement algorithm.

(*i.e.*, parentheses of the form $[_f$ and $]_f$) and performs additional checking in the case of multiple open and close parentheses for a single field. We describe the implementation of this policy and its correctness in more detail below.

match ***edges*** Our algorithm skips sub-paths of p by way of additional match *edges*. A match edge connects the source of some $[_f$ edge to the sink of any matching $]_f$ edge. In Figure 4.1, match edges are shown as dashed edges. We use match edges in two key ways in our refinement algorithm:

1. Our algorithm *only* skips sub-paths whose source and sink are connected by a match edge. This choice is key for algorithm correctness, discussed in more detail below [*fix?*].

2. Our algorithm refines its precision by removing match edges from the graph, which removes sub-path skipping opportunities and forces the algorithm to inspect more of p's edges.

Our analysis approximates L_{sf}-reachability by computing reachability over a language L_{sfr} (r for refinement) that includes match edges:

$$L_{\mathsf{sfr}} : T \rightarrow [_f\ T\ ]_f \mid [_g\ T\ ]_g \mid \ \ldots\ \mid \mathsf{match} \mid T\ T \mid \epsilon$$

(a) A graph for which our algorithm returns an incorrect result.

(b) The graph of (a) fixed to be well-structured.

Figure 4.2: An example showing why our algorithm requires well-structured graphs.

L_{sfr} is identical to L_{sf} except for the additional match production. Thus, L_{sfr} is a superset of L_{sf}, and computing L_{sfr}-reachability approximates L_{sf}-reachability.

Computing L_{sfr}-reachability Each pass of our refinement algorithm computes L_{sfr}-reachability by using match edges to skip sub-paths of p whenever possible. If a node x on p is encountered with a single outgoing match edge $x \xrightarrow{\mathsf{match}} v$, our algorithm *only* follows the match edge, saving time by avoiding inspection of all edges on p between x and v. Figure 4.1(b) shows the L_{sfr}-path discovered by a pass of our analysis when given Figure 4.1(a) as input. The analysis skips the entire original path using the $o \xrightarrow{\mathsf{match}} x$ edge and concludes that x is L_{sfr}-reachable from o. Note that x is not L_{sf}-reachable from o, showing how L_{sfr}-reachability can over-approximate L_{sf}-reachability; this over-approximation can be removed through refinement.

Note that as described, the algorithm above does not correctly compute L_{sfr}-reachability when certain key match edges are missing. In particular, the algorithm fails when given path p with some edge $x \xrightarrow{[_f} y$, the following conditions hold:

1. There are multiple edges on p with the matching label $]_f$.
2. x has at least one outgoing match edge.
3. The match edge connecting x to the *balancing* $]_f$ parenthesis is missing.

For example, consider the graph in Figure 4.2(a). We have an initial edge $o \xrightarrow{[_f} t_0$ and two subsequent edges with the matching $]_f$ label (condition 1). The graph includes

an outgoing match edge from o (condition 2), but is missing the $o \xrightarrow{\text{match}} x$ edge corresponding to the balancing $t_2 \xrightarrow{]_f} x$ edge (condition 3). Our algorithm only follows the $o \xrightarrow{\text{match}} t_2$ edge from o and ignores other edges between o and t_2. Therefore, it encounters the $t_2 \xrightarrow{]_f} x$ without having observed a matching $[_f$ edge and incorrectly concludes that x is not L_{sfr}-reachable from o (incorrect since the original path has balanced parentheses and hence is an L_{sfr} path).

To avoid this incorrectness, we require that each pass of our refinement algorithm work on a *well-structured graph*:

Definition 1. *A graph including a path p and corresponding* match *edges is* well structured *iff each node x on p either (1) has no outgoing* match *edges or (2) has all possible outgoing* match *edges.*

The graph in Figure 4.2(a) is not well-structured, since only some possible match edges from o are present. In contrast, the graph of Figure 4.2(b) is well-structured. When a node has multiple outgoing match edges, our algorithm checks if *any* of the match edges is on an L_{sfr}-path from the source to sink of p, ensuring that an L_{sfr}-path is found if it exists.

Figure 4.3 presents pseudocode for computing L_{sfr}-reachability in each refinement pass. The input to the CheckPath procedure is a well-structured graph G that includes a path p with edges labels from Σ_P (see §4.1.3), starting at node *src* and ending at *sink*. CheckPath returns TRUE iff *sink* is L_{sfr}-reachable from *src* in G. Note that if CheckPath returns FALSE, then p cannot be a L_{sf}-path, since L_{sfr}-reachability over-approximates L_{sf}-reachability. CheckPath is computed recursively via the CheckPathHelper procedure, which maintains a set V of visited nodes and a stack S of thus-far unmatched open parentheses.

For the most part, the CheckPathHelper procedure performs straightforward checking for balanced parentheses using the stack S. Two aspects of its pseudocode

```
CheckPath(src)
1  return CheckPathHelper(src, ∅, ⟨⟩)

CheckPathHelper(x, V, S)
 1  if edge x -[f-> y exists for some f
 2     then if x has outgoing match edges
 3             then for each edge x -match-> z
 4                     do if CheckIfUnvisited(z, V, S)
 5                           then return TRUE
 6                  return FALSE                ▷ no success on any match edge
 7             else return CheckIfUnvisited(z, V, S.f)
 8  elseif edge x -]f-> y exists for some f
 9     then if S = T.f for some (possibly empty) T
10             then return CheckIfUnvisited(z, V, T)
11             else return FALSE                ▷ mismatched parentheses
12  else return S = ⟨⟩                          ▷ end of path, so stack must be empty

CheckIfUnvisited(x, V, S)
1  if x ∈ V
2     then return FALSE
3     else return CheckPathHelper(x, V ∪ {x}, S)
```

Figure 4.3: Pseudocode for a single pass of our refinement algorithm, as applied to the single-path problem.

merit particular mention. The first is the handling of match edges (line 2 through line 6). When outgoing match edges are present, the algorithm checks all the match edges to see if any are on a path with balanced parentheses. The algorithm returns FALSE at line 6 iff *none* of the match edges are on a path with balanced parentheses.

The second interesting aspect of CheckPathHelper is the use of the CheckIfUnvisited procedure to ensure nodes are not visited twice. This check is necessary because some match edges may connect nodes to predecessors on the original path,

```
CheckPathRefine(src)
1  while TRUE
2        do if ¬CheckPath(src)
3              then return FALSE
4              else  if there exists node x with outgoing match edges
5                        then remove all outgoing match edges from x
6                        else  return TRUE
```

Figure 4.4: Outer loop of our refinement algorithm for the single-path problem.

creating cycles in the graph. For example, consider the following input path:

$$t \xrightarrow{[_f} x \xrightarrow{]_f} y \xrightarrow{[_f} z$$

Since y is both the source of an edge labeled $[_f$ and the sink of an edge labeled $]_f$, the input graph will include a $y \xrightarrow{\mathsf{match}} y$ edge. Without checking for previously visited nodes, this input would lead to non-termination in the algorithm.

Refinement Refinement in our algorithm is accomplished by removing match edges from the graph, forcing checking of more parentheses on the original path. Given edge $e = t \xrightarrow{[_f} u$, if outgoing match edges from t are removed, our algorithm can no longer "skip" over e. Instead, the algorithm must check for a sub-path labeled $[_f\ T\ ]_f$ from t (T being the start symbol for L_{sfr}), and in the process it may discover unbalanced parentheses that were skipped when match edges from t were present. Figure 4.1(c) shows the sub-path of Figure 4.1(a) explored by our algorithm after removing the $o \xrightarrow{\mathsf{match}} x$ edge. This removal exposes the unbalanced parentheses $[_g$ and $]_j$, leading the analysis to conclude that p is not an L_{sf}-path. Note that the unbalanced parentheses are still discovered without analyzing the whole path (the $t_1 \rightsquigarrow t_3$ sub-path was skipped), indicating the possible performance benefits of removing the "right" match edges from the graph.

Figure 4.4 gives pseudocode for the outer refinement loop of our algorithm. The

algorithm assumes a well-structured input graph G that includes *all possible* match edges. The CheckPathRefine procedure returns TRUE iff p is an L_{sf}-path, since it continues refinement until all match edges are removed.

In each iteration, the algorithm of Figure 4.4 first computes L_{sfr}-reachability using CheckPath (line 2). If CheckPath returns FALSE, then the algorithm immediately returns FALSE (line 3), as the path cannot have balanced parentheses. If CheckPath returns TRUE, refinement is performed by removing all outgoing match edges from some node x on the path (line 5). The removal of *all* outgoing match edges from the node ensures that the graph remains well-structured, key to correctness. If no match edges can be removed, the algorithm returns TRUE (line 6), since at that point no more refinement is possible.

Note that the pseudocode in Figure 4.4 leaves out certain details irrelevant to correctness. For example, it does not specify which outgoing match edges are removed in each pass, though this choice is important for efficiency in practice. Also, the pseudocode continues refinement until all match edges are removed, while in practice, some budget would be used to terminate the algorithm earlier for some queries.

4.1.5 Proofs of Termination and Soundness

Here, we present proofs of termination and soundness for the refinement algorithm shown in Figure 4.4 in §4.1.4.

Termination The termination argument for our refinement algorithm is straightforward. First, the outer loop in Figure 4.4 can only run for a finite number of iterations. Each iteration must either terminate the loop immediately or remove some match edges from the graph; when no match edges remain, the loop is terminated. Since the input graph has a finite number of match edges, the loop can only run for a finite number of iterations.

The call to CheckPath in each loop iteration of Figure 4.4 must also terminate. As seen in Figure 4.3, CheckPathHelper can visits each node on the path at most once, due to the check in CheckIfUnvisited. Since the path has a finite number of nodes, each call of CheckPath clearly terminates.

Soundness Proving soundness requires showing that our algorithm never claims that a path has unbalanced parentheses when in fact they are balanced. This result would be unsound since it filters a possibly valid flow from consideration, while points-to analysis clients rely on points-to sets over-approximating all valid flows. More formally, we prove the following soundness theorem:

Theorem 1. *If* CheckPathRefine *(defined in Figure 4.4) returns* false*, indicating that the input path p has unbalanced parentheses, then p cannot have balanced parentheses.*

To prove Theorem 1, we show that (1) assuming the input graph is well-structured (see Definition 1 in §4.1.4), the CheckPath procedure (from Figure 4.3) always returns a sound result, and that (2) the outermost CheckPathRefine procedure always invokes CheckPath with a well-structured graph. We state these two conditions conditions for proving Theorem 1 as lemmas:

Lemma 1. *If* CheckPath *(defined in Figure 4.3) returns* false *for a well-structured graph, then the original input path cannot have balanced parentheses.*

Lemma 2. *At line 2 of* CheckPathRefine *(Figure 4.4), the graph containing the node src is always well structured.*

Lemma 2 clearly holds: we assume the graph contains all possible match edges initially (and hence is well structured), and line 5 in Figure 4.4 only removes all outgoing match edges from a node, maintaining well-structuredness. Hence, we have

reduced the problem of proving Theorem 1 to that of proving the soundness of each individual refinement pass, *i.e.*, Lemma 1.

We first argue that the CHECKPATH procedure is sound for graphs with no match edges, *i.e.*, that the following lemma holds:

Lemma 3. *For input graphs consisting of a single path without* match *edges,* CHECKPATH *returns* FALSE *iff the path has unbalanced parentheses.*

For inputs with no match edges, CHECKPATHHELPER from Figure 4.3 essentially simulates a push-down automaton checking for a balanced parentheses string. We elide the straightforward soundness proof for Lemma 3, focusing on the more interesting case of inputs with match edges.

We shall prove Lemma 1 by contradiction, making an inductive argument that CHECKPATH cannot return FALSE for a well-structured graph in which the input path has balanced parentheses. An inductive proof requires a way to relate the behavior of our algorithm on longer input paths to its behavior on shorter paths. We do so by proving that in certain cases, *pruning* the path skipped by a match edge has no effect on the result of the CHECKPATH procedure. We first define the pruning operation:

Definition 2. *Given an input path p with node x and subsequent node y,* pruning *the sub-path from x to y yields a graph with the following properties:*

1. *All nodes between x and y on p are removed, and all incoming and outgoing edges to those nodes (including* match *edges) are removed.*

2. *Nodes x and y are merged into a single node z, such that incoming edges to x go to z and outgoing edges from y start from z.*

Given this definition, we can now state and prove our pruning lemma:

Lemma 4. *Consider a graph G containing input path p and corresponding* match *edges. Let x be the* first *node on p with outgoing* match *edges, and let $x \xrightarrow{\text{match}} y$ be some outgoing* match *edge from x. Consider the graph G' with path p' obtained by pruning the sub-path of p from x to y. If* CheckPath *returns* true *for p' in G', then it must return* true *for p in G.*

Proof. First, we have that for the CheckPathHelper calls for x on p and for z (the merged version of x and y) on p', the V and S arguments are identical. This holds because x was the *first* node on p with outgoing match edges, and hence both x and z are reached by the algorithm before any match edges are traversed, and the sub-path preceding x on p and z on p' are identical. Second, when handling the $x \xrightarrow{\text{match}} y$ edge in G, CheckPathHelper is invoked for y (through a call to CheckIfUnvisited) with the V and S arguments unchanged from those passed in for x (line 4 in Figure 4.3). Hence, the successors of y on p and the successors of z on p' are reached with identical V and S arguments in CheckPathHelper calls.

So, we know that the sub-path of p after y and the sub-path of p' after z are both checked in the same algorithm state. If the graphs examined in both these cases are identical, then the lemma clearly holds. One graph difference is possible: p' may be missing backward match edges from nodes following y to nodes between x and y on p, since the target nodes were pruned. However, we are only concerned with cases in which CheckPath returns true for p'. The loop handling match edges in CheckPathHelper (line 3 through line 5 in Figure 4.3) returns true if a balanced path is found with *any* match edge: additional backward match edges would not affect this result. So, we have that in all cases, if CheckPath returns true for p', it also returns true for p, as desired. □

We are finally ready to prove the soundness of CheckPath (Lemma 1):

Proof of Lemma 1. Our proof is by induction on the length of the original input path

p.

Base Case: In the base case, p has 0 edges and consists of a single node. In this case, CHECKPATHHELPER simply returns TRUE at line 12. Since the premise of Lemma 1 requires CHECKPATH to return FALSE, Lemma 1 is vacuously true in the base case.

Inductive Case: Here, we assume that CHECKPATH is sound for all paths up to length n, and prove that it is sound for paths of length $n+1$. For the purposes of contradiction, assume that Lemma 1 holds for paths up to length n but does not hold for all paths of length $n+1$. Let p be some path of length $n+1$ with balanced parentheses for which CHECKPATH unsoundly returns FALSE. We shall show that in fact, CHECKPATH must return TRUE for p. Note that we assume the well-structured graph containing p has some match edges; if not, we get an immediate contradiction by Lemma 3.

We first claim that for the first node x on p with outgoing match edges, CHECKPATHHELPER(x, V, S) must return FALSE. To see why, assume (for contradiction) that CHECKPATHHELPER(x, V, S) returns TRUE. Then, in order for CHECKPATH to return FALSE for p, CHECKPATHHELPER must return FALSE for some node preceding x on p. However, from Lemma 3 we know that CHECKPATH is sound for paths with no match edges. Hence, since we assumed p has balanced parentheses, and there are no nodes with outgoing match edges before x, CHECKPATHHELPER *must* return TRUE for nodes preceding x, and we have a contradiction. So, we have shown that CHECKPATHHELPER returns FALSE for the first node x on p with outgoing match edges.

Given x, the assumption that p is balanced, and the assumption that the graph containing p is well structured (see Definition 1), there must be an edge $x \xrightarrow{\text{match}} v$ such that the corresponding open and close parenthesis edges for the match edge balance each other on p. Consider the path p' obtained from pruning the sub-path from x

to v from p. We have two cases, corresponding to whether p' does or does not have balanced parentheses. We shall show that each case leads to a contradiction.

In the first case, assume that p' has balanced parentheses. Since p' must have length less than or equal to n, we know that CHECKPATH returns TRUE for p' by the inductive hypothesis. But, by Lemma 4, CHECKPATH must also return TRUE for p. We have a contradiction, eliminating this case from consideration.

We are left with the case in which p has balanced parentheses and p' has unbalanced parentheses. Let the $x \xrightarrow{\mathsf{match}} v$ correspond to $[_f$ and $]_f$ edges for some field f, and let the concatenated labels of the edges between x and v on p be α. Since p is balanced and p' is not, it must be the case that parentheses in α balance other parentheses appearing before x and/or after v, *i.e.*, we have a balanced parentheses string of the form $[_g\ [_f\ \alpha\ ]_f\ ]_j$ for some fields g and j (without loss of generality). However, recall our stipulation that the $x \xrightarrow{\mathsf{match}} v$ edge correspond to the $[_f$ and $]_f$ edges that *balance* each other on p. Hence, due to proper nesting of parentheses, the $[_g\ [_f\ \alpha\ ]_f\ ]_j$ string cannot possibly be balanced. The $[_g$ parenthesis cannot be balanced within α since then we have an unbalanced string of the form $[_g\ [_f\ ]_g\ ]_f$; a similar argument holds for the $]_j$ parenthesis. Hence, we have another contradiction.

Since all cases lead to a contradiction, we have proved that Lemma 1 holds for paths of length $n+1$, completing the inductive proof of Lemma 1. □

4.2 Regular Approximation

In this section, we present an algorithm that computes only the initial approximation of our refinement algorithm, *i.e.*, reachability over a graph with all possible match edges. We show that this approximation requires reachability over a *regular* language R_{F}, and hence is asymptotically less expensive than CFL-reachability. We give a simple and efficient demand-driven algorithm RegularPT, essentially depth-first search,

for finding points-to information based on R_{F}-reachability. §4.4 will show that RegularPT achieves most of the precision of L_{F}-reachability within a time budget of only 1 second per query.

4.2.1 Regular Reachability

Here, we formulate a regular approximation of L_{F}-reachability (the field-sensitive, context-insensitive analysis formulated in §3.2.2) via match edges. Recall our definition of a *flowsTo*-path for L_{F}:[2]

$$\mathit{flowsTo} \rightarrow \mathsf{new}\ (\ \mathsf{assign}\ \mid\ \mathsf{putfield}[f]\ \ \mathit{alias}\ \ \mathsf{getfield}[f]\ )^*$$

The context-free aspect of L_{F}-reachability is checking putfield$[f]$ and getfield$[f]$ edges, the balanced parentheses of L_{F}, for the field-sensitivity conditions specified in §2.2.2. The most expensive part of field-sensitivity is checking for an *alias*-path between the base variables of putfield$[f]$ and getfield$[f]$ edges, since the path may be complex and must itself have balanced parentheses.

We approximate the search for *alias*-paths between matched parentheses by conservatively assuming that such a path always exists, using match edges. As in §4.1, match edges connect matched parenthesis edges, going from the source of each putfield$[f]$ edge to the target of each getfield$[f]$ edge on the same field f. Figure 4.5 shows an example graph with all match edges included. Given a graph with all possible match edges, we can over-approximate L_{F}-reachability with *regular reachability* using language R_{F}, defined as follows:

$$\mathit{flowsToReg} \rightarrow \mathsf{new}\ (\ \mathsf{assign}\ \mid\ \mathsf{match}\ )^*$$

[2]Note that since this chapter is concerned only with context-insensitive analysis, we assume here that assignglobal, param$[i]$, and return$[i]$ edges are labeled assign in all input graphs.

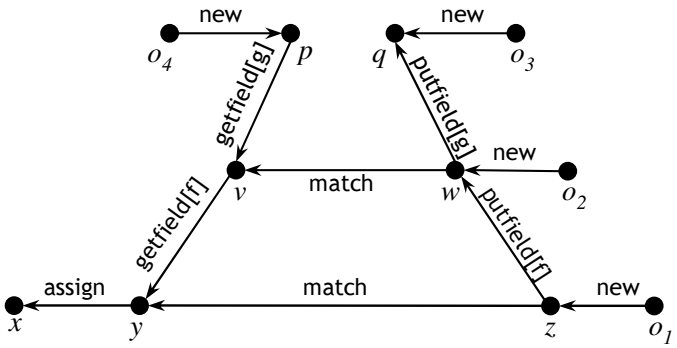


Figure 4.5: A graph illustrating match edges.

Given graph G, let G_R be G with match edges added from the source of each putfield$[f]$ edge to the target of each getfield$[f]$ edge for all fields f. To show that R_{F}-reachability over-approximates L_{F}-reachability and thus is sound, we must show that if x is L_{F}-reachable from o in G, then x is R_{F}-reachable from o in G_R. Note that the *flowsToReg* production differs from the *flowsTo* production only in that "putfield$[f]$ *alias* getfield$[f]$" has been replaced by match. So, the soundness proof reduces to showing that whenever nodes x and y in G are connected by a path labeled putfield$[f]$ *alias* getfield$[f]$, G_R includes a match edge from x to y. This clearly holds by construction, since we add match edges between putfield$[f]$ and getfield$[f]$ edges in G_R regardless of whether their base variables are connected by an *alias*-path.

Since R_{F}-reachability over-approximates L_{F}-reachability, node x may be R_{F}-reachable from node o in G_R but not L_{F}-reachable from o. For example, in Figure 4.5, v is R_{F}-reachable from o_2 but not L_{F}-reachable, since there is no *alias*-path from q to p. In general, precision is lost in cases where an R_{F}-path includes an *invalid* match edge m, where the base variables of the field accesses corresponding to m are not connected by an *alias*-path. In §4.3, we show how refinement can be used to recover most of the precision lost by using match edges.

Note that a points-to analysis technique that, like R_{F}-reachability, handles field

accesses by checking only for matching fields is in fact a field-based analysis (*cf.* §2.2.2). This technique has been shown to have precision relatively close to that of a field-sensitive analysis in previous work [LPH01, LH03], a result reproduced in our experiments.

Since R_{F} is regular, answering the single-source R_{F}-problem is asymptotically cheaper than the single-source L_{F}-problem ($O(NE)$ for regular reachability vs. $O(N^3/\log N)$ for CFL-reachability, as discussed in §3.1). Note the simplicity of this regular expression for *flowsToReg*, relative to the grammar of Figure 3.2. One consequence of our use of match edges is that we no longer need to consider both standard and barred edges when determining R_{F}-reachability; this leads to both conceptual simplicity and a much simpler reachability algorithm (depth-first search) than what is required for L_{F}-reachability.

4.2.2 RegularPT

Figure 4.6 gives pseudocode for an algorithm RegularPT that determines points-to information on demand using R_{F}-reachability. The RegularPT procedure takes a node x and returns the set of all nodes o such that x is R_{F}-reachable from o. This returned set is a points-to set for x, since R_{F}-reachability over-approximates the flows-to relation. The RegularPT procedure is a standard worklist-based depth-first search, traversing incoming assign and match edges to find reachable new edges and their abstract locations.

The worst-case complexity of RegularPT is $O(E + M)$, where E is the number of edges in G and M is the number of match edges in G_R. The derivation of this bound is straightforward, as G_R has $E + M$ edges and RegularPT essentially performs depth-first search on G_R. In real-world Java programs, E is typically $O(N)$ (where N is the number of nodes in G), since variables are typically assigned very few times.

```
RegularPT(x)
 1  pointsTo ← ∅
 2  marked ← ∅
 3  worklist ← [ ]
 4  Propagate(x, marked, worklist)
 5  while worklist ≠ [ ]
 6        do w ← Pop(worklist)
 7           for each edge w ←new— o
 8                 do AddTo(pointsTo, o)
 9           for each edge w ←assign— y
10                 do Propagate(y, marked, worklist)
11           for each edge w ←match— y
12                 do Propagate(y, marked, worklist)
13  return pointsTo


Propagate(x, marked, worklist)
1  if x ∉ marked
2     then AddTo(marked, x)
3          Push(worklist, x)
```

Figure 4.6: Pseudocode for the RegularPT algorithm.

While M can be $O(N^2)$ in the worst-case, a space blowup can be avoided with an implicit representation of match edges in which each field is mapped to corresponding read and write edges.

4.2.3 Improving Precision with Types

The Java type system can be used to improve the precision of RegularPT, similar to previous work [LH03]. We say types `A` and `B` are *incompatible* if `A` is not a subtype of `B` and `B` is not a subtype of `A`.[3] If variable x has declared type `A`, and variable y has incompatible declared type `B`, then any *flowsToReg*-path ending at x that passes through y can safely be ignored, since the type system (both at compile time and

[3] If `A` is an interface, we must check that all classes implementing `A` are incompatible.

through runtime downcast checks) prohibits a flow of objects from y to x. Such *flowsToReg*-paths can exist in the graph because of downcasts. For example, given statements `Object o = a; b = (B)o`, we have a *flowsToReg* b, even if a and b have incompatible declared types.[4] When answering a query for variable x with declared type `A`, RegularPT does not add nodes whose declared types are incompatible with `A` to the worklist.

We can also decrease the number of added match edges using types. It is possible for the base variables of a getfield$[f]$ edge and putfield$[f]$ edge (e.g. nodes v and w in Figure 4.5) to have incompatible types, even though the f field is accessed on both variables. For example, if we have class `A` with field f, class `B extends A`, and class `C extends A`, then `B` and `C` are incompatible in spite of both having field f. When the base variables of a getfield$[f]$ edge $v \rightarrow y$ and a putfield$[f]$ edge $z \rightarrow w$ have incompatible types, we can safely avoid adding a match edge from z to y, since there will *never* be an *alias*-path between v and w. Empirically, these type-based tests considerably improved precision on our benchmarks, which was consistent with results in past work on Java points-to analysis that used similar techniques [LH03].

4.3 Refinement

Here we show how the refinement technique described in §4.1 allows us to recover most of the precision lost by approximating L_{F}-reachability with R_{F}-reachability. Our evaluation shows the precision of R_{F}-reachability to be relatively close to that of L_{F}-reachability for the clients we tested (see §4.4). However, in cases where the precision of R_{F}-reachability is insufficient and time constraints are tight, our experiments show that our refinement technique is more precise than computing fully field-sensitive

[4] Opportunities for type filtering stem primarily from context- and flow-insensitivity in the points-to analysis.

L_{F}-reachability.

In this section, we first formulate the reachability problem that allows for refinement by removing match edges. We then give an algorithm RefinedRegularPT that solves this reachability problem through iterative refinement. In each iteration, RefinedRegularPT adds precision by removing more match edges, terminating when either the points-to analysis client is satisfied or all inspected match edges have been removed. When all relevant match edges are removed, RefinedRegularPT can provide nearly the same precision as L_{F}-reachability.

Note that Chapter 5 describes a different refinement algorithm that could also be applied to the context-insensitive analysis considered in this chapter. The algorithm improves on RefinedRegularPT in some ways but makes different performance trade-offs (see §5.3.2 for more discussion). We present RefinedRegularPT since it may be the best algorithm for certain clients and it was used for the experimental evaluation of §4.4.

4.3.1 Refining through match edge removal

When match edges introduce imprecision relative to computing L_{F}-reachability, *refinement* can recover the lost precision. Recall that the soundness of R_{F}-reachability relies on the fact that whenever there is a "putfield$[f]$ *alias* getfield$[f]$"-path from x to y in the original graph G, there is a match edge from x to y in G_R. Our technique refines by removing some match edge m *and* checking for the existence of an *alias*-path between the base variables of the corresponding putfield$[f]$ and getfield$[f]$ edges.

R_{F}-reachability is used to approximate the search for an *alias*-path during refinement, making use of the remaining match edges. The *alias* production $\overline{\mathit{flowsTo}}\ \mathit{flowsTo}$, given in Figure 3.2, requires possibly long-running L_{F}-reachability computations, as an *alias*-path does not contain match edges. Instead of searching for *alias*-

paths, we refine a match edge by looking for *aliasReg*-paths, defined with a simple extension of the R_{F} grammar (see §4.2.1):

$$\textit{aliasReg} \rightarrow \overline{\textit{flowsToReg}}\ \textit{flowsToReg}$$
$$\overline{\textit{flowsToReg}} \rightarrow (\ \overline{\mathsf{assign}}\ \mid\ \overline{\mathsf{match}}\ )^*\ \ \overline{\mathsf{new}}$$

Finding *aliasReg*-paths instead of *alias*-paths during refinement is clearly sound, by an argument similar to the soundness argument for R_{F}-reachability.

An *aliasReg*-path may itself contain match edges, and could therefore connect two nodes that are not connected by an *alias*-path. However, we can once again regain precision using refinement, this time refining the match edges on the *aliasReg*-path. Our iterative refinement algorithm is based on repeatedly discovering *aliasReg*-paths while refining match edges, and then refining the match edges on those *aliasReg*-paths.

As an example, let us consider refining by removing the match edge $z \rightarrow y$ in Figure 4.5. To do so, we search for an *aliasReg*-path from w to v, and we find one via o_2 labeled $\overline{\mathsf{new}}$ new match. This path includes the match edge $w \rightarrow v$, which we can in turn refine by searching for an *aliasReg*-path from q to p. No such path exists, so the match edge $w \rightarrow v$ can be safely removed from the graph. This removal eliminates the only *aliasReg*-path from w to v, meaning the match edge $z \rightarrow y$ can also be safely removed.

4.3.2 RefinedRegularPT

Figure 4.7 gives pseudocode for the RefinedRegularPT algorithm, an extension of the RegularPT algorithm that performs iterative refinement based on the needs of the points-to analysis client (previously described in §4.1.2). The RefinedRegularPT procedure takes as input a node x and returns TRUE if the query has been positively

getfieldsToRefine : Set of getfield[f] edges
getfieldsSeen : Set of getfield[f] edges

```
DoTraversal(x)
 1  pointsTo ← ∅
 2  marked ← ∅
 3  worklist ← [ ]
 4  Propagate(x, marked, worklist)
 5  while worklist ≠ [ ]
 6        do w ← Pop(worklist)
 7           for each edge w ←new— o
 8                 do AddTo(pointsTo, o)
 9           for each edge w ←assign— y
10                 do Propagate(y, marked, worklist)
11           for each edge w ←getfield[f]— p
12                 do if e ∉ getfieldsToRefine
13                       then AddTo(getfieldsSeen, e)
14                            for each edge q —putfield[f]→ y
15                                  do Propagate(y, marked, worklist)
16                       else RemoveFrom(getfieldsToRefine, e)
17                            ptOfP ← DoTraversal(p)
18                            for each edge q ←putfield[f]— y
19                                  do ptOfQ ← DoTraversal(q)
20                                     if ptOfP ∩ ptOfQ ≠ ∅
21                                        then Propagate(y, marked, worklist)
22                            AddTo(getfieldsToRefine, e)
23  return pointsTo
```

```
RefinedRegularPT(x)
1  getfieldsToRefine ← ∅
2  while TRUE
3        do getfieldsSeen ← ∅
4           pointsTo ← DoTraversal(x)
5           if PositivelyAnswered(pointsTo)
6              then return TRUE
7              else if getfieldsSeen ⊆ getfieldsToRefine
8                      then return FALSE
9                      else getfieldsToRefine ← getfieldsToRefine ∪ getfieldsSeen
```

Figure 4.7: Pseudocode for the RefinedRegularPT algorithm.

answered, and FALSE otherwise.[5] The DoTraversal procedure is similar in function to the RegularPT procedure of RegularPT (seen in Figure 4.6), computing a points-to set for x through a depth-first traversal of the graph; each call to DoTraversal computes a new points-to set. The Propagate procedure is identical to that of Figure 4.6. The definitions of Pop, AddTo, and RemoveFrom are straightforward and we elide them for clarity.

Note that match edges are not explicitly represented in the pseudocode of Figure 4.7. Instead, an implicit representation is used, with the assumption data structures exist for finding reads and writes to any field in constant time (our implementation builds such data structures). Line 14 and line 15 in Figure 4.7 do the equivalent of traversing all incoming match edges for a node, by finding all putfield$[f]$ edges matching an incoming getfield$[f]$ edge and propagating their sinks.

Refinement Lines 16-22 of Figure 4.7 perform refinement of match edges. The pseudocode checks for an *aliasReg*-path between a $p \xrightarrow{\text{getfield}[f]} w$ edge and a $y \xrightarrow{\text{putfield}[f]} q$ edge by finding points-to sets $pt(p)$ and $pt(q)$ (lines 17 and 19), and then checking if $pt(p) \cap pt(q) \neq \emptyset$. The points-to sets include nodes that are reachable along $\overline{\textit{flowsToReg}}$-paths from p and q. Therefore, for any $o \in pt(p) \cap pt(q)$, an *aliasReg*-path from p to q through o can be constructed. For a given getfield$[f]$ edge $p \rightarrow w$, there may be many putfield$[f]$ edges on the same field, and therefore many incoming match edges to w. Instead of refining each such match edge individually, RefinedRegularPT refines them together, thereby avoiding redundant computation of $pt(p)$ for each edge.

We have found an alternate strategy for refining match edges to be empirically more efficient in certain cases. The alternate strategy first finds $pt(p)$, but then finds the set of nodes Q that are reachable along *flowsToReg*-paths from nodes in $pt(p)$; if $q \in Q$, an *aliasReg*-path from p to q clearly exists. We observed that in practice

[5] For positively answered queries, our implementation also makes the computed points-to set available to the client.

this alternate strategy traverses fewer nodes when there are more than two matching putfield[f] edges for a getfield[f] edge, since it does not compute a points-to set for the target of each putfield[f] edge. Our implementation employs the appropriate strategy based on the number of matching putfield[f] edges.

Choosing* match *edges to refine RefinedRegularPT focuses analysis effort on match edges that have already been observed to possibly reflect imprecise flow. RefinedRegularPT first tries to answer a query on some variable x without any refinement, just using R_{F}-reachability. Consider the case where the points-to set for x found using R_{F}-reachability cannot positively answer the query. Let K be the set of match edges on any R_{F}-path from some abstract location o to x. If using R_{F}-reachability to find x's points-to set is less precise than L_{F}-reachability, then K must contain at least one invalid match edge (*i.e.*, a match edge with no corresponding *alias*-path). In its next pass, RefinedRegularPT only refines match edges in K, aiming to positively answer the query with this (typically) small amount of refinement; we have found this amount of refinement to often be sufficient in practice. Proving that some match edge in K is invalid may however require refinement of match edges outside of K, leading to an iterative refinement process.

In the pseudocode of Figure 4.7, we maintain a set *getfieldsToRefine*, containing the getfield[f] edges whose corresponding match edges should be refined, and a set *getfieldsSeen*, containing getfield[f] edges whose match edges were traversed but not refined in the current iteration of the algorithm. *getfieldsToRefine* is maintained across refinement iterations for a single query, while *getfieldsSeen* is cleared on each refinement iteration (line 3 of RefinedRegularPT). If the points-to result computed by an iteration of the algorithm is sufficient for a positively answered query, we terminate and return true (lines 5-6); the PositivelyAnswered procedure is provided by the client. Otherwise, we add the getfield[f] edges in *getfieldsSeen* to *getfieldsToRefine* (line 9), and begin a new iteration. When we cannot add any new

getfield$[f]$ edges to *getfieldsToRefine*, we give up on positively answering the query and return FALSE (lines 7-8).

While refining match edges corresponding to a getfield$[f]$ edge e, we remove e from *getfieldsToRefine* (lines 23 and 31 of Figure 4.7). To see why, consider refining a match edge m for a getfield$[next]$ edge $e = x \rightarrow x$, corresponding to the statement `x = x.next`. The recursive call to DoTraversal at line 24 will pass x as its argument, and if e remained in *getfieldsToRefine*, RefinedRegularPT would again try to refine m, leading to an infinite loop.

Removing a getfield$[f]$ edge from *getfieldsToRefine* during refinement of a corresponding match edge m can lead to imprecision. With the getfield$[f]$ edge removed, RefinedRegularPT may find *aliasReg*-paths during refinement of m that include m itself, as m will not be refined again when encountered. If all *aliasReg*-paths discovered during refinement of m include m, then m is still invalid, as m cannot be used to justify its own existence. However, in such cases RefinedRegularPT is unable to show that m is invalid, losing precision relative to a fully field-sensitive analysis. In general, if RefinedRegularPT refines all match edges, it may compute a less precise result than L_{F}-reachability in cases where G_R contains cyclic paths that include field dereferences, *e.g.*, the cyclic getfield$[next]$ edge $x \rightarrow x$ for `x = x.next`. We have not found the precision loss due to this aspect of our algorithm to be significant in practice.

In the worst-case, a single iteration of RefinedRegularPT may require $O(M^M E)$ time, with E and M defined as in §4.2.2. This worst case occurs when all match edges are being refined, and refining one match edge requires refining $M - 1$ other match edges, each of which requires refining $M - 2$ match edges, and so on. We have not encountered this worst-case behavior in practice, and since we envision clients using strict time budgets and early termination with our algorithms (see §4.1.2), it is not a practical concern.

4.4 Evaluation

We evaluate the behavior of RegularPT and RefinedRegularPT with two clients and several benchmarks. Our evaluation validates the following experimental hypotheses about the algorithms:

The algorithms are precise We show that RegularPT has precision close to that of field-sensitive Andersen's analysis. It resolves more than 89% of the virtual calls that field-sensitive Andersen's analysis can across our benchmarks, and more than 96% of those virtual calls that are not in dead code. RefinedRegularPT provides more precision than RegularPT, resolving nearly all of the virtual calls in live code that field-sensitive Andersen's can. We also show that an intraprocedural version of RegularPT resolves far fewer calls, indicating that our results cannot be obtained with purely intraprocedural analysis.

Precision retained under early termination We show that RegularPT and RefinedRegularPT retain almost all their precision when run with small time budgets and early termination. For nearly all benchmarks, the two algorithms can resolve 90% of the virtual calls that field-sensitive Andersen's can within a 50 node traversal limit (2ms / query). We show that an adaptation of a previously presented demand-driven algorithm [HT01a] that uses full field-sensitivity does not perform nearly as well within a small budget. RegularPT and RefinedRegularPT also answer *all* virtual call and aliasing queries in hot methods of the SPEC benchmarks as precisely as field-sensitive Andersen's analysis, requiring less than 108 nodes of traversal per query.

The algorithms meet our performance goals Since our algorithms perform well with small time budgets, timeouts can be used to ensure good performance while maintaining precise results. For example, we can answer all virtual call queries

in hot methods of the javac benchmark 16x faster than exhaustive field-based Andersen's analysis and 34x faster than exhaustive field-sensitive Andersen's analysis. The memory consumption of RegularPT and RefinedRegularPT is also much less than that of an exhaustive algorithm.

4.4.1 Experimental Configuration

Implementation We implemented our analyses using the Soot 2.2.1 [VRHS+99] and SPARK [LH03] frameworks. We re-used the pointer assignment graph built by SPARK, thereby leveraging their existing analyses for determining reachable code. To handle method calls, we configured SPARK to build a conservative call graph using a class-hierarchy analysis [DGC95, BS96]. We used assign edges rather than param$[i]$, return$[i]$, and assignglobal edges of §3.2.1 to model parameter and return value flow and assignments to globals, as our analysis is context-insensitive. A best effort was made to handle reflective constructs and native methods, as discussed previously in §3.2.3.

To compare against the state-of-the-art, we also implemented a demand-driven algorithm FullFS that uses the same techniques as the algorithm in [HT01a], but works for Java pointer constructs. The basic idea of the algorithm is to find points-to sets for only the variables necessary to answer a top-level points-to query. The points-to sets are found by iterating over relevant statements, applying inference rules to introduce new points-to queries for relevant variables and to propagate abstract locations to queried variables. The algorithm is similar to exhaustive propagation algorithms for Java [LH03, WL04], except that it only propagates the abstract locations relevant to the query. We chose to make FullFS treat fields with full field-sensitivity, thereby computing L_{F}-reachability. The C algorithm in [HT01a] is field-sensitive for the unnamed field accessed by the C $*$ operator, but is field-based for structure fields.

Algorithm	Description
RegularPT	See §4.2.2
RefinedRegularPT	See §4.3.2
FullFS	Adaptation of algorithm in [HT01a]; See §4.6
ExhaustiveFB	Exhaustive field-based Andersen's from SPARK [LH03]
ExhaustiveFS	Exhaustive field-sensitive Andersen's from SPARK [LH03]

Table 4.1: Descriptions of points-to analysis algorithms used in our experiments.

Since the * operator is so frequently used in C programs, full field-sensitivity seemed to be the analogous handling of Java fields. Details of FullFS appear in §4.6.

Table 4.1 lists all points-to analysis algorithms used in our experiments. ExhaustiveFB and ExhaustiveFS are efficient implementations of exhaustive field-based and field-sensitive Andersen's analysis respectively, as provided by SPARK [LH03]; to the best of our knowledge, their speed is competitive with any published implementation of Andersen's analysis for Java.

All experiments were run on a machine with a Pentium 4 Xeon 2.4GHz processor and 2GB RAM, running Redhat Linux 9. We used the Java 1.4.2 JVM as the underlying VM for our experiments, but we analyzed the 1.3.1_01 libraries, to be consistent with [LH03] and because Soot provides models for the native methods in those libraries.

Benchmarks and Clients The characteristics of our benchmarks are presented in Table 4.2. We used the SPEC JVM98 benchmark suite, two benchmarks from the Ashes suite [Ash], `soot` and `sablecc`, and `jedit` [jEd], an open-source text editor. Subsets of these benchmarks were also utilized in previous Java pointer analysis studies [LPH01, RMR01, WL02, LH03, WL04].

The "# Methods" column reports the number of methods found reachable by

Benchmark	# Methods	# Vars	# Stmts
`soot`	6089	51853	146292
`compress`	12244	95463	269289
`jess`	12878	101332	289514
`raytrace`	12378	96873	271980
`db`	12249	95665	270571
`javac`	13385	107753	318411
`mpeg`	12456	98458	276062
`jack`	12502	98579	278965
`sablecc`	14065	110292	352338
`jedit`	17510	144062	412835

Table 4.2: Information about our benchmarks.

SPARK's class-hierarchy analysis (these numbers differ from those in [LH03] due to improvements in the handling of reflection in Soot 2.2.1). "# Vars" is the number of variables (locals or static fields) in the program, and "# Stmts" is the number of assignment statements (the number includes the temporary variables and assignments introduced to make each assignment one of our simple forms). Our largest benchmark `jedit` is comparable in size with the largest benchmarks used in other pointer analysis studies [BLQ+03, WL04].

We evaluated our analyses using two clients, *virtcall* and *localalias*. *virtcall* attempts to resolve virtual calls to a single target by finding the points-to set of the call's receiver. We only consider calls where cheaper type-based techniques cannot resolve the call. *localalias* attempts to disambiguate pairs of local variables in methods that are potentially involve in conflicting field reads / writes. For variables `x` and `y` and field `f`, we will query `x` and `y` if we see writes to both `x.f` and `y.f` or a write (read) of `x.f` and a read (write) of `y.f`. This information can be useful for a variety of optimizations, including eliminating redundant loads and dead stores [FKS00].

For the SPEC benchmarks, we checked *virtcall* and *localalias* queries for the *hot methods* of each benchmark, those methods that execute frequently at runtime. We

found the hot methods by running the benchmarks through Jikes RVM [AAB+00], and observing which methods get recompiled with the optimizing compiler (at any optimization level) in its adaptive optimization system. This experiment reflects the queries likely to be raised by an optimizing JIT compiler. To simulate how inlining may affect our analysis results, we modified our graph to inline all getter (e.g. `Obj getFoo() { return this.foo; }`) and setter methods. This transformation essentially adds context-sensitivity for getter and setter methods, and possibly makes analysis more difficult for RegularPT, since there are more putfield and getfield statements and potentially more invalid match edges. We also ran the *virtcall* client for all virtual calls in the program (including the Java libraries) with no inlining, to reflect an IDE client where a developer wishes to navigate to the invoked method for some virtual call.

For the *virtcall* client, a query is positively answered (see §4.1.2) when the points-to analysis shows that the call has 0 or 1 targets. A virtual call can have 0 targets if it resides in a method that is included in the initial call graph but that the points-to analysis can prove is dead. The *localalias* client actually raises two points-to queries, as it is checking if two variables can point to some common object. Together, these queries are positively answered if they show that the queried variables cannot be aliased. Handling the paired queries of *localalias* in RefinedRegularPT required minor, straightforward modifications to its refinement loop.

When measuring the precision of our algorithms, we used field-sensitive Andersen's analysis as a "gold standard," *i.e.*, we measured how much of the precision of computing field-sensitive Andersen's could be obtained quickly by our demand-driven algorithms. In the rest of this section, we refer to the set of queries positively answered by field-sensitive Andersen's as the *feasible queries*, since these are the only queries that our demand algorithms can hope to positively answer. Table 4.3 shows the total number of virtual calls in our benchmark (excluding those resolvable

Benchmark	Virt	FeasVirt
`soot`	2812	1051
`compress`	5428	1801
`jess`	5540	1861
`raytrace`	5438	1803
`db`	5450	1819
`javac`	6334	1952
`mpeg`	5451	1800
`jack`	6022	2370
`sablecc`	6101	1898
`jedit`	7480	2612

Table 4.3: Number of virtual calls unresolvable by types in each benchmark (the Virt column), and the number of such calls resolvable by field-sensitive Andersen's (the FeasVirt column).

with types alone), and the number of those that are feasible queries (virtual calls that field-sensitive Andersen's resolved). The exclusion of calls resolvable with types alone makes the field-sensitive Andersen's analysis look less precise than in previous work [LH03], since we exclude many easy queries. Also note that some of the calls unresolved by field-sensitive Andersen's actually have multiple targets at runtime; these calls are unresolvable by any analysis without more precision (*e.g.*, context sensitivity).

Table 4.4 gives data on our queries in hot methods. Although the number of queries raised is small, their importance is potentially very high since they all occur in hot methods. For example, if an alias query is positively answered, it may allow for a load to be eliminated in frequently executing code, which could have a significant impact on performance.

Benchmark	Hot	Virt	FeasVirt	Alias	FeasAlias
`compress`	7	0	0	0	0
`jess`	28	9	3	5	0
`raytrace`	23	4	0	6	4
`db`	4	5	5	9	0
`javac`	95	115	30	68	10
`mpeg`	45	2	0	1	0
`jack`	22	12	9	3	0

Table 4.4: Information on *virtcall* and *localalias* queries in hot methods. Hot gives the number of hot methods. Virt gives the number of virtual calls in hot methods, and FeasVirt gives the number of those calls that can be resolved by field-sensitive Andersen's (the number of feasible queries). Alias and FeasAlias are analogous, but for *localalias* queries. We did not collect hot method information for the other three benchmarks because our experimental infrastructure did not support it.

4.4.2 Experimental Results

Precision Table 4.5 shows the results of measuring the precision of our algorithms for the *virtcall* client. The table shows the percentage of feasible *virtcall* queries that an intraprocedural field-based analysis, RegularPT (a field-based analysis), and RefinedRegularPT also positively answered. RefinedRegularPT can take very long time to answer some queries, so we timed each query out at 5 seconds, well above the tolerable time budgets of our target clients. RegularPT positively answered more than 89% of feasible queries in all cases, and more than 96% if restricted to code that could not be proven dead by the analysis (since it contains a virtual call with 0 targets). These results are consistent with previous work studying field-based analysis [LPH01, LH03]. RefinedRegularPT could answer nearly all feasible queries. We show below that nearly all of the precision of RegularPT and RefinedRegularPT is preserved under early termination of queries. The purely intraprocedural field-based analysis did much worse than RegularPT, showing that virtual calls that cannot be resolved with the type system usually cannot be resolved with a purely local analysis.

Benchmark	Intra (Live)	Reg (Live)	RefReg (Live)
`soot`	18.4 (16.0)	94.1 (98.5)	96.9 (99.8)
`compress`	26.0 (23.1)	89.1 (96.4)	93.7 (98.9)
`jess`	25.4 (22.5)	89.4 (96.6)	93.9 (99.0)
`raytrace`	26.1 (23.1)	89.1 (96.4)	93.7 (98.9)
`db`	25.7 (22.7)	89.3 (96.5)	93.7 (98.9)
`javac`	25.3 (22.3)	89.9 (96.7)	94.1 (98.8)
`mpeg`	26.0 (23.1)	89.1 (96.4)	94.4 (98.9)
`jack`	27.0 (25.1)	91.8 (97.5)	95.2 (99.2)
`sablecc`	23.9 (20.6)	89.7 (96.3)	93.9 (98.8)
`jedit`	21.7 (19.0)	92.7 (99.1)	97.2 (99.9)

Table 4.5: RegularPT and RefinedRegularPT have nearly the precision of field-sensitive Andersen's. The table gives the percentage of *virtcall* queries positively answered by an intraprocedural field-based analysis (the Intra column), RegularPT (the Reg column), and RefinedRegularPT with a 5 second time limit per query (the RefReg column), as a percentage of those answered positively by field-sensitive Andersen's. The parenthesized Live numbers indicate the result if limited to queries in code that cannot be proven dead by the points-to analysis.

Figure 4.8 shows some code adapted from the `jedit` benchmark that illustrates why RegularPT had nearly the precision of field-sensitive Andersen's. Consider a query to resolve the call to `remove()` on the `propTable` variable in `setProperty()`; possible targets are in `Hashtable` or one of its subclasses. RegularPT handles the query by immediately traversing across the incoming match edge from the source of the putfield[*properties*] edge corresponding to the field write in the `Buffer` constructor. It then finds a new edge from a `Hashtable` abstract location and resolves the call to the implementation of `remove()` in the `Hashtable` class.

This pattern of a field being written only once in a constructor or other initialization method occurs frequently in Java programs, and RegularPT handles it well, as it immediately traverses to the write upon encountering any read of the field. In general, the precision of RegularPT for resolving virtual calls is less than that of a field-sensitive algorithm only when the algorithm encounters a field read `x = y.f`

```
class Buffer {
  private Hashtable properties;

  public Buffer() {
    this.properties = new Hashtable();
  }

  public void setProperty(String name,
                          Object val) {
    Hashtable propTable = this.properties;
    propTable.remove(name);
    ...
    propTable.put(name,val);
  }
}
```

Figure 4.8: A typical example where RegularPT succeeds, but FullFS does too much work, derived from code in the `jedit` benchmark.

such that objects of multiple types are written into the f field (and such types lead to multiple targets for the call), and not all such objects can be written into y's f field. Such polymorphic uses of fields occur relatively rarely, and hence the precision of RegularPT for resolving virtual calls is close to that of field-sensitive Andersen's.

Precision under early termination We evaluated the precision of RegularPT, RefinedRegularPT, and FullFS under early termination with varying time budgets, to simulate the strict time constraints of IDEs and JIT compilers. Recall from §4.1.2 that under early termination, if a points-to analysis exceeds some time budget for answering a query for variable x, the analysis is terminated and the client is told that x can point to any location. To simplify implementation, instead of using actual timeouts to enforce budgets, we limit the number of nodes that can be traversed by a particular query. Note that we count a node as traversed each time it is removed from the worklists seen in Figure 4.6 and Figure 4.7; RefinedRegularPT can visit a node multiple times, and each visit is counted against the traversal budget.

To study behavior under early termination, we computed the cumulative distribu-

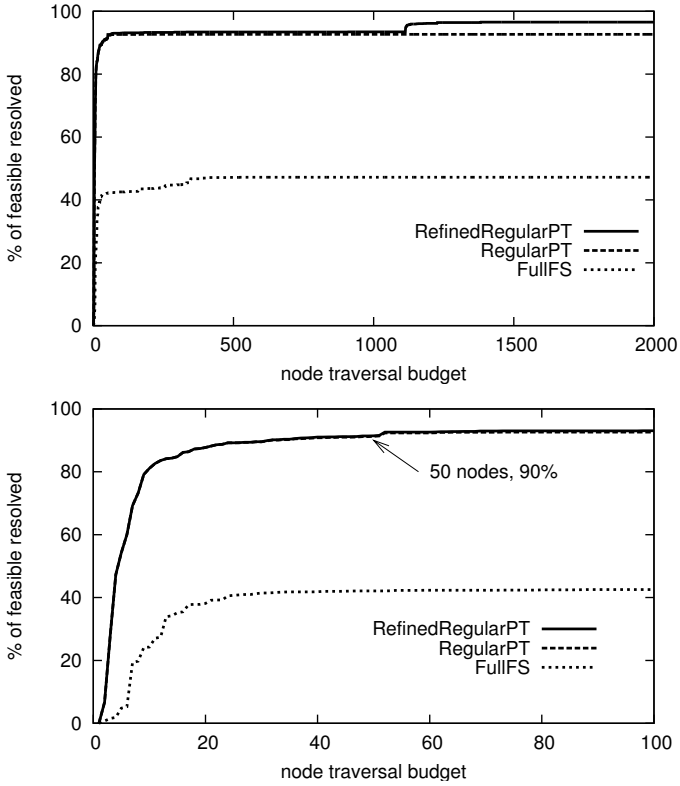


Figure 4.9: RegularPT and RefinedRegularPT performed very well under early termination. We give cumulative distribution of percentage of feasible queries positively answered vs. node traversal budget for the *virtcall* client on `jedit`, for all three algorithms. The top graph shows the distribution from 0 to 2000 nodes traversed, while the bottom graph focuses on 0 to 100 nodes traversed. The distributions for other benchmarks look very similar.

tion of positively answered queries vs. node traversal budget for each of our clients, benchmarks, and algorithms. Figure 4.9 shows the cumulative distributions for `jedit`, using the *virtcall* client to query all virtual calls in the program and library; the distributions for other benchmarks look very similar. The vertical axis is the percentage of feasible queries that were positively answered, and the horizontal axis is the amount of allowed traversal. Recall from Table 4.5 that the maximum possible percentage that RegularPT can reach is 92.7%, and 97.2% for RefinedRegularPT. FullFS should reach 100% if allowed enough traversal.

RegularPT and RefinedRegularPT both performed very well under early termination, retaining most of their precision even under tight time budgets. Figure 4.9 shows that even with a 50 node traversal limit (where queries take 2ms or less), both algorithms positively answered more than 90% of feasible queries. RegularPT and RefinedRegularPT behaved similarly since the first iteration of RefinedRegularPT is exactly RegularPT, and it therefore positively answered all queries that RegularPT did with the same amount of traversal. RegularPT reached its maximum percentage of 92.7% with 182 nodes of traversal; for `jedit`, traversing more than this amount is pointless for RegularPT. A similar result was seen across benchmarks, with the largest amount of traversal required to get all positive answers with RegularPT being 259 nodes for `javac`. Traversing 250 nodes takes under 5ms with our untuned implementation.

RefinedRegularPT resolved many of the queries that RegularPT cannot as the traversal limits grow, reaching 96.5% of feasible queries with a traversal budget of 1250 nodes. Traversing 1250 nodes took 20ms or less with our implementation. ExhaustiveFS (described in Table 4.1) took almost 30 seconds to analyze `jedit`, in which time RefinedRegularPT could answer 1500 queries with a 1250 node budget. Therefore, RefinedRegularPT could be very useful in an application that required more precision than RegularPT and only needed pointer information for a subset of the program, perhaps constructing a call graph for part of the libraries to aid in program understanding.

FullFS did not perform well under early termination. The plateau for FullFS was reached at a 522 node limit, and at this point it only positively answered 47.2% of feasible queries. The algorithm required more than 30000 nodes to positively answer any more queries, and many queries required several hundred thousand nodes of traversal, taking more than 10 seconds of analysis time (sometimes longer than the time required to run ExhaustiveFS).

Budget	Benchmark	Reg	RefReg	FullFS
50 nodes	soot	93.7	93.7	46.6
	compress	88.7	89.7	41.6
	jess	89.1	89.9	41.0
	raytrace	88.8	89.6	41.8
	db	88.9	89.7	42.3
	javac	88.9	89.3	42.5
	mpeg	88.8	89.0	41.7
	jack	91.5	92.2	44.2
	sablecc	89.4	89.6	41.0
	jedit	91.4	91.6	42.1
1250 nodes	soot	94.1	95.9	55.7
	compress	89.1	92.9	50.3
	jess	89.4	93.2	49.3
	raytrace	89.1	93.0	50.5
	db	89.3	93.0	50.9
	javac	89.9	93.3	51.9
	mpeg	89.1	92.9	50.3
	jack	91.8	94.6	59.3
	sablecc	89.7	92.1	49.9
	jedit	92.7	96.4	47.2

Table 4.6: Precision of the demand-driven algorithms with traversal budgets of 50 nodes and 1250 nodes. The columns give the percentage of feasible *virtcall* queries positively answered by RegularPT (the Reg column), RefinedRegularPT (the RefReg column), and FullFS (the FullFS column).

Benchmark	FeasVirt	Reg	FullFS
`jess`	3	3	2
`db`	5	5	5
`javac`	30	30	15
`jack`	9	9	9

Table 4.7: Results for *virtcall* queries in hot methods, showing that RegularPT positively answers the same number of queries as field-sensitive Andersen's. The FeasVirt column gives the number of feasible queries (repeated from Table 4.4), the Reg column the number resolved by RegularPT with a 250 node traversal budget, and the FullFS column the number resolved by FullFS with a 500 node traversal budget.

Benchmark	FeasAlias	Reg	FullFS
`raytrace`	4	4	4
`javac`	10	10	1

Table 4.8: Results for *localalias* queries in hot methods, showing RegularPT matching field-sensitive Andersen's. FeasAlias gives the number of queries resolved by field-sensitive Andersen's (repeated from Table 4.4). Reg gives the number resolved by RegularPT, and FullFS the number resolved by FullFS, with the same traversal budgets used for Table 4.7.

Table 4.6 shows the precision of RegularPT, RefinedRegularPT, and FullFS for the *virtcall* client on all our benchmarks, with traversal budgets of 50 nodes (2ms per query) and 1250 nodes (20ms per query). With a 50 node traversal budget, the precision of RegularPT and RefinedRegularPT was almost identical, 88.8-93.7% of field-sensitive Andersen's. A traversal budget of 1250 nodes allowed RefinedRegularPT to positively answer 1.8-3.8% more queries than RegularPT, relative to field-sensitive Andersen's. FullFS could not answer more than 59.3% of feasible queries with a 1250 node traversal budget, and as with `jedit`, a much larger traversal budget (30000 nodes or more) was required on all benchmarks to substantially improve this precision, well beyond the constraints of our target environments.

Table 4.7 and Table 4.8 give results for *virtcall* and *localalias* queries in hot methods; benchmarks with 0 queries are not listed. For this experiment, we ran RegularPT

with a traversal budget of 250 nodes, and FullFS with a traversal budget of 500 nodes. We gave FullFS a larger budget since in our implementation it seemed to process nodes faster, and with these budgets the time allowed for each query is roughly even for the two algorithms (about 5ms per query). RegularPT positively answered all feasible queries in the hot methods; there is no need to show RefinedRegularPT since its results were exactly the same. FullFS did well on some benchmarks, but even with the larger traversal budget, it could not positively answer many queries in `javac`, the largest of the benchmarks and the one with the most queries. RegularPT traversed 108 nodes or less for all positively answered queries, and therefore could have been run with a smaller traversal budget with no precision penalty.

Our results lead to the slightly counter-intuitive conclusion that within tight time budgets, greater precision can be obtained with an overall less precise algorithm. Consider again the example of Figure 4.8. Answering the *virtcall* query on the `propTable.remove()` call with full field-sensitivity, as FullFS does, leads to extra work, since the object written to the `properties` field in the constructor of `Buffer` will certainly flow to any read of the field, and hence all match edges involving `properties` cannot be removed by refinement. Furthermore, this extra work can increase costs substantially, as RegularPT only requires traversing 3 nodes to answer this query. The example reflects a common case, and illustrates that the greater precision of our algorithms is due to both the relatively small difference in precision between field-based and field-sensitive analysis *and* to the fact that RegularPT often requires very little traversal to answer a query.

We inspected several of the feasible virtual call queries that RefinedRegularPT could not quickly answer by hand, and found that the reasons for their difficulty were independent of field-sensitivity. Some queries involved a parameter of a function nested deeply in the libraries that gets called from many places; avoiding long traversals in these cases would be difficult, and the likelihood of resolving such calls

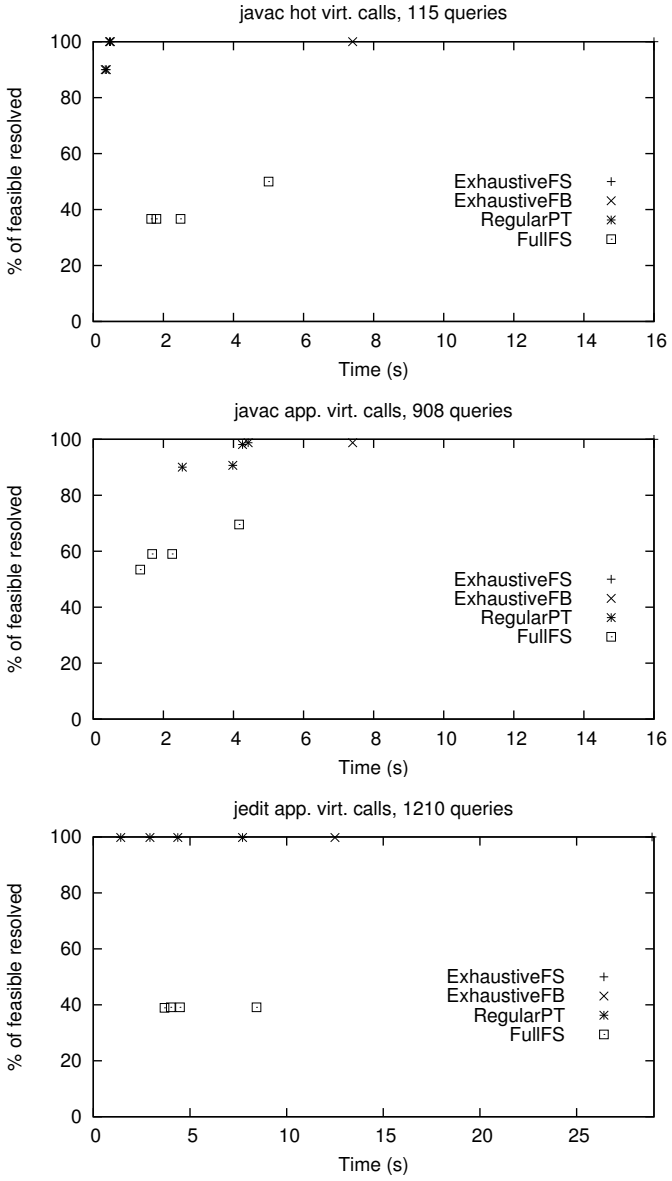


Figure 4.10: Complete time/precision comparison of several demand algorithm configurations and exhaustive algorithms on *virtcall* queries. The x axis is analysis time in seconds, and the y axis is the percentage of feasible queries that were positively answered. For RegularPT and FullFS the data points left to right are for traversal budgets of 50 nodes, 100 nodes, 200 nodes, and 500 nodes. Times are also give for ExhaustiveFB and ExhaustiveFS (described in Table 4.1). From top to bottom, the graphs show virtual calls in hot methods in `javac`, all virtual calls in `javac` application code, and all virtual calls in `jedit` application code.

to a single target is lower. In other cases, imprecision in the conservative call graph lead to traversals of excess methods; refining this call graph on-the-fly is possible (and is done by the algorithm described in Chapter 5), but it is unclear if the extra work involved would be worthwhile with strict time constraints. Context-insensitivity also caused excessive traversal; when the traversal enters a call to `ArrayList.get()`, for example, it exited at all of the many call sites of the method. Context-sensitive analysis could be used to avoid this issue, but again, this requires extra work that may not be beneficial with very tight time budgets.

Performance Since we run our demand algorithms with early termination, time performance can be adjusted depending on how much precision is necessary for the client. Figure 4.10 shows that high precision *and* performance can be obtained with RegularPT through aggressive early termination. We ran experiments where we measured the total time required for RegularPT and FullFS to process all *virtcall* queries in hot methods and all *virtcall* queries in application code (*i.e.*, excluding the Java libraries). The latter experiment simulated some potential program understanding functionality in an IDE where a call graph is built for the application. We compared these total running times to the time required to run ExhaustiveFB and ExhaustiveFS, described in Table 4.1. We show graphs of running time vs. percentage of feasible queries resolved for three representative benchmarks and clients: calls in hot methods in `javac`, calls in application code in `javac`, and calls in application code in `jedit`. In all cases, the precision of a field-based analysis was nearly exactly that of a field-sensitive analysis, so we excluded RefinedRegularPT from the graphs (its performance is almost identical to that of RegularPT).

RegularPT gave significant speedups over ExhaustiveFB in all cases without sacrificing precision. For `javac` virtual calls in hot methods, RegularPT had the same precision as a field-sensitive analysis with a running time of .46 seconds, a 34x speedup over the 16 seconds for ExhaustiveFS and a 16x speedup over the 7.4 sec-

onds for ExhaustiveFB. For virtual calls in application code, speedups over ExhaustiveFS (ExhaustiveFB) ranged from 3.62x (1.68x) for `javac` (the smallest speedup in our benchmarks) to 20.4x (8.8x) for `jedit`. FullFS failed to provide nearly the same precision within the same time budgets given to RegularPT.

The memory consumption of our algorithms was quite reasonable. Given a budget of 250 nodes of traversal, neither RegularPT nor RefinedRegularPT allocated more than 50 kilobytes of memory across our benchmarks, and our implementation could be more memory efficient. In contrast, even with an efficient BDD representation, exhaustive field-sensitive Andersen's analysis takes 23 MB for `javac` and 28 MB for `jedit` [BLQ+03], since all points-to sets need to be represented.

Other Factors We have evaluated our algorithms in a static environment, where all of the benchmark code is available and the graph representation of pointer assignments is built up-front. In a JIT compiler or IDE, such representations may not be readily available. Since our graph representation essentially matches the assignment statements in the program, it can be constructed efficiently. In an IDE, the representation for a method can simply be rebuilt from scratch after its code changes; no complex incremental update is required. Such rebuilding can occur in the background while the user continues working. In a JIT compiler, the graph can be constructed immediately from an intermediate representation. If a method has no intermediate representation because it has only been interpreted, its graph can be constructed at query time, and the cost of building the graph can be factored into early termination heuristics.

Because new code may become available after analysis on our environments (via editing in an IDE or dynamic class loading in a JIT compiler), analysis results may become invalid. If any analysis results are cached in an IDE, they can simply be flushed and the corresponding queries re-run. A JIT compiler presents a greater challenge, since some (possibly running) code may have already been optimized based

on previous analysis results. There are three ways in which dynamic loading of class C can invalidate the results of some query q:

1. C provides a new target for some method invocation that was traversed in answering q. This could affect analysis results by returning some new value that was not previously possible, for example.

2. C calls some existing method m whose parameters were traversed in answering q (since new values could now be passed into m).

3. C has a putfield to a field f, and a getfield on f was encountered when answering q. This new putfield could lead to new match edges that must be considered.

Since our algorithms traverse a representation close to the statements of the program, they could potentially keep track of which statements could be affected by dynamic class loading, and then add appropriate guards to such statements. If a guard later failed due to dynamic class loading, the optimized code could be invalidated, using on-stack replacement [CU91, FQ03] if necessary. See [HDDH07] for an enumeration of the issues related to running pointer analysis in a JIT compiler.

4.5 Java vs. C

Given the effectiveness of our refinement technique for Java points-to analysis, an interesting question is whether the same technique could be used to improve C points-to analysis. Unfortunately, due to C's less restricted constructs for pointer accesses, a straightforward use of our refinement algorithm for C pointer analysis is unlikely to yield good results, since the natural initial approximation would match *any* head read and write. Here we discuss the reasons why our refinement technique cannot easily be applied to C; an effective adaptation is future work.

As formulated previously by Melski and Reps [Rep98], Andersen's analysis for C does not seem to be a balanced parentheses problem.[6] Recall that our refinement technique uses the balanced parentheses of heap reads and writes in Java to construct an initial approximation to be refined as needed (*i.e.*, the graph representation with all possible match edges). For C, the statements most closely related to Java field reads and writes are pointer reads and writes through the * operator, respectively of the form `x = *y` and `*x = y`. However, due to the address-of operator `&`, these reads and writes can create a points-to relation without being balanced, for example in the following program:

```
z = &p;
y = &z;
x = *y;
```

With this program, we have the points-to sets $pt(z) = p$, $pt(y) = z$, and $pt(x) = p$. Hence, in contrast to Java, a points-to relationship ($pt(x) = p$) was caused by the heap read `x = *y` without a balancing heap write.

Though recent work presents an alternate balanced-parentheses formulation of C points-to analysis, our initial match edge approximation would likely be too coarse to use as an effective basis for refinement. The lack of balanced parentheses in the Melski-Reps formulation of Andersen's analysis is not a fundamental problem; recently, Zheng and Rugina have presented an alternate formulation of C alias analysis with balanced parentheses [ZR07]. However, in this formulation a write to the heap could potentially be balanced by *any* read from the heap. So, in the initial graph representation for refinement, the source of each `*z = w` statement would be connected by a match edge to the sink of each `x = *y` statement. The * operator is often used to access many disparate data structures in a C program, in which case this initial

[6] Our discussion ignores pointer arithmetic, as does most previous work on flow-insensitive C points-to analysis.

approximation would likely be too coarse to yield good results. In Java, field names can be used to initially match a much smaller set of heap read and write pairs, since the type system ensures that different object fields are not memory aliases. In order to apply our refinement-based technique to C points-to analysis, an initial approximation of aliasing better than what can be found syntactically would be needed; further investigation of this issue is future work.

4.6 FullFS Details

Here we give the details of FullFS, an algorithm employing the techniques used by the demand-driven points-to analysis algorithm of Heintze et al. [HT01a]. We present inference rules for FullFS in Figure 4.11, showing which points-to queries must be raised, given an initial set of queries and the statements in a particular program. The left column presents the statement types relevant to Java points-to analysis, and the right column gives the inference rules that can be instantiated when the corresponding statement is present. We adopt the notation of [HT01a], where $x \hookrightarrow .$ means that a query has been raised to find what x points to. So, rule (1) states that if there is a points-to query for p and a statement `p = new T()`, then add $p \hookrightarrow o_a$ to the points-to relation (`a` is some label for the statement). Rule (3) states that given statement `p = r` and a query $p \hookrightarrow .$, we must add the query $r \hookrightarrow .$ to see what r points to. Once we discover $r \hookrightarrow o_1$, rule (4) will add $p \hookrightarrow o_1$.

Handling getfield and putfield statements is more complex. Given a getfield statement `p = r.f` and a query $p \hookrightarrow .$, we must find the values written into the f field of abstract locations that r can point to. Rule (6) introduces the query $r \hookrightarrow .$ to find what r points to. Given that $r \hookrightarrow o_1$, introduces the pointed-to-by query $. \hookrightarrow o_1(f_w)$ to find what variables point to o_1. In [HT01a], there are queries of the form $. \hookrightarrow x$ for variables x, since there can be pointers to variables in C through the `&` operator. In

Java, we only have such pointed-to-by queries for allocation sites. The parenthesized f_w gives the *reason* for this query; the form is a field name f with subscript w for write and r for read. Here, we are looking for writes to field f, or f_w, since the getfield statement reads from field f. We keep these reasons to avoid unnecessary work (see below). Rule (8) says that once the analysis has discovered that $r \hookrightarrow o_1$ and $o_1.f \hookrightarrow o_2$, we can add the fact $p \hookrightarrow o_2$.

Putfield statements `p.f = r` are handled as follows. If we have $p \hookrightarrow o_1$ and $. \hookrightarrow o_1(f_w)$, the statement is relevant since we must discover what is written into the f field of o_1. In such a case, rule (10) introduces the query $r \hookrightarrow .$, and once we find $r \hookrightarrow o_2$, rule (11) adds $o_1.f \hookrightarrow o_2$. This latest points-to fact allows rule (8) to trigger for the appropriate getfield statements, flowing o_2 across the field. Notice that if we did not have the reason f_w and just had pointed-to-by queries of the form $. \hookrightarrow o_1$, then when we encounter a statement `p.g = r`, we would still introduce $r \hookrightarrow .$ to be sound, since a read from the g field may have been encountered. Keeping the reason ensures we only do work for relevant field writes.

Rules (2), (5), (9), (12), and (13) concern properly handling the pointed-to-by queries. Rule (2) says that if there is a pointed-to-by query $. \hookrightarrow o_a(any)$ (where (any) means for any reason), we must add $p \hookrightarrow o_a$; rule (5) similarly handles assignments. Rules (9), (12), and (13) handle the case when we have $. \hookrightarrow o_2(any)$ and o_2 is written into a field. For statement `p.f = r`, rule (12) adds the query $p \hookrightarrow .$, since we must which abstract locations have o_2 written into their f field. When we find such an abstract location o_1, rule (13) adds the fact $o_1.f \hookrightarrow o_2$, and introduces the pointed-to-by query $. \hookrightarrow o_1(f_r)$ since we are looking for reads of f. Rule (9) handles the case when we find a statement `p = r.f` that reads f from o_1, adding the fact $p \hookrightarrow o_2$.

Figure 4.12 gives some sample code and partial derivation of the fact $x \hookrightarrow o_6$, assuming $x \hookrightarrow .$ is the initial query. Uses of inference rules are labeled with the corresponding statement. Notice that all leaves of the derivation will be $x \hookrightarrow .$, as

statement	FullFS inference rules
`a: p = new T`	$\dfrac{p \hookrightarrow .}{p \hookrightarrow o_a}$ (1) $\dfrac{. \hookrightarrow o_a(any)}{p \hookrightarrow o_a}$ (2)
`p = r`	$\dfrac{p \hookrightarrow .}{r \hookrightarrow .}$ (3) $\dfrac{p \hookrightarrow . \qquad r \hookrightarrow o_1}{p \hookrightarrow o_1}$ (4) $\dfrac{r \hookrightarrow o_1 \qquad . \hookrightarrow o_1(any)}{p \hookrightarrow o_1}$ (5)
`p = r.f`	$\dfrac{p \hookrightarrow .}{r \hookrightarrow .}$ (6) $\dfrac{p \hookrightarrow . \qquad r \hookrightarrow o_1}{. \hookrightarrow o_1(f_w)}$ (7) $\dfrac{\begin{array}{c} p \hookrightarrow . \qquad r \hookrightarrow o_1 \\ o_1.f \hookrightarrow o_2 \end{array}}{p \hookrightarrow o_2}$ (8) $\dfrac{\begin{array}{cc} r \hookrightarrow o_1 & . \hookrightarrow o_1(f_r) \\ o_1.f \hookrightarrow o_2 & . \hookrightarrow o_2(any) \end{array}}{p \hookrightarrow o_2}$ (9)
`p.f = r`	$\dfrac{p \hookrightarrow o_1 \qquad . \hookrightarrow o_1(f_w)}{r \hookrightarrow .}$ (10) $\dfrac{\begin{array}{cc} p \hookrightarrow o_1 & . \hookrightarrow o_1(f_w) \\ r \hookrightarrow o_2 & \end{array}}{o_1.f \hookrightarrow o_2}$ (11) $\dfrac{r \hookrightarrow o_2 \qquad . \hookrightarrow o_2(any)}{p \hookrightarrow .}$ (12) $\dfrac{\begin{array}{cc} r \hookrightarrow o_2 & . \hookrightarrow o_2(any) \\ p \hookrightarrow o_1 & \end{array}}{o_1.f \hookrightarrow o_2 \qquad . \hookrightarrow o_1(f_r)}$ (13)

Figure 4.11: Inference rules for FullFS.

```
1  x = y.f;
2  w = new T();
3  y = w;
4  z = w;
5  z.f = p;
6  p = new T();
```

$$\dfrac{\dfrac{x \hookrightarrow .}{y \hookrightarrow .}\,1 \quad \dfrac{\dfrac{y \hookrightarrow .}{w \hookrightarrow .}\,3 \quad w \hookrightarrow o_2}{y \hookrightarrow o_2}\,2 \quad \begin{array}{c} x \hookrightarrow . \\ \vdots \\ o_2.f \hookrightarrow o_6 \end{array}}{x \hookrightarrow o_6}\,1$$

Figure 4.12: Example code and partial derivation for FullFS.

expected.

Pseudocode for FullFS appears in Figure 4.13. This algorithm is Heintze and Tardieu's two-set algorithm [HT01a] adapted for field-sensitive Java points-to analysis. We adopt their notation for points-to sets [HT01a]: X represents the points-to set of variable x, and X' represents its "tracked set," defined as $\{o : o \in X \wedge . \hookrightarrow o(any)\}$. Similarly, we denote the points-to set and tracked set of each abstract location field $o.f$ as $O.f$ and $O.f'$ respectively. The algorithm iterates over graph edges, storing the results of points-to and pointed-to-by queries in point-to sets and tracked sets respectively, following the inference rules of Figure 4.11.

Certain basic optimizations must be implemented to obtain a scalable implementation of FullFS. First, worklists should be used to iterate only over necessary graph edges, a standard technique for such analyses. The points-to set data structure should have efficient methods for iteration and adding all elements from another set; we re-used SPARK's hybrid points-to set representation [LH03]. Finally, incrementalization (*i.e.*, only propagating newly added abstract locations in each iteration) can also greatly improve performance, as seen in other work [LH03, WL04].

Due to excessive caching, FullFS is not an ideal basis for a context-insensitive refinement algorithm. FullFS could easily be adapted for the purposes of refinement by adding appropriate handling of match edges. However, FullFS caches points-to

```
FullFS(v)
 1  Add query v_s ↪ .
 2  repeat
 3          for each edge x ←new— o
 4              do if x ↪ . then add o to X
 5                 if . ↪ o(any) then add o to X'
 6          for each edge x ←assign— y
 7              do if x ↪ .
 8                    then add y ↪ .
 9                         add all of Y to X
10                 if Y' ≠ ∅ then add all of Y' to X'
11          for each edge x ←getfield[f]— y
12              do if x ↪ .
13                    then add y ↪ .
14                         for each o ∈ Y
15                             do add . ↪ o(f_w)
16                                add all of O.f to X
17                 for each o ∈ Y'
18                     do if . ↪ o(f_r) then add all of O.f' to X'
19          for each edge x ←putfield[f]— y
20              do for each o ∈ X'
21                     do if . ↪ o(f_w)
22                           then add y ↪ .
23                                add all of Y to O.f
24                 if Y' ≠ ∅
25                    then add x ↪ .
26                         for each o ∈ X
27                             do add all of Y' to O.f'
28                                add . ↪ o(f_r)
29     until no change
30  return V
```

Figure 4.13: Pseudocode for the FullFS algorithm.

sets for all queried variables. We found that for our context-insensitive refinement technique, this caching imposes and memory overhead without improved running times. RegularPT and RefinedRegularPT do not cache intermediate results and provide better performance. For context-sensitive refinement, caching of intermediate results becomes much more important, and hence our context-sensitive refinement algorithm

is based on FullFS, as discussed in Chapter 5.

4.7 Adaptation of Tabulation Algorithm

While arbitrary CFL-reachability problems can be solved in worse-cast $O(N^3/\log N)$ time [Cha06], the *tabulation algorithm* of Reps *et al.* [RHSR94, RHS95] can run asymptotically faster for reachability over a balanced parentheses language. Given our formulation of field-sensitive Andersen's analysis for Java as a balanced-parentheses problem, a natural question arises: can the tabulation algorithm be adapted to create a fast Java points-to analysis? Here, we show that the answer to the question is mixed. In §4.7.1, we give an adapted version of the tabulation algorithm for solving L_{F}-reachability and show that it has worst-case $O(NE)$ complexity for realistic Java programs, asymptotically faster than the standard $O(N^3)$ worst-case bound for Andersen's analysis. Then, in §4.7.2, we discuss the reasons why existing propagation-based implementations of Andersen's analysis are likely to be faster than the tabulation algorithm in practice.

4.7.1 Tabulation Algorithm for Points-To Analysis

Here we present an adaptation of the Reps-Horwitz-Sagiv tabulation algorithm [RHSR94, RHS95] for computing all-pairs L_{F}-reachability, *i.e.*, field-sensitive Andersen's analysis for Java. We first describe the key idea of the algorithm, namely the efficient addition of *vmatch* edges to the graph. In contrast to match edges, *vmatch* edges are only present in the graph when an *alias*-path exists between the base pointers of the corresponding field accesses. On a graph with *vmatch* edges, L_{F}-reachability can be computed more cheaply through reachability over a *regular* language. We then give pseudocode for the adapted tabulation algorithm, and finally

discuss its worst-case complexity.

Algorithm Idea The key idea of the tabulation algorithm is to add a *summary edge* between each parenthesis source and sink connected by a path with balanced parentheses [RHS95]. Balanced-parentheses reachability problems over the input graph can be posed as *regular* reachability problems over a graph with summary edges. The same results are obtained since whenever a balanced path exists from an open to close parenthesis in the input graph, a summary edge is present in the modified graph. Say that the balanced parentheses production in the context-free language is $S \rightarrow (_i\ S\ )_i$ and that the summary edge is labeled s. In essence, after the summary edges are added, the grammar production can be modified to be $S \rightarrow \mathsf{s}$. The modified reachability problem is regular since the parentheses have been eliminated from the grammar.

For points-to analysis, recall again the definition of a *flowsTo*-path in L_{F}:

$$\mathit{flowsTo} \rightarrow \mathsf{new}\ \ (\ \mathsf{assign}\ \mid\ \mathsf{putfield}[f]\ \ \mathit{alias}\ \ \mathsf{getfield}[f]\ )^*$$

For L_{F}, we are interested in adding summary edges from $\mathsf{putfield}[f]$ edge sources and $\mathsf{getfield}[f]$ edge sinks, the same nodes connected by match edges for our refinement algorithm. However, while match edges are added for each pair of accesses of the same field, summary edges are only added when those accesses are connected by an *alias*-path, as in the above grammar. We call these summary edges *vmatch edges*, since they can be thought of as match edges for which we have verified the existence of a connecting *alias* path. More formally, we have $\mathsf{putfield}[f]\ \mathit{alias}\ \mathsf{getfield}[f] \Leftrightarrow \mathit{vmatch}$, while match edges had a weaker guarantee, $\mathsf{putfield}[f]\ \mathit{alias}\ \mathsf{getfield}[f] \Rightarrow \mathsf{match}$.

Given a graph with all possible *vmatch* edges, L_{F}-reachability can be computed via reachability over a regular language. The key invariant of *vmatch* edges is that there is an edge $x \xrightarrow{\mathit{vmatch}} y$ in the graph iff there is a path from x to y labeled

$\mathsf{putfield}[f]$ *alias* $\mathsf{getfield}[f]$. Given this invariant, we can construct a regular language R_{VF} such that x is R_{VF}-reachable from o iff x is L_{F}-reachable from o. The grammar for R_{VF} is quite similar to that of R_{F} from §4.2.1:

$$\mathit{flowsToVReg} \rightarrow \mathsf{new}\ \ (\ \mathsf{assign}\ \mid\ \mathit{vmatch}\ )^*$$

While R_{F}-reachability over-approximated L_{F}-reachability, R_{VF}-reachability is equivalent to L_{F}-reachability because of the use of *vmatch* edges. We shall show shortly that both the addition of *vmatch* edges and the computation of all-pairs R_{VF}-reachability can be done in $O(NE)$ time, yielding an $O(NE)$ algorithm for Andersen's analysis for Java.

Algorithm Idea Given a graph with all *vmatch* edges, all-pairs R_{VF}-reachability can be computed in $O(NE)$ time since R_{VF} is a regular language (as discussed in §3.1). Here, we show how *vmatch* edges can be added to the graph by the tabulation algorithm in $O(NE)$ time in practice, yielding an overall efficient algorithm for Andersen's analysis.

At a high-level, the algorithm inserts a *vmatch* edge whenever it finds an *alias*-path from a sink v of a $\mathsf{putfield}[f]$ edge and a source w of a matching $\mathsf{getfield}[f]$ edge. The algorithm looks for such paths with purely *regular* searches starting from each such node v. The label of each explored path must meet the following regular expression R_{exp}:

$$R_{exp} = (\overline{\mathsf{assign}}|\overline{\mathit{vmatch}})^*\ \ \overline{\mathsf{new}}\ \ \mathsf{new}\ \ (\mathsf{assign}|\mathit{vmatch})^*$$

We obtain R_{exp} by first substituting *vmatch* into the *flowsTo* production based on the

invariant $\mathsf{putfield}[f]$ *alias* $\mathsf{getfield}[f] \Leftrightarrow$ *vmatch*:

$$\begin{aligned} \mathit{flowsTo} &\rightarrow \mathsf{new}\ (\,\mathsf{assign} \mid \mathsf{putfield}[f]\ \mathit{alias}\ \mathsf{getfield}[f]\,)^* \\ &\rightarrow \mathsf{new}\ (\,\mathsf{assign} \mid \mathit{vmatch}\,)^* \end{aligned}$$

Then, given the production $\mathit{alias} \rightarrow \overline{\mathit{flowsTo}}\ \ \mathit{flowsTo}$, we substitute for *flowsTo* and $\overline{\mathit{flowsTo}}$ appropriately, yielding R_{exp}. Once all *vmatch* edges are present in the graph, all nodes connected by an *alias*-path will also be connected by a path matching R_{exp}, *i.e.*, $\mathit{alias} \Leftrightarrow R_{exp}$.

Initially, no *vmatch* edges are present, so the algorithm only finds paths whose labels meet the regular expression R_1:

$$\overline{\mathsf{assign}}^*\ \ \overline{\mathsf{new}}\ \ \mathsf{new}\ \ \mathsf{assign}^*$$

As *vmatch* edges are added, the algorithm includes those edges in its searches, eventually discovering and adding all *vmatch* edges as desired.

Pseudocode Figure 4.14 gives pseudocode for our adapted version of the tabulation algorithm. The algorithm of Figure 4.14 discovers paths meeting the above regular expression R_{exp} using worklists and path-edge sets. *Path edges* record the graph exploration already performed by the algorithm, allowing for avoiding repeated work. The path-edge sets maintain the following invariant: when *backPathEdges* (*forwPathEdges*) contains an edge $x \rightarrow y$, then an R_b-labeled (R_f-labeled) path has been found from x to y, where

$$\begin{aligned} R_b &= (\overline{\mathsf{assign}} | \overline{\mathit{vmatch}})^*\ \ (\overline{\mathsf{new}} | \epsilon) \\ R_f &= (\overline{\mathsf{assign}} | \overline{\mathit{vmatch}})^*\ \ \overline{\mathsf{new}}\ \ (\mathsf{new} | \epsilon)\ \ (\mathsf{assign} | \mathit{vmatch})^* \end{aligned}$$

Essentially, the *backPathEdges* set records the exploration of partial R_{exp}-paths that

forwPathEdges, *backPathEdges*: Set of Edge
forwWorklist, *backWorklist*: Set of Edge

AddVMatches()
1 **for** each w that is the sink of a putfield$[f]$ edge
2 **do** add $w \rightarrow w$ to *backWorklist*
3 add $w \rightarrow w$ to *backPathEdges*
4 **while** *backWorklist* $\neq \emptyset \vee$ *forwWorklist* $\neq \emptyset$
5 **do while** *backWorklist* $\neq \emptyset$
6 **do** remove $v \rightarrow w$ from *backWorklist*
7 **for** each edge $w \xleftarrow{\text{assign}} x$ and $w \xleftarrow{vmatch} x$
8 **do** Propagate$(v \rightarrow x, backPathEdges, backWorklist)$
9 **for** each edge $w \xleftarrow{\text{new}} o$
10 **do** Propagate$(v \rightarrow o, forwPathEdges, forwWorklist)$
11 **while** *forwWorklist* $\neq \emptyset$
12 **do** remove $v \rightarrow w$ from *forwWorklist*
13 **for** each edge $w \xrightarrow{\text{new}} x$, $w \xrightarrow{\text{assign}} x$ and $w \xrightarrow{vmatch} x$
14 **do** Propagate$(v \rightarrow x, forwPathEdges, forwWorklist)$
15 AddVMatchesForNodes(v, w)

AddVMatchesForNodes(v, w)
1 **for** each edge $x \xrightarrow{\text{putfield}[f]} v$
2 **do for** each edge $w \xrightarrow{\text{getfield}[f]} y$
3 **do** add edge $x \xrightarrow{vmatch} y$ if needed
4 **for** each $a \rightarrow y \in backPathEdges$
5 **do** Propagate$(a \rightarrow x, backPathEdges, backWorklist)$
6 **for** each $b \rightarrow x \in forwPathEdges$
7 **do** Propagate$(b \rightarrow y, forwPathEdges, forwWorklist)$

Propagate$(e, pathEdges, worklist)$
1 **if** $e \notin pathEdges$
2 **then** add e to *pathEdges*
3 add e to *worklist*

Figure 4.14: Tabulation algorithm of Reps *et al.* [RHSR94, RHS95], adapted for Java points-to analysis.

only include barred edges, while the *forwPathEdges* set records explored partial (or full) R_{exp}-paths with both barred and non-barred edges. The check at line 1 in

PROPAGATE ensures that the algorithm only attempts to complete such partial paths once. The worklists maintain path edges for which further exploration from the sink node is required.

As the algorithm adds vmatch edges to the graph, traversals previously blocked may be able to continue using the new edges. To "restart" such traversals, whenever a vmatch edge is added, we check for traversals that reached either side of the newly added edge, and appropriately update the worklists (line 5 and line 7 in ADDVMATCHESFORNODES in Figure 4.14).

Complexity To determine the complexity of this algorithm, we observe the key invariant: the number of visits to each edge (whether present in the original graph or inserted later as a *vmatch* edge) is bounded by the number of nodes inserted into the worklist in line 2 of ADDVMATCHES in Figure 4.14 (these are the sink nodes of $\mathsf{putfield}[f]$ edges). The bound is obvious once we realize that each node w can be the sink of a patch edge in the worklist at most as many times as the number of getfield sink nodes. Note that vmatch edges, once inserted, are treated exactly like the originally present edges. Also, we are assuming suitable data structures that require a constant amount of work for each edge traversed.

So, for each $\mathsf{putfield}[f]$ sink node, we perform $O(S + V)$ work, where S is the number of assignment statements in the program and V is the number of *vmatch* edges added. In the worst case, there is a *vmatch* edge between every matched $\mathsf{getfield}[f]$ and $\mathsf{putfield}[f]$ edge. Hence, V is bounded above by $W = \sum_f G(f)P(f)$, where $G(f)$ is the number of $\mathsf{getfield}[f]$ edges and $P(f)$ is the number of $\mathsf{putfield}[f]$ edges.

Since there are at most N getfield sink nodes, where N is the number of nodes in the input graph or variables in the program, the addition of *vmatch* edges has overall worst-case time complexity $O(N(S+W))$. As previously discussed, computing all-pairs R_{VF}-reachability on the resulting graph requires $O(NE)$ time. Since R_{VF}-reachability is computed on a graph with *vmatch* edges, we have $E = S + W$ in the

worst-case. Therefore, we have a worst-case $O(N(S+W))$ algorithm for field-sensitive Andersen's analysis for Java.

Let us discuss the typical complexity for Java programs. In practice, $S = O(N)$. Similarly, we have observed in practice that while W can get large, typically $V = O(N)$. An intuitive reason for V not growing quadratically is the prevalence of getter and setter methods in Java programs, *i.e.*, methods that respectively wrap reads and writes of fields. When a field f is accessed through a getter and setter, there is a single $\mathsf{getfield}[f]$ and $\mathsf{putfield}[f]$ edge for f, and hence only one possible corresponding *vmatch* edge. If all fields in a program are accessed solely through getter and setter methods (and most are in typical Java programs), then the maximum number of *vmatch* edges will equal the number of fields, which will be $O(N)$ for any realistic Java program. Since $V = O(N)$ and $S = O(N)$ in practice, the tabulation-based points-to analysis runs in $O(N^2)$ time, asymptotically better than the worst-case cubic bound of the standard closure algorithm.

4.7.2 Discussion

The tightest worst-case bound proved for the most efficient existing Java points-to analysis algorithms [LH03, BLQ+03, WL04] is still $O(N^3)$ (to the best of our knowledge),[7] allowing for the tabulation algorithm to be more efficient. Nevertheless, in practice we have found that existing algorithms tend to outperform the tabulation algorithm. Here we discuss how the existing efficient algorithms work, why their worst-case cubic behavior tends not to arise in practice, and why they tend to run faster than the tabulation algorithm.

Efficient existing algorithms for Java points-to analysis are based on *propagation* of points-to sets, like the FullFS algorithm of §4.6. Like the FullFS pseudocode

[7] The aforementioned $O(N^3/logN)$ algorithm [Cha06] has not yet been implemented and shown to scale well in practice.

shown in Figure 4.13, these algorithms maintain points-to sets for each variable and abstract-location field in the program, and then process each assignment statement by propagating abstract locations between the corresponding points-to sets. Efficiency is gained through type filters (*i.e.*, only allowing abstract locations of the appropriate static type into a points-to set), smart iteration ordering over the statements (*i.e.*, processing statements in topological order), and incrementalization, described in more detail in [LH03]. Nevertheless, no worst-case bound better than $O(N^3)$ has been proved for these algorithms as far as we know, so in theory the tabulation algorithm could be more efficient.

Though the tabulation algorithm has better asymptotic complexity than propagation, realistic Java programs do not expose the worst-case cubic behavior of propagation-based algorithms, lessening the opportunity for tabulation to yield a performance advantage. Cubic behavior for a propagation-based algorithm could arise in the following situation. Say a program has N abstract locations (allocation sites) of a class with N different fields, such that each field of each abstract location can point to all N other abstract locations. Then, each of the N abstract locations appears in N^2 points-to sets, which could require $O(N^3)$ work to compute. However, no realistic program would cause this behavior. In particular, Java's type system often limits the set of abstract locations in a points-to set to a small subset of all abstract locations, and points-to analyses can use type filters to keep these sets small. Hence, propagation-based points-to analyses tend not to exhibit cubic behavior in practice.[8]

The Master's thesis of Lhotak [Lho02] shows an empirical comparison between a standard propagation-based algorithm and an "alias edge" algorithm quite similar to the tabulation algorithm. The alias edge algorithm in [Lho02] computes aliasing relationships between base pointers of field accesses rather than keeping points-to sets

[8]It would be an interested theoretical result to prove a tighter worst-case bound for propagation algorithms on realistic Java programs.

for object fields, just like the tabulation algorithm. It differs in using points-to sets for variables rather than path edges to record existing analysis work, a choice that we believe does not make performance worse in practice.[9] The performance comparison in Chapter 5 of [Lho02] confirms the trade-offs discussed above. With type filters enabled, the standard propagation algorithm tends to be slightly faster than the alias edge algorithm. However, without type filters, the size of object-field points-to sets increases dramatically, making the propagation algorithm run out of memory while the alias edge algorithm still terminates. In practice, type filters will always be used with a propagation-based algorithm for Java, meaning that the tabulation algorithm is unlikely to give any performance gain.

[9] In fact, due to the structure of graphs arising from real Java programs, recording points-to sets rather than path edges is likely to be a performance win.

Chapter 5

Context-Sensitive Points-To Analysis

This chapter describes a context-sensitive refinement-based points-to analysis for Java. Our analysis has three types of context sensitivity: (1) filtering out of *unrealizable paths* (see §2.2.4), (2) a *context-sensitive heap abstraction* (see §2.2.5), and (3) a *context-sensitive call graph* (see §2.2.3 and §2.2.4). Previous work [LH06] has shown that all three properties are important for precisely analyzing large programs, *e.g.*, to show safety of downcasts. Existing context-sensitive analyses typically give up one or more of the properties for scalability, as discussed in §1.2.2 and §2.2. Through refinement, our analysis retains all three properties while using orders-of-magnitude less memory than less precise existing analyses. Thus, our refinement-based analysis is the most precise heap analysis available for realistic Java programs.

The rest of this chapter is organized as follows. First, §5.1 gives a high-level overview of our context-sensitive refinement algorithm, similar to the overview of our context-insensitive algorithm in §4.1. Then, §5.2 formulates the context-sensitive result of our algorithm, a decidable approximation of the formulation given in §3.4.4.

§5.3 presents our refinement-based points-to analysis algorithm in detail. Finally, §5.4 gives results from our experimental evaluation of our algorithm.

5.1 Algorithm Overview

This section provides an overview of our context-sensitive refinement-based algorithm in a manner similar to the overview of our context-insensitive algorithm in §4.1. First, we present a simplified version of our analysis formulation in §5.1.1. This simplified formulation shows the essence of the analysis problem, namely reachability over the intersection of two balanced-parentheses languages. Then, we present our context-sensitive refinement algorithm in §5.1.2, an extension that adds context sensitivity to the refinement algorithm of §4.1.4. Finally, §5.1.3 illustrates how the analysis works on a Java code example. We show that in typical programs our refinement explores nested data structures in a hierarchical progression, visiting only a small part of the program to obtain sufficient precision for the client.

5.1.1 Simplified Formulation

Here, we present a simplified formulation of the context- and field-sensitive points-to analysis problem, approximated for decidability. Recall that fully context- and field-sensitive points-to analysis is undecidable (see §3.3). In this section, we skirt the undecidability issue by assuming that all analyzed programs have no recursive method calls. Under this assumption, checking for balanced method call parentheses can be formulated as a *regular* reachability problem, since the number of open call parentheses on any path is bounded. Hence, the problem of checking for both balanced field and method call parentheses requires reachability over the intersection

of a context-free language and a regular language, guaranteed to remain context-free and decidable. We discuss our approximate handling of programs with recursion in §5.2.

The present formulation adds context sensitivity to the simplified formulation of §4.1.3. In addition to the simplifications introduced in §4.1.3, the formulation simplifies context sensitivity by (unsoundly) disallowing partially balanced call parentheses (sound formulations appear in §3.3 and §5.2). Also, we assume the presence of an ahead-of-time call graph, again to make the key aspects of our algorithm clear.

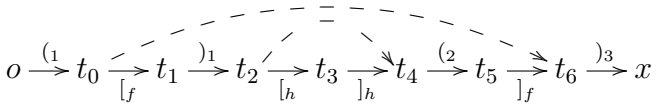
We formulate our analysis as reachability over the intersection of a slightly modified version of L_{sf} (from §4.1.3) and a language for context sensitivity. Let Σ_P be the alphabet of open and close brackets, respectively representing heap writes and reads, and open and close parentheses, representing method call entries and exits:

$$\Sigma_P = \big\{[_f\ ,\ ]_f \mid f \text{ is a field}\big\} \cup \big\{(_i\ ,\ )_i \mid i \text{ is a call site}\big\}$$

Our simplified points-to analysis requires CFL-reachability with language $L_{\mathsf{scf}} = L_{\mathsf{sf}} \cap R_{\mathsf{sc}}$ over Σ_P, with L_{sf} and R_{sc} respectively representing key properties of L_{F} and L_{C} (defined in Chapter 3):

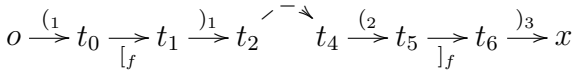
$$L_{\mathsf{sf}} : F \to [_f\ F\ ]_f \mid [_g\ F\ ]_g \mid \ \ldots\ \mid F\ F \mid (_i \mid)_i \mid \ \ldots\ \mid \epsilon$$
$$R_{\mathsf{sc}} : C \to (_i\ C\ )_i \mid (_j\ C\ )_j \mid \ \ldots\ \mid C\ C \mid [_f \mid]_f \mid \ \ldots\ \mid \epsilon$$

L_{sf} is modified from §4.1.3 to allow call parentheses to be mixed anywhere in its strings, and R_{sc} similarly ignores field parentheses. Note that although it has a balanced-parentheses grammar, R_{sc} is regular, since we assume recursion-free programs and hence the number of open call parentheses in a string is bounded.



(a) Initial path with match edges.

(b) Path examined by first pass of algorithm.



(c) Path examined by second pass of algorithm.

Figure 5.1: Paths to illustrate the behavior of our context-sensitive refinement algorithm.

5.1.2 Context-Sensitive Refinement Algorithm

Here we extend the refinement-based algorithm of §4.1.4 with context sensitivity, again illustrating the algorithm on a simplified single path problem. Here, the single path problem requires determining if a path p is a L_{scf}-path, where $L_{\mathsf{scf}} = L_{\mathsf{sf}} \cap R_{\mathsf{sc}}$. As in §4.1.4, the algorithm selectively skips checking sub-paths of p using match edges, which connect matched field parentheses. We refer the reader to §4.1.4 for an explanation of the basic features of this algorithm; here we focus on extending the algorithm to handle method call parentheses.

Figure 5.1(a) gives an example input path for our simplified problem, along with all possible match edges. The path is not an R_{sc}-path, as the $(_2$ and $)_3$ call parentheses are mismatched. We will illustrate how our analysis can discover this fact without inspecting the entire path.

Since a match edge can skip over an arbitrary sequence of method call edges, our algorithm only checks for balanced call parentheses on sub-paths free of match edges. Consider the path of Figure 5.1(b), which skips much of the original path of Figure 5.1(a) using the $t_0 \xrightarrow{\mathsf{match}} t_6$. While it may be tempting to conclude that $(_1$ and $)_3$ are mismatched on this path, $(_1$ is in fact balanced on the path of Figure 5.1(a). To

avoid unsoundly concluding that a path has unbalanced call parentheses, our analysis handles match edges by assuming that any possible sequence of call parentheses may appear on the skipped sub-path. For Figure 5.1(b), the analysis assumes that $)_1(_3$ may have been skipped by the match edge, and hence approximately answers that p may be an L_{scf}-path.

Removal of match edges allows for simultaneous refinement of method call and field parentheses handling, as it exposes more of both of them for checking. In the example of Figure 4.1 in §4.1.3, we showed how removing a match edge exposed mismatched field parentheses. For Figure 5.1, say that after inspecting the path of Figure 5.1(b), the analysis refined by removing the $t_0 \xrightarrow{\mathsf{match}} t_6$. In the subsequent iteration, the analysis would traverse the path of Figure 5.1(c). Here, the match edge removal exposes the mismatched $(_2$ and $)_3$ parentheses, allowing the analysis to conclude that x is not L_{scf}-reachable from o. This ability to refine both types of parentheses through the single mechanism of match edge refinement is key to the efficacy of our algorithm, as we will illustrate further in §5.1.3. Again, the path of Figure 5.1(a) was shown to be unbalanced without inspecting all of its edges, illustrating the performance benefits of refinement.

5.1.3 Refinement on Java programs

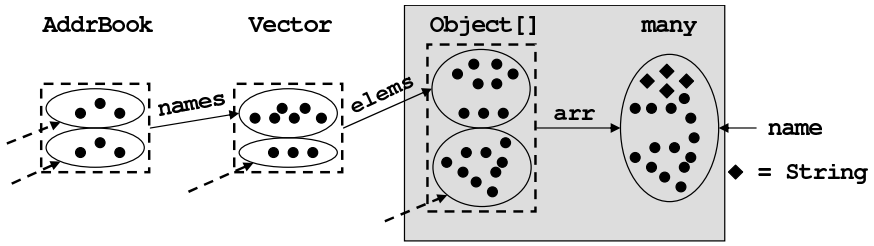
In §5.1.2, we showed how our algorithm is able to save work by skipping inspection of certain sub-paths and refine to obtain a precise answer. Here, we show what the skipped paths correspond to in typical programs (*i.e.*, we show what code is typically analyzed at a particular approximation level) and what code is not visited at all. In particular, the analysis typically only applies full sensitivity for classes whose object contents must be distinguished to precisely answer a query, *e.g.*, to find the contents

```
1  class Vector {
2    Object[] elems; int count;
3    Vector() { t = new Object[10];
4               this.elems = t; }
5    void add(Object p) {
6      t = this.elems;
7      t[count++] = p; // writes t.arr
8    }
9    Object get(int ind) {
10     t = this.elems;
11     return t[ind]; // reads t.arr
12   } ...
13 }
14 class AddrBook {
15   private Vector names;
16   AddrBook() { t = new Vector();
17                this.names = t; }
18   void addEntry(String n, ...) {
19     t = this.names; ...;
20     t.add(n);
21   }
22   void update() {
23     t = this.names;
24     for (int i = 0; i < t.size(); i++) {
25       Object name = t.get(i);
26       // is this cast safe?
27       String nameStr = (String)name;
28       ...
29     }
30   }
31 }
32 void useVec() {
33   Vector v = new Vector();
34   Integer i1 = new Integer();
35   v.add(i1);
36   Integer i2 = (Integer)v.get(0);
37 }
```

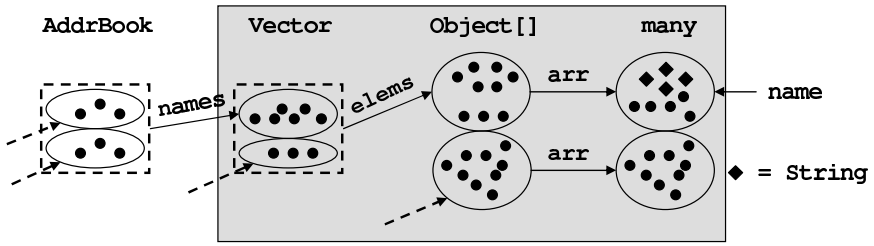
Figure 5.2: Example code for illustrating our points-to analysis algorithm.

of a particular `Vector` object. We will show that both field and context sensitivity are necessary for sufficiently precise results, and that encapsulation allows us to analyze only a small amount of code precisely.

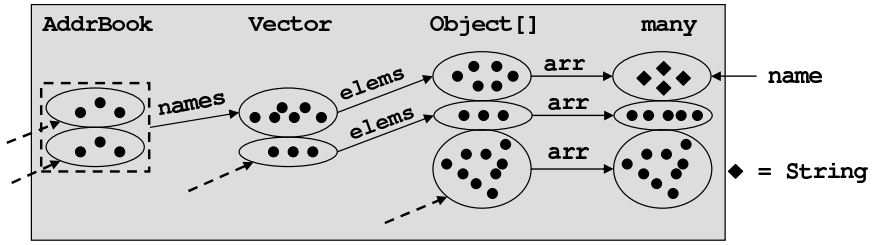
We use the example in Figure 5.2, a partial implementation of an `AddrBook` class



(a) Initial analysis result.



(b) Result after distinguishing `Object[]` contents.



(c) Result after also distinguishing `Vector` contents.

Figure 5.3: Analysis result at different stages of approximation for proving safety of the cast at line 27 of Figure 5.2.

with a `Vector` of names, to illustrate the effects of refinement. We consider an analysis client that aims to statically prove that downcasts cannot fail, in particular the downcast to `String` at line 27. To prove this cast safe with points-to analysis, it suffices to show that the `name` local variable in `update()` can only point to `String` objects.

Figure 5.3(a) through Figure 5.3(c) give an abstract view of the analysis result at each refinement stage while computing $pt(\text{name})$. Each graph shows how the analysis

computes points-to sets of the object fields that are read to obtain the value assigned to `name`: `AddrBook.names` at line 23, `Vector.elems` at line 10, and finally `arr` (a pseudo-field for modeling array accesses) at line 11. Ovals in the graphs enclose points-to sets; pt(name) is shown on the right. A dashed arrow indicates a pointer from some class not shown in Figure 5.2. Figure 5.3(c) shows that after two passes of refinement, the analysis proves that pt(name) contains only `String` objects, shown as diamonds.

The effects of* match *edges The initial analysis result, shown in Figure 5.3(a), is imprecise due to match edges, which cause the analysis to merge the contents of the corresponding fields across objects. match edges cause merging since the analysis uses them to jump from a field read to all writes of the field, ignoring which object's field is being accessed. (This is a *field-based* treatment of fields, as defined in §2.2.2.) Initially, all possible match edges are present in the graph representation of the program, and hence the analysis initially merges the contents of any object field f across all objects.

In the graphs of Figure 5.3, a dashed box indicates that field contents for objects inside the box have been merged due to match edges. Figure 5.3(a) indicates that the `arr` fields for arrays from `Vector.elems` as well as those from some other data structure have been merged; fields `elems` and `names` are similarly collapsed. When computing pt(name), the analysis finds that `name` gets its value from a read of the `arr` field at line 11 of Figure 5.2. Through match edges, the analysis then concludes that any object written into `arr` can flow to `name`, essentially merging the contents of all arrays.

While imprecise, the initial analysis skips inspection of field accesses deeper than those of `arr`, thereby finishing quickly and allowing time for more precise analysis. The gray shading in the graphs of Figure 5.3 indicates which of the shown field dereferences are inspected by that pass of the analysis. In Figure 5.3(a), only accesses of the `arr` field are inspected, while fields `elems` and `names` are skipped. This occurs because when the analysis reaches the array read at line 11 of Figure 5.2, it can jump

immediately to the array write at line 7 using a match edge, and to all other array writes (not shown) using other match edges. Through this skipping, the analysis saves times by avoiding analysis of other code in `Vector` and `AddrBook` and code that uses `AddrBook` objects.

Refining through* match *edge removal Our analysis refines by removing all match edges for all fields of some class `T`, with the goal of distinguishing the contents of different instances of `T`. In the case of our running example, our analysis removes match edges for accesses to `arr` after its first pass, yielding the result in Figure 5.3(b). The dashed box around `Object` arrays has disappeared, and the analysis now distinguishes the `arr` field of arrays stored in `Vector.elems` (*i.e.*, the internal arrays of `Vector`s), from other arrays in the program. The `elems` field is still merged across `Vector`s because of match edges; hence, the analysis concludes that `name` can point to any object stored in the internal array of any `Vector`, still too imprecise a result for the cast-checking client. However, the match edges on `elems` allow the analysis to skip inspection of accesses to `names` and code that uses `AddrBook`s, again saving time and allowing for more refinement.

In its third pass, our analysis succeeds in showing safety of the downcast by removing match edges on `elems`, exposing calls to `Vector` methods for context-sensitive handling. With match edges on `elems` present, the analysis would exit `Vector`'s methods on a match edge (*e.g.*, from the read of `elems` at line 10 in `get()` to the write at line 4 in `Vector()`), skipping an unknown sequence of calls and returns and hence forcing approximation of context sensitivity. Context sensitivity for calls to these methods is required to distinguish contents of different `Vector` instances (see §5.3 for details). After removing match edges on `elems`, the analysis yields the result in Figure 5.3(c), showing that `name` only gets its value from a `Vector` stored in `AddrBook.names`; since such `Vector`s only contain `String`s, this is sufficient to show the downcast at line 27 cannot fail, and the analysis terminates.

Computing a precise result for the example of Figure 5.2 requires unrealizable-path filtering, a context-sensitive heap abstraction, and a context-sensitive call graph. If the analysis did not filter out unrealizable paths, the parameters and return values of the calls to `Vector.get()` at line 25 and line 36 of Figure 5.2 would be conflated, preventing the analysis from proving that the two calls can return distinct objects. The context-sensitive heap abstraction is required to distinguish the internal arrays of the two `Vector`s, both allocated at line 3. A context-insensitive heap abstraction represents both of these arrays with a single abstract object, and hence merges the contents of the `Vector` objects. A context-sensitive call graph is needed to avoid approximation due to spurious recursion, which appears in a context-insensitive call graph when the full `java.util.Vector` class is used; this phenomenon is discussed further in §5.2.2.

Our refinement technique exploits encapsulation in object-oriented code for better performance. In Figure 5.2, the `names` field is *encapsulated*, *i.e.*, the field and the `Vector` it points to cannot be directly accessed outside the `AddrBook` class. Similarly, `Vector` encapsulates its `elems` field. When fields are encapsulated, match edges for those fields can only connect accesses in the same class, limiting the scope of our analysis. For example, with match edges present for `elems` in Figure 5.3(b), the analysis processed code in `Vector`, but not code in `AddrBook` that uses a `Vector`. Furthermore, encapsulated fields allow the refinement to explore data structures in a hierarchical progression: in the example, we explore the array, then the `Vector` pointing to the array, and finally the `AddrBook` pointing to the `Vector`. Note that our analysis exploits encapsulation without any code annotations, instead discovering encapsulation automatically. When fields are not encapsulated, our analysis can still provide precise results, but it may run slower, as more code needs to be analyzed with full precision.

$$
\begin{array}{llll}
S & \xrightarrow{callEntry[i]} & S.i & \\
\epsilon & \xrightarrow{callExit[i]} & \epsilon & \\
S.i & \xrightarrow{callExit[i]} & S & \\
S.j & \xrightarrow{callExit[i]} & error & (i \neq j)
\end{array}
\qquad
\begin{array}{lll}
S & \xrightarrow{\mathsf{assignglobal}} & \epsilon \\
S & \xrightarrow{\overline{\mathsf{assignglobal}}} & \epsilon \\
S & \xrightarrow{\mathsf{match}} & \epsilon \\
S & \xrightarrow{\overline{\mathsf{match}}} & \epsilon
\end{array}
$$

Figure 5.4: State transitions in the FSM for language R_{C}.

5.2 Decidable Formulation

This section gives a decidable formulation of context- and field-sensitive Java points-to analysis, approximating the precise but undecidable analysis formulated in §3.3. First, §5.2.1 presents the regular language R_{C} for computing context-sensitive analysis on recursion-free programs; decidable context- and field-sensitive analysis requires computing $(L_{\mathsf{F}} \cap R_{\mathsf{C}})$-reachability. Then, §5.2.2 discusses how our analysis approximates for programs with recursion.

5.2.1 Context-Sensitive Analysis in CFL-Reachability

Here we describe a regular language R_{C} for computing a context-sensitive points-to analysis on recursion-free programs. We will show how to answer both a projected query "is $o \in pt(x)$?" and a context-sensitive query "is $\langle o, c' \rangle \in pt'(\langle x, c \rangle)$?". Rather than explicitly constructing an inlined program P', context sensitivity is achieved by filtering out paths in our graph representation that correspond to unrealizable paths, as in the precise (but undecidable) formulation of §3.3.

Figure 5.4 defines the transitions in the finite-state machine for R_{C}. In understanding this FSM, it may be useful to refer back to our original formulation of context-sensitive points-to analysis in §3.3, which defines the *callEntry*$[i]$ and *callExit*$[i]$ nonterminals and provides more intuition on the formulation. Each state in the FSM

represents a finite stack of *callEntry*[i] edges, the open parentheses for method calls. Though it checks for balanced parentheses, R_{C} is a regular language because we assume the input program is recursion-free, and hence there are a finite number of possible stack configurations. Transitions in the FSM manipulate the stack in the usual way. The initial state is an empty stack configuration. All states except the *error* state are accept states, and all transitions not shown are self-transitions (*i.e.*, the state machine ignores edges that match the *nonCallTerm* non-terminal of Figure 3.9).

Note that R_{C} allows *partially balanced parentheses*, just like L_{C} in §3.3. An R_{C}-path p is allowed to contain a prefix with unbalanced closed parentheses, due to the $\epsilon \xrightarrow{callExit[i]} \epsilon$ transition, and a suffix with unbalanced open parentheses, as only mismatched *callExit*[i] edges cause a transition to *error*.

The transitions in Figure 5.4 for match and assignglobal edges (and their barred equivalents) show that no checking for balanced call parentheses is done across such edges. Reads and writes to global variables are represented by assignglobal and $\overline{\mathsf{assignglobal}}$ edges, which "skip" the sequence of calls and returns between the accesses (discussed previously in §3.3). Hence, the FSM for R_{C} transitions from any state to ϵ at such edges, "forgetting" any stored call stack information.[1] We transition to ϵ since R_{C} accepts any *callExit*[i] symbol in that state. The same transition is used for match and $\overline{\mathsf{match}}$ edges, since as discussed in §5.1.2, call parentheses cannot soundly be checked across such edges.

R_{C} provides a context-sensitive heap abstraction by treating abstract location nodes identically to variable nodes. Since new and $\overline{\mathsf{new}}$ are not mentioned in Figure 5.4, R_{C} has self-transitions from all states on both symbols. Hence, the R_{C} stack is maintained at abstract location nodes, yielding a context-sensitive heap abstraction in an elegant manner.

[1] With such a transition, the self-edges on globals described in §3.3 are no longer necessary for soundness.

Given R_{C}, a context-sensitive points-to analysis algorithm can be stated concisely. The answer to a projected query "is $o \in pt(x)$?" for program P, represented by graph G, is found by checking for the existence of a *flowsTo*-path p from o to x in G such that p is also an R_{C}-path. In other words, we compute CFL-reachability on G with language $L_{\mathsf{RF}} = L_{\mathsf{F}} \cap R_{\mathsf{C}}$, where L_{F} was developed in Section 3.2. The answer to the context-sensitive query "is $\langle o, c' \rangle \in pt(\langle x, c \rangle)$?" is obtained by modifying R_{C}: we set the initial state of the state machine for R_{C} to c and make c' the only accepting state, thereby mapping the nodes o and x to the appropriate inlined copies $\langle o, c \rangle$ and $\langle x, c' \rangle$. Note that while the call strings of §2.2.4 must start at the root of the call graph, our analysis allows queries with partial calling contexts, since they are valid R_{C} states. Since R_{C} *only* filters paths with mismatched call entries and exits, computing points-to information for program P using L_{RF}-reachability is equivalent to the projected result of computing L_{F}-reachability for fully inlined program P', as desired.

L_{RF}-reachability can be computed by tracking the state of R_{C} for each explored path while computing L_{F}-reachability; paths which cause R_{C} to reach its *error* state are excluded. This technique essentially explodes the input graph for R_{C} [RHS95], creating a node (x, s) when node x is reached with state s of R_{C}. As $|R_{\mathsf{C}}|$ is exponential in the size of the program (due to the exponential number of paths in the call graph), this algorithm has worst-case exponential time complexity. In practice the algorithm does not scale, motivating our refinement approach.

5.2.2 Handling Recursion

Up to this point in the chapter, we have assumed that the input programs for our analysis are recursion-free; here, we discuss how we handle recursive programs. We leverage our demand-driven approach to only approximate recursive calls in the program subset relevant to a query, reducing the number of calls that must be handled

imprecisely.

As context-sensitive and field-sensitive analysis for programs with recursion has been shown undecidable [Rep00], any analysis with both properties must approximate to guarantee termination. One approximation approach is to use k-limiting, *i.e.*, tracking at most k levels of calling context [Shi88]; however, this approach approximates more than what is necessary to achieve decidability. A less drastic approximation is to treat calls within strongly-connected components (SCCs) of the call graph as gotos [WL04, LLA07]. This approach essentially re-labels $\mathsf{param}[i]$ and $\mathsf{return}[i]$ edges within an SCC with assign, collapsing the SCC into a single method and making the program recursion-free, yielding decidability.

With a context-insensitive call graph, the SCC-collapsing approximation leads to a large precision loss. As shown in [LH06], for large programs a context-insensitive call graph typically has an SCC with more than 1000 methods, including methods whose handling is critical to analysis precision, *e.g.*, those of `Vector`. A more precise call graph is necessary to avoid approximate handling of all calls within this large SCC.

Our analysis only approximates handling of method calls that are recursive in a call graph computed during the demand-driven analysis. As our analysis only touches edges in the graph representation relevant to the current query, the recursive cycles it encounters are a subset of all the recursive cycles in the call graph. Consider a query for the objects possibly returned by a call to this simplified version of `Vector.elementAt()` from the Java standard library:

```
Object elementAt(int index) {
  if (index >= numElements) {
    throw new OutOfBoundsException(index + " too big"));
  }
  return elems[index];
```

```
}
```

Our analysis considers only the array access and the read of the `elems` field in the **`return`** statement, ignoring the calls to the `OutOfBoundsException` constructor and `String` and `StringBuffer` methods from the string concatenation. In a context-insensitive call graph that considers all control flow, these ignored calls lead to `elementAt()` being included in the aforementioned large SCC. Our analysis approximates less due to recursion since it constructs a context-sensitive call graph in the program subset relevant to each query, ignoring many potentially recursive calls.

5.3 Refinement Algorithm

In this section, we give a detailed description of our refinement-based points-to analysis algorithm. In §5.1, we presented our refinement algorithm for the problem of checking a single path's string for membership in a simplified version of our reachability languages. In §5.3.1, we show how the refinement algorithm of §5.1.2 extends to the full L_{RF} language, using a detailed example. Then, §5.3.2 presents pseudocode for our algorithm as applied to arbitrary graphs.

5.3.1 Refinement for L_{RF}-reachability

Here we explain our refinement-based algorithm for full L_{RF}-reachability using a detailed example. We use the `AddrBook` example of Figure 5.2 to illustrate the algorithm, again considering a client trying to prove safety of the cast of `name` to `String` at line 27, which requires computing $pt(\text{name})$. Figure 5.5 shows the relevant part of the graph representation for the code in Figure 5.2. To prove the cast safe, the analysis must discover that there is no $(L_{\mathsf{F}} \cap R_{\mathsf{C}})$-path from o_{34}, an `Integer` object, to the $\texttt{name}_{\texttt{update}}$ node. The columns in Figure 5.5 indicate how the analysis explores the graph during

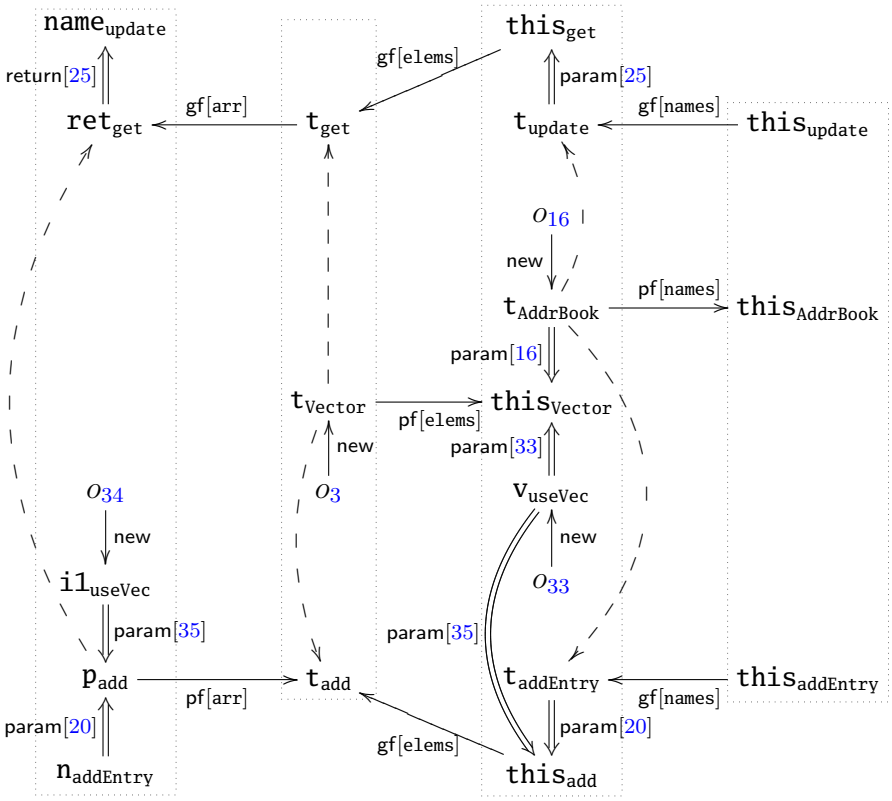


Figure 5.5: Relevant portion of graph for code in Figure 5.2. Solid edges represent program statements, with single edges for intraprocedural statements and double edges for call entry and exit statements. Variables are subscripted with the name of the enclosing method, and line numbers in labels refer to call sites or allocation sites. The dashed edges are match edges. For space, getfield and putfield are abbreviated gf and pf.

refinement. The initial pass of the algorithm visits the left-most column, and each subsequent pass additionally visits the next column to the right, until the safety of the cast is proved.

Approximation As discussed previously (*e.g.*, in §4.1.4), we use match and $\overline{\text{match}}$ edges between corresponding field accesses to enable approximation. L_{F} (defined in §3.2) has two types of matched parentheses pairs, (putfield$[f]$,getfield$[f]$) and ($\overline{\text{getfield}}[f]$,$\overline{\text{putfield}}[f]$); we add match edges for the former type (shown as dashed edges in Figure 5.5, and $\overline{\text{match}}$ edges for the latter type. The match edges *define* the

columns of Figure 5.5, as the analysis uses match edges to limit its scope in each pass. This correspondence was shown previously in Figure 5.3, where in each analysis pass, the gray shading (indicating analyzed code) stops at the first field whose accesses still have match edges.

Our language for computing reachability with match edges is L_{FR} (R for refinement), with start symbol *flowsToR*:

$$\begin{aligned} \mathit{flowsToR} &\rightarrow \mathsf{new}\ (\ \mathsf{assign}\ \mid\ \mathsf{putfield}[f]\ \ \mathit{aliasR}\ \ \mathsf{getfield}[f] \\ &\qquad\qquad \mid\ \mathsf{match}\ )^* \\ \mathit{aliasR} &\rightarrow \overline{\mathit{flowsToR}}\ \ \mathit{flowsToR} \end{aligned}$$

flowsToR is exactly the language used for context-insensitive refinement (*cf.* §4.3). Our algorithm uses match edges whenever possible to skip the (potentially expensive) check for an *aliasR* path. For Figure 5.5, the algorithm skips from $\texttt{ret}_{\texttt{get}}$ to $\texttt{p}_{\texttt{add}}$ using a match edge in its first pass, never leaving the first column of the graph. However, this leads the analysis to conclude that o_{34} is in the points-to set, necessitating refinement. Note that the analysis cannot filter out the param[35] edge in this pass, as it reaches `add()` through a match edge and hence cannot rule out flow from call site 35.

Refinement Selective refinement is accomplished by *removing* match edges from the graph, as in §5.1. Removal of match edges forces the analysis to search for more *aliasR*-paths, as it can no longer skip between the corresponding field accesses; this search can yield a more precise analysis result. In the example, the second pass of our analysis refines by removing the $\texttt{ret}_{\texttt{get}} \xleftarrow{\mathsf{match}} \texttt{p}_{\texttt{add}}$ in Figure 5.5. This removal forces the analysis to handle the read and write of `arr` by checking for an *aliasR*-path from $\texttt{t}_{\texttt{get}}$ to $\texttt{t}_{\texttt{add}}$. Since match edges in the second column remain, the analysis only explores the first two columns of the graph. In this case, an *aliasR*-path is found by connecting $\overline{\mathit{flowsToR}}$-path $\texttt{t}_{\texttt{get}} \xrightarrow{\overline{\mathsf{match}}} \texttt{t}_{\texttt{Vector}} \xrightarrow{\overline{\mathsf{new}}} o_3$ with *flowsToR*-path $o_3 \xrightarrow{\mathsf{new}} \texttt{t}_{\texttt{Vector}} \xrightarrow{\mathsf{match}} \texttt{t}_{\texttt{add}}$.

Again, the analysis cannot filter the path using the param[35] edge, as `add()` is reached through a match edge. So, further refinement is needed.

Context Sensitivity As described in §5.1, a match edge can skip over an arbitrary sequence of method calls and returns, requiring approximation of context sensitivity. As shown in Figure 5.4, R_C "forgets" calling context information when "traversing" match edges using the transition $S \xrightarrow{\text{match}} \epsilon$ and treats $\overline{\text{match}}$ analogously. Each pass of our analysis algorithm computes $(L_{FR} \cap R_C)$-reachability, using R_C to filter unrealizable *flowsToR*-paths as much as possible, *i.e.*, by checking that sub-paths between match edges are R_C-paths. As previously noted, the first two passes of the analysis of the example were unable to disregard the param[35] edge, as R_C state was cleared across a match edge before reaching `add()`.

Removal of match edges can lead to filtering of paths because it exposes more call entry and exit edges for analysis. After the second pass, our analysis further refines field-sensitivity by removing match edges on the `elems` field, *i.e.*, the $\texttt{t}_{\texttt{Vector}} \xrightarrow{\text{match}} \texttt{t}_{\texttt{get}}$ and $\texttt{t}_{\texttt{Vector}} \xrightarrow{\text{match}} \texttt{t}_{\texttt{add}}$ edges in the second column of Figure 5.5. So, in the third pass the analysis must find *aliasR*-paths from $\texttt{this}_{\texttt{get}}$ to $\texttt{this}_{\texttt{Vector}}$ and from $\texttt{this}_{\texttt{Vector}}$ to $\texttt{this}_{\texttt{add}}$. Clearly, the **`this`** parameters of the `Vector` methods can be aliased; hence, the key to precision will be to use the state of R_C tracked along the *aliasR*-paths between the **`this`** parameters to do filtering.

The tracking of R_C state in the third pass of our analysis yields a result precise enough for proving the downcast safe in the example. When searching for an *aliasR*-path from $\texttt{this}_{\texttt{get}}$ to $\texttt{this}_{\texttt{Vector}}$, the analysis starts with R_C call stack $\langle 25 \rangle$, a call site for `Vector.get()` in Figure 5.2. The *aliasR*-path to $\texttt{this}_{\texttt{Vector}}$ combines the $\overline{\textit{flowsToR}}$-path $\texttt{this}_{\texttt{get}} \xrightarrow{\overline{\text{param}}[25]} \texttt{t}_{\texttt{update}} \xrightarrow{\overline{\text{match}}} \texttt{t}_{\texttt{AddrBook}} \xrightarrow{\overline{\text{new}}} o_{16}$ and the *flowsToR*-path $o_{16} \xrightarrow{\text{new}} \texttt{t}_{\texttt{AddrBook}} \xrightarrow{\text{param}[16]} \texttt{this}_{\texttt{Vector}}$. Tracking R_C state along this *aliasR*-path yields call stack $\langle 16 \rangle$ at $\texttt{this}_{\texttt{Vector}}$; essentially, the analysis has concluded that the `get()` call at line 25 of Figure 5.2 must be on a `Vector` allocated at line 16, *i.e.*, a `Vector` used by an

`AddrBook`.

Similarly, after finding an *aliasR*-path from $\texttt{this}_{\texttt{Vector}}$ to $\texttt{this}_{\texttt{add}}$, the analysis reaches $\texttt{this}_{\texttt{add}}$ with R_{C} call stack $\langle 20 \rangle$, concluding that `add()` must be entered from the call site at line 20. The analysis can then filter out the `add()` call at line 35, thereby filtering o_{34} from $pt(\text{name})$ and yielding the desired result, *i.e.*, that $\texttt{name}_{\texttt{update}}$ is not $(L_{\mathsf{F}} \cap R_{\mathsf{C}})$-reachable from o_{34}. Notice that the result is computed without touching the fourth column of Figure 5.5, illustrating how our technique can compute a sufficiently precise result while limiting analysis scope.

5.3.2 Pseudocode

Here we present pseudocode for our refinement-based context-sensitive points-to analysis. We first discuss the outer refinement loop of the analysis (§5.3.2.1), which removes match edges from the graph until a sufficiently precise answer is computed for the analysis client. Then, in §5.3.2.2 we present in detail the algorithm for computing points-to information in each refinement pass, a modification of the FullFS algorithm from §4.6. Finally, §5.3.2.3 presents our refinement policy for removing match edges.

Interface to Clients The presented algorithm computes $(L_{\mathsf{OTF}} \cap R)$-reachability for L_{OTF} as defined in Figure 3.8 and any regular language R.[2] For context-sensitive analysis, the R_{C} language of Figure 5.4 would be used.[3] The algorithm takes as inputs some variable v and some state s from an automaton describing R, notated v_s. It also requires a function ClientSatisfied that returns true when a computed points-to set for v_s satisfies the client. As an example, a client wishing to use context-sensitive points-to analysis to prove that a cast `y = (A)x` can never fail would query x_ϵ (where ϵ represents the empty call stack of R_{C}) with a ClientSatisfied function

[2] There are minor restrictions on the regular language for the presented pseudocode, for simplicity. The restrictions are described along with the corresponding pseudocode.

[3] Other regular languages may be useful, for example to detect sensitive sinks in tainting analysis [LL05]; developing such languages is future work.

```
CSRefinePTo(v_s)
1  while TRUE
2        do pto ← RefinePass(v_s)
3           if ClientSatisfied(pto)
4              then return TRUE
5              else  if ¬UpdateRefinementPolicy() then return FALSE
```

Figure 5.6: Pseudocode for the refinement loop of our points-to analysis.

that returns TRUE iff the types of all abstract locations in the computed points-to set are subtypes of `A`. The algorithm then computes increasingly precise points-to sets using refinement until (1) the client is satisfied, (2) no more refinement is possible, or (3) some time budget has been exhausted.

5.3.2.1 Refinement Loop

Figure 5.6 gives pseudocode for the outermost refinement loop of our algorithm. Each loop iteration first computes an points-to set for the input variable and state v_s at line 2, approximated based on which match edges are refined. (RefinePass, which actually computes the points-to set, is defined in Figure 5.7 and will be discussed shortly.) If the computed points-to set satisfies the client, the loop terminates (line 3 and line 4). Otherwise, the refinement policy for match edges is updated (line 5). If UpdateRefinementPolicy returns FALSE, then no more match edges can be refined—*i.e.*, the algorithm has computed the most precise points-to set possible given its abstraction—and the algorithm terminates. Otherwise, a more precise points-to set is computed in the next iteration.

Our implementation adds two features not present in the pseudocode of Figure 5.6. First, in the case where the client is satisfied with a computed points-to set, the set itself is made available to the client for further processing. Second, the implementation allocates a resource budget for each query, such that if the budget is exceeded in

computing points-to sets, the query is terminated early. We discuss budgets more in §5.4.1.

5.3.2.2 Computing Points-To Sets

Pseudocode for RefinePass, which computes a points-to set in each pass of the refinement algorithm, is presented in Figure 5.7, with helper functions in Figure 5.8. The algorithm is an extended version of the FullFS algorithm of Figure 4.13. We chose to extend FullFS for the context-sensitive algorithm since caching of intermediate results is essential for good performance (unlike the context-insensitive case), and FullFS integrates such caching nicely. Our extensions to FullFS are:

- Handling of the regular language R to be intersected with L_{OTF}, which requires computing state transitions and checking for the error state in several places.
- Handling of match edges.
- On-the-fly call graph refinement, which requires the graph representation of Table 3.2 (including concType$[T]$, receiver$[i][s]$, paramForType$[T][s]$, and returnForType$[T][s]$ edges). The algorithm computes $(L_{\mathsf{OTF}} \cap R)$-reachability, where L_{OTF} was defined in Figure 3.8.

We will explain each of these extensions in turn. The discussion assumes an understanding of the pseudocode in Figure 4.13, and the reader may benefit from reviewing that figure again before proceeding.

Handling the Regular Language Our algorithm computes $(L_{\mathsf{OTF}} \cap R)$-reachability by *exploding* the input graph G for R. The exploded graph $G_R^\#$ is constructed such that computing standard graph reachability on $G_R^\#$ is equivalent to computing R-reachability on G [RHS95]. The nodes and edges of $G_R^\#$ incorporate the states and transitions of some FSM F for R:

```
RefinePass(v_s)
 1  Add query v_s ↪ .
 2  repeat
 3        for each edge x ←new— o and state s
 4            do if x_s ↪ .
 5                  then s' ← Trans(s, new-bar)
 6                       if s' ≠ error then add o_s' to X_s
 7               if . ↪ o_s(any)
 8                  then s' ← Trans(s, new)
 9                       if s' ≠ error then add o_s to X'_s'
10        for each edge x ←assign— y and state s
11            do HandleCopy(x, y, assign, s)
12        for each edge x ←assignglobal— y and state s
13            do HandleCopy(x, y, assignglobal, s)
14        for each edge x ←param[i]— y and state s
15            do HandleCopy(x, y, param[i], s)
16        for each edge x ←return[i]— y and state s
17            do HandleCopy(x, y, return[i], s)
18        for each edge x ←getfield[f]— y and state s
19            do if UseMatchEdges(f)
20                  then for each edge x ←match— p
21                           do HandleCopy(x, p, match, s)
22                  else  if x_s ↪ .
23                           then add y_s ↪ .
24                                for each o_s' ∈ Y_s
25                                    do add . ↪ o_s'(f_w)
26                                       add all of O_s'.f to X_s
27        for each edge x ←putfield[f]— y and state s
28            do for each o_s' ∈ X'_s
29                  do if . ↪ o_s'(f_w)
30                           then add y_s ↪ .
31                                add all of Y_s to O_s'.f
32               if Y'_s ≠ ∅
33                  then add x_s ↪ .
34                       for each o_s' ∈ X_s
35                           do add all of Y'_s to O_s'.f'
36                              add . ↪ o_s'(f_r)
37        HandleOTF()
38    until no change
39  return V_s
```

Figure 5.7: Pseudocode for a single iteration of our context-sensitive refinement algorithm. See Figure 5.8 for the HandleCopy and HandleOTF procedures.

```
HandleCopy(x, y, label, s)
    ▷ Handle a copy edge x <--label-- y,
    ▷ considering the cases of either x or y in state s
1   if x_s ↪ .
2      then s' ← Trans(s, label-bar)
3           if s' ≠ error
4              then add y_s' ↪ .
5                   add all of Y_s' to X_s
6   if Y'_s ≠ ∅
7      then s' ← Trans(s, label)
8           if s' ≠ error then add all of Y'_s to X'_s'
```

```
HandleOTF()
    ▷ Partial pseudocode; other cases are similar
 1  for each edge f <--paramForType[T][i]-- a and state s
 2      do if f_s ↪ .
 3            then s' ← Trans(s, paramForType-bar[T][i])
 4                 if s' = error then continue
 5                 add r_s' ↪ ., where a --receiver[j][i]--> r exists
 6                 for each o_s'' ∈ R_s'
 7                     do if no edge o --concType[T]--> o then continue
 8                        add . ↪ o_s''(call)
 9                        for each state {s_c : o_s'' ∈ R'_s_c}
10                            do add a_s_c ↪ .
11                               add all of A_s_c to F_s
```

Figure 5.8: Pseudocode for HandleCopy and HandleOTF, helper functions for the pseudocode in Figure 5.7.

- Each node n_s in $G_R^\#$ corresponds to some node n in G and some state s in F.
- An edge $n_s \xrightarrow{\mathsf{f}} n'_{s'}$ exists in $G_R^\#$ iff (1) G has an edge $n \xrightarrow{\mathsf{f}} n'$ and (2) F has a transition from s to s' on f.

Determining whether n' is $(L_{\mathsf{OTF}} \cap R)$-reachable from n in G is equivalent to checking if n'_a is L_{OTF}-reachable from n_i in $G_R^\#$, where i and a are respectively initial and accept states for F.

As seen in Figure 5.7, our algorithm modifies FullFS to operate on nodes in the exploded graph. States from the FSM for R are associated with both variable nodes (*e.g.*, the initial query $v_s \hookrightarrow .$ at line 1) and abstract location nodes (*e.g.*, $o_{s'}$ at line 6). Points-to sets and tracked sets are also maintained for exploded graph nodes, *e.g.*, X_s at line 6 and $X'_{s'}$ at line 9.

For scalability, our algorithm *lazily* explodes the input graph during query computation. Constructing a fully exploded graph for R is both unnecessary, since many node-state combinations do not arise during a query, and intractable for large languages (*e.g.*, R_{C}, whose FSM has an exponential number of states). Lazy explosion requires computing state transitions in the FSM for R while traversing the original graph G. This computation is performed by calls to the TRANS procedure, *e.g.*, at line 2 of HANDLECOPY in Figure 5.8. Given edge $n \xrightarrow{\mathsf{f}} n'$ and node n_s from $G_R^{\#}$, the corresponding successor in $G_R^{\#}$ is computed to be $n'_{\text{TRANS}(s,\mathsf{f})}$. Note that it is possible that $\text{TRANS}(s, \mathsf{f}) = \mathit{error}$, indicating that the FSM has no transition from s on f. In such a case, R has filtered out the current path, at which point the pseudocode aborts exploration of the path.

The HANDLECOPY helper procedure in Figure 5.8 performs this generic handling of state transitions across different types of edges. As an example, consider the call to HANDLECOPY with a $\mathsf{param}[i]$ edge at line 15, assuming the regular language is R_{C} for context sensitivity. In this case, x and y in HANDLECOPY are respectively the formal and actual parameter associated with the edge. If x_s has been queried (line 1 in HANDLECOPY), then s' is computed for the $\overline{\mathsf{param}}[i]$ according to the rules of Figure 5.4 (line 2). This transition will result in an error state if the top of the call stack represented by s is not i. Otherwise, propagation across the edge is handled as in FullFS (line 4 and line 5). Tracked sets are handled in a similar manner in line 6 through line 8.

Handling* match *edges The only change required for handling match edges appears in line 19 through line 21. The UseMatchEdges procedure, provided by the refinement policy, returns **true** if match edges are present in the graph for accesses of field f. (We discuss our refinement policy further in §5.3.2.3.) We have found that in practice, removing or keeping all match edges for particular fields (instead of particular field accesses) is sufficient for devising an effective refinement policy. If match edges are present for some field, they are handled through a call to HandleCopy at line 21.

On-the-fly Call Graph Recall from §2.2.3 that on-the-fly call graph refinement requires determining virtual call targets directly in the points-to analysis, by computing points-to information for receiver arguments. We formulated this problem in Chapter 3 as adding edges reflecting virtual call dispatch semantics to our graph representation and finding paths to compute call targets. Since here we aim to perform context-sensitive refinement of the pre-computed call graph, our algorithm searches for $\mathit{virtParam}[i]$ and $\mathit{virtReturn}[i]$ paths (and their inverses) from the L_{OTF} language, as defined in Figure 3.8.[4]

The HandleOTF procedure in Figure 5.8 has pseudocode for finding $\overline{\mathit{virtParam}}[s]$ paths (finding $\mathit{virtParam}[s]$, $\mathit{virtReturn}[s]$, and $\overline{\mathit{virtReturn}}[s]$ paths requires similar logic), according to the following productions from Figure 3.8:

$$\overline{\mathit{virtParam}}[s] \;\rightarrow\; \overline{\mathit{dispatch}}[i][s]\ \mathit{flowsTo}\ \overline{\mathsf{receiver}}[i][s]$$
$$\overline{\mathit{dispatch}}[i][s] \;\rightarrow\; \overline{\mathsf{paramForType}}[T][s]\ \mathsf{receiver}[i][s]\ \mathit{pointsTo}\ \mathsf{concType}[T]$$

Recall that i represents the parameter position, while s identifies the call site. Line 3 in HandleOTF (Figure 5.8) tracks the state along the $\overline{\mathsf{paramForType}}[T][i]$ edge to actual parameter a, and line 5 follows the $\mathsf{receiver}[j][i]$ edge to the receiver argument

[4]Note that an implementation can use a separate refinement policy to select a subset of virtual call sites to refine; for simplicity, our pseudocode refines *all* virtual call sites.

r.[5] Line 5 also introduces a points-to query for r, thereby finding *pointsTo*-paths from r to abstract locations. For each such abstract location o, line 7 checks for the existence of the requisite $o \xrightarrow{\mathsf{concType}[T]} o$ edge. Finally, line 8 finds a *flowsTo*-path from o back to r, and line 10 continues points-to computations at a, since from line 5 it is known that a $r \xrightarrow{\overline{\mathsf{receiver}}[j][i]} a$ must exist, completing the $\overline{virtParam}[i]$ path.

Line 8 and line 9 in HANDLEOTF, which follow a *flowsTo*-path from o to r, are necessary to compute the proper regular language state at the virtual call site. The pointed-to-by query $. \hookrightarrow o_{s''}(call)$ uses a new reason *call* to avoid unnecessary processing at field accesses. At first glance, this traversal of the *flowsTo*-path may seem unnecessary, since its existence has already been proved by the discovery of a *pointsTo*-path from r to o (line 5 and line 6). However, the algorithm must traverse the *pointsTo* *flowsTo* path to determine the correct regular language state at r, which may differ from the s' state of line 3. As a simple example, if we have the path $r \xrightarrow{\overline{\mathsf{param}}[j]} o \xrightarrow{\mathsf{param}[j]} r$, where the regular language is R_{C} for context sensitivity and the initial state is ϵ, the state at the end of the path will change to $\langle j \rangle$ (due to the admittance of partially balanced parentheses by R_{C}).

How does this additional tracking of regular language states affect analysis precision? Consider the example of Figure 5.9. Say there is a query for the points-to set of $p_{B.foo}$, *i.e.*, the `p` parameter of `B.foo()`. The virtual call at line 8 may invoke `B.foo()`, and hence value flow from $p_{callFoo}$ must be considered. However, only the `callFoo()` call at line 16 should be considered for this query, as the previous call at line 15 does not lead to `B.foo()` being invoked at line 8 (since for that call $a_{callFoo}$ has concrete type `A`). Without the tracking of regular language states at line 8 and line 9 of HANDLEOTF, flow from both `callFoo()` call sites would be considered in this example, leading to the result $pt(p_{B.foo}) = \{o_{13}, o_{14}\}$ instead of the more precise

[5] Note that for simplicity, we do not allow state machine transitions on receiver or concType edges. Adding support for such transitions would be straightforward, but such support is unnecessary for context-sensitive analysis.

```
1  class A {
2    void foo(Object p) { ... }
3  }
4  class B extends A {
5    void foo(Object p) { ... }
6  }
7  void callFoo(A a, Object p) {
8    a.foo(p);
9  }
10 main() {
11   A a = new A();
12   B b = new B();
13   Object p1 = new Object();
14   Object p2 = new Object();
15   callFoo(a, p1);
16   callFoo(b, p2);
17 }
```

Figure 5.9: Example illustrating the need to properly track states at virtual call sites.

answer $pt(p_{B.foo}) = \{o_{14}\}$.

Recursion Detection In our implementation, we detect recursive calls during computation of points-to information, as discussed in §5.2.2. When a cycle in the input graph corresponding to recursive method calls is detected, all call sites in the cycle are marked as recursive. Once marked as recursive, call sites are never again pushed on a call stack in our state machine code, in effect collapsing the recursive cycles. A question arises of how to handle existing call stacks (stored as states in exploded graph nodes) which contain call sites that were later detected as recursive. One solution is to remove recursive call sites from existing call stacks whenever they are inspected by the algorithm. Since we have found in practice that recursive cycles rarely arise for the clients we tested, we instead use a simpler solution and just re-start a query computation after each recursive cycle is detected.

Performance Considerations As with FullFS, certain implementation details are important for good performance. Worklists are used to only process necessary graph edges (as opposed to iterating over the entire graph). Rather than iterating over all

states as in the loops of Figure 5.7, the implementation only considers those states in which a node has been queried. Finally, an efficient bit-vector set implementation is used for points-to sets and tracked sets, to ensure that set unions can be computed quickly.

5.3.2.3 Refinement Policy

We have yet to specify how we choose match edges to remove after each analysis pass. We remove match edges with the goal of distinguishing the field contents of different objects of some class `T`; as shown in Figure 5.3, match edges for accesses of field f cause merging of the contents of f across objects. Our method for choosing `T` is straightforward, but empirically effective: we choose the enclosing class for the field corresponding to the first match edge encountered in the previous analysis pass, and then remove match edges on all fields of this class.[6] In the example of Figure 5.5, we encounter a match edge on `arr` in the first pass and `elems` in the second pass, leading to removal of match edges on those fields. Removing these match edges allows the analysis to distinguish the contents of the internal `Object` array of a particular `Vector`, as desired.

5.4 Evaluation

Our experiments validated the following three experimental hypotheses:

Some clients need context sensitivity. We confirmed (as shown previously [LH06]) that context-insensitive analysis does not have enough precision for the cast-checking client, as it could only prove 7.8% of the downcasts in our benchmarks safe.

[6] We also remove match edges for fields in superclasses and inner classes of `T`, as they also tend to be relevant.

Our refinement approach is precise. Our refinement algorithm proved 61% more casts safe on average than one of the most precise existing algorithms [LH06], and refinement was critical for this precision gain. Also, our algorithm proved a disjointness property of objects allocated in some factory methods, requiring precision beyond that of the existing algorithm.

Our refinement approach is scalable. With the analysis budget we chose, our algorithm checked all application downcasts in under 13 minutes on all benchmarks. Furthermore, our algorithm required no more than 35MB of memory for any of the benchmarks, an order of magnitude less than the memory requirements for existing comparable analyses.

5.4.1 Experimental Configuration

Implementation We implemented our analysis using the Soot 2.2.1 [VRHS+99] and Spark [LH03] frameworks. For our graph representation, we augment the pointer assignment graph built by Spark with $\mathsf{param}[i]$ and $\mathsf{return}[i]$ edges for context sensitivity. We analyzed the Sun JDK 1.3.1_01 libraries, as Soot provides models of this version's native methods. All experiments were performed on a machine with a Xeon 2.4GHz processor and 2GB RAM, running Fedora Core 1 Linux.

Our Soot-based implementation differs from the pseudocode of Figure 5.7 code in some minor ways. First, the implementation uses recursive calls rather than worklists to traverse the graph for a query. We found the behavior of the recursive implementation easier to reason about during algorithm design, and the performance impact was small. Also, the implementation sometimes uses an alternate strategy for discovering aliasing between base pointers of field accesses. Given base pointer p of a field read and q of a field write, the algorithm of Figure 5.7 searches for a *flowsTo*-path from any abstract location in $pt(p)$ to q. The alternate strategy computes $pt(p)$ and $pt(q)$

and then checks if $pt(p) \cap pt(q) \neq \emptyset$, tracking R_{C} state appropriately. This alternate strategy sometimes requires far less graph traversal, and the implementation uses it when the default strategy fails.

We give experimental results for the following analyses:

DemRef: our demand-driven, refinement-based algorithm.

Full: our demand-driven algorithm configured to treat all code with full precision, rather than refining.

1H: a 1-limited object-sensitive analysis [MRR05] (*i.e.*, limited to 1 level of object sensitivity) with a (1-limited) context-sensitive heap abstraction and call graph, provided as part of the Paddle framework for BDD-based analysis [Lho06].

The 1H algorithm was chosen because in recent work [LH06], it was shown to be the most precise of a set that included the Zhu and Calman and Whaley and Lam algorithms [ZC04, WL04] and a call string approach [Shi88]. We were unable to run the 1H algorithm on the `chart` benchmark within 2GB of RAM; the result for `chart` in Table 5.2 is taken from [LH06], as its results for other benchmarks exactly matched our observations.

To compare with an analysis that handles assignments with equality constraints, we also implemented data structure analysis [LLA07], a context-sensitive analysis for C that we adapted to Java. We implemented the analysis both with and without on-the-fly context-sensitive call graph construction. We found that without a context-sensitive call graph, the analysis was much too imprecise for our clients; *e.g.*, it could not prove any casts safe in most benchmarks. This imprecision stemmed from the collapsing of call graph SCCs by the analysis (see §5.2.2), which are large in a context-insensitive call graph [LH06]. We were unable to sufficiently scale the algorithm variant with context-sensitive call graph construction to analyze our benchmarks; the most similar published analysis for Java [O'C00] had similar scalability issues.

Configuration We configure our analysis to refine the precision of context-insensitive field-sensitive Andersen's analysis with an on-the-fly call graph, as implemented in Spark [LH03]. We use the context-insensitive analysis to answer queries that require less precision, and to rule out certain paths in our analysis. For example, if we are trying to prove a cast of x to type `T` safe, we do not traverse to nodes y where the context-insensitive analysis shows that all locations in $pt(y)$ are subtypes of `T`. The analysis is scalable, analyzing all benchmarks in under 3.5 minutes, including the time required to construct the graph representation. We include the context-insensitive analysis time in all presented running times for our analysis.

Our refinement analysis is best run with a *budget*: after some fixed amount of time for each query, the analysis terminates and returns a conservative result to the client [SGSB05]. This budget prevents the analysis from running excessively long on queries it cannot hope to answer precisely, *e.g.*, those that require flow-sensitive precision. For our experiments, we configured our analysis to traverse at most 75000 nodes per query, divided evenly among a maximum of 10 refinement iterations, a sweet spot for the tested clients; doubling the budget yielded a negligible precision gain.

Benchmarks Our benchmark suite is described in Table 5.1. We use the same suite as that of [LH06], to compare with its object-sensitive analysis. The size of the benchmarks are comparable to those used in other recent Java points-to analysis studies [LH03, WL04].

Clients We evaluated our analysis using three clients. The first was a client that checked the safety of downcasts in application code; as in [LH06], library casts were excluded to make benchmark differences clear, but the library was still analyzed when necessary for application casts. As illustrated in §5.1.3, downcast checking is an exacting test of points-to analysis precision, especially of the ability to distinguish the contents of different data structures; an analysis that fares poorly at proving

Benchmark	Methods	Statements
compress	2722	36690
db	2741	37243
jack	2996	42729
javac	3916	77619
jess	3354	47645
mpegaudio	2927	41009
mtrt	2873	39180
soot-c	4979	90355
sablecc-j	8853	164056
polyglot	6227	120634
antlr	4021	77934
bloat	5415	106629
chart	7323	110594
jython	4560	69026
pmd	7388	115857
ps	5320	106718

Table 5.1: Information about our benchmarks. We include the SPECjvm98 suite, `soot-c` and `sablecc-j` from the Ashes suite [Ash], several benchmarks from the DaCapo suite version beta050224 [DaC], and the Polyglot Java front-end [NCM03]. The "Statements" column gives the number of edges in the graph representation. The numbers include the reachable portions of the Java library, determining using a call graph constructed on the fly with Andersen's analysis [And94] by Spark [LH03].

downcast safety is unlikely to satisfy other demanding clients.

We also experimented with a client that tries to prove disjointness of the contents of objects allocated in *factory methods*, *i.e.*, methods that return a newly-allocated object for each call. For example, an `iterator()` method typically allocates a new `Iterator` object for each call. The client looks for factory methods using simple pattern matching, and then tries to prove disjointness of method return values for objects allocated in different calls to these methods. For `iterator()`, the client tries to show that calls to `next()` on `Iterator` objects allocated by different calls to `iterator()` can return distinct objects. Proving such disjointness properties could be important, *e.g.*, to reduce false positives for a verification client. Furthermore, this client requires greater precision than the 1H algorithm of [LH06] can provide (since it requires at least 2 levels of object-sensitivity), and hence illustrates the benefits of having a more precise analysis.

Finally, to further test performance, we ran a client that queried the DemRef analysis for all application variables where the 1H analysis yielded a more precise result than context-insensitive analysis, representing a client that requires near-exhaustive points-to information.

5.4.2 Experimental Results

Imprecision of context-insensitive analysis We found context-insensitive Andersen's analysis (from Spark [LH03]) to be insufficient for proving downcasts safe in our benchmarks. The analysis could prove an average of only 7.8% of casts safe, ranging from 0% for `compress` to 31.7% for `sablecc-j`. This result is consistent with previous work [LH06], and shows the client's need for more precise analysis.

Precision for cast-checking Table 5.2 shows that our refinement algorithm provides more precision for the cast-checking client than the 1H algorithm. The refine-

Benchmark	Casts	DemRef Time (s)	DemRef Safe	Full Safe	1H Safe
`compress`	6	44.8	33.3	33.3	0.0
`db`	24	44.4	79.2	37.5	25.0
`jack`	135	62.8	52.6	23.0	31.1
`javac`	315	150.3	20.6	12.4	13.3
`jess`	76	63.7	72.4	6.6	57.9
`mpegaudio`	12	58.8	25.0	25.0	33.3
`mtrt`	10	47.4	50.0	40.0	40.0
`soot-c`	906	387.8	28.0	14.1	8.3
`sablecc-j`	362	315.8	18.5	5.5	11.9
`polyglot`	3482	750.3	88.1	6.8	72.5
`antlr`	281	118.2	50.9	2.8	21.7
`bloat`	1217	472.6	12.6	5.2	6.7
`chart`	535	283.5	38.5	9.0	30.5
`jython`	464	84.9	8.8	2.8	6.5
`pmd`	1135	571.7	15.1	10.0	11.2
`ps`	659	131.1	6.2	5.5	41.0

Table 5.2: Results for the cast safety client. The "Casts" column gives the number of downcasts that context-insensitive analysis cannot prove safe; these numbers differ from those in Lhotak's work because we exclude casts of non-pointers (*e.g.*,`float` to `int`), as they cannot cause a runtime exception. The three rightmost columns respectively give the percentage of these casts proved safe by our refinement algorithm ("DemRef"), our demand-driven algorithm configured to treat all code precisely ("Full"), and the object-sensitive analysis of Lhotak's work [LH06] ("1H"). The "DemRef Time" column gives the running time for the refinement algorithm in seconds.

ment technique proved an average of 1.61x as many casts safe as the 1H algorithm (excluding `compress` where 1H proved no casts safe), ranging from 0.15x for `ps` to 3.39x for `soot-c`. The large precision benefit for the `soot-c` benchmark stems primarily from precise handling of iterators. For example, proving the cast to `Foo` safe is beyond the capabilities of the 1H algorithm for the following code:[7]

```
Iterator i = x.iterator(); o = (Foo)i.next();
```

[7]Two levels of object-sensitivity (including the heap abstraction) would suffice for this case, but that analysis does not yet scale in the Paddle framework [Lho].

Benchmark	# Factory	Dist (%)	Time (s)
db	1	100.0	44.2
jack	1	100.0	52.7
javac	7	28.6	63.1
jess	12	91.7	52.6
soot-c	19	63.2	58.0
sablecc-j	9	55.6	106.4
polyglot	22	31.8	183.6
antlr	3	100.0	56.8
bloat	13	30.8	63.0
chart	19	36.8	110.3
jython	14	21.4	55.4
pmd	20	25.0	101.6
ps	1	100.0	69.5

Table 5.3: Results for the factory method client. The "# Factory" column gives the number of detected factory methods, and the "Dist" column gives the percentage of those factory methods for which the analysis could distinguish the contents of the allocated objects. The "Time" column gives the running time in seconds.

The refinement algorithm is significantly more precise for cast checking (within the same budget) than a demand-driven analysis that treats all code precisely (the "Full" algorithm), as shown in Table 5.2. Given the analysis budget of 75000 nodes, the algorithm with refinement proved 4.25x more casts safe than without refinement, ranging from 1x for `mpegaudio` to 17.88x for `antlr`; doubling the analysis budget had a negligible impact on this result. The algorithm without refinement is often even less precise than then 1H algorithm, showing the importance of using both the demand-driven approach and refinement.

There are several reasons why some casts cannot be proved safe by our analysis. DemRef was less precise than 1H for the `ps` benchmark due to 181 casts of objects read from an operator stack mutated in many parts of the program; the large amount of relevant code led to DemRef choosing incorrect fields to refine for these casts. Proving certain casts safe requires flow- or path-sensitivity, *e.g.*, for casts dominated by an

Benchmark	# Queries	Av. Query (s)	Time (s)
compress	35	0.08	46.9
db	98	0.09	53.3
jack	350	0.19	119.2
javac	2727	0.32	942.2
jess	587	0.43	303.6
mpegaudio	69	0.08	61.4
mtrt	84	0.07	50.5
soot-c	2481	0.58	1503.6
sablecc-j	5704	0.50	2957.0
polyglot	18893	0.26	5118.6
antlr	3143	0.32	1070.7
bloat	3354	0.31	1103.4
chart	—	—	—
jython	3127	0.19	647.3
pmd	8079	0.68	5593.3
ps	3859	0.16	690.7

Table 5.4: Results for querying all application variables for which the 1H algorithm of Lhotak's work [LH06] yielded a more precise result than a context-insensitive analysis. The "# Queries" column gives the number of such variables, the "Av. Query" column gives the average query time in seconds, and the "Time" column gives the total time in seconds. Results for chart are not shown, as we could not run the 1H algorithm on it in available memory.

instanceof check that ensures their safety; many such casts can be proved safe by an extra intraprocedural analysis [O'C00, WS01]. Sometimes, context-sensitive call graph construction consumes the bulk of analysis time for DemRef, but is unnecessary for a precise result; automatically determining which virtual call sites require precise handling is future work.

Precision for factory methods Our analysis proved the contents of many factory-allocated objects disjoint, as shown in Table 5.3. Excluding benchmarks with fewer than 5 factory methods, the analysis proved disjointness for an average of 42.8% of the methods in each benchmark, ranging from 21.4% for `jython` to 91.7% for `jess`. This result shows that precision greater than that provided by the 1H algorithm is required for realistic clients besides downcast checking, and that our analysis can provide that precision.

Scalability of refinement approach Due to our demand-driven approach, the memory requirements of our analysis are significantly less than those of previous approaches. In the experiments, our analysis never consumed more than 5MB of memory for any query, and our implementation does no caching between queries. The memory required to store the results from the context-insensitive analysis pre-pass is less than 30MB using BDDs [BLQ+03], yielding a maximum memory requirement of 35MB for these benchmarks. In comparison, we could not run the object-sensitive algorithm of [LH06] on the `chart` benchmark within 2GB of RAM, and a precise equality-based analysis requires 1GB of RAM on large benchmarks [Ste].

Table 5.2 shows that the refinement algorithm scaled well for the cast checking client, taking under 13 minutes for each benchmark. The factory method client took under 4 minutes per benchmark (as seen in Table 5.3), as it raised few queries. Table 5.4 gives the data for the client that queried all application variables for which the 1H algorithm gave a more precise answer than context-insensitive analysis. The longest running time for this client was was 94 minutes for `pmd`, with an average

query time of 0.68 seconds. Our current implementation computes each query result from scratch, and we believe that for large numbers of queries, performance could be significantly improved through more caching.

Chapter 6

Thin Slicing

> *"Thin-slicing is part of what makes the unconscious so dazzling. But it's also what we find most problematic about rapid cognition. How is it possible to gather the necessary information for a sophisticated judgment in such a short time?"* Malcolm Gladwell, *Blink: The Power of Thinking Without Thinking*

This chapter presents *thin slicing*, a refinement-based program understanding tool based on ideas from our points-to analysis. As discussed in §1.3, thin slicing aims to use refinement to focus programmer attention on those statements most relevant to her development task, just as our points-to analysis uses refinement to focus analysis effort on statements most relevant to the client property. In fact, the technique for explaining heap-based value flow in thin slices is based directly on the match-edge-based refinement in our points-to analysis. Here, we present the details of how thin slices are defined, refined, and computed; see §1.3 for a high level overview of thin slicing.

The rest of this chapter is organized as follows. §6.1 defines producer statements and the thin slicing process, and §6.2 defines thin slices using traditional dependences.

§6.3 describes our technique for expanding (refining) thin slices to explain heap-based value flow and control dependences. §6.4 presents algorithms for computing thin slices as variants of a traditional slicing algorithm. Finally, §6.5 gives results from our experimental evaluation.

6.1 Defining Thin Slices

In this section, we define the producer statements included in a thin slice. We also show how statements excluded from the thin slice explain why the producer statements affect the seed. A simple example, seen in Figure 6.1, is used to illustrate these concepts. §6.2 defines the statements in a thin slice using traditional notions of dependence.

Slicing determines the parts of a program "relevant" to some *seed* statement. In traditional slicing, relevance is defined as any statement possibly affecting the values computed by the seed. As originally stated by Weiser [Wei79], this relevance definition requires the slice to include an *executable subset* of the program in which the seed always performs the same computation as in the original program. Thin slicing differs from classical slicing primarily in its more selective notion of relevance.

With thin slicing, only *producer statements* for the seed are relevant. We define producer statements in terms of *direct uses* of memory locations (variables or object fields in Java). A statement s directly uses a location l iff s uses l for some computation other than a pointer dereference. For example, the statement `y = x.f` does not directly use `x`, but it does directly use $o.f$, where `x` points to o. A statement t is a *producer* for a seed s iff (1) $s = t$ or (2) t writes a value to a location directly used by some other producer.

Consider computing a thin slice for line 7 in the toy example of Figure 6.1. Line 7 directly uses an object field written at line 5 (since `w` and `z` are aliased), and therefore,

```
1  x = new A();
2  z = x;
3  y = new B();
4  w = x;
5  w.f = y;
6  if (w == z) {
7    v = z.f;  // the seed
8  }
```

Figure 6.1: A small program to illustrate thin slicing. Directly-used locations (see §6.1) in the thin slice for line 7 are underlined.

line 5 is a producer. Similarly, line 5 directly uses `y`, which is written at line 3, making line 3 a producer as well. Hence, line 5 and line 3 are comprise the thin slice for line 7 (along with line 7 itself). In contrast, the traditional slice for line 7 is the entire example.

We call the non-producer statements in the traditional slice *explainer statements*. These statements show *why* the producer statements can affect the seed. Explainer statements can show one of two things about the producers:

Heap-based value flow When values flow between producers through heap locations, the locations are accessed using aliased pointers. Explainer statements show how these base pointers may become aliased.

Control flow The remaining explainer statements show the conditions (*i.e.*, the expressions in conditional branches) under which producer statements actually execute.

Consider again the example of Figure 6.1. Line 2 and line 4 show that `w` and `z` both point to the `A` object allocated at line 1. Hence, these lines are explainers for the heap-based value flow between line 5 and line 7 in the thin slice. Line 6 explains control flow, showing the condition under which the seed statement actually executes.

Thin slicing's separation of producer and explainer statements provides a natural, structured method for exploring a traditional slice. Traditional slices must include *transitive* explainer statements (*i.e.*, explainers for the explainers and so on), since any statement possibly affecting the seed is relevant for such a slice. While this transitivity can lead to an overwhelming number of explainer statements, thin slices structure them into a manageable hierarchy. Explainers for heap-based value flow in a thin slice can be shown using two additional thin slices, as shown in §6.3.1. The behavior of a conditional guarding a thin slice statement can also be understood through an additional thin slice. In this manner, more and more thin slices can be used to show explainers, in the limit yielding the traditional slice.

In practice, we have found that very few explainers are needed to accomplish typical debugging and understanding tasks. In our evaluation, over half the tasks could be completed with a thin slice alone. In most other cases, only one or two explainer statements were required, and these explainers were lexically close to thin slice statements (further discussed in §6.3.2). Hence, thin slicing is highly effective at identifying the statements in a traditional slice most relevant to developer tasks.

6.2 Thin Slices as Dependences

In §6.1, we defined thin slices in terms of producer statements. Here we define thin slices in terms of the dependences typically used to define traditional slices. The thin slice for a seed s is a subset of those statements upon which s is transitively *flow dependent* (also known as *data dependent*), obtained by ignoring uses of base pointers in dereferences.

A statement s is flow dependent on statement t if the following three conditions hold [HPR89]:

1. s can read from some storage location l.

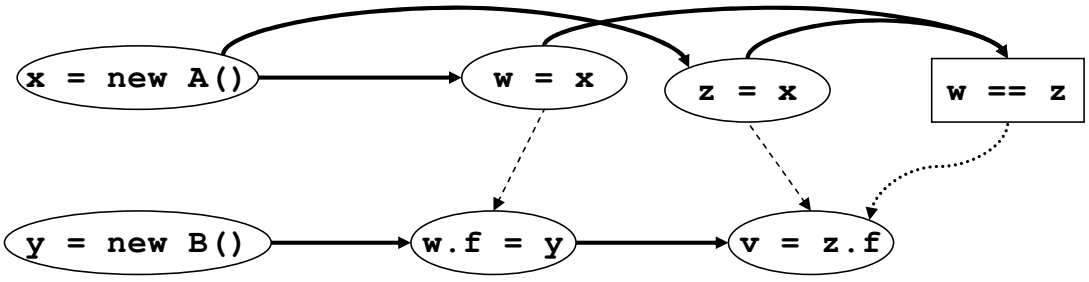


Figure 6.2: A dependence graph for the program of Figure 6.1. Thick edges indicate non-base-pointer flow dependences, used for thin slicing. Traditional slicing also uses base pointer flow dependences (the dashed edges) and control dependences (the dotted edge).

2. t can write to l.

3. There exists a control-flow path from t to s on which l is not re-defined.

For Java-like languages, storage locations are either variables (local or global) or object fields, with the latter accessed through some field dereference of the form `x.f`. Traditional slices must include the transitive flow dependences of the seed.

Thin slices ignore *base pointer flow dependences*, thereby excluding statements explaining heap-based value flow. A base pointer flow dependence is a flow dependence due solely to the use of a pointer in a field dereference. For the statement `y = x.f`, flow dependences due to the use of `x` are base pointer flow dependences. Similarly, a statement of the form `p.f = q` has base pointer flow dependences due to the use of `p`. Ignoring base pointer flow dependences leaves only *producer flow dependences*, which transitively connect a statement to its producers. For example, `y = x.f` would have a producer flow dependence to some statement `z.f = w`, where `x` and `z` may be aliased.

Figure 6.2 shows an example dependence graph for the program of Figure 6.1. Nodes represent statements, and edges represent dependences between statements. As is standard for dependence graphs [HRB88, RHSR94], edges are drawn in the direction opposite of the dependences, so thin slicing requires computing backwards reachability. In Figure 6.2, the solid edges indicate the producer flow dependences,

while the dashed edges indicate ignored base pointer flow dependences. The dotted edge is a control dependence, to be discussed shortly. The seed `v = z.f` is only reachable from `w.f = y` and `y = new B()` via solid edges, and these statements are the producers for the seed, as expected.

An interesting correspondence exists between producer flow dependences and edges in pointer analysis graph representation with match edges. The producer flow dependences for a field write `x.f = y` are a subset of those field reads `w = z.f` such that an edge $y \xrightarrow{\text{match}} w$ exists in our initial pointer analysis graph (which includes all possible match edges). (There may not be a producer flow dependence for each possible match edge since a more precise alias analysis may be used to compute the flow dependences.) Ignoring base pointer flow dependences in thin slicing allows the user to view heap-based value flow without necessarily understanding why the corresponding base pointers are may-aliased. This abstraction corresponds exactly to how match edges allow our pointer analysis to selectively skip checking for aliasing of base pointers.

Note that thin slices also exclude control dependences, explainers of control flow. Intuitively, statement s is control dependent on conditional e if e can affect how many times s executes (Tip's survey [Tip95] has a more formal definition). Figure 6.2 has a dotted control dependence edge from conditional `w == z` to `v = z.f`, the statement in its **if** block in Figure 6.1. §6.3 describes our empirical observation that important control dependences are nearly always lexically close to thin slice statements, and hence can be discovered easily.

6.3 Expanding Thin Slices

Here, we discuss in more detail how thin slices can be refined, or *expanded*, to show explainer statements, as discussed in §6.1. To review, explainer statements can answer

questions of the following form about a thin slice T:

1. Given statements `x := y.f` and `w.f := z` in T such that `w` and `y` are aliased (causing value flow from `z` to `x`), what statements cause the aliasing?

2. Under what conditions can some statement s in T execute?

A thin slicing tool answers these questions when requested by the user. §6.3.1 discusses a technique for explaining aliasing using two additional thin slices. §6.3.2 discusses how relevant control dependences are usually "close" to thin slice statements, making their discovery relatively straightforward.

Example We use the example in Figure 6.3, a simple program fragment manipulating a file, to illustrate thin slice expansion. The example displays only a small part of the `File` implementation, the tracking of whether the file is open using a **`boolean`** field. The `readFromFile()` function throws an exception if the file passed to it is not open. Finally, the `main()` method creates a file, erroneously closes it, and then passes it to `readFromFile()`, causing the exception. The `File` object is read from a `Vector` before being passed to `close()` and `readFromFile()`, complicating discovery of the bug.

6.3.1 Question 1: Explaining Aliasing

When a thin slice includes statements that copy a value through the heap, sometimes the user needs to understand why those statements access the same heap location. For the example of Figure 6.3, suppose that the user asks for a thin slice from line 10 to determine why line 11 threw an exception. The computed thin slice will be $\{3, 4, 5, 9, 10\}$ (highlighted with underlines), the only statements that can produce the boolean `open` value. Clearly, these statements fail to diagnose the bug completely: the user still does not know which `File` is passed to `close()` before being passed to

```
1  class File {
2    boolean open;
3    File() { ...; this.open = true; }
4    isOpen() { return this.open; }
5    close() { ...; this.open = false; }
6    ...
7  }
8  readFromFile(File f) {
9    boolean open = f.isOpen();
10   if (!open)
11     throw new ClosedException();
12   } ...
13 }
14 main() {
15   File f = new File();
16   Vector files = new Vector();
17   files.add(f);
18   ...;
19   File g = (File)files.get(i);
20   g.close();
21   ...;
22   File h = (File)files.get(i);
23   readFromFile(h);
24 }
```

Figure 6.3: An example for showing expansion of thin slices, similar to an example we saw in our evaluation. The bug is an exception thrown at line 11, and understanding the bug requires an explanation of aliasing (§6.3.1) and following a control dependence (§6.3.2). We use single underlines to highlight relevant expressions in the initial thin slice, and double underlines for expressions in explainer statements for aliasing.

`isOpen()`. To diagnose this bug, the user must determine which statements cause the '**`this`**' pointers of `close()` and `isOpen()` to be aliased.

We can expand thin slices to explain aliasing by computing additional thin slices for the base pointers in question. Given aliased base pointers `x` and `y`, we compute thin slices seeded with the statements defining `x` and `y` (unique assuming SSA form). These thin slices will show why some common object o can flow to both `x` and `y`, causing them to be aliased. For Figure 6.3, the common object for the '**`this`**' parameters of `close()` and `isOpen()` is the `File` allocated at line 15. Double underlines in Figure 6.3

indicate the statements added to explain the flow of the `File` (the `Vector` class is elided for clarity). Note line 16 is still omitted, as it does not touch the `File` object. Given these thin slices, the user sees that line 20 closes the `File`, and that the bug could be fixed by either not closing the file or by removing it from the `Vector`.

Note that this expansion technique for explaining aliasing closely matches our technique for refining match edges in our points-to analysis. Expanding a thin slice to explain a particular alias has an effect similar to removing the match edge between the corresponding field access edges in our points-to analysis graph representation. Removing this match edge forces our points-to analysis to see if the corresponding base pointers are may-aliased, but other match edges are still used to skip the alias checking for other field accesses. Similarly, recursive thin slices are used to explain may-aliasing to the user in thin slice expansion, thereby avoiding expansion for other related field accesses.

Explaining aliasing using additional thin slices yields an intuitive hierarchical structure to heap-based flow, making it more understandable for the user. Suppose that statements `x := y.f` and `w.f := z` appear in a thin slice. Expanding the thin slice to show flow into `x` and `w` adds one more level of data dependences to the slice. If during expansion, statements `a := b.g` and `c.g := d` are added, the aliasing of `b` and `c` could be explained with another level of data dependences, and so on. If Figure 6.3 were changed so that the flow of the `Vector` to the `add()` and `get()` in `main()` was complex (*e.g.*, it got stored in a data structure), another level of thin slices would explain that flow. The ability to show these different levels of aliasing in a structured manner relies on the fact that only field reads and writes can dereference pointers in Java; in C, which allows for creating pointers to pointers and taking addresses of variables, explanations of why two statements access the same memory location may not be so simple.

Array accesses can require explainer statements beyond those showing the aliasing

of the array pointers. Say that we have statements `a = b[i]` and `c[j] = d` in the thin slice, such that there is value flow from `d` to `a`. In trying to understand this heap-based flow, the user may wonder both (1) how `b` and `c` can be aliased (the same question as with field accesses), and additionally (2) how the array indices `i` and `j` can have the same value. The latter question can be answered through thin slices on each of the array index expressions (with any necessary expansion).

Two additional technical points about explaining aliasing merit mention. First, the thin slices explaining aliasing should be restricted to only show the flow of objects that can flow to both base pointers, filtering statements showing flow of an object to just one of them. This filtering eliminates some statements irrelevant to explaining the aliasing. Second, context sensitivity may be necessary to focus the aliasing explanations in some cases. For example, if the code of Figure 6.3 were part of a large program where many `File` objects were used, the user would likely want to ask about aliasing '**`this`**' in `isOpen()` for the particular call at line 9, rather than for all calls.

We encountered one case in which an explanation of aliasing was necessary in our experiments, and we believe that many similar situations often arise in practice. In our programming experience, we have found that when such bugs arise, they can be tricky to debug, as values can be mutated in unexpected places. Analyses that find typestate bugs [DLS02, FYD+06], *e.g.*, reading from a file after closing it, could benefit from using thin slices to explain bugs that involve aliasing. Such tools sometimes hide error reports that involve aliasing, since there is no mechanism in the tool for explaining the aliasing succinctly [MSA+04].

6.3.2 Question 2: Control Dependence

In our experience, when a debugging or program understanding task requires viewing control dependences, the control-relevant statements usually lie lexically close to some

statement in the thin slice. In Figure 6.3, the bug manifests at line 11, which throws the exception. As no value flows into the **`throw`** statement, a thin slice from the throw statement will not aid debugging. However, code inspection immediately shows that the condition of the **`if`** statement at line 10 is relevant to the bug, as it directly controls whether the exception is thrown. With this information, the obvious next step is to thin slice from line 10 to learn more about the bug, as described in §6.3.1.

While this example may seem contrived, our experiments show that Figure 6.3 reflects the common case. For nearly all tasks in our evaluation, at most one or two control dependences were relevant, and they all lay syntactically close to statements in the thin slice. We also found that the vast majority of control dependences are unnecessary for understanding the seed behavior. Hence, we believe that in practice, simply showing the thin slice statements in the source code suffices for identifying any relevant control dependences; the user can take additional thin slices from relevant conditionals to understand their behavior. Additional tool support may be useful for indicating non-obvious control dependences, *e.g.*, due to exceptions.

6.4 Computing Thin Slices

Computing a thin slice entails computing a statement's transitive flow dependences, ignoring uses of base pointers (as discussed in §6.2). As in previous work on slicing [HRB88, Rep98], we compute thin slices using variants of graph reachability. Here, we first describe the basics of constructing our graph representation, a subset of system dependence graphs [HRB88] (§6.4.1). Then, we briefly present two simple algorithms to compute thin slices, one context insensitive (§6.4.2) and one context sensitive (§6.4.3).

6.4.1 Graph Construction

In both thin slice algorithms, we first compute a subset of the system dependence graph (SDG) program representation of Horwitz et al. [HRB88]. Previous work [LH96, AH03] has described how to compute SDGs for Java-like languages, and we mostly re-use those techniques (slight differences are discussed in §7.2). Our implementation handles the full Java Virtual Machine bytecode language, excluding concurrency. Our representation differs in that we (1) exclude control dependence edges and (2) handle heap-based flow dependences differently, depending on the thin slicing algorithm (details in §6.4.2 and §6.4.3).

SDG construction relies on the results of a points-to analysis. We use the points-to analysis to compute a call graph for the program, necessary for tracking interprocedural dependences. We also use the points-to analysis to determine which heap locations can be defined (used) by field writes (reads), in order to track heap-based value flow. §6.5 shows that precise points-to analysis is key for effective thin slicing of Java programs.

Our representation of data dependences for local variables and method parameters is straightforward. At a high level, we represent such dependences as follows:

1. For a statement `x = e`, where `x` is a local, we add edges to all statements using `x`, excluding uses in pointer dereferences of the form `x.f`. We operate on an SSA representation, so these edges are added flow sensitively.

2. For an actual parameter node for a call to method `m()`, we query the call graph to find the possible call targets $m_1, \ldots, m_k$. Then, for each m_i, we add an edge from the actual parameter node to the corresponding formal parameter node. Return values are handled similarly.

Our thin slicing algorithms differ from the standard SDG handling of data dependence, and from each other, in their treatment of definitions of heap locations (*i.e.*,

statements of the form `x.f := e`) as described below.

6.4.2 Context-Insensitive Thin Slicing

Our first algorithm computes traditional (context-insensitive) graph reachability on our SDG variant to compute thin slices. In this approach, we represent data dependences for heap access statements as follows:

- For a statement `x.f := e`, we add an edge to each statement with an expression `w.f` on its right-hand side, such that the points-to analysis indicates `x` *may-alias* `w`.

Note that we add direct edges to statements *in other procedures*. In contrast, the traditional SDG only includes interprocedural edges for parameter passing and return values. The advantage of this approach is that we need not model heap accesses using additional parameters and return values, as is done with traditional slicing [HRB88]. In practice, not using heap parameters dramatically increases scalability without significant loss in precision (discussed further in §6.4.3 and §6.5).

Having computed the graph, a simple transitive closure gives the thin slice for a particular seed. It is straightforward to construct the graph and do the traversal in a demand-driven fashion. A potential disadvantage of this approach is that it may return unrealizable paths [RHS95] due to lack of context sensitivity (§6.5 shows this issue is not significant in practice).

6.4.3 Context-Sensitive Thin Slicing

The context-sensitive thin slicing algorithm uses an SDG variant closer to that used in traditional slicing, created compositionally from program dependence graphs (PDGs) for each procedure. Intraprocedurally, this approach handles heap accesses as follows:

- For a statement `x.f := e`, we add an edge to each statement with an expression `w.f` on its right-hand side *in the same procedure* such that the points-to analysis indicates `x` *may-alias* `w`.

We handle interprocedural heap flow in the same way as the standard SDG, with heap reads and writes modeled as extra parameters and return values to each procedure [AG98, HRB88]. Our implementation introduces such parameters using the same heap partitions used by the preliminary pointer analysis. Discovering the appropriate set of parameters for each procedure requires an interprocedural mod-ref analysis [RLS+01], computed using the result of the points-to analysis.

Having built the graph, we compute context-sensitive reachability as a partially balanced parentheses problem [Rep98]. Our implementation relies on a backwards, demand-driven tabulation algorithm [RHS95].

In our experience, constructing an SDG using heap parameters can be very expensive for large programs. Furthermore, we found that for realistic usage patterns, context sensitivity did not provide much benefit for thin slicing. See §6.5 for details.

6.5 Evaluation

We now present an empirical evaluation of thin slicing for debugging and program understanding tasks. Our experiments validate four hypotheses:

- **Thin slices lead the user to desired statements.** For the tasks we considered, thin slices often contain the desired statements (*e.g.*, the buggy statement for a debugging task). When statements explaining pointer aliasing or control flow were relevant, the statements were always lexically close to statements in the thin slice. Subjectively, we also found a thin slicer very useful for understanding one set of benchmarks.

- **Thin slices focus better on desired statements than traditional slices.** We compared context-insensitive thin slicing to context-insensitive traditional slicing (the context-sensitive configurations did not scale) with identical handling of control dependences and a breadth-first strategy for inspecting statements, simulating real-world use of a program understanding tool. The experiments showed that finding desired statements in a traditional slice required inspecting 3.3 times more statements than a thin slice for the debugging tasks, and 9.4 times more statements for the program understanding tasks.

- **A precise pointer analysis is key to effective thin slicing.** We used a pointer analysis with object-sensitive handling [MRR05] of key collections classes for the thin slicer. With a less precise pointer analysis, up to 17.2X more statements required inspection in thin slices to find desired statements.

- **Thin slices can be computed efficiently.** Our context-insensitive thin slicing algorithm scaled well to large programs, with the cost of computing thin slices insignificant compared to the pre-requisite call graph construction and pointer analysis. We were unable to scale a context-sensitive traditional slicer [HRB88] to our larger benchmarks.

The remainder of this section proceeds as follows. §6.5.1 describes our experimental configuration and methodology. §6.5.2 presents details of the debugging experiment, which studied the effectiveness of thin and traditional slicing for locating injected bugs in a standard suite for evaluating debugging tools [DER05]. Finally, §6.5.3 describes the program understanding experiment, which tests the effectiveness of thin and traditional slicing for discovering why certain downcasts cannot fail in the SPECjvm98 benchmark suite.

Program	Methods	Bytecode Size (KB)	Call Graph Nodes	SDG Statements
Software-Artifact Infrastructure Repository				
`nanoxml`	541	35	817	22205
`jtopas`	337	24	397	23766
`ant`	11147	632	20164	584155
`xmlsec`	11192	678	17075	525886
SPECjvm98				
`mtrt`	470	32	514	19699
`jess`	1061	67	1466	46037
`javac`	1610	118	2127	71041
`jack`	592	55	1088	38114

Table 6.1: Benchmark characteristics, derived from methods discovered during on-the-fly call graph construction, including Java library methods. The number of call graph nodes exceeds the number of distinct methods due to limited cloning-based context-sensitivity in the points-to analysis. SDG Statements reports the number of scalar statements, but excludes parameter passing statements introduced to model the heap.

6.5.1 Configuration and Methodology

We implemented the thin and traditional data slicers using the IBM T.J. Watson Libraries for Analysis (WALA) [WAL]. We utilized call graph construction and pointer analysis algorithms provided by WALA, along with its tabulation solver for context-sensitive analysis [RHS95]. We analyzed our benchmarks with the Sun JDK 1.4.2_09 standard library code, for which WALA provides models of important native methods. WALA uses heuristics to analyze the most common uses of reflection in Java, but in general reflection and native methods may still cause some unsoundness, as is typical in Java static analysis implementations. All experiments were performed on a Lenovo ThinkPad t60p with dual 2.2GHz Intel T2600 processors and 2GB RAM. The analyzer ran on the Sun JDK 1.5_07 using at most 1GB of heap space.

Table 6.1 provides information about the programs used in our experiments. For pointer analysis and call graph construction, we used a variant of Andersen's analysis

with on-the-fly call graph construction [And94, RMR01], with fully object-sensitive cloning [MRR05] for objects of key collections classes, as described in [FYD+06] (the importance of this precision is discussed later in the section). We excluded from the call graphs a few large standard libraries (*e.g.*,`javax.swing`, `java.awt`) which we deemed *a priori* uninteresting for the tasks at hand, since none of our tested tasks involved those libraries. For all experiments reported, call graph construction and pointer analysis ran in under 5 minutes.

Note that the points-to analysis information needed for thin slicing could be naturally computed on-demand along with thin slices. We outlined the connections between thin slicing and our refinement-based points-to analysis in §6.2 and §6.3. In a realistic IDE setting, pre-computing an exhaustive pointer analysis would probably not be suitable from a performance standpoint, making demand-driven pointer analysis critical for usability. At the time of this evaluation, our refinement-based points-to analysis was not implemented in WALA, and hence we decided to base our experiments on existing WALA points-to analyses. We plan to have a version of thin slicing based on refinement-based points-to analysis implemented in the near future.

Scalability For the dependence graph traversal, we considered both the context-insensitive (flat graph reachability) and context-sensitive (tabulation) algorithms presented in §6.4.

In all cases, the time and space to compute the thin slice or traditional slice with the context-insensitive algorithm was insignificant compared to the preliminary pointer analysis. Context-insensitive thin slicing took under 6 seconds for all tests except `ant`, which took 47 seconds since a large number of interprocedural heap dependence edges had to be added. These low running times are not surprising, as context-insensitive slicing (thin or traditional) reduces to simple graph reachability on a demand-driven construction of the SDG program representation.

Our implementation of context-sensitive traditional slicing [RHSR94] scales to

handle most experiments on the smaller test cases (`nanoxml`, `jtopas`, `mtrt`, `jack`). For the larger codes, our implementation could not complete in reasonable time and/or space. We believe our implementation is fairly well-tuned, as the analysis engine (based on tabulation [Rep98]) has evolved over several years and been used in several studies reporting scalable interprocedural dataflow analyses (*e.g.*, [FYD+06]). For slicing, the key bottleneck comes from handling of the heap; as programs grow larger, the number of SDG statements introduced to model heap parameter-passing quickly explodes, dramatically increasing space and time requirements. For our larger benchmarks, the full SDG grew to over 10 million nodes before exhausting available memory; we suspect the number of nodes would grow much larger given adequate space. Note that heap parameters are also a scalability bottleneck in a commercial slicing tool [Tei].

In all results reported, we compare results from the context-insensitive thin slicer to a context-insensitive traditional slicer, which scaled to all benchmarks. This provides an apples-to-apples comparison, as all experimental parameters match exactly for the two algorithms; the only difference was how each handled data dependences.

Measuring Slice Size Nearly all existing work measures the precision of a slice by its full size. However, in practice, once a user of a program understanding tool has discovered all of the desired statements for her original problem (*e.g.*, those causing some bug), she will not inspect the rest of the slice. Our experiments aim to simulate this realistic usage pattern.

For each task, we identify both a seed statement for the slice and a set of *desired statements*, *i.e.*, those statements whose discovery suffices for completing the task. For example, for a debugging task, the seed is the point of failure, and the desired statement is the cause of the bug. We then aim to measure how many statements in the slice the user must inspect to discover the desired statements.

We use a breadth-first traversal strategy to simulate the order in which statements

are inspected by the user, as in the work of Renieris and Reiss [RR03]. Intuitively, statements "closer" to the seed seem more likely to be relevant to its behavior. Hence, we assume the user gradually explores statements of increasing distance (defined by the dependence graph of the technique) from the seed until the desired statements are found; a breadth-first search of the dependence graph simulates this strategy. Note that CodeSurfer [Cod], perhaps the most widely-used slicing tool, supports such dependence-graph browsing for viewing slices. The BFS evaluation metric has also been used in other recent work [RR03, ZGG06b, ZJL+06]. For thin and traditional slicing, our tables report the number of statements inspected using this breadth-first inspection strategy.

To our knowledge, ours is the first work to compare static slicing algorithms using a measure intended to simulate the usage of a realistic tool, rather than just comparing the full slice sizes. We note that the two measures produce qualitatively different results. For example, in one of our smaller test cases, `nanoxml-1`, context sensitivity reduces the traditional slice size from 8067 statements to 381 statements, but the number of statements explored in the traversal decreases only from 32 to 26. We observed similar results for thin slices. Given these results, the context-sensitive algorithm of §6.4.3 does not seem beneficial for thin slicing as likely used in practice.

Control Dependence As discussed in §6.3.2, relevant control dependences were observed to be always lexically close to statements in the thin slice, as in the example of Figure 6.3. Furthermore, most control dependences were not useful for the tested tasks, and it is not obvious how to automatically expose important control dependences. Hence, we manually pre-determined the important control dependences for our tasks, and counted only those control dependences as inspected for both the thin and traditional slicers. This handling of control dependences allowed us to focus on the effectiveness of thin slicing's handling of data dependences compared with a traditional slicer's.

Threats to Validity One threat to the validity of our results is that our study of debugging tasks (§6.5.2) uses injected bugs from the SIR suite [DER05], which may not accurately reflect the characteristics of real bugs. Several techniques were used to make the injected bugs in the SIR suite realistic, described in detail in [DER05]. The bugs were of a wide variety: they could alter both the control and data flow of the program, and the resulting failures ranged from program crashes to incorrect output. Nevertheless, we intend to do a future study with real bugs to confirm that thin slicing still provides a significant benefit.

Our use of breadth-first search on the dependence graph to simulate programmer exploration of the slice may not accurately reflect how developers would use a slicing tool. If most developers are able to very quickly prune statements in a traditional slice irrelevant to their tasks, then the BFS metric would overstate the advantage of thin slicing. In the future, we aim to do a user study to obtain more definitive answers on the productivity benefits of thin slicing.

Finally, our use of whole-program pointer analysis and call graph construction for the thin slicer may not scale to larger benchmarks. These analyses also may not be suitable for use inside a development environment, as code edits could require expensive re-computation of the pointer analysis results. We plan to employ demand-driven, refinement-based pointer analysis [SB06] in the next version of the thin slicer to overcome these drawbacks.

6.5.2 Experiment: Locating Bugs

Our first experiment tested locating several bugs, (1) to see if thin slices include the buggy statement when slicing from the seed, and (2) to compare the number of inspected statements for thin and traditional slices. We investigated several injected bugs in the Java programs in the Software-Artifact Infrastructure Repository

(SIR) [DER05]. SIR provides both several injected bugs for each program and test suites that can be used to expose the bugs. For each injected bug, we ran the corresponding test suite to discover a failure. Then, we ran both thin and traditional slicing from the failure point, measuring how many statements had to be inspected to find the bug (as described in §6.5.1).

Three points should be noted about the SIR programs and injected bugs. First, we were unable to include two SIR programs in these experiments, `jmeter` and `siena`. We could not determine the appropriate library dependences to build `jmeter`, and in our runs, no test cases exposed the injected bugs in `siena`. Also, the suite provides several versions of each benchmark; we chose bugs from the most recent versions. Finally, some of the injected bugs represent bugs of omission, *i.e.*, bugs that deleted necessary code. If the omission bug removed an assignment to a local or a conditional branch, we chose as the desired target statements the immediate data or control dependent successor statements, respectively. We excluded bugs that deleted field writes, as there was no obvious relationship between the deleted write and the surrounding code in the method.

Table 6.2 presents results for our debugging experiment. Several of the injected buggy statements were quite close to the failure points of the programs, and hence both the traditional and thin slicers found the bugs very quickly. For example, with `jtopas-1`, the buggy statement itself fails with a `NullPointerException`. These sorts of bugs can be easily debugged without tool support, but we include them for completeness.

Using the traditional slicer required inspecting 1 to 4.52 times more statements than thin slicing to find the bug. The total number of inspected statements for traditional slicing was 3.3 times higher than with thin slicing, a measure of the total inspection effort saved. The injected bugs in `nanoxml` in particular often required tracing a value as it is inserted and later retrieved from one or two `Vector`s, as in the

Bug	# Thin	# Trad.	Ratio	# Control	# ThinCIPA	# TradCIPA
nanoxml-1	12	32	2.67	0	12	32
nanoxml-2	25	113	4.52	0	431	1675
nanoxml-3	29	123	4.24	0	472	1883
nanoxml-4	12	33	2.75	1	17	44
nanoxml-5	35	156	4.46	1	159	45
nanoxml-6	12	52	4.33	0	35	90
jtopas-1	1	1	1	0	1	1
jtopas-2	2	2	1	1	2	2
ant-1	2	2	1	1	2	2
ant-2	4	5	1.25	0	4	5
ant-3	34	55	1.62	15	251	501
ant-4	3	3	1	2	3	3
xml-security-1	2	2	1	1	2	2

Table 6.2: Evaluation of thin slicing for debugging. For each bug, we show the number of statements that must be inspected in the thin slice (the "Thin" column) and the traditional slice (the "Trad" column) to discover the bug using BFS traversal (see §6.5.1). We also give the ratio of traditional statements to thin slice statements, and the number of control dependences that must be exposed to find the bug; the numbers for thin and traditional slices include these control dependences. Finally, we give the number of inspected statements for thin and traditional slicing when context-insensitive points-to analysis is used, without object-sensitive handling for containers (the "ThinCIPA" and "TradCIPA" columns). Slicing of any kind was not useful for five bugs in `xml-security` and one bug in `ant`; these bugs do not appear in the table.

example of Figure 1.3. Tracing this flow by hand can be difficult and time-consuming, and hence we think that thin slicing can have the greatest impact for this type of bug.

Debugging `nanoxml-5` required exposing statements causing aliasing (see §6.3.1), for reasons similar to those of the example in Figure 6.3. To simulate this user action, we ran the thin slicer in a configuration that included statements explaining one level of indirect aliasing. The results show that exposing such statements in this controlled manner is useful, as we still inspected significantly fewer statements than the traditional slice.

Few control dependences were relevant for these debugging tests, validating our

decision to ignore control dependence in thin slices. For all but one bug, the number of control dependences that need to be followed is 2 or less. These control dependences were always obvious from code surrounding the thin slice (as discussed in §6.3.2). The high number of control dependences for `ant-3` is due to the fact that the buggy function has 12 return statements, and one of them is directly control dependent on the bug; we included one control dependence for each return, as it is not obvious which one caused the bug. Nevertheless, all the control dependences were still near statements in the thin slice.

The precision of our preliminary points-to analysis was key to the effectiveness of the thin slicer. The "ThinCIPA" and "TradCIPA" columns in Table 6.2 show our results to be considerably worse with a points-to analysis that does not treat container classes like `Vector` object sensitively. In cases involving such data structures, the number of statements inspected with the thin slice increased by up to a factor of 17.2X with the less precise analysis, likely making the thin slicing tool unusable.

Finally, for five bugs in `xml-security` and one bug in `ant`, no type of slicing could help the user find the bug, and hence they do not appear in the table. The `xml-security` bugs all followed the same pattern:

```
long hash = computeHash(input); // buggy
assert hash == expectedHash; // fails
```

In `xml-security`, the `computeHash()` equivalent is complex, spanning several .class files, and the injected bugs were buried in the algorithm internals. In such cases, slicing from this assertion failure (whether static or dynamic) will inevitably bring in most or all of the code that computes the hash function. This example illustrates that slicing of course is not a panacea; delta debugging [Zel02] or refactoring to test at a finer granularity may help in these situations. We find the fact that thin slicing was useful for 13 out of 19 inspected bugs encouraging.

```
1  class Node {
2    final int op;
3    static int ADD_NODE_OP = 1;
4    Node(int op) { this.op = op; }
5  }
6  class AddNode extends Node {
7    AddNode(...) {
8      super(ADD_NODE_OP); ...
9    }
10 }
11 void simplify(Node n) {
12   int op = n.op;
13   switch (op) {
14   case ADD_NODE_OP:
15     AddNode add = (AddNode) n;
16   ...
17   }
18 }
```

Figure 6.4: An example illustrating a tough cast. Expressions in the thin slice used to understand the safety of the cast are underlined.

In summary, we found that for these injected bugs, thin slices very often contain the buggy statements, and the bugs could be found more quickly with a thin slicer than a traditional slicer. Also, 11.5 statements on average required inspection with the thin slicer (ranging from 1 to 35), quite a manageable number; the average for the traditional slicer was significantly larger at 54.8 statements, ranging from 1 to 156.

6.5.3 Experiment: Understanding Tough Casts

Our second experiment involved using slicing to hand-validate the safety of *tough casts* in the SPECjvm98 benchmarks. A tough cast is a downcast in a program that cannot be verified by precise and scalable pointer analysis (we used the same pointer analysis used to construct our call graph). For example, the cast at line 15 in Figure 6.4, adapted from the `javac` benchmark, is a tough cast. This cast cannot fail because the value of the `op` field of `AddNode` objects is `ADD_NODE_OP`, as guaranteed

by line 8, and no other subclasses of `Node` (not shown) have `ADD_NODE_OP` in their `op` field. Typically, tough casts are those that (1) are not used to cast values retrieved from a container (due to lack of generics) and (2) are not dominated by an explicit **`instanceof`** check ensuring their safety.

Tough casts present a good test of the efficacy of thin slicing in aiding program understanding. The safety of tough casts is often due to some global invariant of a program. These invariants are often (in our experience) undocumented, and discovering the invariants can aid the programmer in understanding the overall structure and behavior of the program. Furthermore, discovering these invariants by hand can be difficult, as it often requires tracing value flow through several disparate parts of the program. Hence, easing the understanding of tough casts with tool support aids overall program understanding and additionally can be useful for refactoring or adding parametrized types or annotations.

Our experimental configuration involved first manually identifying those statements that showed each tough cast could not fail (the desired statements of §6.5.1) with the help of the thin slicer, and then comparing the BFS traversal sizes of the thin and traditional slices from the cast to these desired statements. In the example of Figure 6.4, we can understand the tough cast through thin slicing by following a control dependence from the cast, and then computing a thin slice for line 12 to see what value `op` gets for different subclasses of `Node`. For each SPEC benchmark, we investigated 10 tough casts at random, or all tough casts if there were fewer than 10.

Note that the `compress` and `db` benchmarks had no tough casts, and `mpegaudio` was excluded since its bytecodes are obfuscated, making understanding its casts difficult. Also, we failed to determine the reason for cast safety for 6 casts in `javac` and one cast in `jess`. In these cases, the safety of the cast relies on some subtle invariant that is not easy to determine for one unfamiliar with the code.

The thin slicer significantly eased the manual process of determining the desired

Cast	# Thin	# Trad.	Ratio	# Control	# ThinCIPA	# TradCIPA
mtrt-1	22	51	2.32	0	22	51
mtrt-2	23	52	2.26	0	23	52
jess-1	6	7	1.17	2	6	7
jess-2	13	39	3	0	25	93
jess-3	6	6	1	2	6	6
jess-4	6	7	1.17	2	6	7
jess-5	6	7	1.17	2	6	7
jess-6	6	6	1	2	6	6
javac-1	57	910	16	1	57	910
javac-2	43	853	19.8	1	43	853
javac-3	65	2224	34.2	1	65	2267
javac-4	45	855	19	1	45	855
jack-1	18	79	4.39	0	303	758
jack-2	57	151	2.65	0	339	647
jack-3	18	69	3.83	0	304	603
jack-4	18	79	4.39	0	304	759
jack-5	57	151	2.65	0	339	647
jack-6	35	132	3.77	0	338	802
jack-7	35	132	3.77	0	338	802
jack-8	35	132	3.77	0	338	802
jack-9	30	79	2.63	0	304	759
jack-10	57	151	2.65	0	339	647

Table 6.3: Evaluation of thin slicing for understanding tough casts. The types of data in the table columns are described with Table 6.2.

statements for each tough cast. Although the code was unfamiliar to us, our thin slicing tool guided us through heap-based value flow, saving a great deal of time. The thin slicer was especially helpful when source code was *not* available, *e.g.*, for the `jack` benchmark, as we had to study a compiler representation of the bytecodes and could not use standard IDE-based source navigation tools.

Results for the tough casts experiment appear in Table 6.3. Thin slicing helped understand tough casts more effectively than traditional slicing: the number of statements examined using a traditional slice exceeded by 1.17 to 34.2 times the number examined using a thin slice. In total, 9.4 times more statements were examined with the traditional slicer than the thin slicer. In `javac`, the casts resembled Fig-

ure 6.4. The code includes a large number of `Node` subclasses used pervasively in the program, resulting in large numbers for the traditional slicer. The importance of object-sensitive container handling in the points-to analysis is seen for the `jack` casts, where the number of inspected statements increased by factors of 5.9-16.9X with less precise analysis.

The absolute numbers of inspected statements exceeded those for the debugging tests, but they remained manageable for a user. The thin slicer required inspecting an average of 29.3 statements (ranging from 6-65), while the traditional slicer required an average of 280 (ranging from 6 to 2224). For `javac`, many of the thin slice statements were writes of opcodes in a large number of constructors (like in Figure 6.4), which could be quickly inspected to ensure that a suitable constant is written. For `jack`, the BFS traversal over-estimated the number of thin slice statements that needed to be inspected; once we understood the benchmark, we could terminate the search early at some statements which we knew would not cause the cast to fail.

In summary, we conclude thin slicing can effectively provide tool support to identify statements that ensure tough casts cannot fail. A traversal based on thin slicing typically touches significantly fewer statements than a traversal based on traditional transitive flow dependence.

Chapter 7

Related Work

Here we discuss the previous work most closely related to our work on points-to analysis and thin slicing. First, we discuss previous work on pointer analysis in §7.1. Then, we discuss work related to thin slicing in §7.2.

7.1 Pointer Analysis Related Work

In §1.2.2, we gave a brief overview of the progression of points-to analysis research over the last 15 years. Here, we give a more detailed comparison of our points-to analysis with closely related work, specifically covering the following key areas:

- Context-insensitive points-to analysis
- Context-sensitive points-to analysis
- Refinement-based points-to analysis
- Demand-driven points-to analysis
- Incremental points-to analysis

- CFL-reachability
- Cast verification

We do *not* aim to comprehensively discuss all work on points-to analysis; see [O'C00, Hin01, GC01, Ryd03] for more comprehensive discussions and comparisons of various points-to analyses.

7.1.1 Context-insensitive points-to analysis

The key difference between the context-insensitive points-to analysis of Chapter 4 and most previous work is our demand-driven, refinement-based approach. Several exhaustive context-insensitive points-to analyses [And94, Ste96, FFSA98, Das00, SFA00, HT01b, RMR01, WL02, LH03, BLQ+03, ZC04] were discussed in §1.2.2. We showed in §4.4 that our analysis could nearly match the precision of Andersen's analysis for tested clients with a budget of 2ms per query, yielding up to a 16X speed improvement over a state-of-the-art exhaustive algorithm [LH03].

Previous work has studied the trade-offs between field-based and field-sensitive points-to analysis for Java (see §2.2.2 for relevant definitions), as we do through refinement with match edges. Liang et al. experimented with a variety of points-to analysis algorithms for Java [LPH01]. They conclude that field-sensitivity yields little benefit over field-based analysis for the extra required effort. Our precision results in the demand-driven setting support this conclusion. The field-based program representation used in the work of Lhoták and Hendren [LH03] is similar to our graph with conservative match edges; instead of a match edge, they create a node for each field, and represent getfields and putfields to assignments from and to the field node. This representation works well for exhaustive propagation of points-to sets, but the match edge representation is more suitable for our refinement techniques.

Algorithm	Eq / Sub	CS CG	CS Heap	Shown to Scale
Zhu / Whaley [WL04, ZC04]	Sub			X
Whaley [WR99]	Sub			
Choi [CGS+99]	Sub			
Fähndrich [FRD00]	Eq	X		X
Rehof [RF01]	Sub	X		
Emami [EGH94]	Sub	X		
Wilson [WL95]	Sub	X		
Cherem [CR04]	Eq		X	
Ruf [Ruf00]	Eq		X	1.1 lib
Liang [LH01]	Eq		X	
Guyer [GL03]	Sub		X	
Lattner [LLA07]	Eq	X	X	
O'Callahan [O'C00]	Eq	X	X	
Steensgaard (CS) [Ste]	Eq	X	X	X
Wang [WS01]	Sub	X	X	1.1 lib
Object-sensitive [MRR05, LH06]	Sub	X	X	1-limited
Naik [NAW06]	Sub	X	X	3-limited
Current paper	Sub	X	X	X

Table 7.1: A comparison of key properties of previous analyses. Algorithms are named by first author unless they have been referred to differently in this paper; note that Steensgaard (CS) [Ste] is different than his original analysis [Ste96]. The "Eq/Sub" column indicates whether assignments are modeled with equality or subset constraints, and the "CS CG" and "CS Heap" columns respectively indicate the use of a context-sensitive call graph and heap abstraction (see §2.2.4 and §2.2.5 for definitions). Finally, the "Shown to Scale" column indicates whether the algorithm has been shown to scale to large Java benchmarks; "1.1 lib" means the smaller Java 1.1 libraries were analyzed, and *k-limiting* [Shi88] is indicated where used.

7.1.2 Context-sensitive points-to analysis

The context-sensitive points-to analysis of Chapter 5 is distinguished from previous work by its ability to scalably compute a context-sensitive heap abstraction and call graph while requiring far less memory than existing approaches. Table 7.1 gives key properties of several other context-sensitive points-to analysis algorithms; here we

summarize some of the approaches taken by these analyses.

While effective for clients like escape analysis, summary-based analyses with subset constraints [WL95, WR99] have only been shown to scale to medium-sized programs. Summary-based analyses that use equality constraints have typically been more scalable [O'C00, FRD00, Ste, LLA07]. The algorithms of O'Callahan [O'C00] and Lattner and Adve [LLA07] scale well with a context-sensitive heap abstraction, but are less scalable when computing a context-sensitive call graph. A similar analysis for C# scales with a context-sensitive call graph [Ste], but still requires more than 1GB of memory on its largest benchmark.

Binary decision diagrams (BDDs) have been used in several recent systems to greatly improve the scalability of context-sensitive analysis. The Zhu and Calman [ZC04] and Whaley and Lam [WL04] algorithm, while quite scalable, uses a context-insensitive heap abstraction and call graph, leading to precision loss [LH06]. We compare extensively with the BDD-based 1-limited object-sensitive analysis of [LH06] in Section 5.4; object-sensitive analysis [MRR05] analyzes methods separately based on the receiver object instead of using call strings, exploiting typical object-oriented code structure for greater precision and scalability.

Naik *et al.* present a static race detection tool based on a scalable 3-limited object-sensitive analysis [NAW06]. It is difficult to compare our analysis with theirs directly, as their race detection client raises object-sensitive queries, which are in general incomparable with context-sensitive queries [MRR05]. As future work we plan to design an object-sensitive version of our analysis, allowing for a better comparison.

Hackett and Aiken [HA06] present a points-to analysis for C with context sensitivity (including a context-sensitive heap abstraction), flow sensitivity, and partial path sensitivity. The analysis is summary based and implemented through a translation to SAT. Their analysis takes several hours to analyze large C programs.

7.1.3 Refinement-based points-to analysis

Plevyak and Chien [PC94] present a type inference algorithm for object-oriented programs that uses refinement whenever a potential imprecision in the analysis result is detected. The refinement is accomplished through cloning, *e.g.*, analyzing a function separately for different call sites. In contrast to their work, which aggressively refines away all imprecision, our analyses only refine in parts of the program deemed likely to yield a more precise result for the client. This more selective refinement is key to scalability.

Guyer and Lin [GL03] present a client-driven points-to analysis for C that detects which statements cause imprecision for a given client, and then re-analyzes the program with greater flow and context sensitivity for those statements. Their results show that they obtain much of the precision benefit of flow and context sensitivity at a small extra cost, and their work was an inspiration for ours. The key difference with our work is similar to the difference with Plevyak and Chien's, namely that their analysis adds sensitivity to all possibly polluting statements when imprecision is detected. This approach does not scale for Java, as it requires too much code to be treated precisely.

7.1.4 Demand-driven points-to analysis

The demand-driven points-to analysis of Heintze and Tardieu [HT01a] is the best known in the literature. The analysis is for C, and they show promising results with a client that tries to resolve calls through function pointers. We show in §4.4 that when adapted to Java, this analysis does not provide satisfactory performance.

7.1.5 CFL-reachability

Our use of CFL-reachability is based on the work of Reps *et al.* on developing and utilizing the CFL-reachability framework [RHSR94, RHS95, HRS95, Rep98]. The L_{F} grammar in Figure 3.2 is an adaptation of the grammar given for Andersen's analysis for C in [Rep98]. Our key insight was to recognize that Andersen's analysis for Java is a balanced-parentheses problem when expressed in terms of CFL-reachability, a structure we exploit in both our context-insensitive and context-sensitive algorithms.

Our match edges are related to the summary edges used by the tabulation algorithm for balanced parentheses languages [RHS95, RHSR94, Rep94]. Summary edges are computed bottom-up as paths between parentheses are found. In contrast, match edges are added exhaustively and then refined by checking for paths between parentheses.

CFL-reachability formulations lead directly to demand-driven algorithms through the use of the magic-sets transformation [Rep94]. The inference rules of FullFS (presented in §4.6), an adaptation of the demand-driven algorithm of Heintze et al. [HT01a], correspond exactly to a magic-sets transformation of the grammar in Figure 3.2 with *pointsTo* as the start symbol.

Recent work of Kodumal and Aiken [KA04, KA07] has elucidated the connection between CFL-reachability and set constraints. Their work shows how to efficiently solve balanced-parentheses CFL-reachability problems with a set constraints solver [KA04], and more recent work shows how to additionally handle an intersected regular language [KA07]. As presented, their work requires explicit construction of the state machine for the regular language, which would make applying their technique to our context-sensitive points-to analysis formulation difficult to scale (the number of states in R_{C} (see §5.2) is exponential in the size of the program).

7.1.6 Incremental points-to analysis

Some recent work makes promising advances in performing incremental points-to analysis. Kodumal and Aiken show how to perform a limited form of incremental analysis in a set constraints solver using backtracking [KA05]. Their technique is most effective in cases where code changes are primarily limited to a small set of source files, which they show is a typical development pattern. Hirzel et al. present a points-to analysis implemented in a JIT compiler that handles all Java language features, which can quickly update its computed results after program changes [HDDH07]. Our approach to handling code changes is to recompute from scratch points-to queries that are affected by a program change. In cases where the number of queried variables is moderate, our approach has the advantages of simplicity, as no engineering for incrementality is needed, and of not needing to cache intermediate analysis results, which can consume significant amounts of memory,

Choi et al. [CGS+99] and Whaley and Rinard [WR99] define flow and context-sensitive points-to and escape analyses for Java. Vivien and Rinard extend the analysis in [WR99] to be incremental [VR01], focusing analysis effort on code deemed to be likely to yield profitable results. Their results show that their incremental analysis obtains most of the effect of an exhaustive analysis in much less time. Our positive results for early termination may be explained by similar underlying principles. It is unclear how long their analysis takes to answer individual queries.

7.1.7 Cast verification

Constraint-based analyses have been developed to convert legacy Java programs to use Java 5 generics, and they have been shown to prove many downcasts safe [DKTE04, FTD+05]. These analyses rely on the generics annotations of Java 5 `java.util` classes to model their behavior. In contrast, our approach determines properties of library

code without annotations, and hence handles application data structures as well. Furthermore, our analysis can be used for more than cast safety, as shown by our factory method client. In other work, Wang and Smith present a context-sensitive constraint-based type analysis [WS01] and show that it is effective at proving downcasts safe. However, they analyze the Java 1.1 libraries, which are significantly smaller than the Java 1.3 libraries used in the present work.

In general, there are various type inference algorithms that are in some cases comparable to points-to analyses. Cartesian-product analysis [Age95] was an early inference algorithm that precisely handled polymorphic code (through context sensitivity), and numerous variants have been proposed since then (*e.g.*, in some of the aforementioned cast verification work [WS01, DKTE04]). For some clients, such type inference algorithms are insufficient, and a points-to analysis is required. For example, tainting analysis (*e.g.*, [LL05]) relies on tracking the flow of individual objects from untrusted sources to trusted sinks, rather than just inferring a precise concrete type for the sink.

7.2 Thin Slicing Related Work

Since first being defined by Weiser in 1979 [Wei79], slicing has inspired a large body of work on computing slices and on applications to a variety of software engineering tasks. We refer the reader to Tip's survey [Tip95] and Krinke's thesis [Kri03] for broad overviews of slicing technology and challenges. Here, we focus on the work most relevant to our own.

Our thin slicing algorithm is a straightforward adaptation of the SDG-based approach first presented by Horwitz et al. [HRB88]. Our implementation of a traditional slicer is in fact tabulation-based, as suggested in [Rep98] and the 20-year Retrospective to [HRB88].

CodeSurfer [Cod] is a program understanding tool for C and C++ based on the analysis techniques of [HRB88, RHSR94]. CodeSurfer also uses pointer analysis to allow navigation from a use of a heap location to potential defs. Our evaluation metric of a breadth-first traversal strategy aims to simulate use of a tool like CodeSurfer, which allows for navigating the dependence graph. While CodeSurfer allows navigation of all control and data dependences, thin slicing emphasizes producer statements and shows explainer statements using additional thin slices (see §6.1); our evaluation has shown that this technique quickly leads users to the most relevant statements.

Atkinson and Griswold [AG98] present a slicer relying on a preliminary flow-insensitive pointer analysis. This work targets C, and so had to deal with difficulties arising from issues such as stack-directed pointers and unsafe memory access, which do not arise in Java. Larsen and Harrold [LH96] presented one of the first slicing approaches for object-oriented software, adding pseudo-parameters for fields to track dependencies through the heap. Our context- sensitive slicer implementation uses a similar approach, but relies on a partially context-sensitive preliminary pointer analysis to disambiguate locations with field- and object-sensitivity, and additional pseudo-parameters to soundly handle all fields that may be accessed transitively by callees.

In recent years, several papers have improved precision by integrating more precise static alias analysis into slicing. Liang and Harrold [LH98] present a novel approach to represent formal parameter objects with trees. Hammer and Snelting [HS04] present an enhancement to this approach, including a criterion for sound limiting of tree sizes for recursive data structures. Both these approaches are more powerful than relying solely on a preceding flow-insensitive alias analysis, since must-alias information on parameters can allow sound strong updates. It is not clear how far these algorithms scale; the experimental results of Hammer and Snelting address programs significantly smaller than the benchmarks considered here. In future work, we plan to incorporate

aspects of Hammer and Snelting's approach for thin slices.

Orso et al. [OSH04] present a classification of data dependence edges in an SDG, based on certainty of may-alias information, and the span (scope) of a program over which a data dependence flows. They propose an incremental slicing procedure to aid debugging, whereby a tool can provide progressively larger slices by including progressively more classes of data dependencies. Our thin slice expansion technique is similar in spirit, but goes in a different direction by expanding slices to include statements that indirectly give rise to the primary alias relations.

Mock et al. [MACE02] showed that for C programs with heavy pointer use, using dynamic points-to data significantly improved slice precision over a conservative flow-insensitive pointer analysis. We suspect Java programs resemble C programs with heavy pointer use with regard to data dependence.

PSE [MSA+04] is a static analysis tool for localizing the causes of typestate errors in C and C++ programs by essentially computing a variant of a backward slice, with extra filtering based on the type of error. Their system is able to perform strong updates on the heap in some situations, using a technique we plan to try in our thin slicer. Their work unsoundly ignores may-aliasing in some configurations, partly due to the fact that traces involving aliasing are hard for developers to understand. If applied to a Java-like language, our technique for explaining aliases in thin slices may help solve this problem, as discussed in §6.3.1.

Recently, Zhang et al. have considerably improved the state-of-the-art in dynamic slicing [ZTG06, ZGZ04, ZGG06a, ZGG06b]. Thin slicing applies naturally to dynamic data dependences, and we believe dynamic thin slices could provide benefits similar to static thin slices. Zhang et al.'s work on improving scalability [ZGZ04, ZTG06] could be leveraged to create a more scalable dynamic thin slicer. Their recent work on pruning dynamic slices [ZGG06a] is complementary to ours: thin slicing and their heuristics for determining when a statement is unlikely to be relevant (based on

which statements output good and bad values) could be fruitfully combined. Recent work [ZGG06b] observes that using dynamic data dependences alone can often identify buggy statements in C programs; we suspect that in fact those data dependences considered by a thin slicer would also be sufficient. Finally, this work [ZGG06b] also suggests exploring statements closer to the seed first when viewing a slice, an idea we also use in our evaluation.

Chapter 8

Conclusions and Future Work

We have presented two refinement-based techniques for analysis and understanding of large object-oriented programs. Our refinement-based points-to analysis exploits the balanced parentheses structure of Java heap accesses to provide both scalability and precision. Thin slicing uses a novel notion of relevance and user-guided refinement to better focus programmer attention on those statements most relevant to development tasks. Together, our points-to analysis and thin slicing enable new types of programming tools that could significantly ease the process of developing and maintaining large-scale object-oriented programs.

Many clients other than those we tested could benefit from the scalability and precision of our refinement-based points-to analysis. An incomplete list of recent analyses that may benefit includes work on type-state verification [FYD+06], race detection [NAW06], reflection analysis [LWL05], and analyses for security bugs [LL05]. By using our pointer analysis, these analyses may be able to much larger programs or reduce their false positive rate through more precise handling of the heap. Additionally, use with a wider variety of clients may expose opportunities to improve the points-to analysis itself, *e.g.*, with a different refinement policy.

A practical IDE-based implementation of our points-to analysis would pose some interesting engineering challenges. One key issue would be how to maintain the global program representations still required by the points-to analysis, *i.e.*, a pre-computed call graph and information on where fields are accessed (for addition of match edges). IDEs like Eclipse do very little caching of program representations like abstract syntax trees to reduce memory overhead. Maintaining a call graph in memory for a large program may be too expensive, necessitating a balance of on-demand computation and limited caching. In addition, any global data like a call graph must be updated incrementally as the programmer edits the code, which may pose another interested engineering challenge.

A parallel implementation of our points-to analysis could yield a large scalability improvement. Since our analysis does no sharing of state across queries, the analysis can be trivially parallelized by running each query on its own processor. The memory usage of our analysis may need to be more carefully engineered to make good use of the cache on a multiprocessor. However, with proper engineering, a parallel implementation of the pointer analysis could provide large real-world speedups, as multicore processors are becoming pervasive.

The next step in validating thin slicing would be to build an IDE-based implementation and test its usefulness with developers. Various user-interface issues may arise during this process. The thin slice could be shown to the user in a standard tree view, but a graph view may be more intuitive, especially in cases where expansion is necessary. Also, assuming the thin slicer is built atop our refinement-based pointer analysis, it remains an open question how to handle budgets and early termination in the user interface.

Ideally, an IDE-based thin slicer would make extensive use of developer feedback. Better incorporation of such feedback into the compilation and analysis process has been a longstanding open problem (*e.g.*, see Knuth's study [Knu71]). Thin slicing

provides a promising avenue for progress on this issue, as users can easily track a small number of statements that the analysis thinks are relevant to the seed statement. One could imagine various ways that the user could help improve the analysis result, *e.g.*, by providing reasons why certain statements are not relevant or by suggesting a different refinement policy for the underlying pointer analysis. Lessons learned from implementing such a thin slicer and evaluating it in a user study may be broadly applicable to software quality tools.

Bibliography

[AAB+00] Bowen Alpern, Dick Attanasio, John Barton, Michael Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Ton Ngo, Mark Mergen, Vivek Sarkar, Mauricio Serrano, Janice Shepherd, Stephen Smith, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeno virtual machine. *IBM Systems Journal, Java Performance Issue*, 39(1), 2000.

[AG98] Darren C. Atkinson and William G. Griswold. Effective whole-program analysis in the presence of pointers. In *Foundations of Software Engineering*, pages 46–55, 1998.

[Age95] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP*, pages 2–26, 1995.

[AH03] Matthew Allen and Susan Horwitz. Slicing Java programs that throw and catch exceptions. In *PEPM '03: Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 44–54, New York, NY, USA, 2003. ACM Press.

[And94] Lars O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, 1994.

[Ash] Ashes suite collection. http://www.sable.mcgill.ca/software/.

[BLQ+03] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[BR01] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[BS96] David Bacon and Peter Sweeney. Fast static analysis of C++ virtual function calls. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), San Jose, CA*, October 1996.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 1999.

[Cha06] Swarat Chaudhuri. CFL-reachability in subcubic time. Technical report, IBM Research Report RC24126, 2006.

[Cod] CodeSurfer. http://www.grammatech.com/products/codesurfer/.

[CR04] Sigmund Cherem and Radu Rugina. Region analysis and transformation for java programs. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, 2004.

[CU91] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 26, pages 1–15, New York, NY, 1991. ACM Press.

[DaC] DaCapo Benchmark Suite. http://www-ali.cs.umass.edu/DaCapo/gcbm.html.

[DADY04] Nurit Dor, Stephen Adams, Manuvir Das, and Zhe Yang. Software validation via scalable path-sensitive value flow analysis. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 12–22, New York, NY, USA, 2004. ACM Press.

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Conference on Programming Language Design and Implementation (PLDI), Vancouver, British Columbia, Canada*, June 2000.

[DER05] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4), October 2005.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, August 1995.

[DGS97] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, November 1997.

[DKTE04] Alan Donovan, Adam Kiezun, Matthew S. Tschantz, and Michael D. Ernst. Converting Java programs to use generic libraries. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004.

[DLFR01] Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. In *SAS*, pages 260–278, 2001.

[DLS02] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: path-sensitive program verification in polynomial time. In *Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[DMM98] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 106–117, New York, NY, USA, 1998. ACM Press.

[Ecl] The Eclipse Platform. http://www.eclipse.org/.

[EcP] Eclipse Plugin Central. http://www.eclipseplugincentral.com.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256, New York, NY, USA, 1994. ACM Press.

[FFA00] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for c. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, pages 175–198, London, UK, 2000. Springer-Verlag.

[FFSA98] Manuel Fändrich, Jeffrey S. Foster, Zhendong Su, and Alex Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada*, June 1998.

[FKS00] Stephen Fink, Kathleen Knobe, and Vivek Sarkar. Unified analysis of array and object reference in strongly typed languages. In *Proceedings of the 2000 Static Analysis Symposium*, June 2000.

[Fow99] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[FQ03] S. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement, 2003.

[FRD99] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. From polymorphic subtyping to CFL-reachability: Context-sensitive flow analysis using instantiation constraints. Technical report, Microsoft Research Technical Report MSR-TR-99-84, 1999.

[FRD00] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[FTD+05] Robert Fuhrer, Frank Tip, Julian Dolby, Adam Kiezun, and Markus Keller. Refactoring techniques for migrating applications to generic java container classes. In *European Conference on Object-Oriented Programming (ECOOP)*, 2005.

[FYD+06] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *International symposium on Software testing and analysis (ISSTA)*, 2006.

[GC01] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, 2001.

[GL03] Samuel Z. Guyer and Calvin Lin. Client-driven pointer analysis. In *International Static Analysis Symposium (SAS), San Diego, CA*, June 2003.

[HA06] Brian Hackett and Alex Aiken. How is aliasing used in systems software? In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 69–80, New York, NY, USA, 2006. ACM Press.

[Har78] Michael A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1978.

[HCXE02] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[HDDH07] Martin Hirzel, Daniel Von Dincklage, Amer Diwan, and Michael Hind. Fast online pointer analysis. *ACM Trans. Program. Lang. Syst.*, 29(2):11, 2007.

[HDWY06] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2006.

[Hin01] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE), Snowbird, Utah*, June 2001.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[HPR89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Conference on Programming Language Design and Implementation (PLDI)*, 1989.

[HRB88] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Conference on Programming Language Design and Implementation (PLDI)*, 1988.

[HRS95] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. In *SIGSOFT'95: Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1995.

[HS04] Christian Hammer and Gregor Snelting. An improved slicer for Java. In *Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 17–22, 2004.

[HT01a] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah*, June 2001.

[HT01b] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah*, June 2001.

[J2E] Java Platform Enterprise Edition. http://java.sun.com/javaee.

[J2S] Java Platform Standard Edition. http://java.sun.com/javase.

[jEd] jEdit: Open source programmer's text editor. http://www.jedit.org.

[KA04] John Kodumal and Alex Aiken. The set constraint/CFL-reachability connection in practice. In *Conference on Programming Language Design and Implementation (PLDI)*, June 2004.

[KA05] John Kodumal and Alex Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS '05: Proceedings of the 12th International Static Analysis Symposium*. London, United Kingdom, September 2005.

[KA07] John Kodumal and Alex Aiken. Regularly annotated set constraints. In *PLDI*, pages 331–341, 2007.

[Knu71] Donald E. Knuth. An empirical study of FORTRAN programs. *Softw., Pract. Exper.*, 1(2):105–133, 1971.

[Kri03] Jens Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, University of Passau, 2003.

[LA05] Chris Lattner and Vikram Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 129–142, New York, NY, USA, 2005. ACM Press.

[Lam04] Butler W. Lampson. Software components: Only the giants survive. In K. Sparck-Jones and A. Herbert, editors, *Computer Systems: Theory, Technology, and Applications*, pages 137–146. Springer-Verlag, 2004.

[LH96] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *International Conference on Software Engineering (ICSE)*, 1996.

[LH98] Donglin Liang and Mary Jean Harrold. Slicing objects using system dependence graphs. In *ICSM*, pages 358–367, 1998.

[LH01] Donglin Liang and Mary Jean Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *Proceedings of the 8th International Symposium on Static Analysis*, 2001.

[LH03] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction (CC), Warsaw, Poland*, April 2003.

[LH04] Ondřej Lhoták and Laurie Hendren. Jedd: a BDD-based relational extension of Java. In *Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[LH06] Ondřej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: Is it worth it? In *International Conference on Compiler Construction (CC)*, 2006.

[Lho] Ondřej Lhoták. Personal communication. 2005.

[Lho02] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[Lho06] Ondřej Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, January 2006.

[LL05] V. Benjamin Livshits and Monica S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, August 2005.

[LLA07] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, California, June 2007.

[LPH01] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE), Snowbird, Utah*, June 2001.

[LPH02] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Evaluating the precision of static reference analysis using profiling. In *Internal Symposium of Software Testing and Analysis (ISSTA), Rome, Italy*, July 2002.

[LT93] Nancy G. Leveson and Clark S. Turner. Investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[LWL05] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for Java. In Kwangkeun Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, volume 3780. Springer-Verlag, November 2005.

[MACE02] Markus Mock, Darren C. Atkinson, Craig Chambers, and Susan J. Eggers. Improving program slicing with dynamic points-to data. *SIGSOFT Softw. Eng. Notes*, 27(6):71–80, 2002.

[McC06] Kevin McCoy. How the IRS failed to stop $200M in bogus refunds. *USA Today*, December 5, 2006.

[MRR05] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[MSA+04] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. PSE: explaining program failures via postmortem static analysis. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 63–72, New York, NY, USA, 2004. ACM Press.

[MXBK05] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[NAW06] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *CC*, pages 138–152, 2003.

[NR69] Peter Naur and Brian Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch,*

Germany, 7-11 Oct. 1968, Brussels. Scientific Affairs Division, NATO, 1969.

[O'C00] Robert O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Carnegie Mellon University, November 2000.

[OJ97] Robert O'Callahan and Daniel Jackson. Lackwit: a program understanding tool based on type inference. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 338–348, New York, NY, USA, 1997. ACM Press.

[OSH04] Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold. Classifying data dependences in the presence of pointers for program comprehension, testing, and debugging. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 13(2):199–239, 2004.

[oST] National Institute of Standards and Technlogy. Software errors cost U.S. economy $59.5 billion annually. NIST News Release 2002-10.

[PC94] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Portland, Oregon*, October 1994.

[Rep94] Thomas Reps. Solving demand versions of interprocedural analysis problems. In *International Conference on Compiler Construction (CC), Edinburgh, Scotland*, April 1994.

[Rep98] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, November/December 1998.

[Rep00] Thomas Reps. Undecidability of context-sensitive data-independence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1):162–186, 2000.

[RF01] Jakob Rehof and Manuel Fähndrich. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2001.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1995.

[RHSR94] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), New Orleans, LA*, December 1994.

[RLS+01] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, and Rita Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Trans. Program. Lang. Syst.*, 23(2):105–186, 2001.

[RMR01] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Tampa Bay, Florida*, October 2001.

[RR03] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *IEEE International Conference on Automated Software Engineering (ASE)*, 2003.

[Ruf00] Erik Ruf. Effective synchronization removal for java. In *Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[Ryd03] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *International Conference on Compiler Construction (CC), Warsaw, Poland*, April 2003.

[SB06] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[SFA00] Zhendong Su, Manuel Fähndrich, and Alexander Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *ACM Symposium on Principles of Programming Languages (POPL), Boston, Massachusetts*, pages 81–95, January 2000.

[SGSB05] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

[Shi88] O. Shivers. Control flow analysis in scheme. In *Conference on Programming Language Design and Implementation (PLDI)*, 1988.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[Ste] Bjarne Steensgaard. Personal communication on analysis for Microsoft Bartok compiler. 2005.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1996.

[Tei] Tim Teitelbaum. Personal communication regarding CodeSurfer. 2007.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[VR01] Frederic Vivien and Martin C. Rinard. Incrementalized pointer and escape analysis. In *Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah*, June 2001.

[VRHS+99] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[WAL] T.J. Watson Libraries for Analysis. http://wala.sourceforge.net.

[Wei79] Mark David Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Conference on Programming Language Design and Implementation (PLDI)*, 1995.

[WL02] John Whaley and Monica Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *International Static Analysis Symposium (SAS), Madrid, Spain*, September 2002.

[WL04] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 1999.

[WS01] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for java. In *15th European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.

[Yan90] Mihalis Yannakakis. Graph-theoretic methods in database theory. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, April 1990.

[YHR99] Suan Hsi Yong, Susan Horwitz, and Thomas W. Reps. Pointer analysis for programs with structures and casting. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 91–103, 1999.

[ZC04] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In *Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[Zel02] Andreas Zeller. Isolating cause-effect chains from computer programs. *SIGSOFT Softw. Eng. Notes*, 27(6):1–10, 2002.

[ZGG06a] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Pruning dynamic slices with confidence. In *Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[ZGG06b] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Software Engineering*, 2006. To appear.

[ZGZ04] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *International Conference on Software Engineering (ICSE)*, 2004.

[ZJL+06] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical debugging: Simultaneous identification of multiple bugs. In *Proceedings of the 23rd International Conference on Machine Learning*, 2006.

[ZR07] Xin Zheng and Radu Rugina. Demand-driven alias analysis for c. Technical report, Cornell University CIS TR2007-2089, 2007.

[ZTG06] Xiangyu Zhang, Sriraman Tallam, and Rajiv Gupta. Dynamic slicing long running programs through execution fast forwarding. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2006.