PRECISE AND PRACTICAL FLOW ANALYSIS OF OBJECT-ORIENTED SOFTWARE

BY ANA MILANOVA

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Barbara Gershon Ryder

and approved by

____

____

____

____

New Brunswick, New Jersey

October, 2003

ABSTRACT OF THE DISSERTATION

Precise and Practical Flow Analysis of Object-Oriented Software

by ANA MILANOVA

Dissertation Director: Barbara Gershon Ryder

Program flow analysis is a technique which determines properties about the run-time behavior of a program by analyzing its source code. Flow information has a wide variety of uses in optimizing compilers and software tools for software understanding, testing and maintenance. There are two important requirements for a flow analysis to be successfully applied in optimizing compilers and software tools: (i) first the analysis needs to be relatively precise and (ii) the analysis needs to be practical. However, typically there is a tradeoff between analysis precision and analysis practicality.

Many existing practical flow analyses are based on inclusion constraints. However, these analyses do not model dimensions of analysis precision that are of crucial importance for the analysis of object-oriented languages and its usability in software tools and compilers.

The first contribution of this thesis is the development of annotated inclusion constraints—a general framework that allows relatively precise and at the same time practical analysis of large programs. The annotated constraint system is parameterized by an annotation language and operations on the annotations. The key idea is to take a relatively imprecise flow analysis that can be expressed using non-annotated inclusion constraints, and add a dimension of precision by choosing appropriate annotations and operations on the

annotations. Using this approach results in analyses that are substantially more precise while remaining efficient and practical.

The second contribution is the formulation and implementation of a field-sensitive points-to analysis for Java in the context of this framework. Points-to analysis for Java is a fundamental flow analysis with a wide variety of uses in optimizing compilers and software tools. Extensive experiments show that our analysis has practical cost and achieves substantial impact on various client applications.

Another contribution is the development of object sensitivity, a new form of context sensitivity for object-oriented languages. We have formulated and implemented several object-sensitive points-to analyses for Java as instances of the framework for annotated constraints. The empirical results show that object sensitivity substantially improves the precision of points-to and side-effect analyses over context-insensitive analyses. At the same time, the annotations model context sensitivity efficiently, achieving practical cost, comparable to the cost of context-insensitive analysis.

The last contribution is the application of relatively precise flow analysis for the construction of interclass dependence diagrams (to be used in integration and regression testing, impact analysis, reverse engineering, etc.). We have developed a general algorithm for diagram construction, and have shown empirically that using field-sensitive points-to analysis improves diagram precision substantially compared to earlier work. The enhanced precision improves the usability of the diagrams in software tools.

Acknowledgements

I would like to thank my advisor, Professor Barbara Gershon Ryder, for her unconditional help and support, and her belief in my abilities. She was always my mentor and she taught me many valuable lessons. I am very grateful to Nasko Rountev for all his help during my studies, and for the many things I learned from him. Without the great support from Barbara and Nasko I probably would not have finished my dissertation. I am also thankful to all the PROLANGS members, and the PROLANGS reading group members for providing interesting discussion and challenging and productive research environment.

I am deeply thankful to my parents and grandparents for their support, and to my parents-in-law, Ana and Petar Milas for taking care of Catherine, during the last year of my studies. Finally, I would like to thank my dearest husband, Antun Milas, for his constant encouragement, help and support, and for always having faith that I could finish my dissertation.

Dedication

To my grandfather Marin who passed away on April 27, 2003.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Program flow analysis is a technique which determines properties about the run-time behavior of a program by analyzing its source code. Traditionally, flow information has been used to enable various compiler optimizations. In addition, it has a wide variety of uses in software engineering tasks. For example, flow analysis can be used to construct the calling structure of an object-oriented software system which is essential for the understanding of the system. Flow information can be used to define test coverage criteria and to evaluate the coverage achieved by a test suite. It can be used to identify code related to a change in a software system which facilitates software maintenance. In addition, flow analysis information can be used to identify security vulnerabilities.

There are two important requirements for a flow analysis to be successfully applied in optimizing compilers and software tools, analysis *precision* and analysis *practicality*, but typically there is a tradeoff between precision and practicality. [1] In compilers analysis that is too imprecise leads to foregone optimizations. In the context of flow-analysis-based software tools, analysis precision is crucial because imprecision results in wasted human time and decreased productivity of the software developers, testers and maintainers. For example, in software testing, analysis that is too imprecise leads to low test coverage due to infeasible testing goals. As a result, the tester may spend valuable time trying to design test cases that exercise such goals until she determines that the low coverage is due to imprecise analysis and not to incomplete testing. In software maintenance, analysis imprecision may result in valuable developer time spent examining code which is unrelated to a change task. Analysis practicality is crucial both for

[1] In this thesis, we use the terms analysis precision and precise analysis to refer to *relative* precision (i.e., the analysis is relatively precise compared to similar analyses). A more detailed discussion of the notion of precision appears in Chapter 2.

optimizing compilers and software tools. In compilers, the analysis cost is constrained by compilation costs. In software tools, the analysis needs to be able to analyze large software systems in practical time because most of the time tool users are not willing to wait long for the analysis to complete; thus, slow analysis undermines the usability of a tool and leads to decreased developer productivity.

Because of the importance of analysis precision as well as its practicality, it is important to investigate techniques for relatively precise, and at the same time practical flow analysis. Many existing practical flow analyses are based on *inclusion constraint systems*. Such analyses examine the source code of the program and construct a system of constraints of the form $X \subseteq Y$, where X and Y are expressions representing sets; the solution of the constraint system provides information about the flow of values in the program. However, these analyses do not model dimensions of flow analysis precision such as field sensitivity (the ability to distinguish flow of values through different object fields) and context sensitivity (the ability to distinguish between different contexts of invocation of a method). These dimensions of precision are of crucial importance for the analysis of object-oriented languages because field insensitivity and context insensitivity inherently compromise analysis precision. This imprecision is due to fundamental object-oriented features and programming idioms such as encapsulation, inheritance, and polymorphism.

We propose *annotated* inclusion constraints which is a general framework that allows the modeling of different dimensions of flow analysis precision using inclusion constraint systems. It is based on constraints of the form $X \subseteq_a Y$ where the annotation a is an attribute of the inclusion relation. The annotated constraint system is parameterized by an annotation language and operations on the annotations; *precise* modeling of various dimensions of flow analysis precision can be achieved by choosing appropriate annotations. Most importantly, extensive empirical evaluation shows that the annotations *efficiently* model precision dimensions, retaining the practicality of the underlying inclusion constraint system.

1.1 Contributions

The work presented in this thesis has the following contributions.

1.1.1 Annotated Inclusion Constraints

The first contribution of this thesis is a general framework which allows analysis designers to design relatively precise flow analyses from analyses that are imprecise with respect to a dimension of precision. The key idea is to take a relatively *imprecise* flow analysis that can be expressed using non-annotated inclusion constraints, and add a dimension of precision by choosing appropriate annotations and operations on the annotations. The analysis generates annotated inclusion constraints, and uses the operations on the annotations to resolve these constraints. The solution of the system of annotated constraints contains the solution of the flow analysis problem which is precise with respect to the dimension of precision. This approach allows the reuse of techniques for efficient constraint representation and constraint resolution, as well as the physical reuse of efficient constraint solvers. Using this approach results in analyses that achieve substantially better precision while remaining efficient and practical.

1.1.2 Field-sensitive Points-to Analysis for Java

The second contribution of this thesis is the formulation and implementation of a field-sensitive *points-to analysis* for Java using *annotated inclusion constraints*. Points-to analysis for Java is a fundamental flow analysis which determines the set of objects pointed to by a reference variable or a reference object field. This information has a wide variety of client applications in optimizing compilers and software engineering tools. The relatively imprecise points-to analysis that our analysis is based on, is the field-insensitive, context-insensitive Andersen's points-to analysis for C [6]. We use constraint annotations to model precisely and efficiently two dimensions of precision (i) the flow of values through object fields and (ii) the semantics of virtual calls (i.e., on-the-fly call graph construction). By solving systems of annotated inclusion constraints, we have been able to perform the first practical and relatively precise points-to analysis for Java.

Similarly to many other static compilers and whole-program analyses for Java [36, 25, 68, 69, 42, 9], our analysis assumes that native methods, reflection and dynamic loading are resolved manually. These restrictions on our program model are discussed in detail later in the thesis (see Section 3.4, Chapter 3).

We evaluate the performance of the analysis on a large set of Java programs. Our experiments show that the analysis runs in practical time and space. We also show that the points-to solution has significant impact on clients such as call graph construction, virtual call resolution, synchronization removal, and stack-based object allocation. Our results demonstrate that the annotations model field sensitivity and virtual dispatch efficiently; the resulting analysis is a realistic candidate for a relatively precise, practical, general-purpose points-to analysis for Java.

1.1.3 Context-sensitive Points-to Analysis for Java

Another contribution of this thesis is the development of *object sensitivity*, a new form of context sensitivity suitable for flow-insensitive points-to analysis for Java. The key idea of our approach is to analyze a method separately for each of the objects on which this method is invoked. To ensure flexibility and practicality, we propose a parameterization framework that allows analysis designers to control the tradeoffs between cost and precision in the object-sensitive analysis.

We have formulated and implemented several object-sensitive points-to analyses as instances of our *annotated inclusion constraints* framework. The relatively imprecise analysis is the context-insensitive analysis outlined in Section 1.1.2, and the dimension of precision modeled by constraint annotations is object sensitivity. We compare these instantiations with the context-insensitive points-to analysis for Java from Section 1.1.2 which is based on Andersen's analysis for C. On a set of 23 Java programs, our experiments show that for the majority of programs these analyses have comparable cost. Our results also show that object sensitivity significantly improves the precision of call graph construction, virtual call resolution, and proving downcast safety. These experiments demonstrate that (i) object-sensitive analyses can achieve significantly better precision than context-insensitive ones, and (ii) the annotations model precision dimensions such as context sensitivity efficiently, allowing the object-sensitive analyses to remain efficient and practical.

Side-effect analysis determines the memory locations that may be modified by the execution of a program statement. This information is needed for various compiler optimizations and software engineering tools. We present a new form of side-effect analysis for Java which is based on object-sensitive points-to analysis, and demonstrate empirically that object sensitivity significantly improves the precision of side-effect analysis.

1.1.4 Applications to Software Engineering Problems

The *Object Relation Diagram (ORD)* of a program is a class interdependence diagram which has applications in a wide variety of software engineering problems (e.g., integration testing, integration coverage analysis, regression testing, impact analysis, program understanding, and reverse engineering). Because the imprecision of the ORD directly affects the feasibility of its usage, it is important to investigate techniques for constructing relatively precise ORDs.

Our work makes the following contributions. First, we develop the *Extended Object Relation Diagram (ExtORD)*, a version of the ORD designed for use in integration coverage analysis. The ExtORD shows the specific statement that creates an interclass dependence, and can be easily constructed by extending techniques for ORD construction. Second, we develop a general algorithm for ORD construction, parameterized by *class analysis*. Class analysis is a fundamental analysis that determines the possible classes of all objects that a reference variable may refer to. Third, we demonstrate empirically on a set of 23 Java programs that relatively precise class analysis based on the field-sensitive points-to analysis outlined in Section 1.1.2 can improve significantly the precision of the ORD and ExtORD compared to earlier work.

1.2 Thesis Outline

The rest of this thesis is organized as follows. The general structure of our framework for annotated inclusion constraints is presented in Chapter 2. Chapter 3 describes the field-sensitive points-to analysis for Java. Chapter 4 describes the context-sensitive analyses. Chapter 5 discusses analysis applications to software engineering problems. Related work is discussed in Chapter 6. Chapter 7 presents a summary of the thesis and directions for future work.

Chapter 2

Annotated Inclusion Constraints

This chapter discusses the major dimensions of precision in the flow analysis of object oriented languages, and describes the general structure of our framework for annotated inclusion constraints. Previous constraint-based flow analyses [21, 65] employ non-annotated inclusion constraints. We develop a new general constraint-based approach that extends previous work by introducing a parameterization mechanism which allows the analysis designer to define problem specific constraint annotations. These annotations can be used to model different dimensions of flow analysis precision, such as field sensitivity and various forms of context sensitivity.

Later in the thesis we present and evaluate specific analyses formulated in the context of the framework. Chapter 3 discusses field-sensitive points-to analysis for Java which employs constraint annotations to model field sensitivity and virtual dispatch. Chapter 4 discusses various novel forms of context sensitivity and the use of constraint annotations to model precisely context sensitivity. (The notions of field sensitivity, context sensitivity and virtual dispatch are discussed below.)

2.1 Dimensions of Precision in the Flow Analysis of Object-Oriented Programming Languages

Flow analysis answers questions of the form "Can a value created at program point p_1 flow to program point p_2?". Examples of flow analyses with wide ranges of uses in optimizing compilers and software tools are points-to analysis, class (type) analysis, and constant propagation.

The two major dimensions in the design space of flow analysis are flow sensitivity and context sensitivity. Intuitively, *flow-sensitive* analyses take into account the flow of control between program points inside a method, and compute separate solutions for these points. *Flow-insensitive* analyses ignore the flow of control between program points, and therefore can be less precise and more efficient than flow-sensitive analyses. *Context-sensitive* analyses distinguish between the different contexts under which a method is invoked, and analyze the method separately for each context. *Context-insensitive* analyses do not separate the different invocation contexts for a method, which improves efficiency at the expense of some possible precision loss. There are various mechanisms used for context sensitivity.

Other dimensions of flow analysis precision are field sensitivity, and the handling of virtual calls. *Field-sensitive* analyses distinguish flow of values through different object fields. *Field-insensitive* analyses do not distinguish between different object fields which results in less precise analysis results that can be more efficient. The handling of virtual calls refers to the method of constructing the calling structure of the program. In *on-the-fly* analyses, the analysis of reference variables for the purposes of call graph construction is intertwined with the flow analysis; values are propagated interprocedurally on a call graph constructed during the analysis. *Fixed-call-graph* analyses employ inexpensive algorithms for call graph construction to achieve a conservative approximation of the call graph[1]; values are propagated interprocedurally on the fixed call graph. On-the-fly analyses are more precise than fixed-call-graph analyses, but the improved precision typically occurs at the expense of decreased performance. A detailed discussion of these and other important dimensions of precision in the analysis of object-oriented languages appears in [58].

We conclude this section with a brief discussion of the notion of *precise analysis*. Historically, program flow analyses described the propagation of values from one program point to another along possible execution paths. In that context, dataflow analysis was

[1]Fixed-call-graph analyses typically employ Class Hierarchy Analysis (CHA) [18], or Rapid Type Analysis (RTA) [7] for call graph construction. CHA is an inexpensive analysis that determines the possible targets of a virtual call by examining the class hierarchy of the program. RTA is another inexpensive and widely used analysis. It performs a reachability computation on the call graph generated by CHA; by keeping track of the classes that have been instantiated, RTA computes a more precise call graph than CHA.

described as *precise up to symbolic execution* if the values are propagated assuming all possible branches are executable [8]. The notion of precise backwards dataflow analysis is similar. Precise forward or backward interprocedural analysis generalize the idea of up to symbolic execution to paths that include procedure calls. Clearly, the notion of precise flow-insensitive analysis can be derived from the above definition by ignoring the sequentiality of statements on a program path.

Useful static analyses such as points-to analysis, that are precise in the traditional sense are undecidable or NP-hard, even in the flow-insensitive case [41, 40, 51, 35]. Since the goal of our work is the development of analyses that run in practical time and achieve acceptable precision in practice, this thesis studies safe approximate analyses that are *relatively* precise. Such relatively precise analyses incorporate several dimensions of precision such as field sensitivity and context sensitivity, and due to these dimensions of precision are more precise than most existing practical analyses for object-oriented languages. In the same time, they remain efficient and practical.

2.2 Annotated Inclusion Constraints

This section defines the general structure of the framework for annotated inclusion constraints. The annotation language and the operations on the annotations are outlined in Section 2.2.1. Sections 2.2.2 and 2.2.3 describe the constraint language and the constraint representation. Section 2.2.4 describes the constraint resolution procedure. Section 2.2.5 generalizes the framework to handle multiple sets of annotations, and Section 2.2.6 discusses the framework implementation.

2.2.1 Annotation Language and Annotation Operations

The set of annotations is a finite set $\mathcal{S}$. One element of the set is designated as the empty annotation and is denoted by ϵ. The following operations defined on elements of $\mathcal{S}$ are necessary for the integration of the annotations in the constraint solution procedure.

$$\begin{aligned} \mathit{match}: &\quad \mathcal{S} \times \mathcal{S} \rightarrow \text{boolean} \\ \mathit{concat}: &\quad \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S} \\ \mathit{transpose}: &\quad \mathcal{S} \rightarrow \mathcal{S} \end{aligned}$$

Operation *concat* requires the binary predicate *match* to hold. The role of these operations in constraint resolution will be explained shortly. The set $\mathcal{S}$, and the operations acting on the elements of $\mathcal{S}$ can be instantiated to model a specific precision dimension. For example, if the annotations are used to model field sensitivity, $\mathcal{S}$ includes the set of field identifiers. An inclusion relation annotated with $f \in \mathcal{S}$ represents flow of values through field f of a given object.

2.2.2 Constraint Language

Flow analyses can be modeled using annotated inclusion constraints of the form $L \subseteq_a R$, where L and R correspond to some program elements and $a \in \mathcal{S}$ is chosen from the given set of annotations described in Section 2.2.1. We use $L \subseteq R$ to denote constraints labeled with the empty annotation. L and R are set expressions, defined by the following grammar:

$$L, R \rightarrow v \mid c(v_1, \ldots, v_n) \mid \mathit{proj}(c, i, v) \mid 0 \mid 1$$

Here v and v_i are set variables, $c(\ldots)$ are constructed terms and $\mathit{proj}(\ldots)$ are projection terms. Each *constructed term* is built from an n-ary constructor c. A constructor is either *covariant* or *contravariant* in each of its arguments; the role of this variance in constraint resolution will be explained shortly. Constructed terms may appear on both sides of inclusion relations. 0 and 1 represent the empty set and the universal set; they are treated as nullary constructors. *Projections* of the form $\mathit{proj}(c, i, v)$ are terms used to select the i-th argument of any constructed term $c(.., v_i, ..)$. Projection terms may appear only on the right-hand side of an inclusion. Intuitively, set variables, constructed terms and projection terms represent some program elements and the inclusion relation represents flow of values between these elements.

$$c(v_1, ..., v_n) \subseteq_a c(v'_1, ..., v'_n) \Rightarrow$$

$$\begin{cases} v_i \subseteq_a v'_i & \text{if } c \text{ is covariant in } i \text{ for } i = 1 \dots n \\ v'_i \subseteq_{transpose(a)} v_i & \text{if } c \text{ is contravariant in } i \text{ for } i = 1 \dots n \end{cases}$$

$$c(v_1, ..., v_n) \subseteq_a proj(c, i, v) \Rightarrow$$

$$\begin{cases} v_i \subseteq_a v & \text{if } c \text{ is covariant in } i \\ v \subseteq_{transpose(a)} v_i & \text{if } c \text{ is contravariant in } i \end{cases}$$

Figure 2.1: Resolution rules for non-atomic constraints.

2.2.3 Annotated Constraint Graphs

Systems of constraints can be represented as directed multi-graphs. Constraint $L \subseteq_a R$ is represented by an edge from the node for L to the node for R; the edge is labeled with the annotation a. There could be multiple edges between the same pair of nodes, each with a different annotation.

The nodes in the graph can be classified as variables, sources, and sinks. *Sources* are constructed terms that occur on the left-hand side of inclusions. *Sinks* are constructed terms or projections that occur on the right-hand side of inclusions. The graph only contains edges that represent *atomic constraints* of the following forms: *Source* $\subseteq_a$ *Var*, *Var* $\subseteq_a$ *Var*, or *Var* $\subseteq_a$ *Sink*. If the constraint system contains a non-atomic constraint, the resolution rules from Figure 2.1 are used to generate new atomic constraints, as described in Section 2.2.4.

We use annotated constraint graphs based on the *inductive form* representation [4]. Inductive form is an efficient sparse representation that does not explicitly represent the transitive closure of the constraint graph. The graphs are represented with adjacency lists $pred(n)$ and $succ(n)$ stored at each node n. Edge (n_1, n_2, a), where a is an annotation, is represented either as a predecessor edge by having $\langle n_1, a \rangle \in pred(n_2)$, or as a successor edge by having $\langle n_2, a \rangle \in succ(n_1)$, but not both. *Source* $\subseteq_a$ *Var* is always a predecessor edge and *Var* $\subseteq_a$ *Sink* is always a successor edge. *Var* $\subseteq_a$ *Var* is either a predecessor or a successor edge, based on a fixed total order τ : *Vars* $\rightarrow \mathcal{N}$. Edge (v_1, v_2, a) is a predecessor edge if and only if $\tau(v_1) < \tau(v_2)$.

2.2.4 Solving Systems of Annotated Inclusion Constraints

Every system of annotated inclusion constraints can be represented by an annotated constraint graph in inductive form. The system is solved by computing the closure of the graph under the following transitive closure rule:

$$\left.\begin{array}{l} \langle L, a\rangle \in \mathit{pred}(v) \\ \langle R, b\rangle \in \mathit{succ}(v) \\ \mathit{match}(a, b) \end{array}\right\} \Rightarrow L \subseteq_{\mathit{concat}(a,b)} R \qquad (\textsc{Trans})$$

The closure rule can be applied locally, by examining $\mathit{pred}(v)$ and $\mathit{succ}(v)$. The new transitive constraint is created *only if* the annotations of the two existing constraints "match"—that is, only if $\mathit{match}(a, b)$ holds, where *match* is the problem specific binary predicate on the set of annotations as described in Section 2.2.1. Intuitively, the annotations define a coloring of the edges in the graph and the Trans rule uses the coloring to filter out infeasible flow of values in the constraint system. The annotation of the new constraint is produced by applying the concatenation operation *concat* on annotations a and b.

If the new constraint generated by the Trans rule is atomic, a new edge is added to the graph. Otherwise, the resolution rules from Figure 2.1 are used to transform the constraint into several atomic constraints and their corresponding edges are added to the graph. For covariant arguments, the direction of flow is preserved and the annotation is also preserved. For contravariant arguments the direction of flow is reversed and a new annotation is produced by applying the *transpose* operation.

The closure of a constraint graph under the Trans rule is the *solved inductive form* of the corresponding constraint system. The least solution of the system is not explicit in the solved inductive form [4], but is easy to compute by examining all predecessors of each variable. For constraint graphs without annotations, the least solution $LS(v)$ for a variable v is

$$LS(v) = \{c(\ldots) \mid c(\ldots) \in \mathit{pred}(v)\} \cup \bigcup_{u \in \mathit{pred}(v)} LS(u)$$

In this case, $LS(v)$ is computed by transitive acyclic traversal of all predecessor edges; the least solution includes all constructed terms reachable backwards from v on paths that consist of predecessor edges [21]. For an annotated constraint graph, the traversal is done similarly, but the annotations are used as in rule TRANS:

$$LS(v) = \{\langle c(\ldots), a\rangle \mid \langle c(\ldots), a\rangle \in \mathit{pred}(v)\} \cup$$

$$\{\langle c(\ldots), \mathit{concat}(y, x)\rangle \mid \langle u, x\rangle \in \mathit{pred}(v) \wedge \langle c(\ldots), y\rangle \in LS(u) \wedge \ \mathit{match}(x, y)\}$$

First, constructed terms that are direct predecessors of v are included in the least solution with the appropriate annotation: $\langle c(\ldots), a\rangle$. The solution also includes tuples $\langle c(\ldots), \mathit{concat}(y, x)\rangle$, where $\langle c(\ldots), y\rangle$ appears in the least solution of a predecessor of v, u, and annotation y "matches" with the annotation x on the predecessor edge from u to v. In general, the least solution depends on the variable order function τ. If the concatenation operation is *associative*, the solution does not depend on τ.

2.2.5 Tuples of Annotations

The mechanism presented in this chapter allows the modeling of a specific dimension of precision through the instantiation of the annotation set $\mathcal{S}$ and the operations on the elements of $\mathcal{S}$. In order to combine multiple dimensions of precision, we consider tuples of constraint annotations $\langle a_1, a_2, \ldots, a_n\rangle$, where each element a_i belongs to a set of annotations $\mathcal{S}_i$ that represents some precision dimension. There are operations match_i, concat_i, and $\mathit{transpose}_i$ defined for each $\mathcal{S}_i$. The operations on tuples, denoted by *Match*, *Concat*, and *Transpose* become component-wise applications of the operations defined on individual sets. *Match* is defined as follows:

$$\mathit{Match}(\langle a_1, a_2, \ldots, a_n\rangle, \langle b_1, b_2, \ldots, b_n\rangle) =$$

$$\mathit{match}_1(a_1, b_1) \ \wedge \ \mathit{match}_2(a_2, b_2) \ \wedge \ \ldots \mathit{match}_n(a_n, b_n)$$

where match_i denotes the match operation for set $\mathcal{S}_i$. Clearly, two tuples of annotations "match" if and only if for every i, a_i and b_i "match"; thus, adding additional dimensions of precision further restricts flow in the system. The concatenation and the transposition operations are generalized in a similar fashion:

$$\mathit{Concat}(\langle a_1, a_2, \ldots, a_n\rangle, \langle b_1, b_2, \ldots, b_n\rangle) =$$

$$\langle \mathit{concat}_1(a_1, b_1), \mathit{concat}_2(a_2, b_2), \ldots \mathit{concat}_n(a_n, b_n)\rangle$$

and

$$Transpose(\langle a_1, a_2, \ldots, a_n\rangle, \langle b_1, b_2, \ldots, b_n\rangle) =$$
$$\langle transpose_1(a_1, b_1), transpose_2(a_2, b_2), \ldots transpose_n(a_n, b_n)\rangle$$

Tuples of annotations are integrated in the constraint resolution by modifying the rules in Figure 2.1 to take *Transpose* instead of the transpose operation specific to a single dimension of precision. Similarly, closure rule TRANS and the rule for the computation of the least solution $LS(v)$ use *Match* and *Concat*.

We conclude this section with a brief discussion about the worst-case complexity of the annotated constraint system. The worst-case complexity of a non-annotated constraint system is $O(n^3)$, where n is the number of nodes in the constraint graph. Typically, this cost is dominated by the cost of performing TRANS. [2] For every node Y, there are at most $O(n^2)$ pairs of constraints $X \subseteq Y \subseteq Z$ (clearly, there are at most $O(n)$ nodes on the left of Y, and at most $O(n)$ nodes on the right). For every node, the analysis does at most $O(n^2)$ work for inferring transitive constraints; thus the complexity of the resolution is $O(n^3)$.

We can generalize the derivation of the worst-case bound in order to account for the tuples of annotations. The number of edges between two variables is bounded by $|\mathcal{S}_1|.|\mathcal{S}_2|\ldots|\mathcal{S}_n|$, where $|\mathcal{S}_i|$ denotes the size of annotation set $\mathcal{S}_i$. Therefore, the worst-case bound on the cost of the resolution is $O(n^3 \prod_{i=1}^{n} |\mathcal{S}_i|^2.(O(match_i)+O(concat_i)))$, where $O(match_i)$ and $O(concat_i)$ denote the complexities of $match_i$ and $concat_i$. Clearly, if the costs of each $match_i$ and $concat_i$ are constant, the worst-case bound becomes $O(n^3 \prod_{i=1}^{n} |\mathcal{S}_i|^2)$. For individual analyses this bound can be usually improved by exploiting properties of the analyses and determining a better bound on the number of edges between two variables. For example, for the analysis in Chapter 3, and some of the analyses in Chapter 4, the costs of *match* and *concat* are constant; in addition, it can be shown that the number of edges between two variables is bounded by a constant. In this case, the worst-case bound on the resolution of the annotated constraint graph remains

[2] In most realistic cases, the number of arguments in a constructor is bounded by a constant; thus, the work for applying the resolution rules in Figure 2.1 is bounded by $O(n^2)$. If each constructor can have at most n arguments, the cost of applying the resolution rules becomes at most $O(n^3)$.

$O(n^3)$.

2.2.6 General Implementation

The constraint-based analysis uses BANE (Berkeley Analyses Engine) [3]. BANE is a toolkit for constructing constraint-based program analyses. The public distribution of BANE (`bane.cs.berkeley.edu`) contains a constraint-solving engine for non-annotated constraints that employs inductive form, and techniques for efficient constraint resolution. We modified the constraint engine to represent and solve systems of annotated constraints. The modified constraint engine is parameterized by (possibly multiple) sets of annotations with appropriate operations.

To the best of our knowledge, this is the first work that attempts to define a general framework for modeling precision dimensions using inclusion constraints. This is accomplished by defining a general annotation language, and operations on the annotations that can be instantiated. This model unifies various relatively precise analyses and allows easy formulation and implementation of new more precise analyses from relatively imprecise ones; it facilitates the reuse of efficient techniques for constraint resolution.

2.3 Discussion

This section discusses the traditional formulation of flow analyses in terms of data-flow frameworks. It relates at an intuitive level problems formulated in terms of data-flow frameworks to problems formulated in terms of set inclusion constraints.

Traditionally flow analyses are expressed in terms of data-flow frameworks of the form $D = \langle G, L, F, M \rangle$ [45]. In this model, G is the control-flow graph of the program, L is the lattice of flow elements, $F = \{f : L \rightarrow L\}$ is the class of transfer functions, and $M : E \rightarrow F$ is a mapping from the edges of G to the elements of F. This model is very general and encompasses data-flow analysis problems of widely varying complexity. Due to the generality of the model, no efficient uniform implementation technique can be developed; thus, data-flow analyses are typically implemented individually.

Set inclusion constraint frameworks present a powerful formalism for expressing certain classes of data-flow analysis problems; they present a powerful machinery for implementing data-flow analyses of these classes. However, it is an interesting and still open question what are the properties of the dataflow problem (properties of the control-flow graph G, the lattice L, and the transfer functions F) that will allow the problem to be formulated in terms of set constraints. The answer to this question will provide valuable insight into what classes of data-flow problems may possibly be implemented efficiently using annotated or non-annotated set inclusion constraint frameworks.

Chapter 3

Field-sensitive Points-to Analysis for Java

Performance improvement through the use of compiler technology is important for making Java a viable choice for production-strength software. In addition, the development of large Java software systems requires strong support from software engineering tools for program understanding, maintenance, and testing. Both optimizing compilers and software engineering tools employ various static analyses to determine properties of run-time program behavior. One fundamental static analysis is *points-to analysis*. For Java, points-to analysis determines the set of objects that a given reference variable or reference object field may refer to. By computing such *points-to sets* for variables and fields, the analysis constructs an abstraction of the run-time memory states of the analyzed program. This abstraction is typically represented by one or more *points-to graphs*. (An example of a points-to graph is shown in Figure 3.1, which is discussed later.)

Points-to analysis enables a variety of other analyses—for example, *side-effect analysis*, which determines the memory locations that may be modified by the execution of a statement, and *def-use analysis*, which identifies pairs of statements that set the value of a memory location and subsequently use that value. Such analyses are needed by compilers to perform well-known optimizations such as code motion and partial redundancy elimination. These analyses are also important in the context of software engineering tools: for example, def-use analysis is needed for program slicing and data-flow-based testing. Points-to analysis is a crucial prerequisite for employing these analyses and optimizations.

In addition to enabling other analyses, points-to analysis can be used directly in optimizing static Java compilers to perform a variety of popular optimizations such as virtual call resolution, removal of unnecessary synchronization, and stack-based object allocation. Typically, each of these optimizations is based on a *specialized* analysis designed for the purpose of this specific optimization. Thus, compilers that employ multiple optimizations need to implement many different analyses. In contrast, using a single *general-purpose* points-to analysis can enable several different optimizations. Furthermore, the cost of the analysis can be amortized across many client optimizations, and the development effort to implement the optimizations can be significantly reduced.

Because of the many applications of points-to analysis, it is important to investigate approaches for precise and efficient computation of points-to information. In this chapter we formulate and evaluate a points-to analysis for Java which is based on Andersen's points-to analysis for C [6].

Andersen's analysis for C is a flow- and context-insensitive analysis with cubic worst-case complexity. Previous work has shown that techniques based on non-annotated inclusion constraints allow efficient implementations of this analysis [21, 65]. We use this analysis for C, which is *field-insensitive*, flow- and context-insensitive, and for Java can contain substantial imprecision, to formulate a relatively precise analysis for Java in the context of our framework. We use two sets of constraint annotations to model two dimensions of precision. *Field annotations* model field sensitivity, and allow separate tracking of the flow of values through the different fields of an object. *Method annotations* are used to model precisely and efficiently the semantics of virtual calls, by representing the relationships between a virtual call, its receiver objects, and its target methods.

One disadvantage of Andersen's analysis is the implicit assumption that all code in the program is executable. Java programs contain large portions of unused library code; including such dead code can have negative effects on analysis cost and precision. In our analysis, like in other analyses [7, 69, 33], we keep track of all methods potentially reachable from the entry points of the program, and we only analyze such reachable methods.

```
class Y {..}
class X {
  Y f;
  void set (Y r)
      { this.f = r; }
  static void main() {
   s1: X p = new X();
   s2: Y q = new Y();
      p.set(q);
  }
}
```

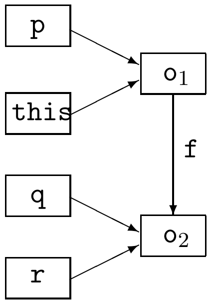


Figure 3.1: Sample program and its points-to graph.

3.1 Semantics of Points-to Analysis for Java

In this section we define the semantics of our points-to analysis for Java; Section 3.3 describes the implementation of the analysis with annotated inclusion constraints. The analysis is defined in terms of three sets. Set R contains all reference variables in the analyzed program (including static variables). Set O contains names for all objects created at object allocation sites; for each allocation site s_i, we use a separate object name $o_i \in O$. Set F contains all instance fields in program classes. Analysis semantics is expressed as manipulations of *points-to graphs* containing two kinds of edges. Edge $(r, o_i) \in R \times O$ shows that reference variable r points to object o_i. Edge $(\langle o_i, f\rangle, o_j) \in (O \times F) \times O$ shows that field f of object o_i points to object o_j. A sample program and its points-to graph are shown in Figure 3.1.

To simplify the presentation, we only discuss the kinds of statements listed below; our actual implementation (described in Section 3.3) handles the entire language. [1]

- Direct assignment: `l = r`
- Instance field write: `l.f = r`
- Instance field read: `l = r.f`
- Object creation: `l = new C`
- Virtual invocation: l = r_0.m(r_1,...,r_k)

[1] We assume that native methods, reflection and dynamic loading are resolved manually.

$$f(G, s_i\colon\ l = \textit{new}\ C) = G \cup \{(l, o_i)\}$$

$$f(G, l = r) = G \cup \{(l, o_i) \mid o_i \in Pt(G, r)\}$$

$$\begin{aligned} &f(G, l.f = r) = \\ &\quad G \cup \{(\langle o_i, f\rangle, o_j) \mid o_i \in Pt(G, l)\ \wedge\ o_j \in Pt(G, r)\} \end{aligned}$$

$$\begin{aligned} &f(G, l = r.f) = \\ &\quad G \cup \{(l, o_i) \mid o_j \in Pt(G, r)\ \wedge\ o_i \in Pt(G, \langle o_j, f\rangle)\} \end{aligned}$$

$$\begin{aligned} &f(G, l = r_0.m(r_1, \ldots, r_n)) = \\ &\quad G \cup \{\textit{resolve}(G, m, o_i, r_1, \ldots, r_n, l) \mid o_i \in Pt(G, r_0)\} \end{aligned}$$

$$\begin{aligned} &\textit{resolve}(G, m, o_i, r_1, \ldots, r_n, l) = \\ &\quad \texttt{let}\ \ m_j(p_0, p_1, \ldots, p_n, \textit{ret}_j) = \textit{dispatch}(o_i, m)\ \ \texttt{in} \\ &\qquad \{(p_0, o_i)\} \cup f(G, p_1 = r_1) \cup \ldots \cup f(G, l = \textit{ret}_j) \end{aligned}$$

Figure 3.2: Points-to effects of program statements.

At a virtual call, name `m` uniquely identifies a method in the program. This method is the *compile-time* target of the call, and is determined based on the declared type of $\mathtt{r}_0$ [28, Section 15.11.3]. At run time, the invoked method is determined by examining the class of the receiver object and all of its superclasses, and finding the first method that matches the signature and the return type of `m` [28, Section 15.11.4].

Analysis semantics is defined in terms of rules for adding new edges to points-to graphs. Each rule represents the semantics of a program statement. Figure 3.2 shows the rules as functions of the form $f : \textit{PtGraph} \times \textit{Stmt} \rightarrow \textit{PtGraph}$. The points-to set (i.e., the set of all successors) of x in graph G is denoted by $Pt(G, x)$. The solution computed by the analysis is a points-to graph that is the closure of the empty graph under the edge-addition rules.

For most statements, the effects on the points-to graph are straightforward; for example, statement $l = r$ creates new points-to edges from l to all objects pointed to by r. For virtual call sites, resolution is performed for every receiver object pointed to by r_0. Function *dispatch* uses the class of the object and the compile-time target of the call to determine the actual method m_j invoked at run time. Variables $p_0, \ldots, p_n$ are the formal parameters of the method; variable p_0 corresponds to the implicit parameter `this`. Variable ret_j contains the return value of m_j.

3.2 Applications of Points-to Analysis for Java

Using points-to analysis in optimizing compilers and software engineering tools has several advantages. A single points-to analysis can enable a wide variety of client applications. The cost of the analysis can be amortized across many clients. Once implemented, the analysis can be reused by various clients at no additional development cost; such reusability is an important practical advantage. In this section we briefly discuss several specific applications of points-to analysis for Java. In our experiments, we have evaluated the impact of our analysis on some of these applications; the results from these experiments are presented in Section 3.4.

3.2.1 Call Graph Construction and Virtual Call Resolution

The points-to solution can be used to determine the target methods of a virtual call by examining the classes of all possible receiver objects. The set of target methods is needed to construct the *call graph* for the analyzed program; this graph is a prerequisite for all interprocedural analyses and optimizations used in Java compilers and tools. If the call has only one target method, it can be *devirtualized* by replacing the virtual call with a direct call; this optimization eliminates the run-time overhead of virtual dispatch. In addition, virtual call resolution allows subsequent inlining of the target method, potentially enabling additional optimizations within the caller.

3.2.2 Synchronization Removal

Synchronization in Java allows safe access to shared objects in multi-threaded programs. Each object has an associated lock which is used to ensure mutual exclusion. Synchronization operations on locks can have considerable run-time overhead; this overhead occurs even in single-threaded programs, because the standard Java libraries are written in thread-safe manner.

Static analysis can be used to detect properties that allow the removal of unnecessary synchronization. For example, no synchronization is necessary for an object that cannot "escape" its creating thread and therefore cannot be accessed by any other thread (i.e., a *thread-local* object).[2] Some escape analyses [14, 11, 12, 73] have been used to identify thread-local objects and to remove the synchronization constructs associated with such objects.

Points-to analysis can be used as an alternative to escape analysis in detecting thread-local objects. Consider an object o_i and suppose that in the points-to graph computed by the analysis, o_i is not reachable from (i) static (i.e., global) reference variables, or (ii) objects of classes that implement interface `java.lang.Runnable`[3]. It can be proven that in this case o_i is not accessible outside the thread that created it. We can identify such thread-local objects by performing a reachability computation on the points-to graph; this approach is similar to the multithreaded object analysis proposed by Aldrich et al. [5].

3.2.3 Stack Allocation

In some cases, an object can be allocated on a method's stack frame rather than on the heap. This transformation reduces garbage collection overhead and enables additional optimizations such as object reduction [27, 24]. Similarly to synchronization removal, static analysis can be used to detect properties that allow stack-based allocation. For example, stack allocation is possible for an object that may never "escape" the lifetime of its creating method and therefore can only be accessed during that lifetime (i.e., a *method-local* object). Some analyses [14, 11, 73, 24] can detect method-local objects; clearly, such objects can be allocated on the stack frames of their creating methods.

Points-to analysis can be used as an alternative to escape analysis in identifying method-local objects. Suppose that object o_i has been classified as thread-local according to the points-to solution (i.e., o_i is not reachable from static variables or from objects

[2]If synchronization operations are removed for objects receiving `wait`, `notify`, or `notifyAll` messages, the modified program may throw `IllegalMonitorStateException`. This problem can be avoided by maintaining the information needed by the notification methods without performing actual synchronization [57].

[3]The `run` methods of such objects are the starting points of new threads.

implementing `Runnable`). Also, suppose that in the computed points-to graph, o_i is not reachable from the formal parameters or the return variable of the method that created o_i. In this case, it can be proven that o_i is method-local; we can identify such method-local objects by traversing the points-to graph.

3.3 Points-to Analysis for Java Using Annotated Constraints

In this section we show how to implement the points-to analysis from Section 3.1 using annotated inclusion constraints. Recall that the analysis is defined in terms of the set R of all reference variables and the set O of names for all objects created at object allocation sites. Every element of $R \cup O$ is essentially an abstract memory location representing a set of run-time memory locations.

To implement our analysis with annotated inclusion constraints, we generalize an approach for modeling Andersen's analysis for C with non-annotated constraints [21, 65]. For each abstract location x, a set variable v_x represents the set of abstract locations pointed to by x. The representation of each location is through a ternary constructor *ref* which is used to build constructed terms of the form $ref(x, v_x, \overline{v_x})$. The last two arguments are the same variable, but with different variance—the overline notation is used to denote a contravariant argument. Intuitively, the second argument is used to read the values of locations pointed to by x, while the last argument is used to update the values of locations pointed to by x. Given a reference variable $r \in R$ and an object variable $o \in O$, constraint

$$ref(o, v_o, \overline{v_o}) \subseteq v_r$$

shows that r points to o.

3.3.1 Modeling Field Sensitivity

Field Annotations

We use *field annotations* to model the flow of values through fields of objects. The set of field annotations is the set of unique identifiers for all instance fields defined in program classes plus the special empty annotations; thus $\mathcal{S} = F \cup \{\epsilon\}$. For any two object variables o_1 and o_2, constraint

$$ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq_f v_{o_1}$$

shows that field f in object o_1 points to object o_2.

The operations on the annotations are defined as follows. The match predicate is shown below: [4]

$$match_f(f_1, f_2) = \begin{cases} \text{true} & \text{if } f_1 \text{ or } f_2 \text{ is the empty annotation } \epsilon \\ \text{true} & \text{if } f_1 = f_2 \\ \text{false} & \text{otherwise} \end{cases}$$

The concatenation operation which requires match to hold and produces the annotation of the new constraint is shown below:

$$concat_f(f_1, f_2) = \begin{cases} f_1 & \text{if } f_2 = \epsilon \\ f_2 & \text{if } f_1 = \epsilon \\ \epsilon & \text{otherwise} \end{cases}$$

Intuitively, a field annotation is propagated until it is matched with another instance of itself (i.e., the two constraints represent accesses through the same field), after which the two instances cancel out. Operation transpose is defined as follows: $transpose_f(a) = a$; thus it preserves the annotation.

Constraints for Assignment Statements

For every program statement, our analysis generates annotated inclusion constraints representing the semantics of the statement. Figure 3.3 shows the constraints generated for assignment statements. The first two generation rules are straightforward. The rule for $l.f = r$ uses the first constraint to access the points-to set of l, and the second constraint to update the values of field f in all objects pointed to by l. Fresh variable u is a new variable introduced in the constraint system; we will refer to such variable as projection variable. Similarly, the last rule uses two constraints to read the values of field f in all objects pointed to by r.

---

[4] The subscript f in $match_f$ is used to denote the match operation for field annotations. The subscript is used in similar fashion with the other operations.

$$\langle l = \mathit{new}\ o_i \rangle \Rightarrow \{\mathit{ref}(o_i, v_{o_i}, \overline{v_{o_i}}) \subseteq v_l\}$$

$$\langle l = r \rangle \Rightarrow \{v_r \subseteq v_l\}$$

$$\langle l.f = r \rangle \Rightarrow \{v_l \subseteq \mathit{proj}(\mathit{ref}, 3, u),\ v_r \subseteq_f u\},\ u \text{ fresh}$$

$$\langle l = r.f \rangle \Rightarrow \{v_r \subseteq \mathit{proj}(\mathit{ref}, 2, u),\ u \subseteq_f v_l\},\ u \text{ fresh}$$

Figure 3.3: Constraints for assignment statements.

```
s1: p = new X();
s2: q = new Y();
    p.f = q;
    r = p.f;
```

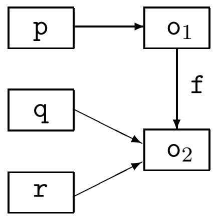


Figure 3.4: Accessing object fields.

Example

Consider the statements in Figure 3.4 and their corresponding points-to graph. After processing the statements, our analysis creates the following constraints:

$$\begin{array}{ll} \mathit{ref}(o_1, v_{o_1}, \overline{v_{o_1}}) \subseteq v_p & \mathit{ref}(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_q \\ v_p \subseteq \mathit{proj}(\mathit{ref}, 3, u) & v_q \subseteq_f u \\ v_p \subseteq \mathit{proj}(\mathit{ref}, 2, w) & w \subseteq_f v_r \end{array}$$

where u and w are fresh variables. For the purpose of this example we assume that the variable order τ (defined in Section 2.2.3 in Chapter 2) is $\tau(v_p) < \tau(v_q) < \tau(v_r) < \tau(v_{o_1}) < \tau(v_{o_2}) < \tau(u) < \tau(w)$. [5] Consider the indirect write in `p.f = q`. Since we have

$$\mathit{ref}(o_1, v_{o_1}, \overline{v_{o_1}}) \subseteq v_p \subseteq \mathit{proj}(\mathit{ref}, 3, u)$$

we can use the TRANS rule with the appropriate instantiations of *match* and *concat* for field annotations, as defined above. Subsequently we can use the resolution rules from

[5] In non-annotated constraint systems, the role of the variable ordering is to reduce the amount of redundant edge additions in the constraint graph. In the presence of annotations, a variable ordering may need to be imposed in order to insure analysis correctness. Section 3.3.3 discusses the variable ordering restrictions needed to preserve the correctness of the field-sensitive analysis.

Figure 2.1 in Chapter 2 to generate a new constraint $u \subseteq v_{o_1}$. Thus,

$$v_q \subseteq_f u \subseteq v_{o_1}$$

and using rule TRANS we generate $v_q \subseteq_f v_{o_1}$. Intuitively, this new constraint shows that some of the values of field f in object o_1 come from variable q. Now we have

$$\mathit{ref}(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_q \subseteq_f v_{o_1}$$

Since both constraint edges are predecessor edges, we cannot apply rule TRANS. Still, in the least solution of the constraint system (as defined in Section 2.2.4), we have the constraint $\mathit{ref}(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq_f v_{o_1}$, which shows that field f of o_1 points to o_2.

To model indirect reads, we use the second argument of the *ref* constructor. For example, for the constraints above we have

$$\mathit{ref}(o_1, v_{o_1}, \overline{v_{o_1}}) \subseteq v_p \subseteq \mathit{proj}(\mathit{ref}, 2, w)$$

and therefore $v_{o_1} \subseteq w \subseteq_f v_r$, which through TRANS generates $v_{o_1} \subseteq_f v_r$. This new constraint shows that the value of r comes from field f of object o_1. Now we have

$$v_q \subseteq_f v_{o_1} \subseteq_f v_r$$

Since the annotations of the two constraints match—that is, they represent accesses to the same field—we generate $v_q \subseteq v_r$ to represent the flow of values from q to r. Thus, in the least solution of the system we have

$$\mathit{ref}(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_r$$

which shows that reference variable r points to o_2. This example illustrates how field annotations allow us to model the flow of values through object fields.

3.3.2 Modeling Virtual Dispatch

Method Annotations

We use *method annotations* to model precisely the semantics of virtual dispatch. The set of all annotations $\mathcal{S}$ equals to $T \cup M \cup \{\epsilon\}$, where T is the set of all classes (types) in the program, and M is the set of all unique identifiers of *compile-time* targets at method calls.

The operations on the annotations essentially represent the *dispatch* function from Section 3.1. They are based on a precomputed *lookup* table. For a given class $c \in T$, and a given *compile-time* target method $m \in M$, the lookup table is used to determine the corresponding *run-time* target method. Such a table is straightforward to precompute by analyzing the class hierarchy; the table is essentially a representation of the *dispatch* function from Section 3.1.

The match predicate is shown below: [6]

$$match_m(c,n) = \begin{cases} \text{true} & \text{if } c \text{ or } n \text{ is the empty annotation } \epsilon \\ \text{true} & \text{if } c \in T,\ n \in M, \text{ and } lookup(c,n) \text{ exists} \\ \text{false} & \text{otherwise} \end{cases}$$

The concatenation operation returns the identifier of the run-time target of the call:

$$concat_m(c,n) = \begin{cases} c & \text{if } n = \epsilon \\ n & \text{if } c = \epsilon \\ lookup(c,n) & \text{otherwise} \end{cases}$$

Constraints for Virtual Calls

For every virtual call in the program, our analysis generates a constraint according to the following rule:

$$\langle l = r_0.m(r_1,\ldots,r_k)\rangle \Rightarrow$$
$$\{v_{r_0} \subseteq_m lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)\}$$

The rule is based on a *lam* (lambda) constructor. The constructor is used to build a term that encapsulates the actual arguments and the left-hand-side variable of the call. The annotation on the constraint is a unique identifier of the *compile-time* target method of the call. This annotation is used during the analysis to find all appropriate *run-time* target methods.

[6]We use the subscript m to denote operations on method annotations.

To model the semantics of virtual calls as defined in Section 3.1, we separately perform virtual dispatch for every receiver object pointed to by r_0. Given the class of the receiver object and the unique identifier for the compile-time target of the virtual call, the lookup table returns the run-time identifier which corresponds to a unique lambda term of the form

$$lam(\overline{v_{p_0}}, \overline{v_{p_1}}, \ldots, \overline{v_{p_k}}, v_{ret})$$

Here p_i are the formal parameters of the run-time target method; p_0 corresponds to the implicit parameter `this`. We assume that each method has a unique variable *ret* that is assigned the value returned by the method (this can be achieved by inserting auxiliary assignments in the program representation). At the beginning of the analysis, lambda terms of the above form are created for all non-abstract methods in the program and are stored in the lookup table.

To model the effects of virtual calls, we define an additional closure rule VIRTUAL. This rule encodes the semantics of virtual calls described in Section 3.1 and makes use of the operations on the annotations defined above. It is used together with the TRANS rule to obtain the solved form of the constraint system. VIRTUAL is applied whenever we have two constraints of the form [7]

$$ref(o, v_o, \overline{v_o}) \subseteq_c v \qquad v \subseteq_m lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)$$

As described in Section 2.2.3, the edge from the *ref* term is a predecessor edge, and the edge to the *lam* term is a successor edge. Thus, the VIRTUAL closure rule can be applied locally, by examining sets $pred(v)$ and $succ(v)$. Whenever two such constraints are detected, the analysis performs *match* to determine if there exists a run-time target method that corresponds to class c and compile-time target m. If c and m "match", operation *concat* returns the run-time identifier; this identifier is used to find the corresponding lambda term for the run-time method. The result of applying VIRTUAL are two new constraints:

$$ref(o, v_o, \overline{v_o}) \quad \subseteq_c \quad v_{p_0}$$

[7]Every constraint of the form $ref(o, v_o, \overline{v_o}) \subseteq_c v$ is annotated with an annotation c which corresponds to the class of object o; this annotation is used when performing lookup.

```
class A {
     X n() {
      ...  return rA; }
      }
class B extends A {
     X n() {
      ...  return rB; }
     }
s1: A a = new A();
s2: B b = new B();
    A c = b;
c1: X x = b.n();
c2: X y = c.n();
    if (...)  a = b;
c3: X z = a.n();
```

$$(1)\quad ref(o_2, v_{o_2}, \overline{v_{o2}}) \subseteq_B v_b \subseteq_{B.n} lam(\overline{0}, v_x) \Rightarrow \{ref(o_2, v_{o_2}, \overline{v_{o2}}) \subseteq_B v_{B.n.this},\ v_{rB} \subseteq v_x\}$$

$$(2)\quad ref(o_2, v_{o_2}, \overline{v_{o2}}) \subseteq_B v_b \subseteq_{A.n} lam(\overline{0}, v_y) \Rightarrow \{ref(o_2, v_{o_2}, \overline{v_{o2}}) \subseteq_B v_{B.n.this},\ v_{rB} \subseteq v_y\}$$

$$(3)\quad ref(o_1, v_{o_1}, \overline{v_{o1}}) \subseteq_A v_a \subseteq_{A.n} lam(\overline{0}, v_z) \Rightarrow \{ref(o_1, v_{o_1}, \overline{v_{o1}}) \subseteq_A v_{A.n.this},\ v_{rA} \subseteq v_z\}$$

$$(4)\quad ref(o_2, v_{o_2}, \overline{v_{o2}}) \subseteq_B v_a \subseteq_{A.n} lam(\overline{0}, v_z) \Rightarrow \{ref(o_2, v_{o_2}, \overline{v_{o2}}) \subseteq_B v_{B.n.this},\ v_{rB} \subseteq v_z\}$$

Figure 3.5: Example of virtual call resolution.

$$lam(\overline{v_{p0}}, \overline{v_{p1}}, \ldots, \overline{v_{p_k}}, v_{ret}) \quad \subseteq \quad lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)$$

The first constraint creates the association between parameter `this` of the invoked method and the receiver object o, which is of class c. The second constraint immediately resolves to $v_{r_i} \subseteq v_{p_i}$ (for $i \geq 1$) and $v_{ret} \subseteq v_l$, plus the trivial constraint $0 \subseteq v_{p0}$. These new atomic constraints model the flow of values from actuals to formals, as well as the flow of return values to the left-hand side variable l used at the call site.

Example

Consider the set of statements in Figure 3.5. For the purpose of this example, assume that $\tau(v_a) < \tau(v_b) < \tau(v_c)$. Since the declared type of `b` is `B`, at call site c_1 the compile-time target method is `B.n`; thus, we have

$$v_b \subseteq_{B.n} lam(\overline{0}, v_x)$$

When rule VIRTUAL is applied as shown in (1), the lookup for class `B` (the class of receiver object o_2) and compile-time target `B.n` produces run-time target `B.n`. The resolution with the *lam* term for `B.n` creates the two new constraints shown in (1).

The declared type of `c` is `A`, and for call site c_2 we have $v_c \subseteq_{A.n} lam(\overline{0}, v_y)$. Thus,

$$ref(o_2, v_{o_2}, \overline{v_{o2}}) \subseteq_B v_b \qquad v_b \subseteq_{A.n} lam(\overline{0}, v_y)$$

where the second constraint is obtained through the TRANS rule.[8] When applying rule VIRTUAL as shown in (2), the lookup for class `B` of receiver object o_2, and compile-time target `A.n` leads to run-time target `B.n`. The two new constraints which result from the resolution are shown in (2). For call site c_3, the receiver object can be either o_1 or o_2. As shown in (3) and (4), separate lookup and resolution is performed for each receiver.

3.3.3 Correctness

Let G be a points-to graph as defined in Section 3.1 and A be an annotated constraint graph in inductive form as described in this section. We will define a *representation relation* α between A and the edges in G. If α holds between A and all edges in G, we will write $\alpha(A, G)$.

Consider a reference variable r and an object variable o such that $e = (r, o)$ is an edge in G. We have $\alpha(A, e)$ if and only if A contains a path

$$\mathit{ref}(o, v_o, \overline{v_o}) \rightarrow v_1 \rightarrow \ldots \rightarrow v_n \rightarrow v_r$$

such that all v_i correspond to reference variables and all edges are predecessor edges with empty annotations.

Similarly, consider two object variables o_i and o_j such that $e = (\langle o_i, f\rangle, o_j)$ is an edge in G. We have $\alpha(A, e)$ if and only if one of the following two conditions is true. The first condition is that A contains a path

$$\mathit{ref}(o_j, v_{o_j}, \overline{v_{o_j}}) \rightarrow v_1 \rightarrow \ldots \rightarrow v_n \xrightarrow{f} v_{o_i}$$

such that all v_i $(1 \leq i \leq n)$ correspond to reference variables, all edges are predecessor edges, and the only non-empty annotation on the path is field annotation f on edge (v_n, v_{o_i}). [9] The second condition is that A contains a path

$$\mathit{ref}(o_j, v_{o_j}, \overline{v_{o_j}}) \rightarrow v_1 \rightarrow \ldots \rightarrow v_n \xrightarrow{f} u \rightarrow v_{o_i}$$

---

[8]Note that if $\tau(v_c) < \tau(v_b)$, instead of propagating the *lam* term to v_b we would propagate the *ref* term to v_c.

[9]To simplify the presentation, whenever one of the annotations is empty, we avoid using the tuple notation. For example, we use f instead of $\langle f, \epsilon\rangle$; similarly we use m instead of $\langle \epsilon, m\rangle$.

such that all v_i $(1 \leq i \leq n)$ correspond to reference variables and u is a "fresh" variable created due to field access (see Figure 3.3). In this path, all edges are predecessor edges, and the only non-empty annotation is f on edge (v_n, u).

Let G^* be the final points-to graph computed by the algorithm in Section 3.1, and A^* be the solved inductive form of the corresponding annotated constraint system. The least solution of the constraint system is obtained through additional traversal of predecessor edges in A^*, as described in Section 2.2.4.

Suppose that $\alpha(A^*, G^*)$. It is easy to show that in this case G^* is contained in the set of points-to pairs extracted from the least solution of the constraint system. For example, for every edge $(\langle o_i, f\rangle, o_j) \in G^*$, the least solution contains $\mathit{ref}(o_j, v_{o_j}, \overline{v_{o_j}}) \subseteq_f v_{o_i}$. To prove that $\alpha(A^*, G^*)$, we use the following lemma:

Lemma 1 *Let $\langle s, G\rangle \Rightarrow G'$ as described in Figure 3.2. If $\alpha(A, G)$, there exists a sequence of applications of closure rules and resolution rules such that A can be transformed into A' for which $\alpha(A', G')$.*

The proof of the this lemma depends on the following restriction on the variable order τ: all variables v_r, where $r \in R$, should have lower order than the rest of the constraint variables. We enforce this restriction as part of building the constraint system.

The proof of the lemma requires case-by-case analysis of all statement kinds. For example, consider the assignment `l = r`. In G', there are new edges of the form (l, o_i), where $(r, o_i) \in G$. In A, we have a path from $\mathit{ref}(o_i, v_{o_i}, \overline{v_{o_i}})$ to v_r containing only predecessor edges with empty annotations. Graph A also contains $v_r \subseteq v_l$, which is represented either as a predecessor edge or as a successor edge (depending on the order of v_r and v_l). If $v_r \in \mathit{pred}(v_l)$, then we have the needed path from $\mathit{ref}(o_i, v_{o_i}, \overline{v_{o_i}})$ to v_l. If $v_l \in \mathit{succ}(v_r)$, then a sequence of applications of rule TRANS creates the needed path.

The rest of the statements are handled in a similar manner. For the cases when we have access to object fields, we have to know that all reference variables have order smaller than the rest of the variables (as described in earlier in this section). This restriction ensures that the appropriate kinds of edges (predecessor or successor) are created when transforming A. Given the above lemma, it is trivial to show that α holds between A^* and G^*.

We conclude this section with a discussion on the complexity of the analysis. It can be shown that method annotations appear if and only if the edge is of one of the following forms: (i) $(\mathit{ref}(o_i, v_{o_i}, \overline{v_{o_i}}), v_r)$, annotated with c, where c denotes the class of o_i, and (ii) $(v_r, \mathit{lam}(\ldots))$, annotated with m, where m denotes the compile time target of the call represented by *lam*. Clearly, the class c of o_i, and the compile-time target m are unique; therefore if there is an edge with method annotation between two terms, then this is the only edge between them.

We now consider field annotations. Let us partition the nodes in the constraint graph in two disjoint sets as follows. The first set, denoted by G_1 contains *ref*, *lam* and *proj* terms, and variables v_r where r is a reference variable. The second set, denoted by G_2 contains projection variables u and variables v_o, where o is an object (recall that v_o represents the points-to sets of the field of object o). We extend the notation for G_1 and G_2 to represent the corresponding graphs (i.e., the edges in G_1 are the edges (n_1, n_2), where $n_1, n_2 \in G_1$, and similarly the edges in G_2 are the edges (m_1, m_2), where $m_1, m_2 \in G_2$). We prove the following lemma:

Lemma 2 *The following invariants hold during resolution: (i) no edge in G_1 or G_2 contains a field annotation, (ii) every edge from a node $n_1 \in G_1$ to a node $m_2 \in G_2$ is a predecessor edge that contains a field annotation, and (iii) every edge from a node $m_1 \in G_2$ to a node $n_2 \in G_1$ is a successor edge with a field annotation.*

The proof of this statement can be carried out by induction on the number of applications of closure rules and resolution rules. By the variable ordering restriction stated earlier in this section, we have $\tau(n \in G_1) < \tau(m \in G_2)$. Therefore, the statement clearly holds for the initial constraint graph (resulting directly after constraint generation and before the application of closure and resolution rules). Suppose that the statement is true after k rule applications. Consider an application of the rule Trans on $X \subseteq Y \subseteq Z$ (recall that closure rule Trans is applied only if (X, Y) is a predecessor edge and (Y, Z) is a successor edge). There are five cases: (i) $X, Y, Z \in G_1$, (ii) $X, Y, Z \in G_2$, (iii) $X \in G_1$, $Y, Z \in G_2$, (iv) $X \in G_2$, $Y, Z \in G_1$, and (v) $X \in G_1$, $Y \in G_2$, and $Z \in G_1$.

Clearly, since the invariant holds before the application of TRANS, in cases (i) and (ii) edges (X, Y) and (Y, Z) do not have field annotation; therefore the resulting edge (X, Z) does not have field annotation either and the invariant is preserved. In case (iii), edge (X, Y) is a predecessor edge with a field annotation, and edge (Y, Z) is a successor edge without a field annotation. The resulting edge (X, Z) is a predecessor edge with a field annotation (recall that we have $\tau(X) < \tau(Z)$). Similarly, in case (iv) edge (Y, Z) is a successor edge with a field annotation which preserves the invariant. In case (v) the edge (X, Y) has a field annotation, and edge (Y, Z) has a field annotation. If the two annotations match, the resulting edge (X, Z) is in G_1 and has empty annotation, thus preserving the invariant.

It remains to be shown that if there is an edge with a field annotation between two variables, this edge is the only edge between them. Consider a predecessor edge $(v_r \in G_1, v_x \in G_2)$ with an annotation f. Clearly, variable r appears at some indirect write of the form `p.f = r`. Without loss of generality we may assume that r does not appear at any indirect write statement that writes a field other than f. [10] Using the lemma stated above, it can be shown that it is impossible to have a predecessor edge with a field annotation other than f from v_r (Clearly, the only way to have such an edge is to combine an edge (v_r, Y) with an edge (Y, Z), where Y and Z are in G_2. Since edge (v_r, Y) cannot have an annotation other than f, and due to the lemma (Y, Z) cannot have a field annotation, this is impossible). Thus, the only annotated outgoing edges from v_r are predecessor edges annotated with f. In a similar manner it can be shown that the only annotated incoming edges into v_q are successor edges with annotation of only one kind. Since we were able to bound by a constant the number of edges between each pair of nodes in the constraint graph, the complexity of the resolution of the annotated constraints is at most $O(n^3)$.

---

[10] This restriction is true for our intermediate representation (`Jimple`). In general it can be easily achieved by introducing a fresh variable at every indirect field write. Each statement of the form `p.f = r` can be broken into two statements: `p.f = r'` and `r' = r`; clearly, this transformation preserves the $O(n)$ bound on the number of variables.

3.3.4 Cycle Elimination and Projection Merging

Cycle elimination [21] and projection merging [65] are techniques that can be used to reduce the cost of Andersen's analysis for C. We have adapted these techniques to allow us to reduce the cost of our points-to analysis for Java.

The idea behind cycle elimination is to detect a set of variables that form a cycle in the constraint graph:

$$v_1 \subseteq v_2 \subseteq \ldots \subseteq v_k \subseteq v_1$$

Clearly, all such variables have equal solutions and can be replaced with a single variable. Whenever a cycle is detected during the resolution process, one variable from the cycle is chosen as a witness variable, and the rest of the variables are redirected to the witness. This transformation has no effect on the computed solution, but can significantly reduce the cost of the analysis.

Cycle detection is performed every time a new edge is added between two variables v_i and v_j. The detection algorithm essentially performs depth-first traversal of the constraint graph and tries to determine whether v_i is reachable from v_j. Cycle detection is partial and does not detect all cycles. Nevertheless, for Andersen's analysis for C this technique has significant impact on the running time of the analysis [21].

Cycle elimination cannot be used directly for the annotated constraint systems presented in this paper. If we performed the standard cycle detection, we would discover cycles in which some edges have field annotations; however, the variables in such cycles do not have the same solution, and cannot be replaced by a single witness variable. To guarantee the correctness of our analysis for Java, we use a restricted form of cycle elimination. The cycle detection algorithm is invoked only when a new edge is added between two reference variables—that is, when the new edge is (v_{r_i}, v_{r_j}), where $r_i, r_j \in R$. It can be proven that in this case, the detected cycle contains only reference variables, and all edges in the cycle have empty annotations. This guarantees that all variables on the cycle have identical points-to sets, and therefore replacing the cycle with a single variable preserves the points-to solution.

Lemma 3 *Let* $(v_{x_1}, v_{x_2}, \ldots, v_{x_n}, v_{x_1})$ *be a cycle detected by the cycle detection algorithm [21]. If* $x_1, x_n \in R$*, the following facts hold true (i)* $x_i \in R$ *for every* $i = 1 \ldots n$*, and (ii)* $(v_{x_i}, v_{x_{i+1}})$ *is an edge with an empty annotation for every* $i = 1 \ldots n\text{-}1$*.*

The proof of the lemma can be carried out by considering two cases. Consider the case when $\tau(v_{x_1}) < \tau(v_{x_n})$. Edge (v_{x_n}, v_{x_1}) which triggers the cycle detection is a successor edge, and the remaining edges in the cycle $(v_{x_i}, v_{x_{i+1}})$, $i = 1 \ldots n\text{-}1$, are predecessor edges. Therefore, $\tau(v_{x_1}) < \tau(v_{x_2}) < \ldots < \tau(v_{x_n})$. Similarly, if $\tau(v_{x_n}) < \tau(v_{x_1})$, we have $\tau(v_{x_n}) < \tau(v_{x_{n-1}}) < \ldots < \tau(v_{x_1})$. The order of variables v_{x_i} is bounded by the larger value of $\tau(v_{x_1})$ and $\tau(v_{x_n})$; thus $x_i \in R$ for every i, because of the restriction on the variable order τ from Section 3.3.3.

Field annotations (with empty method annotations) appear only on edges of kinds (v_r, v_x) and (v_x, v_r), where v_x represents an object variable or a projection variable, and v_r represents reference variable. Similarly, method annotations (with empty field annotations) appear only on edges of the form $(\mathit{ref}(\ldots), v_r)$ and $(v_r, \mathit{lam}(\ldots))$. Therefore, edges (v_{r_i}, v_{r_j}), where $r_i, r_j \in R$ are edges with empty annotations.

Given the above lemma it can be shown that using this restricted form of cycle elimination preserves analysis correctness.

Projection merging is a technique for reducing redundant edge additions in constraint systems [65]. It combines multiple projection constraints for the same variable into a single projection constraint. For example, constraints $v \subseteq \mathit{proj}(c, i, u_1)$ and $v \subseteq \mathit{proj}(c, i, u_2)$ are replaced by

$$v \subseteq \mathit{proj}(c, i, w) \quad w \subseteq u_1 \quad w \subseteq u_2$$

where w is a special projection variable. For points-to analysis for C, constraints of the form $w \subseteq u_i$ are represented only as successor edges; this restriction guarantees a bound on the number of projection variables w. The analysis ensures the restriction by assigning to w a high index in the variable order τ [65]. In this case, projection merging is beneficial because it is coupled with cycle elimination.

In our annotated constraint systems, projection merging does not interact with cycle elimination. In our case, the high indices from [65] (which necessitate the interaction) are not required. The high indices become unnecessary because the bound on the number of special projection variables is ensured by the variable ordering restriction from Section 3.3.3. Thus, the special projection variables can be treated similarly to the rest of the variables in the constraint system.

3.3.5 Tracking Reachable Methods

Andersen's analysis implicitly assumes that all code in the program is executable. Since Java programs heavily use libraries that contain many unused methods, we have augmented our analysis to keep track of reachable methods, in order to avoid analyzing dead code. Thus, we take into account the effects of statements in a method body only if the method has been shown to be reachable from one of the entry methods of the program. The set of entry methods contains (i) the `main` method of the starting class, (ii) the methods invoked at JVM startup (e.g., `initializeSystemClass`), and (iii) the class initialization methods `<clinit>` containing the initializers for static fields [44, Section 3.9].

During the analysis, we maintain a list of reachable methods; whenever a method becomes reachable, all statements in its body are processed and the appropriate constraints are introduced in the constraint system. Any call to a constructor also generates a corresponding call to the appropriate `finalize` method. For multi-threaded programs, a call to `Thread.start` is treated as a call to the corresponding `run` method.

3.4 Empirical Results

We use the Soot framework (`www.sable.mcgill.ca`), version 1.0.0, to process Java bytecode and to build a typed intermediate representation [71]. The constraint-based analysis uses BANE (Berkeley ANalysis Engine) [3]. We instantiated the general constraint engine to represent and solve systems of constraints with field and method annotations (as described in Section 2.2.6 in Chapter 2). The analysis works on top of the constraint engine, by processing newly discovered reachable methods and generating the appropriate constraints. The points-to effects of JVM startup code and native methods (for

Program	User Class	Size (Kb)	Whole-program Class	Whole-program Method	Whole-program Stmt
proxy	18	56.6	565	3283	58837
compress	22	76.7	568	3316	60010
db	14	70.7	565	3339	60747
jb-6.1	21	55.6	574	3393	60898
echo	17	66.7	577	3544	62646
raytrace	35	115.9	582	3451	62755
mtrt	35	115.9	582	3451	62760
jtar-1.21	64	185.2	618	3583	65112
jlex-1.2.5	25	95.1	578	3381	65437
javacup-0.10	33	127.3	581	3564	66463
rabbit-2	52	157.4	615	3770	68277
jack	67	191.5	613	3573	69249
jflex-1.2.2	54	198.2	608	3692	71198
jess	160	454.2	715	3973	71207
mpegaudio	62	176.8	608	3531	71712
jjtree-1.0	72	272.0	620	4078	79587
sablecc-2.9	312	532.4	864	5151	82418
javac	182	614.7	730	4470	82947
creature	65	259.7	626	3881	83454
mindterm1.1.5	120	461.1	686	4420	90451
soot-1.beta.4	677	1070.4	1214	5669	92521
muffin-0.9.2	245	655.2	824	5253	94030
javacc-1.0	63	502.6	615	4198	102986

Table 3.1: Characteristics of the data programs. First two columns show the number and bytecode size of user classes. Last three columns include library classes.

JDK 1.1.8) are encoded in stubs included in the analysis input. Dynamic class loading (e.g., through `Class.forName`) and reflection (e.g., calls to `Class.newInstance`) are resolved manually; similar approaches are typical for static whole-program compilers and tools [36, 25, 68, 69].

All experiments were performed on a 900MHz Sun Fire-280R shared machine with 4Gb physical memory. The reported times are the median values out of three runs. We used 23 publicly available data programs, ranging in size from 56Kb to about 1Mb of bytecode. We used programs from the SPECjvm98 suite, other benchmarks used in previous work on analysis for Java, as well as programs from an Internet archive (`www.jars.com`) of popular publicly available Java applications.

Table 3.1 shows some characteristics of the data programs. The first two columns show the number of user (i.e., non-library) classes and their bytecode size. The next three columns show the size of the program, including library classes, after using class

Program	Time (sec)	Memory (Mb)	Time-nf (sec)	Memory-nf (Mb)
proxy	4.8	35.1	6.9	35.1
compress	8.3	39.6	25.7	60.1
db	9.2	40.6	26.3	62.1
jb	6.0	36.7	10.0	39.7
echo	18.7	49.2	75.2	125.0
raytrace	7.8	42.2	29.8	64.7
mtrt	9.4	42.1	30.1	64.7
jtar	16.8	50.3	40.0	80.1
jlex	6.7	39.8	32.9	66.0
javacup	23.2	55.8	27.1	66.7
rabbit	9.1	46.2	33.7	65.7
jack	28.7	54.8	162.2	105.1
jflex	28.5	63.5	79.8	130.1
jess	35.8	59.4	75.0	154.7
mpegaudio	11.6	44.0	35.7	73.4
jjtree	8.6	46.8	15.3	58.8
sablecc	34.5	78.5	423.6	157.0
javac	100.5	110.0	150.7	209.7
creature	64.3	94.3	457.9	255.2
mindterm	37.2	78.5	153.7	228.2
soot	139.4	117.8	162.8	182.1
muffin	120.7	133.9	-	-
javacc	99.6	96.6	61.6	121.2

Table 3.2: Running time and memory usage of the analysis (with and without field annotations).

hierarchy analysis (CHA) [18] to filter out irrelevant classes and methods.[11] The number of methods is essentially the number of nodes in the call graph computed by CHA. The last column shows the number of statements in Soot's intermediate representation (`Jimple`).

3.4.1 Analysis Cost

Our first set of experiments measured the cost of the analysis, as shown in Table 3.2. The first two columns show the running time of the analysis and the amount of memory used. For 18 out of the 23 programs, the analysis runs in less than a minute. For all programs, the running time is less than three minutes and the memory usage is less than 150Mb. These results show that our analysis is practical in terms of running time and

[11]CHA is an inexpensive analysis that determines the possible targets of a virtual call by examining the class hierarchy of the program.

memory usage, as evidenced on a large set of Java programs. This practicality means that the analysis can be used as a relatively precise general-purpose points-to analysis for advanced static compilers and software engineering tools for Java.

Analysis cost can be reduced further if the library code is analyzed in advance. This would allow certain partial analysis information about the Java libraries to be computed once and subsequently used multiple times for different client programs. We intend to investigate this approach in our future work.

We also investigated a version of our analysis in which no field annotations are used, and therefore individual object fields are not distinguished. The last two columns in Table 3.2 show the cost of this *no-fields* version. The running time is between 62% and 1228% (average 343%, median 286%) of the running time of the original analysis; the memory usage is between 105% and 291% (175% on average). Typically, the *no-fields* version is significantly more expensive; for one of the larger programs, it even ran out of memory. These results show the importance of distinguishing object fields: the improved precision produces smaller points-to sets, which in turn reduces analysis cost. By using field annotations, we have been able to distinguish object fields in a simple and efficient manner. The annotations restrict infeasible flow, which in this case results in a smaller constraint graph compared to the non-annotated constraint graph. A substantially smaller graph, means less work for the analysis and substantially improved performance.

3.4.2 Analysis Precision

Call Graph Construction and Virtual Call Resolution

To measure the precision with respect to call graph construction and virtual call resolution, we compared our points-to analysis with Rapid Type Analysis (RTA) [7]. RTA is an inexpensive and widely used analysis for call graph construction. It performs a reachability computation on the call graph generated by CHA; by keeping track of the classes that have been instantiated, RTA computes a more precise call graph than CHA.

Program	(a) Removed Targets		(b) Resolved Call Sites	
	Points-to	RTA	Points-to	RTA
proxy	8.7	5.7	60%	7%
compress	6.9	2.9	55%	18%
db	6.4	2.7	59%	18%
jb	8.8	6.0	48%	8%
echo	3.5	1.6	43%	20%
raytrace	6.4	2.8	56%	21%
mtrt	6.4	2.8	56%	21%
jtar	5.3	3.0	35%	17%
jlex	9.9	6.5	59%	13%
javacup	7.3	4.0	60%	11%
rabbit	8.3	3.7	52%	13%
jack	3.0	1.1	83%	11%
jflex	6.7	3.3	42%	13%
jess	5.9	2.4	55%	14%
mpegaudio	7.2	2.7	50%	17%
jjtree	8.5	5.8	48%	15%
sablecc	7.7	1.2	38%	4%
javac	2.9	1.2	31%	14%
creature	3.8	2.2	48%	26%
mindterm	2.7	1.3	45%	25%
soot	4.6	1.0	40%	1%
muffin	4.1	1.3	65%	12%
javacc	4.7	2.5	79%	11%
Average	6.1	2.9	53%	14%

Table 3.3: Improvements for CHA-unresolved virtual call sites. (a) Average reduction in the number of target methods per call site. (b) Percentage of uniquely resolved call sites.

Both our analysis and RTA improve the call graph computed by CHA by identifying sets of methods reachable from the entry points of the program; this reachability computation reduces the number of nodes in the call graph. For brevity, we summarize this reduction without explicitly showing the number of nodes for each program. The average reduction in the number of nodes is 54% for our analysis and 47% for RTA. On average, the call graph computed by our analysis has 14% less nodes than the call graph computed by RTA. This reduction allows subsequent analyses and optimizations to safely ignore portions of the program.

To determine the improvement for call graph edges, we considered call sites that could not be resolved to a single target method by CHA. Let V be the set of all CHA-unresolved call sites that occur in methods identified by our analysis as reachable. For each site from V, we computed the difference between the number of target methods according

Program	Thread-local	Method-local
proxy	44%	19%
compress	52%	27%
db	53%	28%
jb	45%	22%
echo	51%	26%
raytrace	56%	31%
mtrt	56%	31%
jtar	49%	24%
jlex	50%	24%
javacup	44%	22%
rabbit	54%	31%
jack	52%	29%
jflex	50%	23%
jess	46%	29%
mpegaudio	29%	15%
jjtree	50%	28%
sablecc	45%	29%
javac	51%	28%
creature	30%	16%
mindterm	49%	27%
soot	57%	35%
muffin	53%	32%
javacc	65%	49%
Average	49%	27%

Figure 3.6: Thread-local and method-local allocation sites.

to CHA and the number of target methods according to RTA and our analysis. The average differences are shown in the first section of Table 3.3. On average, our analysis removes more than twice as many targets as RTA; this improved precision is beneficial for reducing the cost and improving the precision of subsequent interprocedural analyses.

The second section of Table 3.3 shows the percentage of call sites from V that were resolved to a single target method. Our points-to analysis performs significantly better than RTA—on average, 53% versus 14% of the virtual call sites are resolved. The increased precision allows better removal of run-time virtual dispatch and additional method inlining.

Synchronization Removal and Stack Allocation

Points-to analysis has a wide variety of client applications, including optimizations for removal of unnecessary synchronization and for stack-based object allocation. To investigate the impact of our analysis on synchronization removal and stack allocation, we identified all object allocation sites that correspond to thread-local and method-local objects, as described in Section 3.2. Figure 3.6 shows what percentage of all allocation sites in reachable methods were identified as thread-local or method-local.

Across all programs, the analysis detects a significant number of allocation sites for thread-local objects—on average, about 49% of all allocation sites. These results indicate that the points-to information can be useful in detecting and eliminating unnecessary synchronization in Java programs. The analysis also discovers a significant number of sites for method-local objects—on average, about 27% of all sites. These results suggest that there are many opportunities for stack-based object allocation that can be detected with our analysis.

Chapter 4

Context-sensitive Points-to Analysis for Java

Recent work [52, 64, 43, 55, 42] has shown that flow- and context-insensitive points-to analysis for Java can be efficient and practical even for large programs, and therefore is a realistic candidate for use in optimizing compilers and software engineering tools. However, context insensitivity inherently compromises the precision of points-to analysis for object-oriented languages such as Java. This imprecision results from fundamental object-oriented features and programming idioms. (Section 4.1 presents several examples that illustrate this point.) The imprecision decreases the impact of the points-to analysis on client optimizations (e.g., virtual call resolution) and leads to less precise client analyses (e.g., def-use analysis). To make existing flow- and context-insensitive analyses more useful, it is important to introduce context sensitivity that targets the sources of imprecision that are specific to object-oriented languages. At the same time, the introduction of context sensitivity should not increase analysis cost to the point of compromising the practicality of the analysis.

In this chapter we propose *object sensitivity* as a new form of context sensitivity for flow-insensitive points-to analysis for Java. Our approach uses *the receiver object at a method invocation site* to distinguish different calling contexts. Conceptually, every method is replicated for each possible receiver object. The analysis computes separate points-to sets for each replica of a local variable; each of those points-to sets is valid for method invocations with the corresponding receiver object. Furthermore, the naming of objects is also object-sensitive: each object allocation site is represented by several object names, corresponding to different receiver objects for the enclosing method.

In addition, we propose a parameterization mechanism that allows the design of flexible object-sensitive analyses. The framework is parameterized in four dimensions.

In the context of this mechanism, analysis designers can select (i) the degree of precision in the object naming scheme, (ii) the set of reference variables for which the analysis maintains multiple points-to sets, (iii) the degree of precision in the context naming, and (iv) the set of call sites that are analyzed context-sensitively. This approach can be used to tune the cost of the analysis and to define *targeted sensitivity* for certain objects, variables, context and call sites for which more precise handling is likely to improve the analysis precision.

In this chapter we discuss parameterized object-sensitive points-to analysis that is based on Andersen-style points-to analysis for Java (as described in Chapter 3). Object sensitivity is modeled precisely and efficiently using annotated inclusion constraints. We employ *object annotations* to allow precise separate tracking of the flow of values through different replicas of a local variable or an object; the use of object annotations avoids potentially expensive actual replication of reference variables and objects, and allows efficient object-sensitive analysis. In addition, the parameterization mechanism can be modeled precisely using appropriate object annotations.

Although we demonstrate our technique on Andersen's analysis, parameterized object sensitivity can be trivially applied to enhance the precision of other flow- and context-insensitive analyses for Java (e.g., analyses that are based on flow- and context-insensitive points-to analyses for C [63, 61, 16]). [1]

4.1 The Imprecision of Context-insensitive Analysis

This section presents several examples of basic object-oriented features and programming idioms for which context-insensitive analysis produces imprecise results.

4.1.1 Encapsulation

Figure 4.1 illustrates the typical situation when an encapsulated field is written through a modifier method. At the call site at line 6, $\mathtt{y}_1$ points to o_3 and $\mathtt{x}_1$ points to o_1. After

[1] Our implementation technique based on inclusion constraints with object annotations does not apply to these analyses. These analyses are less precise than inclusion constraint-based ones; typically they are implemented using some form of a less expensive constraint-based mechanism.

```
  class X {...}
  class Y {
     X f;
1    void set(X x) { this.f = x; } }
2 s1:X x1 = new X();
3 s2:X x2 = new X();
4 s3:Y y1 = new Y();
5 s4:Y y2 = new Y();
6    y1.set(x1);
7    y2.set(x2);
```

Figure 4.1: Imprecision due to field encapsulation.

the analysis applies the transfer function for the virtual call (as shown in Figure 3.2), the implicit parameter `this` of method `set` points to o_3 and formal parameter `x` points to o_1. After the analysis processes the call at line 7, `this` points to o_4 and `x` points to o_2. Thus, at statement `this.f=x` at line 1, the analysis erroneously infers points-to edges $(\langle o_3, f\rangle, o_2)$ and $(\langle o_4, f\rangle, o_1)$.

The imprecision can be avoided if the analysis distinguishes invocations of `set` on o_3 from invocations of `set` on o_4. This could be achieved if the analysis were able to associate *multiple* points-to sets with `this` and with `x`, one for each of the objects on which `set` is invoked. This would allow statement `this.f=x` to be analyzed separately for each of the receiver objects, and would avoid creating spurious points-to edges. In Section 4.2.1 we show how object-sensitive analysis achieves this goal.

During context-insensitive analysis, there is a single copy of every method for all possible invocations. Therefore, field `f` of *each* receiver object will point to *all* objects passed as arguments to the method which sets the value of `f`. In object-oriented languages, encapsulation and information hiding are strongly supported, and fields are almost always accessed indirectly through method invocations. As a result, context-insensitive analysis can incur significant imprecision.

4.1.2 Inheritance

Consider the example in Figure 4.2. At line 2, which is executed after the constructor at line 10 is invoked, `B.this` points to o_3 and `B.xb` points to o_1. After the analysis processes the call to the superclass constructor, `A.this` and `A.xa` point to o_3 and o_1,

```
   class X { void n() {...} }
   class Y extends X { void n() {...} }
   class Z extends X { void n() {...} }
   class A {
      X f;
1     A(X xa) { this.f = xa; } }
   class B extends A {
2     B(X xb) { super(xb); ...  }
      void m() {
3        X xb = this.f;
4        xb.n(); } }
   class C extends A {
5     C(X xc) { super(xc); ...  }
      void m() {
6        X xc = this.f;
7        xc.n(); } }
8  s1:Y y = new Y();
9  s2:Z z = new Z();
10 s3:B b = new B(y);
11 s4:C c = new C(z);
12    b.m();
13    c.m();
```

Figure 4.2: Field assignment through a superclass.

respectively. Because of the call at line 5, `A.this` will point to o_4 and `A.xa` will point to o_2. Thus, at statement `this.f=xa` at line 1, spurious edges $(\langle o_3, f\rangle, o_2)$ and $(\langle o_4, f\rangle, o_1)$ are added to the graph. The imprecision propagates further, as the analysis infers that `xb` at line 3 points to both o_1 (of class `Y`) and o_2 (of class `Z`). Therefore, it appears that the possible targets of the virtual call at line 4 are `Y.n` and `Z.n` (the same problem also occurs at line 7). As a result, the calls at lines 4 and 7 cannot be devirtualized using the solution computed by the context-insensitive analysis. The imprecision is due to statement `this.f=xa` in the constructor of superclass `A`, which merges the information for all possible receiver objects.

In the presence of inheritance, instance fields are often located in superclasses and are written through invocations of superclass constructors or methods. During context-insensitive analysis, fields of subclass instances are perceived to point to objects intended for instances of other subclasses. In the presence of wide and deep inheritance hierarchies, context insensitivity can lead to substantial imprecision.

```
class Container {
    Object[] data;
    Container(int size) {
     s1:Object[] data_tmp = new Object[size];
        this.data = data_tmp; }
    void put(Object e,int at) {
        Object[] data_tmp = this.data;
        data_tmp[at] = e; }
    Object get(int at) {
        Object[] data_tmp = this.data;
        return data_tmp[at]; } }

s2:Container c1 = new Container(100);
s3:Container c2 = new Container(200);
s4:X x = new X();
   c1.put(x,0);
s5:X y = new Y();
   c2.put(y,1);
```

Figure 4.3: Simplified container class.

4.1.3 Collections and Maps

Consider the example in Figure 4.3. There is a single object name o_1 which represents the `data` arrays of both instances of `Container`. Therefore, objects stored in individual containers appear to be shared between the two containers. In order to avoid this imprecision, the `data` array of every instance of `Container` should be represented by a distinct object name. In addition, the analysis should be able to assign distinct points-to sets to `put.this` and `put.e` for every possible receiver object of `put`.

Context insensitivity causes data that is stored in one instance of a collection or a map to be retrieved from every other instance of the same class, and very likely from all instances of its subclasses. Since collections (e.g., `Vector`) and maps (e.g., `Hashtable`) are commonly used in Java, context insensitivity can seriously compromise analysis precision.

4.2 Object-sensitive Analysis

In context-sensitive analysis, a method is analyzed separately for different calling contexts. We define a new form of context-sensitive points-to analysis for Java which we refer to as *object-sensitive analysis.* With object sensitivity, each method and each constructor is analyzed separately for each object on which this method/constructor may be invoked. More precisely, the analysis uses a set of *object names* to represent objects allocated at run time. If a method/constructor may be invoked on run-time objects represented by object name o, the object-sensitive analysis maintains a separate contextual version of that method/constructor that corresponds to invocation context o.

Our object-sensitive analysis is based on Andersen's analysis for Java from Chapter 3. However, the same approach can be trivially applied to other flow- and context-insensitive analyses for Java (e.g., analyses derived from flow- and context-insensitive points-to analyses for C [63, 61, 16]). Section 4.2.1 defines the semantics of our object-sensitive analysis. Intuitively, it distinguishes calling context at a given call site by receiver object (distinguished by allocation site). Section 4.2.2 discusses why object sensitivity is appropriate for flow-insensitive analysis of object-oriented programs, and compares this approach with other context sensitivity mechanisms.

4.2.1 Semantics of Object-sensitive Analysis

The object-sensitive analysis is defined in terms of several sets. Recall from Section 3.1 in Chapter 3 that set R contains all reference variables (including static variables) and set F contains all instance fields. Set S contains all object creation sites in the program. We use a set of object names $O' \subseteq S \times (S \cup \{\epsilon\})$ and a set of replicated reference variables R'; these sets will be defined shortly.

For every method m we use a set of *reaching contexts* C_m. If m is an instance method or constructor, C_m is defined as the set of allocation sites of objects $o \in O'$ such that it is determined during the analysis that o can be receiver of m. If m is a static method, C_m is the union over all C_n such that n is the instance method or constructor first reached backwards on the call graph from m (i.e., traversal stops when an instance method/constructor is reached); if no instance method is reached on a path, C_m includes $\{\epsilon\}$, where ϵ denotes the special static context (e.g., $\{\epsilon\}$ is the context of start-up methods such as `main`, and the static initializers `<clinit>`).

For every allocation site $s_i \in S$ in method m the objects allocated at s_i are represented by a set of object names $\{o_{i,j} \mid o_j \in C_m\}$. For example, name $o_{i,j}$ represents all run-time objects that were created at s_i when instance method m was invoked on a receiver created at s_j. Consider allocation site s_1 in Figure 4.3 which appears in constructor `Container`. Sites s_2 and s_3 create instances of `Container`; therefore there are two object names $o_{1,2}$ and $o_{1,3}$ corresponding to that allocation site.

Set $\mathcal{C} = S \times \{\epsilon\}$ represents the space of all possible contexts for our object-sensitive analysis. Conceptually, separate analysis is achieved by maintaining multiple replicas of reference variables for each possible context. The set of replicated reference variables R' is defined by a function $map : R \times \mathcal{C} \rightarrow R'$. Static variables are mapped to themselves while every local variable r in method m is mapped to a "fresh" variable r^c for every reaching context $c \in C_m$. For the rest of the chapter we will refer to elements of R' as *context copies*.

The object-sensitive analysis constructs a points-to graph where the nodes in the graph are elements of O' and R'. The analysis semantics is given in Figure 4.4. Starting from a set of entry methods (e.g., `main`, class initializers `<clinit>`, and JVM startup methods such as `initializeSystemClass`) the analysis examines program statements and applies the transfer functions in Figure 4.4. Let C denote all sets of reaching contexts C_m. The effects of $F(G, C, s)$ for assignment statements are equivalent to applying the corresponding $f(G, s)$ from Figure 3.2 for each context from C_m, where m is the enclosing method of s. At virtual calls, in addition to adding new points-to edges due to parameter flow for the appropriate contexts, the transfer function has a side effect: the set of reaching contexts C_{n_j} of the callee n_j is augmented to include the new object context reaching the callee. Similarly, for static calls, the set of reaching contexts for the callee is updated appropriately.

Example

Consider the set of statements in Figure 4.2. Because of the constructor calls at line 10 and line 2 we have $o_3 \in C_B$ and $o_3 \in C_A$, and therefore

$$\{\texttt{B.this}^{o_3},\ \texttt{B.xb}^{o_3},\ \texttt{A.this}^{o_3},\ \texttt{A.xa}^{o_3}\} \subseteq R'.$$

$$F(G, C, s_i : l = new\ C) = G \cup \bigcup_{c \in C_m \wedge c = o_j} \{(l^{o_j}, o_{i,j})\}$$

$$F(G, C, l = r) = G \cup \bigcup_{c \in C_m} f(G, l^c = r^c)$$

$$F(G, C, l.f = r) = G \cup \bigcup_{c \in C_m} f(G, l^c.f = r^c)$$

$$F(G, C, l = r.f) = G \cup \bigcup_{c \in C_m} f(G, l^c = r^c.f)$$

$$\begin{aligned} & F(G, l = r_0.n(r_1, \ldots, r_n)) = \\ & \quad G \cup \bigcup_{c \in C_m} \{resolve(G, n, o_{i,k}, r_1^c, \ldots, r_n^c, l^c) \mid o_{i,k} \in Pt(G, r_0^c)\} \end{aligned}$$

$$\begin{aligned} & resolve(G, n, o_{i,k}, r_1^c, \ldots, r_n^c, l^c) = \\ & \qquad \texttt{let}\ \ c' = o_i \\ & \qquad n_j(p_0, p_1, \ldots, p_n, ret_j) = dispatch(o_{i,k}, n)\ \ \texttt{in} \\ & \qquad\qquad C_{n_j} = C_{n_j} \cup \{c'\} \\ & \qquad\qquad \{(p_0^{c'}, o_{i,k})\} \cup f(G, p_1^{c'} = r_1^c) \cup \ldots \cup f(G, l^c = ret_j^{c'}) \end{aligned}$$

$$\begin{aligned} & F(G, l = n(r_0, \ldots, r_n)) = \\ & \quad C_n = C_n \cup C_m \\ & \quad G \cup \bigcup_{c \in C_m} \{f(G, p_0^c = r_0^c) \cup \ldots \cup f(G, l^c = ret^c)\} \end{aligned}$$

Figure 4.4: Points-to effects of statements for the object-sensitive analysis. C_m is the set of reaching contexts for m, the enclosing method of the statement. r^c denotes $map(r, c)$. In the rule for virtual calls, n_j denotes a possible dynamic target of the call; c' is the new context for n_j.

Similarly,

$$\{\texttt{C.this}^{o_4},\ \ \texttt{C.xc}^{o_4},\ \ \texttt{A.this}^{o_4},\ \ \texttt{A.xa}^{o_4}\} \subseteq R'.$$

At line 2, $\texttt{B.this}^{o_3}$ points to $o_{3,\epsilon}$ and $\texttt{B.xb}^{o_3}$ points to $o_{1,\epsilon}$. When the analysis processes the call to `A.A` at line 2, `A.this` and `A.xa` are mapped to the context copies corresponding to o_3, and points-to edges ($\texttt{A.this}^{o_3}$, $o_{3,\epsilon}$) and ($\texttt{A.xa}^{o_3}$, $o_{1,\epsilon}$) are added to the graph. Similarly, $\texttt{A.this}^{o_4}$ points to $o_{4,\epsilon}$ and $\texttt{A.xa}^{o_4}$ points to $o_{2,\epsilon}$. Statement `this.f=xa` at line 1 occurs in reaching contexts o_3 and o_4. Thus, we have

$$\texttt{A.this}^{o_3} = \texttt{A.xa}^{o_3} \qquad\qquad \texttt{A.this}^{o_4} = \texttt{A.xa}^{o_4}$$

which produces edges $(\langle o_{3,\epsilon}, f\rangle, o_{1,\epsilon})$ and $(\langle o_{4,\epsilon}, f\rangle, o_{2,\epsilon})$.

4.2.2 Advantages of Object Sensitivity

In object-oriented languages such as Java, one of the primary roles of instance methods is to access or modify the state of the objects on which they are invoked. Instance methods typically work on encapsulated data, using implicit parameter `this` to modify or retrieve data from the object structure rooted at the receiver object. If points-to analysis does *not* distinguish the different receiver objects of instance methods, the states of these objects are essentially merged and any access/modification of the state of one object is propagated to all other objects. Therefore, it is crucial to distinguish the different objects pointed to by `this` and to analyze instance methods separately for different receiver objects. Similarly, the role of a constructor is to create the initial object state. To avoid merging the initial states of all objects pointed to by `this`, points-to analysis should distinguish the different objects on which a constructor is invoked.

Context sensitivity mechanisms of finer granularity than a receiver object may create redundant contextual versions. For example, one of the most popular mechanisms for context sensitivity is the *call string* approach, which represents invocation context using a string of k enclosing call sites. For $k = 1$, a method is analyzed separately for each call site that invokes that method. For many statements, it is redundant to distinguish between *distinct* call sites that have the *same* receiver object. For example, if statement `this.f=formal` were analyzed separately by flow insensitive analysis, for distinct call sites that have the same receiver object, the effect would be the same as if it were analyzed once for that object: field f of the receiver would point to all objects in the points-to sets of the corresponding actual parameters at all call sites. Clearly, because of the flow insensitivity of the analysis, the effects of the distinct per-call-site versions of the statement will be merged. The same kind of redundancy also occurs for statements that read the value of any field of the receiver object (e.g., `l=this.f`), as well as for certain method invocations on the receiver (e.g., `l=this.m()`). Therefore, such redundancies cause the call string approach to incur increased analysis cost without any precision gain. On the other hand, object-sensitive analysis performs exactly the necessary amount of work for such statements.

In certain cases, distinguishing calling context by a chain of enclosing call sites can be less precise than distinguishing context per receiver object. To illustrate such a case, recall the set of statements from Figure 4.2. Suppose that the following new statement is added at line 14:

14 s_5 : `C` $\mathtt{c}_2$ `= new C(y);`

If calling context is distinguished per call site ($k = 1$), the effects of constructor `A.A` invoked at line 5 are merged for receivers o_4 and o_5. Thus, there are redundant points-to edges $(\langle o_4, f\rangle, o_1)$ and $(\langle o_5, f\rangle, o_2)$. The imprecision propagates and affects both the points-to analysis and its clients; for example, the virtual call at line 7 cannot be resolved.

This example can be easily generalized to demonstrate that for any fixed k, distinguishing calling context by a chain of k enclosing call sites can be less precise than distinguishing context per receiver object. Consider a hierarchy of $k+1$ classes, denoted by $A_0, A_1, \ldots, A_k$. Classes A_0 and A_i, for i=1 ... k are defined as follows:

```
class A0 {
    X f;
    A0(X xa0) { this.f = xa0; } }

class Ai extends Ai-1 {
    Ai(X xai) { super(xai); } }
```

Consider the following statements

s_1 : `X x = new X();`
s_2 : $\mathtt{A}_k$ $\mathtt{a}_1$ `= new` $\mathtt{A}_k$`(x);`
s_3 : `Y y = new Y();`
s_4 : $\mathtt{A}_k$ $\mathtt{a}_2$ `= new` $\mathtt{A}_k$`(y);`

If calling context is distinguished by k enclosing call sites, the effects of constructor $\mathtt{A}_0.\mathtt{A}_0$ are merged for receivers o_2 and o_4. Thus, there are redundant points-to edges $(\langle o_4, f\rangle, o_1)$ and $(\langle o_3, f\rangle, o_2)$. The object-sensitive analysis is able to analyze this constructor separately for receivers o_2 and o_4 and avoids the imprecision.

In other cases, context-sensitive analysis based on the call string approach can produce more precise results than the object-sensitive analysis. For example, if statement `return formal` in an instance method were analyzed object-sensitively for distinct call sites that have the same receiver but different actual arguments, the analysis will imprecisely propagate the values of each corresponding actual to the left-hand side of every call. On the other hand, if calling context is distinguished per call site, the analysis will propagate to the left-hand side of the call only the actual passed at the same call. Our observations from examining large number of Java programs have indicated that field assignment through a chain of superclass constructors occurs frequently in Java programs, while return of parameter values occurs very infrequently. However, more experiments are needed to determine if the object-sensitive analysis (an instance of the functional approach to context sensitivity) performs better in practice than an analysis based on the call string approach [62].

4.3 Object-sensitive Points-to Analysis Using Annotated Constraints

In this section we show how to implement the object-sensitive analysis from Section 4.2.1 using annotated inclusion constraints. Recall that the object-sensitive analysis is defined in terms of five sets. Set R contains all reference variables, set F contains all instance fields, and set S contains all object creation sites in the program. Set O' is the set of precise object names, and set R' is the set of replicated reference variables. To simplify the presentation we use the notations for set S and set O interchangeably (recall from Section 3.1 in Chapter 3 that O denotes the set of all object names corresponding to allocation sites).

The analysis uses a term $\mathit{ref}(o, v_o, \overline{v_o})$ for each object name $o \in O$, and a set variable v_r for each reference variable $r \in R$. We model the precise naming scheme and the reference variable replication by using constraint annotations.

4.3.1 Modeling Object Sensitivity

Object Annotations

We use *object annotations* to model the flow of values through different context copies of reference variables and objects. The set of annotations $\mathcal{S}$ is the set of tuples of contexts plus the empty annotation; thus $\mathcal{S} = \{[o_1 \times o_2] \mid o_1, o_2 \in O \cup \{*\}\} \cup \{\epsilon\}$.

The role of the annotations is (i) to distinguish between the points-to sets associated with the same variable but different object contexts and (ii) to distinguish between objects allocated at the same site, but with different enclosing receivers. An annotated inclusion constraint $X \subseteq^{[o_1 \times o_2]} Y$ represents flow from the context copy of X for object context o_1, to the context copy of Y for object context o_2. For example, $\mathit{ref}(o_i, v_{o_i}, \overline{v_{o_i}}) \subseteq^{[o_1 \times o_2]} v_r$ means that $r^{o_2} \in R'$ points to $o_{i,1} \in O'$. Similarly $\mathit{ref}(o_i, v_{o_i}, \overline{v_{o_i}}) \subseteq_f^{[o_1 \times o_2]} v_{o_j}$ means that field f of object $o_{j,2} \in O'$ points to object $o_{i,1} \in O'$, and $v_r \subseteq^{[o_1 \times o_2]} v_l$ represents that the points-to set of $r^{o_1} \in R'$ is a subset of the points-to set of $l^{o_2} \in R'$.

The wild card symbol $*$ has standard meaning; it represents all reaching contexts associated with a given variable or object. Intuitively, it is used to model that the flow is valid for all context copies of the side of the constraint associated with $*$. For example, constraint $\mathit{ref}(o_i, v_{o_i}, \overline{v_{o_i}}) \subseteq^{[* \times o_2]} v_r$ means that r^{o_2} points to every $o_{i,j}$ where o_j is a reaching context for the enclosing method of object allocation site s_i.

The operations on the annotations are defined as follows. The match_o predicate is shown below:

$$\mathit{match}_o(t_1, t_2) = \begin{cases} \text{true} & \text{if } t_1 \text{ or } t_2 \text{ is the empty annotation } \epsilon \\ \text{true} & \text{if } t_1{=}[c_{r_1} \times c_{l_1}],\ t_2{=}[c_{r_2} \times c_{l_2}] \text{ and } c_{l_1}{=}c_{r_2} \text{ or } c_{l_1}{=}* \text{ or } c_{r_2} {=}* \\ \text{false} & \text{otherwise} \end{cases}$$

The concatenation operation, which produces the object annotation of the new constraint, and is applied only if $\mathit{match}_o(t_1, t_2)$ holds, is shown below:

$$\mathit{concat}_o(t_1, t_2) = \begin{cases} t_1 & \text{if } t_2 = \epsilon \\ t_2 & \text{if } t_1 = \epsilon \\ [c_{r_1} \times c_{l_2}] & \text{if } t_1{=}[c_{r_1} \times c_{l_1}] \text{ and } t_2{=}[c_{r_2} \times c_{l_2}] \end{cases}$$

An annotation is propagated until it is "matched" with another annotation. A new transitive constraint $X \subseteq Z$ is created from $X \subseteq^{t_1} Y \subseteq^{t_2} Z$ only if the constraint on the left and the constraint on the right respectively represent flow into, and flow from the *same* context copy of the intermediate variable Y. $\mathit{Transpose}_o$ is defined as follows:

$$\mathit{transpose}_o(t) = \begin{cases} t & \text{if } t = \epsilon \\ [c_l \times c_r] & \text{if } t = [c_r \times c_l] \end{cases}$$

Constraints for Assignment Statements

For assignment statements that involve local variables (i.e., non-static variables) the analysis generates constraints with appropriate object annotations that are identical to the ones shown in Figure 3.3 in Section 3.3.1, Chapter 3. The points-to set and the object annotations reaching a variable on the left-hand-side of such constraint are propagated to the right-hand-side of the constraint. For example, for assignment `l = r`, the analysis creates constraint $v_r \subseteq v_l$, where the empty annotation (identical to the analysis in Chapter 3) is used to represent all possible contexts in which local variables `r` and `l` appear. Intuitively, the object annotations reaching v_r are propagated to v_l. These constraints represent intraprocedural flow and model the semantics of the assignment statements in Figure 4.4.

Without loss of generality we may assume that statements that involve static variables are only of kind `l = r`. For such assignments the analysis generates the following constraints: [2]

$$\langle l = r \rangle \Rightarrow \begin{cases} \{v_r \subseteq^{[\epsilon \times *]} v_l\} & \text{if } r \text{ is static and } l \text{ is local} \\ \{v_r \subseteq^{[* \times \epsilon]} v_l\} & \text{if } r \text{ is local and } l \text{ is static} \end{cases}$$

Here ϵ represents the special static contexts associated with static variables. These constraints reflect the semantics described in Section 4.2.1. For example, when a static variable is assigned to a local variable, its value needs to be propagated to every possible context copy of the local variable; the wild card symbol models this semantics.

[2]We do not consider assignments `l = r`, where both l and r are static; clearly, such assignment can be substituted by a pair of statements `x = r`, `l = x` where x is a local variable. Our intermediate representation, derived from `Jimple`, does not include assignments that involve two static variables.

Constraints for Virtual Calls

Recall from Section 3.3.2, Chapter 3 that for every virtual call the analysis generates a constraint according to the following rule:

$$\langle l = r_0.m(r_1, \ldots, r_k)\rangle \Rightarrow$$
$$\{v_{r_0} \subseteq_m lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)\}$$

where m is the unique *compile-time* target of the call. In order to model the semantics of *resolve* in Figure 4.4, closure rule VIRTUAL from Chapter 3 is modified by introducing object annotations. If object annotations t_1 and t_2 "match", and method annotations c and m "match" in

$$ref(o_i, v_{o_i}, \overline{v_{o_i}}) \subseteq_c^{t_1} v \qquad v \subseteq_m^{t_2} lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)$$

the analysis finds a lambda term $lam(\overline{v_{p_0}}, \overline{v_{p_1}}, \ldots, \overline{v_{p_k}}, v_{ret})$ which corresponds to the definition of the run-time target method. [3] Let $concat_o(t_1, t_2) = t$. There are two cases. First, consider that t is non-empty, i.e., $t = [c_r \times c_l]$. In this case, the receiver is propagated to the call *interprocedurally* [4], and c_r represents the context of the receiver and c_l represents the context of the call site. The following constraints are introduced:

$$\begin{aligned} ref(o_i, v_{o_i}, \overline{v_{o_i}}) &\subseteq^{[c_r \times o_i]} \; v_{p_0} \\ lam(\overline{v_{p_0}}, \overline{v_{p_1}}, \ldots, \overline{v_{p_k}}, v_{ret}) &\subseteq^{[o_i \times c_l]} \; lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l) \end{aligned}$$

Second, consider the case when t is empty. In this case the allocation site of the receiver and the call site occur in the same context, and the receiver reaches the call site *intraprocedurally.* According to the semantics in Figure 4.4, $resolve(G, n, o_{i,c}, r_1^c, \ldots, r_k^c, l^c)$ is applied for every context c reaching the enclosing method; thus, the points-to sets of every context copy r_j^c of an actual argument, is propagated to $p_j^{o_i}$, and similarly ret^{o_i} is propagated to every l^c. The analysis models this semantics by using the wild card

[3] Note that variable v is not necessarily the same as v_{r_0} (where v_{r_0} was used in the constraint generation rule for virtual calls shown above). If $\tau(v) < \tau(v_{r_0})$, the *lam* term is propagated to v which may result in non-empty object annotation t_2.

[4] The call may be propagated to the receiver as well.

symbol in the following constraints:

$$\begin{array}{rcl} \mathit{ref}(o_i, v_{o_i}, \overline{v_{o_i}}) & \subseteq^{[*\times o_i]} & v_{p0} \\ \mathit{lam}(\overline{v_{p0}}, \overline{v_{p1}}, \ldots, \overline{v_{p_k}}, v_{ret}) & \subseteq^{[o_i\times *]} & \mathit{lam}(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l) \end{array}$$

Example: Field Assignment Through a Superclass

Consider the statements in Figure 4.2. At line 10, the analysis creates constraints:

$$\mathit{ref}(o_3, v_{o_3}, \overline{v_{o_3}}) \subseteq v_b \subseteq_{B.B} \mathit{lam}(\overline{0}, \overline{v_y})$$

Applying VIRTUAL results in the following constraints:

$$\begin{array}{rcl} \mathit{ref}(o_3, v_{o_3}, \overline{v_{o_3}}) & \subseteq^{[*\times o_3]} & v_{B.this} \\ \mathit{lam}(\overline{v_{B.this}}, \overline{v_{B.xb}}) & \subseteq^{[o_3\times *]} & \mathit{lam}(\overline{0}, \overline{v_y}) \end{array}$$

The second constraint requires application of the rules for structural constraints in Figure 2.1. Due to the contravariance of the arguments of *lam*, operation $\mathit{transpose}_o$ is applied to $[o_3 \times *]$ and the application of the rules results in the trivial constraint $0 \subseteq^{[*\times o_3]} v_{B.this}$ and constraint $v_y \subseteq^{[*\times o_3]} v_{B.xb}$. We have the following non-trivial constraints:

$$\mathit{ref}(o_3, v_{o_3}, \overline{v_{o_3}}) \subseteq^{[*\times o_3]} v_{B.this} \qquad v_y \subseteq^{[*\times o_3]} v_{B.xb}$$

After line 2 we have

$$\mathit{ref}(o_3, v_{o_3}, \overline{v_{o_3}}) \subseteq^{[*\times o_3]} v_{B.this} \subseteq_{A.A} \mathit{lam}(\overline{0}, \overline{v_{B.xb}})$$

The analysis applies VIRTUAL on the above constraints. Clearly, $[* \times o_3]$ and ϵ match, and $\mathit{concat}_o([* \times o_3], \epsilon) = [* \times o_3]$. As a final result of VIRTUAL we have:

$$\begin{array}{rcl} \mathit{ref}(o_3, v_{o_3}, \overline{v_{o_3}}) & \subseteq^{[*\times o_3]} & v_{A.this} \\ \mathit{lam}(\overline{v_{A.this}}, \overline{v_{A.xa}}) & \subseteq^{[o_3\times o_3]} & \mathit{lam}(\overline{0}, \overline{v_{B.xb}}) \end{array}$$

After applying the rules in Figure 2.1 on the second constraint, we have the following non-trivial constraints:

$$\mathit{ref}(o_3, v_{o_3}, \overline{v_{o_3}}) \subseteq^{[*\times o_3]} v_{A.this} \;\; (1) \qquad v_{B.xb} \subseteq^{[o_3\times o_3]} v_{A.xa} \;\; (2)$$

Applying the same sequence of rules at line 11, analogously to line 10, results in

$$\mathit{ref}(o_4, v_{o_4}, \overline{v_{o_4}}) \subseteq^{[*\times o_4]} v_{C.this} \qquad v_z \subseteq^{[*\times o_4]} v_{C.xc}$$

After line 5, analogously to line 2, we have

$$\mathit{ref}(o_4, v_{o_4}, \overline{v_{o_4}}) \subseteq^{[*\times o_4]} v_{A.this} \quad (3) \quad v_{C.xc} \subseteq^{[o_4\times o_4]} v_{A.xa} \quad (4)$$

At line 1 the analysis generates the following constraints:

$$v_{A.this} \subseteq \mathit{proj}(\mathit{ref}, 3, u) \quad (5) \qquad v_{A.xa} \subseteq_f u \quad (6)$$

The analysis applies TRANS to (1) and (5). Clearly, $\mathit{match}_o([* \times o_3], \epsilon)$ holds, and $\mathit{concat}_o([* \times o_3], \epsilon) = [* \times o_3]$. Following the application of TRANS, the resolution rules in Figure 2.1 are applied on constraint $\mathit{ref}(o_3, v_{o_3}, \overline{v_{o_3}}) \subseteq^{[*\times o_3]} \mathit{proj}(\mathit{ref}, 3, u)$ and after the application of $\mathit{transpose}_o([* \times o_3])$, we have $u \subseteq^{[o_3\times *]} o_3$ (recall that constructor *ref* is contravariant in its third argument). Similarly, from (3) and (5) we have $u \subseteq^{[o_4\times *]} o_4$. Clearly, the annotations are used to distinguish between flow from variable u in two separate object contexts. In the first case, the constraint represents flow from the context copy of u associated with object context o_3 and in the second case it represents flow from the context copy associated with o_4. Therefore we have

$$\mathit{ref}(o_1, v_{o_1}, \overline{v_{o_1}}) \subseteq v_y \subseteq^{[*\times o_3]} v_{B.xb} \subseteq^{[o_3\times o_3]} v_{A.xa} \subseteq_f u \subseteq^{[o_3\times *]} v_{o_3} \quad (7)$$
$$\mathit{ref}(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq v_z \subseteq^{[*\times o_4]} v_{C.xc} \subseteq^{[o_4\times o_4]} v_{A.xa} \subseteq_f u \subseteq^{[o_4\times *]} v_{o_4} \quad (8)$$

For the purposes of this example we assume that the variable order τ is: $\tau(v_y) < \tau(v_z) < \tau(v_{B.xb}) < \tau(v_{C.xc}) < \tau(v_{A.xa}) < \tau(u) < \tau(v_{o_3}) < \tau(v_{o_4})$. The least solution for variables v_{o_3} and v_{o_4} is computed by transitive acyclic traversal (recall Sections 2.2.4 and 2.2.5 in Chapter 2). From the above constraints the least solution for u is: [5]

$$LS(u) \quad = \quad \{\ \langle \mathit{ref}(o_1, v_{o_1}, \overline{v_{o_1}}), \langle f, \_, [* \times o_3]\rangle\rangle, \quad \langle \mathit{ref}(o_2, v_{o_2}, \overline{v_{o_2}}), \langle f, \_, [* \times o_4]\rangle\rangle\ \}$$

The least solution for v_{o_3} is computed by examining the least solution for u. Since $[*\times o_3]$ and $[o_3 \times *]$ "match", there is constraint $\mathit{ref}(o_1, v_{o_1}, \overline{v_{o_1}}) \subseteq_f^{[*\times *]} v_{o_3}$ in the least solution

[5]We use the notation $\langle f, m, o\rangle$ to denote tuples of field, method, and object annotations. Although precisely the method annotations should be present in the tuples for the least solution, they are omitted to shorten the notation and emphasize the manipulation of object annotations.

of the system (implied by the constraints in line (7)). This constraint implies that there is a field edge f from each object created at allocation site 3 (under any possible object context) to each object created at site 1 (under any possible context). In this case, allocation sites 1 and 3 appear only in the special context of `main`, which is denoted by ϵ; therefore, the only points-to edge that the above constraint implies is $(\langle o_{3,\epsilon}, f\rangle, o_{1,\epsilon})$. Similarly, the least solution for v_{o4} is computed by examining the least solution for u, and since $[* \times o_4]$ and $[o_4 \times *]$ "match" there is constraint $\mathit{ref}(o_2, v_{o_2}, \overline{v_{o2}}) \subseteq_f^{[*\times*]} v_{o4}$ (implied by the constraints in line (8)). This constraint implies a field edge $(\langle o_{4,\epsilon}, f\rangle, o_{2,\epsilon})$.

When $LS(u)$ is examined for the computation of the least solution of v_{o_3}, $[* \times o_4]$ and $[o_3 \times *]$ do not "match" and the analysis avoids inferring constraint $\mathit{ref}(o_2, v_{o_2}, \overline{v_{o2}}) \subseteq_f^{[*\times*]} v_{o_3}$. The last constraint erroneously implies a field edge between $o_{3,\epsilon}$ and $o_{2,\epsilon}$. Similarly, because $[* \times o_3]$ and $[o_4 \times *]$ do not match, the constraint that implies a spurious field edge between $o_{4,\epsilon}$ and $o_{1,\epsilon}$ is avoided as well. Clearly, such constraints represent infeasible flow; due to the use of the annotations they are avoided during the analysis.

Example: Containers

Consider the example in Figure 4.3 which illustrates flow of values through container objects. After processing the constructor call on line 7, and statements 1 and 2 we have:

$$\mathit{ref}(o_2, v_{o_2}, \overline{v_{o2}}) \subseteq^{[*\times o_2]} v_{Container.this} \subseteq \mathit{proj}(\mathit{ref}, 3, u)$$
$$\mathit{ref}(o_1, v_{o_1}, \overline{v_{o1}}) \subseteq v_{Container.data\_temp} \subseteq_{data} u$$

Applying Trans on the first two constraints results in constraint $\mathit{ref}(o_2, v_{o_2}, \overline{v_{o2}}) \subseteq^{[*\times o_2]} \mathit{proj}(\mathit{ref}, 3, u)$ (clearly, $\mathit{match}_o([*\times o_2], \epsilon)$ holds, and $\mathit{concat}_o([*\times o_2], \epsilon) = [*\times o_2]$). Subsequently, the application of the rules for structural constraints results in $u \subseteq^{[o_2\times *]} v_{o_2}$. For the purposes of this example, we assume the following variable order: $\tau(v_{Container.data\_temp}) < \tau(v_{o_2}) < \tau(u)$. Applying Trans on the constraints given on the last line above, and $u \subseteq^{[o_2\times *]} v_{o_2}$ results in

$$\mathit{ref}(o_1, v_{o_1}, \overline{v_{o1}}) \subseteq v_{Container.data\_temp} \subseteq_{data}^{[o_2\times *]} v_{o_2} \quad (1)$$

Note that application of Trans stops due to the variable ordering which forces the last constraint to be represented as a predecessor edge. These constraints imply that field *data* of object $o_{2,\epsilon}$ points to array object $o_{1,2}$ (in this case, the only possible enclosing context for allocation site 2 is ϵ). After processing the call to `put` on line 10 we have

$$ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq^{[*\times o_2]} v_{put.this} \qquad v_x \subseteq^{[*\times o_2]} v_{put.e} \tag{2}$$

For statement 3 in `put` the analysis generates constraints $v_{put.this} \subseteq proj(ref, 2, v)$ and $v \subseteq_{data} v_{put.data\_temp}$; for statement 4 it generates $v_{put.data\_temp} \subseteq proj(ref, 3, w)$ and $v_{put.e} \subseteq_{ar} w$. [6] We rewrite these constraints together with the ones in (2):

$$ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq^{[*\times o_2]} v_{put.this} \subseteq proj(ref, 2, v) \tag{3}$$

$$v \subseteq_{data} v_{put.data\_temp} \subseteq proj(ref, 3, w) \tag{4}$$

$$v_x \subseteq^{[*\times o_2]} v_{put.e} \subseteq_{ar} w \tag{5}$$

Applying Trans to (3) followed by the rules for structural constrains results in $v_{o_2} \subseteq^{[*\times o_2]} v$. Note that operation $transpose_o$ is not applied, because *ref* is covariant in the second argument. Rewriting this constraint with (4) results in

$$v_{o_2} \subseteq^{[*\times o_2]} v \subseteq_{data} v_{put.data\_temp} \subseteq proj(ref, 3, w)$$

Assuming $\tau(v_{put.data\_temp}) < \tau(v_{o_2}) < \tau(v)$, after applying Trans once we have

$$v_{o_2} \subseteq_{data}^{[*\times o_2]} v_{put.data\_temp} \subseteq proj(ref, 3, w)$$

Rewriting this constraint with (1) results in

$$ref(o_1, v_{o_1}, \overline{v_{o_1}}) \subseteq v_{Container.data\_temp} \subseteq_{data}^{[o_2\times *]} v_{o_2} \subseteq_{data}^{[*\times o_2]} v_{put.data\_temp} \subseteq proj(ref, 3, w)$$

and after application of Trans (multiple times) followed by the rules for structural constraints we have $w \subseteq^{[o_2\times o_2]} v_{o_1}$. Rewriting this constraint with (5) results in

$$v_x \subseteq^{[*\times o_2]} v_{put.e} \subseteq_{ar} w \subseteq^{[o_2\times o_2]} v_{o_1} \tag{6}$$

The constraints in (6) imply $v_x \subseteq_{ar}^{[*\times o_2]} v_{o_1}$ which represents that the points-to set of `x` in `main` is contained in the array $o_{1,2}$ (in this case the only possible context for `x` is the special context of `main`, denoted by ϵ). Similarly, the analysis infers $v_y \subseteq_{ar}^{[*\times o_3]} v_{o_1}$ which represents that the points-to set of `y` is contained in the array $o_{1,3}$.

[6] Here *ar* denotes a special field used to model array accesses.

Suppose that the following new statement is added to the program

```
13 X x₁ = c₁.get();
```

Processing this call and the statements in `get` results in $v_{o_1} \subseteq_{ar}^{[o_2\times *]} v_{x_1}$. The analysis infers $v_x \subseteq^{[*\times *]} v_{x_1}$, which implies correctly that the points-to set of `x` is a subset of the points-to set of $\mathtt{x}_1$. However, since $[*\times o_3]$ and $[o_2 \times *]$ do not match, the analysis avoids inferring constraint $v_y \subseteq^{[*\times *]} v_{x_1}$, which represents infeasible flow from `y` to $\mathtt{x}_1$.

4.3.2 Correctness and Precision

Let G be a points-to graph as defined in Section 4.2.1, and C_{m_i} be the corresponding sets of reaching contexts for the methods m_i in the program. Let A be an annotated constraint graph in inductive form. The sets of reaching contexts $C_{m_i}^A$ implied by A are inferred as follows:

- $o_i \in C_m^A$ if m is an instance method and A contains edge $\mathit{ref}(o_i, v_{o_i}, \overline{v_{o_i}}) \xrightarrow{t} v_{this_m}$
- $o_i \in C_m^A$ if m is a static method reachable through a sequence of static invocations from instance method m' and $o_i \in C_{m'}^A$
- $\epsilon \in C_m^A$ if m is a static method reachable through a sequence of static invocations from the entry methods (as defined in Section 4.2.1).

We define a subsumption relation between an object annotation in the context of an edge in A (e.g., $t_r \xrightarrow{t} v_l$), and tuples of object contexts. Intuitively, the relation defines the tuples of contexts represented by an object annotation that contains a wild card, or an empty annotation (recall that such annotations may represent multiple tuples of object contexts).

$$t \text{ subsumes} \begin{cases} \text{every } [c\times c] \text{ such that } c \in C_{m(r,l)}^A & \text{if } t \text{ is empty} \\ [c_r \times c_l] & \text{if } t = [c_r \times c_l] \\ \text{every } [c_r \times c_l] \text{ such that } c_r \in C_{m(r)}^A & \text{if } t = [*\times c_l] \\ \text{every } [c_r \times c_l] \text{ such that } c_l \in C_{m(l)}^A & \text{if } t = [c_r \times *] \\ \text{every } [c_r \times c_l] \text{ such that } c_r \in C_{m(r)}^A \text{ and } c_l \in C_{m(l)}^A & \text{if } t = [*\times *] \end{cases}$$

Correctness

We define *representation relations* α', between the reaching contexts implied by A, denoted by C^A, and the reaching contexts C, and α between A and the edges in G. Consider an object context o_i and method m, such that $o_i \in C_m$. We have $\alpha'(C^A, o_i \in C_m)$ if and only if $o_i \in C^A_m$. If α' holds between C^A and all contexts $o_i \in C_{m_i}$ we will write $\alpha'(C^A, C_{m_i})$. Similarly, if α' holds between C^A and all C_{m_i} we will write $\alpha'(C^A, C)$.

Consider a reference variable $r^{o_i} \in R'$ and an object $o_{j,k} \in O'$ such that $e = (r^{o_i}, o_{j,k})$ is an edge in G. We have $\alpha(A, e)$ if and only if A contains a path

$$\mathit{ref}(o_j, v_{o_j}, \overline{v_{o_j}}) \xrightarrow{t_0} v_1 \xrightarrow{t_1} \ldots \to v_n \xrightarrow{t_n} v_r$$

such that adjacent non-empty t_i, t_j match and $t = t_0 \oplus t_1 \oplus \ldots t_n$ subsumes $[o_k \times o_i]$. [7] Similarly, consider two objects $o_{i,j}$ and $o_{k,l}$ such that $e = (\langle o_{i,j}, f\rangle, o_{k,l})$ is an edge in G. We have $\alpha(A, e)$ if and only if A contains a path

$$\mathit{ref}(o_k, v_{o_k}, \overline{v_{o_k}}) \xrightarrow{t_0} v_1 \xrightarrow{t_1} \ldots \to v_n \xrightarrow{\langle f,\epsilon,t_n\rangle} v_{o_i}$$

such that adjacent non-empty t_i, t_j match, and $t = t_0 \oplus t_1 \oplus \ldots t_n$ subsumes $[o_l \times o_j]$. If α holds between A and all edges in G we will write $\alpha(A, G)$.

Let G^* be the final points-to graph, and C^* be the final set of contexts computed by the algorithm in Section 4.2.1. Let A^* be the solved inductive form of the corresponding annotated constraint system. The least solution of the constraint system is obtained through additional traversal of predecessor edges in A^*, as described in Section 2.2.4.

Suppose that $\alpha(A^*, G^*)$. It is easy to show that in this case G^* is contained in the set of points-to pairs extracted from the least solution of the constraint system. For example, for every edge $(\langle o_{i,j}, f\rangle, o_{k,l}) \in G^*$, the least solution contains $\mathit{ref}(o_k, v_{o_k}, \overline{v_{o_k}}) \subseteq^t_f v_{o_i}$, where t subsumes $[o_l \times o_j]$, i.e., points-to edge $(\langle o_{i,j}, f\rangle, o_{k,l})$ can be extracted from A^*. To prove that $\alpha(A^*, G^*)$, we use the following lemma:

Lemma 4 **(Correctness)** *Let $\langle s, G, C\rangle \Rightarrow \langle G', C'\rangle$ as described in Figure 4.4. If $\alpha(A, G)$*

[7] For the rest of this chapter we will use the short notation $\oplus$ instead of $concat_o$ to denote concatenation of object annotations. Due to the associativity of $\oplus$ we can omit parentheses in expressions of the form $t_0 \oplus t_1 \oplus \ldots t_n$.

and $\alpha'(C^A, C)$, there exists a sequence of applications of closure rules and resolution rules such that A can be transformed into A' for which $\alpha(A', G')$ and $\alpha'(C^{A'}, C')$.

The proof of the lemma requires case-by-case analysis of all statement kinds. Consider the assignment `l = r`. We consider three cases. First, suppose that both `l` and r are local variables. In G' there is a new edge $(l^{o_i}, o_{j,k})$ for every $(r^{o_i}, o_{j,k}) \in G$. In A we have a path $\mathit{ref}(o_j, v_{o_j}, \overline{v_{o_j}}) \rightarrow\overset{t}{\ldots}\rightarrow v_r$ such that t subsumes $[o_k \times o_i]$ for every $(r^{o_i}, o_{j,k})$. In A' we have an edge $v_r \rightarrow v_l$. If it is a predecessor edge then we have the needed path; otherwise it is created by a sequence of applications of Trans. Second, suppose that r is a local variable and l is a static variable. In G' there is a new edge $(l, o_{j,k})$ for every $(r^{o_i}, o_{j,k}) \in G$. In A we have a path $\mathit{ref}(o_j, v_{o_j}, \overline{v_{o_j}}) \rightarrow\overset{t}{\ldots}\rightarrow v_r$ such that t subsumes $[o_k \times o_i]$ for every edge $(r^{o_i}, o_{j,k}) \in G$. In A' we have an edge $v_r \xrightarrow{[*\times\epsilon]} v_l$. Since $*$ matches every object context, in A' we will have a path with an annotation $t \oplus [* \times \epsilon]$ which subsumes $[o_k \times \epsilon]$. Similarly consider the third case, when l is a local variable and r is a static variable. In A, there is a path $\mathit{ref}(o_j, v_{o_j}, \overline{v_{o_j}}) \rightarrow\overset{[C_r\times\epsilon]}{\ldots}\rightarrow v_r$, such that $[C_r \times \epsilon]$ subsumes $[o_k \times \epsilon]$ for every edge $(r, o_{j,k}) \in G$. In A' we have an edge $v_r \xrightarrow{[\epsilon\times *]} v_l$ which leads to a path $\mathit{ref}(o_j, v_{o_j}, \overline{v_{o_j}}) \rightarrow\overset{[C_r\times *]}{\ldots}\rightarrow v_l$. Since $\alpha'(C^A, C_{m(l)})$, $*$ represents all possible contexts for the enclosing method of l. Therefore, $[C_r \times *]$ subsumes $[o_k \times o_i]$ and A' correctly represents every points-to edge $(l^{o_i}, o_{j,k}) \in G'$.

Consider the assignment `l.f = r`. In G' we have new edges $(\langle o_{j,k}, f\rangle, o_{l,m})$ for every $(l^{o_i}, o_{j,k}) \in G$ and $(r^{o_i}, o_{l,m}) \in G$. In A we have a path $\mathit{ref}(o_j, v_{o_j}, \overline{v_{o_j}}) \rightarrow\overset{t_1}{\ldots}\rightarrow v_l$ such that t_1 subsumes $[o_k \times o_i]$ for every $(l^{o_i}, o_{j,k}) \in G$; similarly, we have a path $\mathit{ref}(o_l, v_{o_l}, \overline{v_{o_l}}) \rightarrow\overset{t_2}{\ldots}\rightarrow v_r$, such that t_2 subsumes $[o_m \times o_i]$ for every $(r^{o_i}, o_{l,m}) \in G$. In A' we have a successor edge with empty object annotation $v_l \overset{\langle f,\epsilon,\epsilon\rangle}{\rightarrow} \mathit{proj}(\mathit{ref}, 3, v_r)$. [8] Therefore, we have

$$\mathit{ref}(o_j, v_{o_j}, \overline{v_{o_j}}) \rightarrow\overset{t_1}{\ldots}\rightarrow v_l \overset{\langle f,\epsilon,\epsilon\rangle}{\rightarrow} \mathit{proj}(\mathit{ref}, 3, v_r)$$

and after applying Trans (possibly multiple times) followed by the resolution rules for

[8] For brevity we omit the use of projection variable u. Clearly, constraint $v_l \subseteq_f \mathit{proj}(\mathit{ref}, 3, v_r)$ has the same effect as $v_l \subseteq \mathit{proj}(\mathit{ref}, 3, u)$ and $u \subseteq_f v_r$.

structural constraints there is an edge $v_r \stackrel{\langle f,\epsilon,transpose_o(t_1)\rangle}{\longrightarrow} v_{o_j}$. Thus we have

$$ref(o_l, v_{o_l}, \overline{v_{o_l}}) \rightarrow .\stackrel{t_2}{.} \rightarrow v_r \stackrel{\langle f,\epsilon,transpose_o(t_1)\rangle}{\longrightarrow} v_{o_j}$$

Since t_2 subsumes $[o_m \times o_i]$ and t_1 subsumes $[o_k \times o_i]$, t_2 and $transpose_o(t_1)$ "match". It is easy to see that in this case $t = t_2 \oplus transpose_o(t_1)$ subsumes $[o_m \times o_k]$; therefore, we have the needed path.

The rest of the statements are handled in similar manner. Given this lemma it is trivial to show that α holds between A^* and G^*. **Q.E.D.**

Precision

In order to prove that the analysis is precise, we define correspondence relations β' and β (the inverses of α' and α respectively). We have $\beta'(o_i \in C^A_m, C)$ if and only if $o_i \in C_m$. Intuitively, β' holds if and only if a context o_i which was determined to reach method m in A, reaches method m according to the algorithm in Section 4.2.1. We will use $\beta'(C^A_m, C)$ to denote that $\beta'(o_i \in C^A_m, C)$ holds for all contexts reaching m in A, and similarly, $\beta'(C^A, C)$ denotes that $\beta'(C^A_m, C)$ holds for all m.

Relation β is a relation between the edges of A and points-to graph G. Consider an object name o_i and a reference variable l such that e: $ref(o_i, v_{o_i}, \overline{v_{o_i}}) \xrightarrow{t} v_l$ is a predecessor edge in A. We have $\beta(e, G)$ if and only if for every $[c_{o_i} \times c_l]$ subsumed by t, $o_{i,c_{o_i}} \in Pt(l^{c_l})$ in G. [9] Consider two reference variables r and l such that e: $v_r \xrightarrow{t} v_l$ is a predecessor or successor edge in A. We have $\beta(e, G)$ if and only if for every $[c_r \times c_l]$ subsumed by t, $Pt(r^{c_r}) \subseteq Pt(l^{c_l})$ in G. Similarly, consider a reference variable r and an object name o_i such that e: $v_r \xrightarrow{\langle f,\epsilon,t\rangle} v_{o_i}$ is a predecessor edge in A. We have $\beta(e, G)$ if and only if for every $[c_r \times c_{o_i}]$ subsumed by t, $Pt(r^{c_r}) \subseteq Pt(\langle o_{i,c_{o_i}}, f\rangle)$ in G. For an object o_i and reference variable l such that e: $v_{o_i} \xrightarrow{\langle f,\epsilon,t\rangle} v_l$ is a successor edge in A, we have $\beta(e, G)$ if and only if for every $[c_{o_i} \times c_l]$ subsumed by t, $Pt(\langle o_{i,c_{o_i}}, f\rangle) \subseteq Pt(l^{c_l})$ in G. If β holds between all edges of the kinds specified above and G we will write $\beta(A, G)$.

Suppose that $\beta(A^*, G^*)$. In this case the set of points-to pairs extracted from A^*

[9] $Pt(x)$ denotes the points-to set of variable x and $Pt(\langle o, f\rangle)$ denotes the points-to set of field f of object o.

is contained in the set of pairs in G^* and the solution computed by the algorithm in Section 4.3 is precise. To prove that $\beta(A^*, G^*)$ we use the following lemma:

Lemma 5 *(***Precision***) Let A be transformed into A' by the application of a sequence of resolution rules and closure rules. If $\beta(A, G^*)$ and $\beta'(C^A, C^*)$, then $\beta(A', G^*)$ and $\beta'(C^{A'}, C^*)$.*

The proof of the lemma is carried out by induction in two steps. First, let A_0 be the constraint graph, after constraint generation and before the application of closure rules and resolution rules. We show that $\beta'(C^{A_0}, C^*)$ and $\beta(A_0, G^*)$ hold. Before the application of resolution rules and closure rules we have non-empty $C_m^{A_0} = \{\epsilon\}$ only for static methods m transitively reachable from entry methods. Clearly, for every such method m we have $\epsilon \in C_m$; therefore $\beta'(C^{A_0}, C^*)$. In order to show that $\beta(A_0, G^*)$ consider an edge with empty annotation $v_r \rightarrow v_l$. We have to show that for every $[c \times c]$ subsumed by ϵ we have $Pt(r^c) \subseteq Pt(l^c)$. In this case ϵ subsumes static contexts $[\epsilon \times \epsilon]$ for local variables r and l in a static method m transitively reachable from an entry method. In G^*, since $\epsilon \in C_m$, we have $Pt(r^\epsilon) \subseteq Pt(l^\epsilon)$ due to statement `l = r`. Therefore, for such an edge we have $\beta(e, G^*)$. It is easy to see that β holds for annotated edges representing constraints that involve static variables.

Second, we use case-by-case analysis of the application of resolution rules and closure rules. Consider an application of rule TRANS that results in a new edge between two reference variables. Consider the sequence of predecessor and successor edges

$$v_p \xrightarrow{t_1} v_r \xrightarrow{t_2} v_l$$

If t_1 and t_2 match, $t = t_1 \oplus t_2$ subsumes $[c_p \times c_l]$ if and only if there exists c_r such that t_1 subsumes $[c_p \times c_r]$ and t_2 subsumes $[c_r \times c_l]$. Because $\beta(A, G^*)$ holds, for every $[c_p \times c_l]$ we have $Pt(p^{c_p}) \subseteq Pt(r^{c_r}) \subseteq Pt(l^{c_l})$ for some c_r. Therefore $Pt(p^{c_p}) \subseteq Pt(l^{c_l})$ for every $[c_p \times c_l]$ subsumed by t and thus we have $\beta(A', G^*)$.

Consider an application of resolution rule $c(v_1, ..., v_n) \subseteq_a proj(c, i, v)$. Consider the sequence of predecessor edges followed by a successor edge

$$ref(o_i, v_{o_i}, \overline{v_{o_i}}) \xrightarrow{t_1} v_{r_1} \xrightarrow{t_2} \ldots \xrightarrow{t_n} v_{r_n} \xrightarrow{\langle f,\epsilon,\epsilon \rangle} proj(ref, 3, v_r)$$

The successor edge represents a constraint with an empty object annotation. The resolution rule is applied if and only if every adjacent non-empty t_i, t_j match. Let $t = t_1 \oplus t_2 \ldots \oplus t_n$. It is easy to see that given $\beta(e_1, G^*)$, $\beta(e_2, G^*)$, etc., where e_i are the predecessor edges in the sequence, we have $o_{i,c_o} \in Pt(r_n^{c_{r_n}})$ for every $[c_o \times c_{r_n}]$ subsumed by t. After the application of the resolution rule (following possibly multiple applications of TRANS), we have $v_r \overset{\langle f,\epsilon,transpose_o(t)\rangle}{\longrightarrow} o_i$. Thus, we have to show that for every $[c_o \times c_{r_n}]$ subsumed by t we have $Pt(r^{c_{r_n}}) \subseteq Pt(\langle o_{i,c_o}, f\rangle)$ in G^*.

Constraint $v_{r_n} \overset{\langle f,\epsilon,\epsilon\rangle}{\rightarrow} proj(ref, 3, v_r)$ is created due to statement s: $\mathtt{r}_n\mathtt{.f=r}$. After applying the transfer function for s, for every $c \in C_{m(r_n,r)}$ and for every $o_{i,c_{o_i}} \in Pt(r_n^c)$ we have $Pt(r^c) \subseteq Pt(\langle o_{i,c_{o_i}}, f\rangle)$ in G^*. From $\beta'(C^A_{m(r_n,r)}, C)$ we have $c_{r_n} \in C_{m(r_n,r)}$, and from $\beta(A, G^*)$ we have $o_{i,c_o} \in Pt(r_n^{c_{r_n}})$. Therefore, $Pt(r^{c_{r_n}}) \subseteq Pt(\langle o_{i,c_o}, f\rangle)$ in G^*.

Consider application of closure rule VIRTUAL. We have

$$ref(o_i, v_{o_i}, \overline{v_{o_i}}) \rightarrow\overset{t}{\ldots}\rightarrow v_{r_0} \overset{\langle\epsilon,m,\epsilon\rangle}{\rightarrow} lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)$$

From $\beta(A, G^*)$ we have that for every $[c_o \times c_{r_0}]$ subsumed by t, $o_{i,c_o} \in Pt(r_0^{c_{r_0}})$. After the resolution we have

$$ref(o_i, v_{o_i}, \overline{v_{o_i}}) \overset{\langle c,\epsilon,t_1\rangle}{\longrightarrow} v_{this_n}$$

$$v_{r_i} \overset{t_2}{\rightarrow} v_{p_i} \qquad\qquad v_{ret} \overset{transpose_o(t_2)}{\longrightarrow} v_l$$

where $lam(\overline{v_{this_n}}, \overline{v_{p_1}}, \ldots, \overline{v_{p_k}}, v_{ret})$ is the lambda term corresponding to the run-time target of the call n. If $t = [C_o \times C_{r_0}]$, then $t_1 = [C_o \times o_i]$ and $t_2 = [C_{r_0} \times o_i]$; otherwise (when $t = \epsilon$) $t_1 = t_2 = [* \times o_i]$. After the resolution we also have $C_n^{A'} = C_n^A \cup \{o_i\}$. We have to show

- for every $[c_o \times o_i]$ subsumed by t_1, $o_{i,c_o} \in Pt(this_n^{o_i})$
- for every $[c_{r_0} \times o_i]$ subsumed by t_2, $Pt(r_j^{c_{r_0}}) \subseteq Pt(p_j^{o_i})$ and $Pt(ret^{o_i}) \subseteq Pt(l^{c_{r_0}})$
- $o_i \in C_n$

Constraint $v_{r_0} \overset{\langle\epsilon,m,\epsilon\rangle}{\rightarrow} lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)$ is created due to s: $\mathtt{l=r_0.m(r_1 \ldots r_n)}$. After applying the transfer function for s

- for every $c \in C_{m(r_0,r_j,l)}$ and every $o_{i,c_{o_i}} \in Pt(r_0^c)$, we have $o_{i,c_{o_i}} \in Pt(this_n^{o_i})$

- $Pt(r_j^c) \subseteq Pt(p_j^{o_i})$ and $Pt(ret^{o_i}) \subseteq Pt(l^c)$
- $C_n = C_n \cup \{o_i\}$

From $\beta'(C_m^A, C)$ we have $c_{r_0} \in C_m$, and from $\beta(A, G^*)$ we have $o_{i,c_o} \in Pt(r_0^{c_{r_0}})$. Therefore, $\beta'(A', C)$ and $\beta(A', G^*)$ hold. **Q.E.D.**

We can use the following optimization to prevent unnecessary multiple edges between two nodes in the constraint graph. Consider a call site `this.m(`$\mathtt{r}_1, \ldots \mathtt{r}_n$`)`, such that each r_i is a formal parameter of the method enclosing the statement. The pattern occurs frequently when invocation of superclass constructors is used(e.g., recall Figure 4.2). In this case, instead of using multiple constraints of the form $lam(\overline{v_{p_0}}, \overline{v_{p_1}}, \ldots, \overline{v_{p_n}}) \subseteq^{[o_i \times o_i]} lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_n}})$, for each reaching context o_i, the analysis can use a single constraint with empty annotation: $lam(\overline{v_{p_0}}, \overline{v_{p_1}}, \ldots, \overline{v_{p_n}}) \subseteq lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_n}})$. It is easy to see that substituting $[o_i \times o_i]$ with ϵ preserves precision, while the number of edges between each pair v_{r_j} and v_{p_j} may be reduced substantially.

We conclude this section with a brief discussion on the worst-case complexity of the analysis. Recall from Section 2.2.5, Chapter 2 that the worst-case complexity of the analysis is at most $O(n^3|\mathcal{S}|^2)$, where $\mathcal{S}$ is the size of the set of object annotations. Since $|\mathcal{S}|$ is $O(n^2)$, the worst-case complexity is at most $O(n^7)$.

4.4 Parameterized Object Sensitivity

In this section we define a parameterized framework for object-sensitive analysis. The framework encompasses a family of analyses that range from the least precise and least costly context-insensitive Andersen's analysis to the most precise and costly object-sensitive analysis described in Section 4.2.1.

The framework is parameterized in four dimensions. First, the analysis designer can select the degree of precision in the naming scheme for object names. This is done by defining a set of object allocation sites for which a more precise naming scheme should be used. The analysis uses multiple object names for the selected sites and single object names for all other sites. Second, the analysis designer can specify the set of reference variables for which multiple points-to sets should be maintained. The analysis replicates only these selected variables. Third, the analysis designer can select the degree of precision in the context naming scheme. This is achieved by defining a set of object contexts for which the analysis maintains copies of replicated objects and variables. The rest of the object contexts are treated *class-sensitively*, i.e., they are represented by their class, and the analysis maintains separate object names and points-to sets per class context instead of per object context. Fourth, the analysis designer can specify the set of virtual call sites that are analyzed context-sensitively; the rest of the virtual calls are treated context-insensitively.

The goal of the parameterization is to enhance the flexibility of the object-sensitive analysis. By varying the object naming scheme, the set of replicated variables, the degree of precision in the context naming, and the set of call sites analyzed context-sensitively, the analysis designer can control directly the size of the points-to graph and the cost of the analysis.

4.4.1 Object Naming

The parameterization for object names is based on set S^*. $S^* \subseteq S$ contains the object allocation sites for which the analysis designer wants to use the more precise naming scheme from Section 4.2.1 which allows multiple names for allocation site. The set of object names $O' \subseteq S \times (S \cup \{\epsilon\})$ is defined as follows. If $s_i \in S^*$ then the more precise object naming scheme from Section 4.2.1 is used during the analysis. For any other s_i, there is a single object name for the allocation site (similarly to Andersen's analysis).

4.4.2 Reference Variables

The parameterization for reference variables is based on a set $R^* \subseteq R$ which contains all variables that should be replicated during the analysis. For the boundary case $R^* = \emptyset$, there is no replication and analysis behavior is similar to Andersen's analysis. Function $map : R \times \mathcal{C} \rightarrow R'$ constructs R' as follows: if $r \in R^*$ is a local variable in method m, r is mapped to a "fresh" variable r^c for every context $c \in O'$ such that $c \in C_m$. Any other variable is mapped to itself. Thus, *map* replicates variables in R^* for all applicable contexts, and preserves variables not in R^* (i.e., $map(r, c) = r$ for any $r \notin R^*$).

The transfer functions from Figure 4.4 can be modified in a straightforward fashion in order to accommodate the changes needed for the parameterized analysis. For example, the transfer function for object creation becomes

$F(G, C, s_i : l = new\ C) =$

$$\begin{cases} G \cup \bigcup_{o_j \in C_m} \{(map(l, o_j), o_{i,j})\} & \text{if } s_i \in S^* \\ G \cup \bigcup_{o_j \in C_m} \{(map(l, o_j), o_{i,\epsilon})\} & \text{otherwise} \end{cases}$$

4.4.3 Context Naming (Class Sensitivity)

In certain cases a large number of objects of one class may be allocated (statically) in the program and the points-to graph may become too large, making the full-blown object-sensitive analysis infeasible. One way to control the cost of the analysis is to use a more coarse-grained context naming scheme by applying class sensitivity selectively. The parameterization for class sensitivity is based on a set $S^{**} \subseteq S$. If object context $o_i \in S^{**}$, it is represented separately and the analysis maintains context copies of $o_j \in S^*$ and $r \in R^*$ for each applicable o_i. Otherwise, the context o_i is represented by its class (more generally, it can be represented by a superclass of its class). Consider the example in Figure 4.2 and suppose that S^{**} is empty. Constructor `B.B` and method `B.m` are analyzed per class `B`; similarly, `C.C` and `C.m` are analyzed per class `C`. Constructor `A.A` is analyzed once in the context of `B`, and once in the context of `C`. In this case, the class-sensitive analysis is as precise as the object-sensitive analysis.

4.4.4 Call Sites

The parameterization for call sites is based on a set of instance method calls $Calls^*$. $Calls^*$ contains the call sites for which the analysis uses value propagation as described in Section 4.2.1. The rest of the calls are treated context-insensitively, meaning that parameter information is propagated to the callee from all possible contexts for the enclosing method; similarly return value information is propagated from the callee to all possible contexts for the caller. The transfer function for virtual calls is modified accordingly to accommodate this change. If $c_i \in Calls^*$ it is identical to the one in

```
   class X {...}
   class Y extends X {...}
   class A {
       X f;
1      void A(X x) { this.f = x; }
2      X get() { X x = this.f; return x;}
      }
   class B {
3      X m(A a) {
4        X x = a.get();
5        return x; }
       }
6  s1:X x = new X();
7  s2:Y y = new Y();
8  s3:A a1 = new A(x);
9  s4:A a2 = new A(y);
10 s5:B b1 = new B();
11 s6:B b2 = new B();
12     X xx = b1.m(a1);
13     X yy = b2.m(a2);
```

Figure 4.5: Context-insensitive calls.

Figure 4.4. Otherwise, it becomes

$$F(G, C, l = r_0.m(r_1, \ldots, r_n)) =$$

$$G \cup \bigcup_{o_i, o_j \in C_m} \{resolve(G, m, o_{k,l}, r_1^{o_i}, \ldots, r_n^{o_i}, l^{o_i}) \mid o_{k,l} \in Pt(G, r_0^{o_j})\}$$

Clearly, this function merges the points-to sets of the actual arguments for all possible reaching contexts of the caller and propagates the merged set to the corresponding formal of the callee; similarly the return values for all object contexts reaching the callee at that call site are merged and the points-to set of each context copy of the left-hand-side is updated with the merged set.

Example: Context-insensitive Call Sites

Consider the example program in Figure 4.5, which illustrates how imprecision can arise from treating a call site context-insensitively. After calls 12 and 4 we have $o_5 \in C_{B.m}$ and $o_3 \in C_{A.get}$, and therefore

$$\{\texttt{m.a}^{o_5},\ \texttt{m.x}^{o_5},\ \texttt{get.this}^{o_3},\ \texttt{get.x}^{o_3}\} \subseteq R'.$$

Similarly after calls 13 and 4 we have $o_6 \in C_{B.m}$ and $o_4 \in C_{A.get}$, and therefore

$$\{\texttt{m.a}^{o_6},\ \texttt{m.x}^{o_6},\ \texttt{get.this}^{o_4},\ \texttt{get.x}^{o_4}\} \subseteq R'.$$

If call site $4 \notin Calls^*$, it is processed using the modified transfer function. In this case we have $\texttt{get.x}^{o_3} \subseteq \texttt{m.x}^{o_5}$ and $\texttt{get.x}^{o_4} \subseteq \texttt{m.x}^{o_5}$ as well as $\texttt{get.x}^{o_3} \subseteq \texttt{m.x}^{o_6}$ and $\texttt{get.x}^{o_4} \subseteq \texttt{m.x}^{o_6}$ although o_4 is not a valid context for `get` when the context of `m` is o_5 and similarly o_3 is not a valid context for `get` when the context of `m` is o_6. As a result, the analysis erroneously infers that `xx` points to $o_{2,\epsilon}$ and `yy` points to $o_{1,\epsilon}$. If $4 \in Calls^*$ this imprecision is avoided because the caller context is distinguished; in this case, at call 4 the analysis precisely infers only $\texttt{get.x}^{o_3} \subseteq \texttt{m.x}^{o_5}$ and $\texttt{get.x}^{o_4} \subseteq \texttt{m.x}^{o_6}$.

4.5 Parameterized Object Sensitivity Using Annotated Constraints

The four dimensions of parameterization can be easily modeled by choosing appropriate annotations. Objects that are not replicated, i.e., their corresponding allocation site $s_i \notin S^*$, and reference variables $r \notin R^*$ are treated similarly to static variables. For such allocation sites and variables the analysis generates constraints using the special context ϵ. For example, the constraint for object creation becomes

$$\langle l = new\ o_i \rangle \Rightarrow \begin{cases} \{ref(o_i, v_{o_i}, \overline{v_{o_i}}) \subseteq v_l\} & \text{if } s_i \in S^* \\ \{ref(o_i, v_{o_i}, \overline{v_{o_i}}) \subseteq^{[\epsilon \times *]} v_l\} & \text{otherwise} \end{cases}$$

To account for merged caller contexts, rule VIRTUAL is modified as follows. Consider

$$ref(o_i, v_{o_i}, \overline{v_{o_i}}) \subseteq_c^{t_1} v \qquad v \subseteq_m^{t_2} lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)$$

such that t_1 and t_2 match, and c and m match. Let $concat_o(t_1, t_2) = t$. Let c_i be the call site that corresponds to lambda term $lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)$. If $c_i \in Calls^*$, the rule from Section 4.3 is applied. Otherwise (if $c_i \notin Calls^*$) we have

$$\begin{aligned} ref(o_i, v_{o_i}, \overline{v_{o_i}}) \quad &\subseteq^{[C_r \times o_i]} \quad v_{p_0} \\ lam(\overline{v_{p_0}}, \overline{v_{p_1}}, \ldots, \overline{v_{p_k}}, v_{ret}) \quad &\subseteq^{[o_i \times *]} \quad lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l) \end{aligned}$$

where $C_r = *$ if t is empty, and $C_r = C_r'$ if t is non-empty and equals $[C_r' \times C_l']$. To accommodate the parameterization for class sensitivity, when creating constraints for virtual calls, if $o_i \notin S^{**}$ the annotation uses the representative class for o_i instead of o_i.

We conclude the section with a brief discussion on the effects of the dimensions of parameterization on the complexity of the analysis. In particular, we consider the

```
input    Stmt: set of statements    map: R × C → R'
         Methods: set of methods    Pt: R' → P(O')
output   Mod: Stmt × C → P(O')
declare  MMod: Methods × C → P(O')
[1]   foreach indirect write s: p.f = q ∈ Stmt do
[2]      foreach context c in which s appears do
[3]         Mod(s,c) := {o | o ∈ Pt(map(p,c))}
[4]         add Mod(s,c) to MMod(EnclMethod(s),c)
[5]   while changes occur in Mod or MMod do
[6]      foreach virtual call s: l = r.m(...) ∈ Stmt do
[7]         foreach context c in which s appears do
[8]            foreach object o ∈ Pt(map(r,c)) do
[9]               Mod(s,c) := Mod(s,c) ∪
                  {o' | o' ∈ MMod(target(o,m),o)}
[10]           add Mod(s,c) to MMod(EnclMethod(s),c)
[11]     foreach static call s: l = C.m(...) ∈ Stmt do
[12]        foreach context c in which s appears do
[13]           Mod(s,c) := Mod(s,c) ∪ MMod(m,c)
[14]           add Mod(s,c) to MMod(EnclMethod(s),c)
```

Figure 4.6: Object-sensitive MOD analysis. $\mathcal{P}(X)$ denotes the power set of X.

case where R^* consists of all variables in instance methods and constructors (i.e., such variables are replicated) and we choose to apply class sensitivity. Let A be a class, and k be a constant. Suppose that we apply class sensitivity in the following way: if there are more than k object allocation sites in the program that create objects of class A or one of the subclasses of A, such object contexts are treated class-sensitively (i.e., the object context is substituted with A). Consider two variables X enclosed by method m, and Y enclosed by method n. There are at most k possible contexts for X (clearly, if m is declared in class C and there are more than k allocation sites that create objects of class C or one of the subclasses of C, these object contexts would be collapsed and there would be only one possible context for X - C). Similarly, there are at most k possible contexts for Y. Thus, there are at most $k * k$ pairs of contexts, and thus at most k^2 edges between each pair of variables. Therefore, in this case, the complexity of the analysis is at most $O(k^4 n^3)$, or $O(n^3)$.

4.6 Side-Effect Analysis

In this section we present a MOD analysis based on object-sensitive points-to analysis. Our MOD algorithm computes a set of modified objects $Mod(s, c) \subseteq O'$ for each statement s and for each context c of the method containing s. The algorithm is shown in Figure 4.6. $Pt(x)$ denotes the set of objects pointed to by context copy x. We say that statement s *appears* in context c if $c \in C_m$. $MMod(m, c)$ stores the sets of objects modified by each contextual version of a method (i.e., objects that are modified when m is invoked with context c). For virtual calls (lines 6–10) the target methods are determined for each receiver object o in context c, based on the class of o and the compile-time target m. In addition, object o determines which set of modified objects associated with the target method will be added to the Mod set at line 9.

Example. Consider the example in Figure 4.2. MOD analysis based on context-insensitive points-to analysis erroneously determines that the *Mod* sets for statements 1, 2, and 5 are $\{o_3, o_4\}$. Consider a MOD analysis based on the object-sensitive points-to analysis from Section 4.2.1. The statement at line 1 appears in two contexts: o_3 and o_4. Therefore, $MMod(A.A, o_3)$ is $\{o_3\}$ and $MMod(A.A, o_4)$ is $\{o_4\}$. The receiver for the call statement at line 2 is o_3; therefore the MOD analysis infers that $Mod(2, o_3)$ is $\{o_3\}$. Similarly $Mod(5, o_4)$ is $\{o_4\}$.

4.7 Empirical Results

We chose to implement two particular object-sensitive points-to analyses. In both cases, R^* contains all non-static reference variables, and S^* contains all allocation sites. We choose to apply class sensitivity only to programs that allocate (statically) a large number of objects (more than 500 allocation sites of non-library classes). [10] The two analyses differ in the fourth dimension of parameterization: for the first analysis we chose to merge the contexts of the callers for calls not made through implicit parameter `this`, while calls through `this` are treated object-sensitively. For the second analysis all calls were

[10] Given this condition, class sensitivity was applied to `jess`, `soot`, and `javac`. For those programs, the degree of class sensitivity was determined using the following heuristic: if there were more than 50 instances of class A or its subclasses, the corresponding object contexts were represented by A.

Program	*And*		*CI-ObjSens*		*ObjSens*	
	Time (sec)	Mem (Mb)	Time (sec)	Mem (Mb)	Time (sec)	Mem (Mb)
proxy	4.8	35.1	5.3	34.8	4.1	35.1
compress	8.3	39.6	10.1	40.1	12.6	43.7
db	9.2	40.6	10.6	42.5	14.3	42.6
jb	6.0	36.7	5.8	36.9	6.5	37.1
echo	18.7	49.2	44.9	66.2	51.8	69.4
raytrace	7.8	42.2	10.8	46.1	12.7	46.8
mtrt	9.4	42.1	11.3	46.2	12.8	46.8
jtar	16.8	50.3	24.4	58.9	28.1	57.7
jlex	6.7	39.8	7.3	40.6	7.5	40.6
javacup	23.2	55.8	21.2	58.5	21.5	58.8
rabbit	9.1	46.2	11.7	45.6	12.8	47.9
jack	28.7	54.8	24.9	56.7	62.9	65.4
jflex	28.5	63.5	30.3	66.4	29.9	67.4
jess	35.8	59.4	87.5	79.6	-	-
mpegaudio	11.6	44.0	10.4	48.4	13.7	50.1
jjtree	8.6	46.8	32.1	64.4	46.2	65.8
sablecc	34.5	78.5	51.2	75.3	404.1	102.6
javac	100.5	110.0	168.5	129.0	180.5	133.4
creature	64.3	94.3	105.5	124.8	109.2	126.5
mindterm	37.2	78.5	51.5	90.5	61.6	93.9
soot	139.4	117.8	115.9	117.9	244.6	144.0
muffin	120.7	133.9	115.1	149.7	170.1	165.1
javacc	99.6	96.6	93.4	101.9	101.8	102.0

Table 4.1: Running time and memory usage of the analyses.

treated object-sensitively. Given that many calls in Java programs are made through `this`, treating such calls object-sensitively can improve analysis precision substantially over context insensitive analysis (recall the example in Figure 4.2). These object-sensitive analyses, which we denote by *CI-ObjSens* and *ObjSens* respectively, were compared with Andersen's context-insensitive analysis (denoted by *And*).

We augmented the constraint engine with object annotations. The instantiated constraint engine was able to represent and solve systems of constraints with field, method, and object annotations (as described in Section 2.2.6, Chapter 2). We use the Soot framework (`www.sable.mcgill.ca`) to process Java bytecode and build a typed intermediate representation [71]. All experiments were performed on a 900MHz Sun Fire-280R shared machine with 4Gb physical memory. We used the same set of 23 programs, described in Chapter 3, Section 3.4. The reported times are the median values out of three runs.

4.7.1 Analysis Cost

Table 4.1 shows running times and memory consumption for *And*, *CI-ObjSens* and *ObjSens*. The empirical results demonstrate that *CI-ObjSens* object-sensitive analysis is practical in terms of running time and memory consumption. For all but one program, *CI-ObjSens* runs in less than two minutes and for the majority of the programs its cost is comparable to the cost of Andersen's analysis. For most programs, the more expensive analysis *ObjSens* has comparable cost to *And* as well. However, for some programs its cost is significantly bigger; for one of the larger programs *ObjSens* did not terminate in 500 seconds, the acceptable upper bound that we set for the analysis to complete. Given that in terms of precision, *CI-ObjSens* and *ObjSens* performed virtually the same (see Section 4.7.2), our results suggest that the use of *ObjSens* may not be justified.

These results show that the annotations are an efficient representation of object context and the annotation overhead is reasonable; the overhead is easily offset by the improved analysis precision which results in smaller points-to sets which means less work and reduced memory consumption for the analysis.

The practicality of *CI-ObjSens* makes it a realistic candidate for use in advanced static optimizing compilers and software tools for Java. These results combined with the results presented in Chapter 3 show that the annotations are an efficient mechanism for implementing different dimensions of flow analysis precision.

4.7.2 Analysis Precision

Call Graph Construction and Virtual Call Resolution

One fundamental application of points-to analysis information is virtual call resolution and call graph construction. The points-to solution can be used to determine the target methods of a virtual call by examining the classes of possible receiver objects. This information is necessary to construct the call graph of the program which is necessary for any interprocedural analysis and optimization. If a given virtual call can have a single target method then the call can be devirtualized and the overhead of dynamic dispatch can be eliminated. In addition, virtual call resolution allows better inlining.

Program	*CI-ObjSens*		*ObjSens*	
	Resolved calls	Removed targets	Resolved calls	Removed targets
proxy	10%	3%	10%	3%
compress	23%	17%	23%	17%
db	19%	18%	19%	18%
jb	68%	10%	68%	10%
echo	12%	16%	13%	16%
raytrace	21%	17%	21%	17%
mtrt	21%	17%	21%	17%
jtar	43%	8%	43%	8%
jlex	53%	6%	53%	6%
javacup	32%	7%	32%	7%
rabbit	35%	12%	35%	12%
jack	5%	16%	5%	16%
jflex	15%	4%	15%	5%
jess	20%	21%	-	-
mpegaudio	24%	21%	24%	21%
jjtree	63%	7%	63%	7%
sablecc	52%	221%	52%	221%
javac	7%	11%	7%	11%
creature	19%	5%	19%	5%
mindterm	6%	10%	8%	10%
soot	10%	9%	11%	7%
muffin	4%	14%	4%	14%
javacc	16%	6%	16%	6%
Average	26%	21%	26%	21%

Table 4.2: Improvements over context-insensitive analysis. Resolved calls shows the increase in the number of resolved call sites. Removed targets shows the reduction in the number of target methods.

To determine the improvements from object-sensitive analysis in the number of resolved calls we considered the set V of CHA-unresolved calls which appear in methods reachable by *ObjSens*. We computed the number of sites from V that were resolved to a single target method, according to *And* and according to *CI-ObjSens* and *ObjSens*. The improvement in the number of resolved call sites for *CI-ObjSens* over *And* is shown in the second column of Table 4.2 and the improvement of *ObjSens* over *And* is shown in the fourth column of Table 4.2. On average, both object-sensitive analyses resolve 26% more sites than *And*. This increased precision allows better removal of redundant run-time virtual dispatch and enables additional method inlining.

We also computed the sum (over all sites in V) of the number of target methods according to *And*, as well as the corresponding sum according to *CI-ObjSens* and *ObjSens*.

Program	*And*	*CI-ObjSens*	*ObjSens*
proxy	24%	67%	67%
compress	24%	71%	71%
db	24%	74%	75%
jb	12%	44%	44%
echo	18%	43%	43%
raytrace	23%	71%	71%
mtrt	23%	71%	71%
jtar	17%	44%	44%
jlex	22%	78%	78%
javacup	9%	85%	86%
rabbit	23%	68%	68%
jack	15%	63%	63%
jflex	4%	62%	62%
jess	20%	73%	-
mpegaudio	23%	68%	70%
jjtree	65%	80%	80%
sablecc	33%	47%	47%
javac	12%	36%	36%
creature	18%	33%	33%
mindterm	25%	47%	47%
soot	17%	25%	26%
muffin	13%	35%	35%
javacc	6%	59%	59%
Average	20%	58%	59%

Table 4.3: Improvements in the number of resolved downcasts.

The reduction in the total number of target methods (i.e., call edges removed at call sites) is shown in the third and fifth columns of Table 4.2 respectively. On average, the object-sensitive analysis remove 21% of the target methods determined by *And*. This improved precision is beneficial for reducing the cost and improving the precision of subsequent interprocedural analyses.

Proving Downcast Safety

The points-to analysis information can be used to determine the set of possible classes of run-time objects corresponding to a variable which is the target of a downcast statement. If all of these classes are direct or transitive subclasses of the class specified at the downcast statement, the cast check operation can be eliminated. This optimization removes the run-time overhead of cast checks [25]; in addition, statically proving that a given cast check cannot throw `ClassCastException` may allow more efficient application

Program	*And*			*CI-ObjSens*			*ObjSens*		
	1-3	4-9	≥10	1-3	4-9	≥10	1-3	4-9	≥10
proxy	19%	6%	75%	76%	14%	10%	76%	14%	10%
compress	23%	4%	73%	68%	9%	23%	68%	9%	23%
db	20%	4%	76%	66%	9%	25%	66%	9%	25%
jb	16%	4%	80%	73%	15%	12%	74%	14%	12%
echo	25%	6%	69%	63%	11%	26%	63%	11%	26%
raytrace	23%	5%	72%	67%	9%	24%	67%	9%	24%
mtrt	23%	5%	72%	67%	9%	24%	67%	9%	24%
jtar	19%	7%	74%	62%	13%	25%	62%	13%	25%
jlex	18%	3%	79%	57%	33%	10%	57%	33%	10%
javacup	14%	3%	83%	54%	37%	9%	54%	37%	9%
rabbit	20%	4%	76%	48%	35%	17%	48%	35%	17%
jack	17%	3%	80%	54%	8%	38%	54%	8%	38%
jflex	18%	4%	78%	64%	23%	13%	64%	23%	13%
jess	16%	5%	79%	63%	8%	29%	-	-	-
mpegaudio	23%	4%	73%	67%	9%	24%	67%	9%	24%
jjtree	8%	2%	90%	32%	26%	42%	32%	26%	42%
sablecc	20%	3%	77%	67%	13%	20%	67%	13%	20%
javac	14%	3%	83%	38%	20%	42%	32%	20%	42%
creature	19%	2%	79%	55%	13%	32%	55%	13%	32%
mindterm	20%	7%	73%	57%	13%	30%	57%	13%	30%
soot	21%	6%	73%	46%	14%	40%	47%	13%	40%
muffin	16%	4%	80%	45%	6%	49%	45%	6%	49%
javacc	10%	1%	89%	29%	49%	22%	29%	49%	22%
Average	18%	4%	78%	57%	17%	36%	57%	17%	36%

Table 4.4: Number of modified objects for program statements. Each column shows the percentage of statements whose number of modified objects is in the corresponding range.

of standard optimizations such as code motion and redundancy elimination. Proving downcast safety is beneficial for the purposes of program understanding as well.

Table 4.3 shows the percentage of cast checks in methods reachable by *ObjSens* that can be eliminated. On average, the object-sensitive analyses resolve nearly 60% of the cast checks while the context-insensitive analysis resolves only 20%. Currently, Java lacks parametric polymorphism and downcasts are heavily used in Java programs for handling containers and maps (e.g., `Vector` and `Hashtable`). The object-sensitive analyses handle precisely flow through container objects, and as a result they are able to resolve a large percentage of downcasts.

Side-Effect Analysis

Using the MOD algorithm described in Section 4.6, we performed measurements for the object-sensitive analyses and *And* in order to estimate the impact of the analyses on MOD analysis. More precise points-to analyses produce a smaller number of modified objects per statement.

We considered all methods that *ObjSens* determined to be reachable. For all statements in such methods, we computed (i) *Mod* sets according to the algorithm from Figure 4.6, and (ii) *Mod* sets using Andersen's analysis and a corresponding context-insensitive version of the algorithm from Figure 4.6. In order to compare the output of the two analyses, for each statement we merged the *ObjSens*-based *Mod* sets for different contexts to obtain a single *Mod* set. For example, the aggregate *Mod* set for line 1 in Figure 4.2 is $\{o_3, o_4\}$, which is the union of $Mod(1, o_3)$ and $Mod(1, o_4)$. Similarly, multiple object names for the same allocation site were treated as the same name.

Table 4.4 shows the distribution of the number of modified objects for the three analyses. Each column corresponds to a specific range of numbers. For example, the first column corresponds to statements that may modify one, two or three objects, while the last column corresponds to statements that may modify at least 10 objects. Each column shows what percentage of statements (counting only statements that modify at least one object) corresponds to the particular range of numbers of modified objects.

The measurements in Table 4.4 show that object sensitivity significantly improves analysis precision. For MOD analysis based on *CI-ObjSens* and *ObjSens*, on average 57% of the statements modify at most three objects. In contrast, for MOD analysis based on *And* this percentage is 18%. It is also significant to note that for *And* nearly 80% of the statements modify at least 10 objects. This indicates substantial imprecision, which can be reduced significantly by using *CI-ObjSens* or *ObjSens*.

The above empirical results show that object-sensitive analysis is a promising candidate for producing useful side-effect information. Such relatively precise information is important for (i) implementing advanced optimizations in aggressive optimizing compilers, and (ii) improving the precision of software productivity tools, with the corresponding reduction in human time and effort spent on software understanding, restructuring, and testing.

Chapter 5

Analysis Applications

Object-oriented systems are characterized by complex interclass interactions. The goal of *integration testing* is to reveal faults that cause some interclass interactions to fail. One fundamental problem in integration testing is how to define the test order (i.e., should class A be tested before or after class B is tested). The goal of *integration coverage analysis* is to show that sufficiently many interclass interactions are covered during testing. Coverage analyzers need to determine which statements trigger interclass dependences and which interclass dependences are triggered at a given statement (i.e., exactly which two classes are involved). The goal of *regression testing* is to show that after a change is made, the program still satisfies its requirements. Regression testing must address the following problems: (i) determining the impact of a change on a given class (i.e., the set of classes affected by the change) and (ii) defining the regression test order (i.e., the order in which affected classes need to be retested).

The *Object Relation Diagram (ORD)* [38] is a model of interclass dependences which can be used to address these testing applications. The ORD can be used to define efficient test order for integration testing. It can be used in coverage analysis to determine pairs of interacting classes. In addition, the ORD can be used in regression testing to find the set of classes affected by a change and to define efficient regression test order.

The ORD of a program P is a directed graph in which nodes represent program classes and edges represent dependences between these classes. There are three kinds of edges. An *inheritance edge* from B to A represents that B depends on A because B is a subclass of A. An *aggregation edge* from B to A represents that B depends on A because instances of class A may be contained in instances of class B. An *association edge* from B to A represents associations between objects of class B and objects of class

A due to method calls or field accesses. (Several diagrams are shown in Figure 5.2 in Section 5.1.)

One disadvantage of the ORD is that there is at most one edge of each kind between two classes. Therefore, coverage analyzers cannot use the ORD to determine which specific statement triggers the dependence. We define the *Extended ORD (ExtORD)* to address this problem. The ExtORD allows multiple edges of each kind, one for each statement that triggers this kind of dependence; therefore, test coverage of such statements can be distinguished.

Imprecise ORDs and ExtORDs contain spurious dependence edges, representing interclass dependences that are impossible and do not correspond to any actual program execution. In integration testing, spurious edges may lead to dependence cycles in the ORD, which complicate the task of defining a test order. In integration coverage analysis, spurious dependence edges result in time spent on trying to execute code in a way which triggers impossible dependences. In regression testing imprecision leads to two problems: (i) when the set of classes determined to be affected by a change is too imprecise, time will be wasted on retesting unaffected classes and (ii) cycles in the ORD complicate the definition of a regression test order. In these cases, as well as for other ORD uses such as program understanding and reverse engineering, significant time and effort could be saved if the ORD and the ExtORD are more precise (i.e., contain fewer spurious edges).

Because of the wide range of applications of these dependence diagrams, it is important to investigate approaches for construction of relatively precise ORDs and ExtORDs. In order to construct these diagrams, it is necessary to have information about the classes of all objects that certain variables may refer to at run time. This information can be obtained by using *class analysis*, which is a popular form of static program analysis for object-oriented languages. The goal of class analysis is to compute for each variable r the set $Cs(r)$ of all classes such that an object of class $C \in Cs(r)$ may be bound to r at run time. There are many class analyses with different tradeoffs between cost and precision, developed primarily in the context of optimizing compilers for object-oriented programming languages. The precision of the dependence diagrams directly depends on the precision of the underlying class analysis which is used during ORD/ExtORD construction.

In this chapter we define a generalized algorithm for ORD construction which is parameterized by class analysis. By varying the underlying class analysis, the algorithm allows the user to build ORDs with differing degrees of precision. For presentation purposes we use Java programs to demonstrate our approach, but the methodology can be used with minor modifications with other object-oriented programming languages.

Previous work used the structure of the program class hierarchy to determine $Cs(r)$ in order to construct the ORD. In our experiments, we compare this simple form of class analysis with one more precise class analyses based on the field-sensitive points-to analysis presented in Chapter 3. On a set of 23 realistic Java programs, our results show that this more precise analysis significantly improves the precision of the ORD and the ExtORD, compared to using the structure of the class hierarchy.

In integration testing, these reductions may result in substantial savings in time and effort. When using coverage analysis tools, substantially less time will be spent trying to exercise impossible interclass dependences. In regression testing, the improved precision leads to (i) smaller number of classes selected for retesting and (ii) less time spent on determining a retest order. Furthermore, the significantly improved precision is beneficial for other uses of the ORD/ExtORD (e.g., for program understanding and reengineering).

5.1 The Object Relation Diagram

5.1.1 Applications of Object Relation Diagrams

The ORD can be used in integration testing, integration coverage analysis, regression testing, impact analysis, program understanding, and reverse engineering. Once constructed, the ORD can be reused by various clients at no additional cost. In this section we briefly discuss several specific client applications of the ORD.

Integration Testing One goal of integration testing is to reveal faults that are triggered by the interactions between classes. Usually, integration testing proceeds in stages. At each stage there is a target set of classes under test. Classes not included in the target set and used by some of the classes in the target set need to be simulated by *stubs*. The stubs simulate the class behavior for the given context, which is usually a small subset of the entire class behavior. One important problem in integration testing is how to determine the order in which classes are tested (i.e., should class A be tested before or after class B is tested).

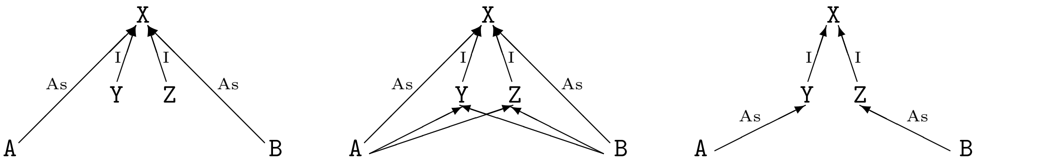
Bottom-up integration testing strategies [10, 38, 39] aim at minimizing the number of stubs, because stub construction requires significant time and effort. In this case, the test order can be derived by bottom-up traversal of the ORD. Clearly, no stub is required for a class that is tested before the classes that depend on it. Intuitively, the independent classes are tested first, then the classes that depend on them, and so on.

Coverage Analysis The goal of coverage analysis is to evaluate the quality of a given test suite by measuring the coverage of program code and of certain aspects of the behavior of that code. Integration testing focuses on faults due to complex interclass dependences. Thus, it is important to ensure that the test suite covers interclass interactions. For example, one coverage requirement is that all interclass method calls (i.e., calls from one class to a method in another class) should be exercised [10]. Another more strict requirement states that every possible interclass dependence should be covered [67]. The Extended ORD (described in Section 5.1.2) is a version of the ORD that can be used by coverage analyzers to determine the set of statements that trigger interclass dependencies and therefore need to be exercised in order to satisfy coverage requirements.

Regression Testing The goal of regression testing is to ensure that when a change is made to a program, the program still satisfies its requirements. Because of complex dependences between classes, when a change is made to a class, this change usually affects other classes in the program. Each of these affected classes needs to be retested. Two fundamental problems in regression testing are (i) how to identify the classes affected by a change and (ii) how to efficiently perform retesting of these affected classes. The ORD can be used in regression testing to identify the set of affected classes. All classes reachable backwards in the ORD from the changed classes are potentially affected by the

```
   class X { void n() {...} }
   class Y extends X { void n() {...} }
   class Z extends X { void n() {...} }
   class A {
       X f;
2      A(X xa) { this.f = xa; ...  }
       void m() {
3          X xa = this.f;
4          xa.n(); } }
   class B {
       X g;
5      B(X xb) { this.g = xb; ...  }
       void m() {
6          X xb = this.g;
7          xb.n(); } }
8  s1: Y y = new Y();
9  s2: Z z = new Z();
10 s3: A a = new A(y);
11 s4: B b = new B(z);
12     a.m();
13     b.m();
```

Figure 5.1: Sample set of statements.



(a) Static dependences (b) Transitive subclasses (c) Precise run-time dependences

Figure 5.2: Object Relation Diagrams corresponding to different methods of construction.

change. In addition, the ORD can be used to determine a test order which minimizes the necessary stubs. Such a test order leads to an efficient retesting strategy, because stub construction requires significant effort. Similarly to determining the test order for bottom-up integration testing, the regression test order can be derived by bottom-up traversal of the ORD (i.e., class A is retested before the classes that depend on it; when the classes that depend on A are retested, stubs are not required for A because it has already been retested).

5.1.2 The ORD by Kung et al.

Kung et al. [38] define the Object Relation Diagram using three kinds of dependence edges. For the rest of the chapter we use the notation $(\langle B, l\rangle, A)$ to denote an edge labeled l from B to A.

Inheritance There is an edge labeled I from class B to class A if and only if B is a direct subclass of A.

Aggregation There is an edge $(\langle B, Ag\rangle, A)$ if and only if statement `this.f = new A(..)` appears in a constructor defined in class B.[1]

Association There is an edge $(\langle B, As\rangle, A)$ if and only if one of the following is true:

- `T m(...,A r,...) {...}` is a definition of a method or a constructor in class B. We refer to such an association as a *parameter association.*
- Field access `r.f` appears in a statement contained in a method or a constructor defined in class B and the declared type of reference variable r is A ($r \neq$ `this`). We refer to such an association as a *field access association.*
- Method invocation `r.m(...)` appears in a statement contained in a method or a constructor in class B and the declared type of r is A ($r \neq$ `this`). We refer to such an association as a *method call association.*

Figure 5.2(a) shows the ORD for the program in Figure 5.1. Clearly, this diagram does not reflect dependences due to dynamic bindings of polymorphic variables. For example, class A depends on class Y because at run time an object of class Y is bound to polymorphic variable `xa` at line 4. However, this dependence is not shown in the ORD from Figure 5.2(a).

The *class firewall for class C*, denoted by $CFW(C)$, is defined as the set of classes reachable from the node corresponding to class C in the transpose of the ORD [38, 39]. Intuitively, the class firewall contains the classes that may be affected when C is modified, and thus should be retested (assuming that the ORD for the program is not modified).

---

[1] According to [38] aggregation can be (i) static, due to encapsulated non-pointer fields and (ii) dynamic, due to pointer fields initialized within constructors. Since all fields in Java are references (i.e., pointers to objects), static aggregation is not possible and we simplify the definition accordingly.

For example, based on the ORD in Figure 5.2(a), $CFW(X) = \{X, Y, Z, A, B\}$ and $CFW(Y) = \{Y\}$.

In this chapter we define the *Extended Object Relation Diagram (ExtORD)* which makes the dependence diagram more informative and more suitable for use in code coverage analysis tools. The ExtORD contains *annotated* association and aggregation edges. There is an annotated aggregation $Ag\!:\!s_i$ if and only if s_i is an object creation statement which triggers an aggregation. There is an annotated association $As\!:\!s_i$ if and only if s_i is a program statement which triggers a field access association or a method call association. There is a non-annotated association As which denotes parameter association. Thus, the ExtORD may contain multiple aggregation and multiple association edges between two nodes, because it makes explicit which program statements trigger the dependences. For example, association edge $(\langle A, As\!:\!4\rangle, X)$ indicates that because of the virtual call at line 4 in Figure 5.1, there is an association between class A and class X. It is easy to see how coverage analyzers can make use of the Extended Object Relation Diagram. Since the ExtORD identifies the interclass dependences together with the statements that create them, a high-quality test suite should exercise these statements and should achieve sufficient coverage of the dependencies triggered by the statements.

5.1.3 The ORD by Labiche et al.

Recall from Section 5.1.2 that the ORD from [38] does not reflect the dependences that arise due to possible dynamic bindings of polymorphic variables. In order to correct this problem, Labiche et al. [39] propose to augment the ORD from [38] with additional edges representing dynamic relationships. If there is an edge labeled As from A to B, there are additional edges with the same label from A to all subclasses of B (including transitive subclasses). For the example in Figure 5.1, there are additional association edges from A and B to both Y and Z. Figure 5.2(b) shows how the diagram in Figure 5.2(a) is augmented with these additional edges. This approach is equivalent to modifying the three rules for association inference from Section 5.1.2 to take into account the possible dynamic bindings of polymorphic variables by considering the structure of the class hierarchy (i.e., by considering all transitive subclasses of the targets of ORD association

```
  class X { void n() {...} }
  class Y extends X {
1     Y(X xy) { xy.n(); ...  } ...  }
  class Z extends X {
2     Z(X xz) { xz.n(); ...  } ...  }
3  s1: X x = new X();
4  s2: Y y = new Y(x);
5  s3: Z z = new Z(y);
```

Figure 5.3: Sample program.

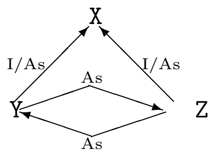
edges).

The ExtORD can be augmented in a similar fashion. If there is an edge labeled $As\!:\!s_i$ from A to B, there are edges labeled $As : s_i$ from A to all subclasses of B (including transitive subclasses). For the set of statements in Figure 5.1, there are additional edges $(\langle A, As : 4\rangle, Y)$ and $(\langle A, As : 4\rangle, Z)$. In order to achieve good coverage of interclass dependences, it may not be enough to exercise a statement once. Due to polymorphism, certain statements (e.g., the method call at line 4 in Figure 5.1) need to be exercised several times to achieve coverage for all dynamic interclass dependences that may result from such statements.

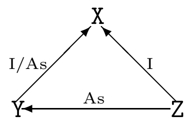
5.1.4 The Disadvantages of Imprecise Analysis

Clearly, the ORD computed by Kung et al. [38] omits dependences and may result in incomplete testing and retesting. On the other hand, the ORD computed by considering the structure of the class hierarchy could be overly conservative. This imprecision results from imprecise analysis of the possible dynamic bindings of polymorphic variables. In Figure 5.1 class A does not depend on class Z because at run time `A.A.xa` at line 2 and `A.m.xa` at line 4 can reference only objects of class Y. The same kind of spurious dependence occurs between class B and class Y. The exact ORD corresponding to this set of statements is shown in Figure 5.2(c).

Analysis imprecision results in spurious dependences and can impair the usefulness of tools that employ the ORD or the ExtORD. When the ORD is traversed bottom-up to choose integration test order or regression test order, imprecision can lead to *dependence cycles*. There are two problems with dependence cycles: (i) complex analysis



(a) Transitive subclasses



(b) Precise run-time dependences

Figure 5.4: ORDs corresponding to Figure 5.3.

and additional work are required to break the cycle [37, 67] and (ii) cycle breaking requires stub construction. Consider the ORDs in Figure 5.4 which correspond to the set of statements in Figure 5.3. The diagram in Figure 5.4(a) is constructed based on the structure of the class hierarchy, and contains a dependence cycle between Y and Z.[2] This cycle is caused by a spurious dependence edge from Y to Z which is due to the imprecise analysis of the possible dynamic bindings of `xy`—according to the class hierarchy, `xy` may refer to instances of X, Y, or Z. Therefore, the test order cannot be determined without cycle breaking. In contrast, the diagram in Figure 5.4(b) clearly shows that the test order should be X, Y, Z.

In coverage analysis, imprecision can lead to incorrect reports of inadequate coverage. For example, if the structure of the class hierarchy is used to compute the ExtORD for the statements in Figure 5.1, the following three association edges would correspond to the method call at line 4:

$$(\langle A, As\!:\!4\rangle, X) \qquad (\langle A, As\!:\!4\rangle, Y) \qquad (\langle A, As\!:\!4\rangle, Z)$$

A coverage analyzer will always report that the first and the third edge are not covered. Therefore, the user will attempt to exercise the call with receivers from classes X and Z and will spend valuable time determining that variable `xa` cannot refer to objects of any other class except Y.

In regression testing, an imprecise ORD can lead to large class firewalls, and thus code not affected by the change may be selected for retesting.

Imprecision in the ORD results in waste of time and effort for activities such as breaking of spurious dependence cycles, trying to execute a statement in a manner that

[2] In Figure 5.4(a) there are self-loop association edges at Y and Z; for simplicity, we omit them from the picture. Clearly, self-loop edges can be ignored when the ORD is used for test order definition.

triggers nonexistent interclass dependences, and retesting parts of the code that are not affected by a change. Therefore, it is important to investigate approaches for constructing more precise ORDs and ExtORDs.

5.2 Class Analysis for ORD Construction

Recall that information about the dynamic bindings of polymorphic variables is necessary during ORD construction. In order to determine all classes for which field access `p.f` triggers field associations with the enclosing class, one needs to find the possible dynamic bindings of reference variable `p` (i.e., the classes of all objects that `p` may refer to at run time). Similarly, for method call associations one needs the set of all possible classes of the receiver object; for parameter associations one needs the set of all classes of objects that can be bound to formals.

This information can be obtained by using *class analysis*, which is a popular form of static program analysis originally developed in the context of optimizing compilers. Class analysis computes a set of classes for each program variable r; this set approximates the classes of all run-time objects that may be bound to r. There is a wide variety of existing class analyses with various degrees of cost and precision [49, 1, 2, 50, 20, 7, 30, 19, 13, 52, 57, 64, 66, 69, 43, 55, 29, 46]. These analyses can be used for ORD and ExtORD construction. More precise underlying class analyses result in fewer spurious dependences and therefore more precise ORDs and ExtORDs. For the set of statements in Figure 5.3, the set of possible classes of all objects that may be bound to variable `xy` at line 1 is $\{X\}$. The set $\{X, Y, Z\}$ computed by examining the class hierarchy is a valid approximation because it includes the only possible run-time class X, but it is imprecise because it also includes Y and Z.

In Section 5.2.1 we present a general algorithm for ORD construction, parameterized by a class analysis. Section 5.2.2 discusses *Class Hierarchy Analysis* (*CHA*) which is the simplest form of class analysis [18]. Section 5.2.3 discusses one class analysis that is more precise than *CHA*.

input *Stmts*: set of statements
Methods: set of methods
Cs: $Vars \rightarrow Powerset(Classes)$
output *ORD*
foreach direct subclass X of class C **do**
  $ORD := ORD \cup \{(\langle X, I\rangle, C)\}$
foreach $s\colon this.f = new\ C \in Stmts$ in a constructor **do**
  $ORD := ORD \cup \{(\langle EnCl(s), Ag\rangle, C)\}$
foreach $s \in Stmts$ containing $p.f$ **do**
  if $p \neq$ `this` **do**
    $ORD := ORD \cup \{(\langle EnCl(s), As\rangle, C) \mid C \in Cs(p)\}$
foreach virtual call $s\colon l = p.m(...) \in Stmts$ **do**
  if $p \neq$ `this` **do**
    $ORD := ORD \cup \{(\langle EnCl(s), As\rangle, C) \mid C \in Cs(p)\}$
foreach method $m \in Methods$ **do**
  foreach explicit formal parameter p of m **do**
    $ORD := ORD \cup \{(\langle EnCl(m), As\rangle, C) \mid C \in Cs(p)\}$

Figure 5.5: ORD construction.

5.2.1 ORD Construction

Figure 5.5 presents an algorithm, parameterized by a class analysis, which computes the ORD starting from an empty ORD. *Cs* is the output of the given class analysis. $Cs(x)$ denotes the set of classes which approximates the classes of all objects that may be bound to variable x. *EnCl* denotes the enclosing class of a given statement or method. Figure 5.5 shows the process of constructing different ORD edges. For example, for each statement containing a virtual call through variable p, the algorithm adds association edges from the node representing the class enclosing the statement to each node representing a class $C \in Cs(p)$. For static calls the analysis creates an association edge from the class containing the call to the class that defines the static method; static field accesses are handled analogously. Clearly, the ExtORD can be constructed by extending the algorithm from Figure 5.5 to add annotated edges.

5.2.2 Class Hierarchy Analysis

Class Hierarchy Analysis (CHA) is the simplest form of class analysis. To determine the possible bindings of polymorphic variables, CHA examines the structure of the class hierarchy. For a given variable r of declared type C, the set of possible classes of objects that may be bound to r is reported to be the set containing C and all direct and transitive subclasses of C (excluding abstract classes). For example, *CHA* computes the following sets $Cs(x)$ for the variables in Figure 3.1:

$$Cs(X.set.r) = \{Y, Z\} \quad Cs(p) = \{X\} \quad Cs(q) = \{Y, Z\}$$

5.2.3 Class Analysis Based on Points-to Analysis

Clearly, the solution derived from a points-to analysis can be used to derive the solution of a corresponding class analysis. The set of possible classes of objects that may be bound to r can be determined by examining the classes of all objects in the points-to set of r.

In this chapter we consider the context-insensitive points-to analysis based on Andersen's analysis presented in Chapter 3 and its corresponding class analysis. We study the impact of this class analysis on ORD and ExtORD precision. For the rest of this chapter this class analysis is denoted by *AND*.

5.3 Empirical Results

We performed experiments on the same set of 23 publicly available Java programs, described in Chapter 3 and used for the experiments in Chapter 3 and Chapter 4. The set includes programs from the SPECjvm98 suite, other benchmarks used in previous work on analysis for Java, as well as programs from an Internet archive (`www.jars.com`) of popular publicly available Java applications. All experiments were performed on a 900MHz Sun Fire-280R shared machine with 4Gb physical memory.

5.3.1 Precision

In our experiments we measured the impact of class analysis on ORD construction with respect to two metrics: (i) the number of edges in the ORD and (ii) the average size of the class firewall. We also measured the impact of class analysis on the number of edges in the ExtORD.

Program	*CHA*	*AND*	Reduction
			AND
proxy	43	29	33%
compress	67	43	36%
db	48	20	58%
jb	188	132	30%
echo	79	52	34%
raytrace	190	121	36%
mtrt	190	121	36%
jtar	284	235	17%
jlex	115	69	40%
javacup	594	170	71%
rabbit	513	212	59%
jack	501	307	39%
jflex	620	184	70%
jess	2275	1504	34%
mpegaudio	437	232	47%
jjtree	604	351	42%
sablecc	38643	18023	53%
javac	12292	7672	38%
creature	619	272	56%
mindterm	1028	423	59%
soot	76223	32581	57%
muffin	11431	2522	78%
javacc	780	554	29%
		Avg	46%

Table 5.1: ORD size.

Java programs typically contain a large portion of standard library code. We believe that in most cases one can assume the correctness of the library code, and therefore interactions between library classes and interactions between library classes and user classes do not need to be tested. In order to assess the impact of the different class analyses on determining the dependencies between user classes, we extract the *user-class-related* portion of the ORD/ExtORD computed by the algorithm in Figure 5.5. The nodes of the user-class-related subgraph are all nodes representing user classes, and the edges are all edges connecting user class nodes. For the remainder of this chapter the terms ORD and ExtORD will be used to refer to these corresponding user-class-related subgraphs.

ORD Size

The size of the ORD is an indication of the precision of the diagram; more precise diagrams contain fewer edges. The improvements of *AND* over *CHA* are shown in the

Program	*CHA*	*AND*	Reduction
			AND
proxy	4	2	50%
compress	6	4	33%
db	4	1	75%
jb	12	8	33%
echo	7	5	29%
raytrace	18	12	33%
mtrt	18	12	33%
jtar	27	26	4%
jlex	13	9	31%
javacup	26	16	49%
rabbit	37	24	35%
jack	30	7	77%
jflex	35	22	37%
jess	143	133	7%
mpegaudio	38	22	42%
jjtree	53	41	21%
sablecc	283	270	5%
javac	152	129	15%
creature	44	42	5%
mindterm	82	68	17%
soot	382	362	5%
muffin	192	136	29%
javacc	32	20	38%
		Avg	31%

Table 5.2: Class firewall size.

last two columns of table 5.1. On average, *AND* reduces the number of ORD edges by about 46%.

These results show that class analysis based on points-to analyses can significantly reduce the number of spurious dependence edges. The improved precision may lead to less time and effort spent on cycle breaking and stub construction when the ORD is used to define a test order in integration testing or a retest order in regression testing.

Class Firewall Size

The size of the class firewall indicates the suitability of the ORD for use in regression testing. Class firewalls computed from a more precise ORD contain fewer classes. The first two columns in Table 5.2 show the average class firewall sizes for our benchmarks. For each user class in a program, we calculated the size of its firewall. Then we took the average of these firewall sizes over all the user classes in the program. The last column

Program	*CHA*	*AND*	Reduction
			AND
proxy	188	143	24%
compress	161	117	27%
db	181	135	25%
jb	1179	1003	15%
echo	315	217	31%
raytrace	1489	961	36%
mtrt	1490	962	36%
jtar	1126	984	13%
jlex	1076	941	12%
javacup	1616	1062	34%
rabbit	1317	864	34%
jack	2289	1808	21%
jflex	2074	1127	46%
jess	5580	4235	24%
mpegaudio	1411	1009	29%
jjtree	7796	7164	8%
sablecc	87632	38023	57%
javac	66072	49208	26%
creature	3932	2962	25%
mindterm	4751	2525	47%
soot	153418	75026	51%
muffin	21186	6203	71%
javacc	8593	7907	8%
		Avg	30%

Table 5.3: ExtORD size.

in the table shows the improvements of *AND* over *CHA*. On average, the more precise class analyses reduce the average class firewall size by 31%.

These results show that class analysis based on points-to analysis produces substantially smaller class firewalls, which may result in less work and less time and effort spent on regression testing. Savings may occur because classes *not* affected by the change which triggered the regression testing will not be retested.

ExtORD Size

In order to estimate the impact of the different analyses on coverage analysis, we computed the size of the ExtORD. This size is an indication of the precision of the diagram; more precise diagrams contain fewer dependence edges (i.e., fewer spurious edges).

The percentage improvements for *AND* over *CHA* are shown in the last column in Table 5.3. On average, *AND* reduces the number of edges in the ExtORD by 30%.

Program	ORD		ExtORD	
	CHA	*AND*	*CHA*	*AND*
compress	45%	2%	46%	2%
db	65%	2%	40%	6%
raytrace	41%	2%	43%	8%

Table 5.4: Absolute analysis precision.

We draw two conclusions from these results. First, the results computed by *CHA* are very imprecise and this leads to a significant number of spurious dependence edges. Attempting to exercise statements in a way that triggers these impossible dependences in order to achieve high coverage will lead to substantial waste of time and effort. Second, this significant reduction shows that *AND* is a good candidate for use in tools for coverage analysis.

5.3.2 Absolute Precision

We performed *absolute* precision experiments, which answer the question how many edges reported by the class analysis are actually infeasible [56, 53]. This information allows to judge the usefulness of a particular class analysis for the purposes of ORD and ExtORD construction.

For three of our benchmarks we compared the ORDs and ExtORDs computed by *CHA* and *AND* with the diagrams that represent actual run-time dependences (computed based on profiling experiments and manual examination of dependence edges). The percentage of infeasible edges for *CHA* and *AND* is shown in Table 5.4. For these programs using *AND* results in small number of infeasible dependence edges, on average 2% for the ORD, and 5% for the ExtORD. Therefore, these results indicate that *AND* may be a good candidate for use in tools for ORD and ExtORD construction.

Chapter 6

Related Work

The work related to ours can be broadly classified in four categories: (i) flow analysis based on solutions of systems of constraints, (ii) points-to and class analysis for object-oriented languages, (iii) side-effect analysis, and (iv) work on interclass dependence diagram construction.

6.1 Constraint-based Flow Analysis

In this section we consider flow analyses that are formulated using systems of constraints. There are two major categories of such analyses: (i) *inclusion* constraint-based, and (ii) *unification* constraint-based. Inclusion based analyses are more precise and more expensive than unification based ones (e.g., a flow- and context-insensitive analysis based on inclusion constraints has cubic worst case complexity, while a flow- and context-insensitive analysis based on unification constraints runs in almost linear time).

6.1.1 Inclusion Constraints

The closest related work from this category are the *inclusion* constraint-based implementations of Andersen's flow- and context-insensitive points-to analysis for C from [21, 65] in which non-annotated constraints are used together with inductive form, cycle elimination, and projection merging. Recently, flow- and context-insensitive analysis for C based on non-annotated inclusion constraints has been presented in [32]; this work introduces new techniques for constraint resolution and constraint representation that allow improvements in analysis performance over [21] and [65].

We extend the work in [21, 65] by introducing a general framework for annotated inclusion constraints which allows analysis designers to model various dimensions of analysis precision precisely and efficiently. We demonstrate several instantiation of the framework. Field annotations are used to track object fields separately during points-to analysis. Method annotations allow us to model the semantics of virtual calls. Object annotations are used to model various forms of context sensitivity. These are not possible with the constraints from [21, 65] and [32]. In addition, we avoid analyzing dead library code by including a reachability computation in the analyses.

Other work in this category includes the context-sensitive version of Andersen's analysis for C presented in [26]. The analysis uses bottom-up traversal of the call graph and constraint copying to model context sensitivity for C. This work draws two conclusions: (i) context sensitivity may not improve the precision of Andersen's analysis for C, and (ii) using copying to model context sensitivity may lead to very expensive analysis. Our work shows that context sensitivity significantly improves the precision of Andersen-like points-to analysis for Java; in addition, it shows that using constraint annotations may be a practical alternative to standard copying methods for the purposes of modeling context sensitivity.

Recent work [31] presents a client-driven Andersen-style analysis for C. Similarly to one of our dimensions of parameterization for object-sensitive analysis, it employs selective context sensitivity by selecting procedures to be analyzed context-sensitively. Unlike our analysis, the selection in [31] is driven by the client analysis. Our analysis is designed to be a relatively precise general-purpose pointer analysis to be reused by different clients; in our case the selection is based on a heuristic that in practice preserves the precision of the fully context-sensitive analysis.

6.1.2 Unification constraints

Constraint indices and constraint polarities have been used to introduce call-site context-sensitivity in unification-based flow analysis for C [23] and in on-demand flow analysis for functional languages [22]. This work has similar flavor to our use of annotations for tracking flow of values through object fields and object contexts. Conceptually, the goal is to restrict the flow of values in constraint systems—either for unification constraints in [23], or for inclusion constraints in our case. Our framework allows the use of annotations to model different dimensions of precision, including various forms of context sensitivity. In addition, it is based on inclusion constraints that are potentially more expensive and more precise than the constraints in [23] and [22]. Other work on context-sensitive analysis for C includes [17] which adds context sensitivity to the analysis from [16], which is an extension of the unification-based analysis. Similarly to [26], one of the conclusions of [17] is that context sensitivity does not improve the precision of the original analysis.

Context-sensitive alias analysis for Java based on unification constraints was presented in [47]. This work has similar flavor to [23]. However, it is unclear whether the analyses in [47] can be used on large Java programs. Our empirical results demonstrate that using our framework, context-sensitive points-to analysis for Java can be relatively precise and practical.

6.2 Points-to and Class Analysis for Object-oriented Languages

In this section we discuss points-to and class analyses related to the field-sensitive points-to analysis in Chapter 3 and to the object-sensitive analysis in Chapter 4.

6.2.1 Points-to Analysis

Flow-insensitive context-sensitive alias analysis for Java has been developed by Ruf [57] in the context of a specialized algorithm for synchronization removal. Ruf's analysis uses method summaries to model context sensitivity and, unlike our object-sensitive analysis, requires bottom-up traversal of the call graph (i.e., a called method is analyzed before or together with its callers). Also, our analysis is based on Andersen's analysis, which has cubic time worst case complexity [6]; Ruf's algorithm is based on the almost-linear Steensgaard's points-to analysis for C [63]. Other context-sensitive points-to analyses for Java are presented in [30, 13]. The algorithm in [13] uses method summaries to model context sensitivity, while [30] uses the call string approach. In general, these analyses are more precise and significantly more costly than ours.

Flow-insensitive context-insensitive points-to analyses for Java are described in [52, 64, 43, 42, 9]. The analysis in [52] is based on Steensgaard's flow- and context-insensitive analysis and is less expensive and less precise than our field-sensitive points-to analysis. Work by Streckenbach and Snelting [64], which postdates our initial report on field-sensitive points-to analysis [54], describes a points-to analysis for Java based on Andersen's analysis for C. Analysis cost is higher than ours, which is most likely due to the different kind of constraints employed by this approach. Another flow- and context-insensitive points-to analysis for Java based on Andersen's analysis is presented in [43], together with several analysis variations. Direct comparison with this work is not possible because it handles the library code in a different manner; based on the size of the analyzed code, our analysis appears to be faster. Recent work by Hendren et. al. presents context-insensitive points-to analysis for Java based on Andersen's analysis [42, 9]. These analyses employ constraints similar to ours, and incorporate several optimization (e.g., filtering based on declared types [42], and bdd's which represent points-to sets efficiently [9]). Due to these optimizations the analyses perform better than ours; in the future we plan to incorporate these new techniques in our analysis for Java. Other recent work that is based on Andersen's analysis is the points-to analysis described in [72], which is context-insensitive and intraprocedurally flow-sensitive.

6.2.2 Class Analysis

Different mechanisms for context sensitivity have been studied in the context of class analysis [49, 48, 1, 50, 2, 30]; these methods typically use some combination of the parameter types to abstract context. The work in [48, 1, 2] presents class analyses for Smalltalk and Self. Similarly to our object-sensitive analysis, these analyses use information about the receiver object in order to create and select contextual method versions. Unlike our analysis, they use additional information (e.g., the method invocation site). The idea of object sensitivity is to use only the receiver object as context; we believe that for the purposes of flow-insensitive points-to analysis for Java, using invocation sites or other information may be redundant in most cases. The non-parameterized object-sensitive analysis from Section 4.2.1 can be expressed in the general framework for context-sensitive class analysis presented in [30]; however, it is not identified or studied in [30]. Various practical context-insensitive class analyses are presented in [20, 7, 19, 69, 66].

6.3 Side-effect Analysis

Conceptually, our MOD analysis is based on similar MOD analyses for C [60, 34, 59]. Razafimahefa [52] presents algorithms for side-effect analysis for Java that are based on context-insensitive information. The more precise of the algorithms is based on context-insensitive points-to analysis for Java derived from Steensgaard's analysis for C [63]. Clausen [15] investigates side-effect analysis for Java in the context of a Java bytecode optimizer. This analysis does not use points-to information, i.e., a modification through field f is assumed to write *all* objects whose class contains field f. This may result in less precise side-effect information.

6.4 Dependence Diagrams

The firewall approach was proposed by White et al. for regression testing of procedural code [74, 75]. Kung et al. [38] define the Object Relation Diagram (ORD) and adapt the firewall approach for object-oriented languages; however, Kung et al.'s work does not consider the effects of dynamic binding of polymorphic variables. Labiche et al. [39] propose an approach for correcting the problem with dynamic bindings. Their work uses the structure of the class hierarchy to determine possible bindings. Our approach uses more precise class analysis and constructs ORDs with significantly fewer spurious dependence edges, compared to ORDs constructed by examining the class hierarchy.

Work by Tai and Daniels [67] and Jeron et al. [37] concentrates on approaches for cycle breaking in the ORD. This work addresses the following question: given the diagram, what cycles should be removed so that the minimum number of stubs will need to be constructed. Our work concentrates on improving the precision of the ORD, which potentially leads to fewer dependence cycles.

There are many existing class analyses and related points-to analyses (see Section 6.2) for object-oriented programming languages. Our work is the first one to investigate the use of these analyses for the purposes of ORD and ExtORD construction.

Work by Tonella and Potrich [70] describe the construction of the UML class diagram for C++. The class diagram focuses on *static* interclass interactions and does not consider the effects of dynamic binding of polymorphic variables (similarly to the ORD by Kung et. al.). It is useful for certain program understanding tasks. The ORD and ExtORD focus on *dynamic* class interactions, particularly targeting ones due to dynamic binding; they are useful primarily for program testing and reverse engineering. Andersen-like points-to analysis is used in [70] to infer types of objects stored in containers (a refinement of the class diagram); in our work it is used for a different purpose, to infer dynamic dependences.

Chapter 7
Summary and Future Work

Many existing practical flow analyses are based on efficient inclusion constraint systems. However, these analyses do not model dimensions of flow analysis precision such as field sensitivity and context sensitivity. These dimensions of precision are of crucial importance for the analysis of object-oriented languages because field insensitivity and context insensitivity inherently compromise analysis precision due to fundamental object-oriented idioms such as encapsulation, inheritance, and polymorphism. To achieve the goal of practical and relatively precise flow analysis of large object-oriented systems, we propose the use of a general framework for annotated inclusion constraints that allow the precise modeling of different dimensions of flow analysis precision.

7.1 Annotated Inclusion Constraints

We propose a general framework which allows analysis designers to design relatively precise flow analyses from analyses that are relatively imprecise. The key idea is to take a relatively *imprecise* analysis that can be expressed using non-annotated inclusion constraints, and add a dimension of precision by instantiating the framework with appropriate annotations and operations on the annotations. The analysis generates annotated inclusion constraints, and uses the operations on the annotations to resolve these constraints. The solution of the system of annotated constraints contains the solution of the relatively *precise* analysis problem. This approach allows the reuse of techniques for efficient constraint representation and constraint resolution, as well as the physical reuse of efficient constraint solvers. Using this approach results in analyses that achieve substantially better precision while remaining efficient and practical.

7.2 Field-sensitive Points-to Analysis for Java

We present a formulation and implementation of a field-sensitive points-to analysis for Java as an instance of the framework for annotated inclusion constraints. The relatively imprecise points-to analysis that our analysis is based on, is the field-insensitive, context-insensitive Andersen's points-to analysis for C [6]. We use constraint annotations to model precisely and efficiently two dimensions of precision (i) the flow of values through object fields and (ii) the semantics of virtual calls. By solving systems of annotated inclusion constraints, we have been able to perform the first practical and relatively precise points-to analysis for Java. Our empirical results demonstrate the practicality of this analysis. They also show that the points-to solution has significant impact on a wide variety of clients applications and client analyses.

7.3 Context-sensitive Points-to Analysis for Java

We present *object sensitivity*, a new form of context sensitivity suitable for flow-insensitive flow analysis for Java. The key idea of our approach is to analyze a method separately for each of the objects on which this method is invoked. To ensure flexibility and practicality, we propose a parameterization framework that allows analysis designers to control the tradeoffs between cost and precision in the object-sensitive analysis.

We have formulated and implemented several object-sensitive points-to analyses as instances of our *annotated inclusion constraints* framework. The relatively imprecise analysis is the context-insensitive analysis in Chapter 3, and the dimension of precision modeled by constraint annotations is object sensitivity. We compare these instantiations with the context-insensitive points-to analysis for Java from Chapter 3 which is based on Andersen's analysis for C. Our extensive experiments show that for the majority of programs these analyses have comparable cost. Our results also show that object sensitivity significantly improves the precision of call graph construction, virtual call resolution, and proving downcast safety. These experiments demonstrate that (i) object-sensitive analyses can achieve significantly better precision than context-insensitive ones, and (ii) the annotations model precision dimensions such as context sensitivity efficiently, allowing the object-sensitive analyses to remain efficient and practical. In addition we present a new form of side-effect analysis for Java which is based on object-sensitive points-to analysis, and demonstrate empirically that object sensitivity significantly improves the precision of side-effect analysis.

7.4 Applications to Software Engineering Problems

Interclass dependence diagrams such as the *Object Relation Diagram (ORD)* has important applications to software engineering problems (e.g., integration testing, integration coverage analysis, regression testing, impact analysis, program understanding, and reverse engineering). It is important to investigate techniques for constructing relatively precise ORDs because the imprecision of the ORD directly affects the feasibility of its usage. Our work makes three contributions. First, we develop the *Extended Object Relation Diagram (ExtORD)*, a version of the ORD designed for use in integration coverage analysis. Second, we develop a general algorithm for ORD construction, parameterized by class analysis. Third, we demonstrate empirically that relatively precise class analysis based on the field-sensitive points-to analysis in Chapter 3 can improve significantly the precision of the ORD and ExtORD compared to earlier work.

7.5 Future Work

One direction of future work is to investigate techniques for further reduction of analysis cost. For example, the cost of various analyses can be reduced if the library code is analyzed in advance. This would allow partial analysis information about the Java libraries to be computed once and subsequently used for different client programs. Analysis performance can be improved by integrating into the framework new techniques for constraint resolution (e.g., the sub-transitive closure from [32]). Also, the cost of points-to analysis for Java can be further reduced by using static types for the purposes of filtering infeasible objects from points-to sets [42], and bdd's as part of a mechanism for efficient points-to set representation [9].

Another possible direction of future work is to investigate other instances of the framework. For example, it can be used to formulate and implement CFA-style and other forms of context sensitivity in points-to and class analysis for object-oriented programs. We are working on the formulation and implementation of a context-sensitive analysis which identifies resources that may be subject to certain hardware faults. The analysis information can be used to direct fault injection for the purposes of testing highly reliable web applications. Another interesting possibility for future work is to investigate analyses that combine different mechanisms of context sensitivity (e.g., CFA-style context sensitivity and object sensitivity). We believe that this can be achieved easily by combining the annotations used to model each form of context sensitivity. In addition, we would like to explore flow analyses such as constant propagation and permission flow analysis.

We also plan to perform theoretical and experimental comparison of object-sensitive analyses with context-sensitive analyses that are based on the call string approach to context sensitivity [62]. In addition, it would be interesting to have theoretical and empirical comparison between object sensitivity and other instances of the functional approach to context sensitivity (e.g., [13, 57]).

In the future we plan to investigate applications of points-to, side-effect, and def-use analyses in the context of software productivity tools (e.g., tools for program understanding and testing). The tradeoff between cost and precision is an important issue in such tools, and we intend to focus our work on this problem. We are also interested in tools for construction of various types of interclass dependence diagrams (e.g., concern graph diagrams) to be used for integration testing, concern identification and other software engineering tasks.

References

[1] O. Agesen. Constraint-based type inference and parametric polymorphism. In *Static Analysis Symposium*, LNCS 864, pages 78–100, 1994.

[2] O. Agesen. The cartesian product algorithm. In *European Conference on Object-oriented Programming*, LNCS 952, pages 2–26, 1995.

[3] A. Aiken, M. Fähndrich, J. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *International Workshop on Types in Compilation*, 1998.

[4] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, June 1993.

[5] J. Aldrich, C. Chambers, E. G. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *Static Analysis Symposium*, LNCS 1694, pages 19–38, 1999.

[6] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[7] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, 1996.

[8] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, 1978.

[9] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using bdds. In *Conference on Programming Language Design and Implementation*, 2003.

[10] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 1999.

[11] B. Blanchet. Escape analysis for object-oriented languages. Applications to Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 20–34, 1999.

[12] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 35–46, 1999.

[13] R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *Symposium on Principles of Programming Languages*, pages 133–146, 1999.

[14] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–19, 1999.

[15] L. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045, 1997.

[16] M. Das. Unification-based pointer analysis with directional assignments. In *Conference on Programming Language Design and Implementation*, pages 35–46, 2000.

[17] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Static Analysis Symposium*, 2001.

[18] J. Dean, D. Grove, and C. Chambers. Optimizations of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-oriented Programming*, pages 77–101, 1995.

[19] G. DeFouw, D. Grove, and C. Chambers. Fast interprocedural class analysis. In *Symposium on Principles of Programming Languages*, pages 222–236, 1998.

[20] A. Diwan, J.Eliot B. Moss, and K. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 292–305, 1996.

[21] M. Fähndrich, J. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Conference on Programming Language Design and Implementation*, pages 85–96, 1998.

[22] M. Fähndrich, J. Rehof, and M. Das. From polymorphic subtyping to CFL reachability: Context-sensitive flow analysis using instantiation constraints. Technical Report MS-TR-99-84, Microsoft Research, 1999.

[23] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Conference on Programming Language Design and Implementation*, pages 253–263, 2000.

[24] Stephen J. Fink, Kathleen Knobe, and Vivek Sarkar. Unified analysis of array and object references in strongly typed languages. In *Static Analysis Symposium*, pages 155–174, 2000.

[25] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An optimizing compiler for Java. *Software: Practice and Experience*, 30(3):199–232, March 2000.

[26] J. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Static Analysis Symposium*, LNCS 1824, pages 175–198, 2000.

[27] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *International Conference on Compiler Construction*, LNCS 1781, 2000.

[28] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[29] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, November 2001.

[30] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–124, 1997.

[31] S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *Static Analysis Symposium*, 2003.

[32] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA. In *Conference on Programming Language Design and Implementation*, pages 254–263, 2001.

[33] M. Hind, M. Burke, P. Carini, and J. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, May 1999.

[34] M. Hind and A. Pioli. Which pointer analysis should I use? In *International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[35] S. Horwitz. Precise flow-insensitive may-alias analysis is np-hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, January 1997.

[36] IBM Corporation. *High Performance Compiler for Java*, 1997. `www.alphaWorks.ibm.com/formula`.

[37] T. Jeron, J.-M. Jezequel, Y. Le Traon, and P. Morel. Efficient strategies for integration and regression testing of OO systems. In *International Symposium on Software Reliability Engineering*, pages 260–269, 1999.

[38] D. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima. Class firewall, test order and regression testing of object-oriented programs. *Journal of Object-Oriented Programming*, 8(2):51–65, 1995.

[39] Y. Labiche, P. Thevenod-Fosse, H. Waeselynck, and M.-H. Durand. Testing levels for object-oriented software. In *International Conference on Software Engineering*, pages 136–145, 2000.

[40] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[41] W. Landi and B. G. Ryder. Pointer induced aliasing: A problem classification. In *Symposium on Principles of Programming Languages*, pages 93–103, 1991.

[42] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*, LNCS 2622, pages 153–169, 2003.

[43] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 73–79, 2001.

[44] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.

[45] T. Marlowe and B. G. Ryder. Properties of data flow frameworks: A unified model. *Acta Informatica*, 28:121–163, 1990.

[46] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *International Symposium on Software Testing and Analysis*, 2002.

[47] R. O'Callahan. *The Generalized Aliasing as a Basis for Software Tools*. PhD thesis, Carnegie Mellon University, 2000.

[48] N. Oxhoj, J. Palsberg, and M. Schwartzbach. Making type inference practical. In *European Conference on Object-oriented Programming*, pages 329–349, 1992.

[49] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 146–161, 1991.

[50] J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–340, 1994.

[51] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, 1994.

[52] C. Razafimahefa. A study of side-effect analyses for Java. Master's thesis, McGill University, December 1999.

[53] A. Rountev. *Dataflow Analysis of Software Fragments*. PhD thesis, Rutgers University, August 2002.

[54] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated inclusion constraints. Technical Report DCS-TR-417, Rutgers University, July 2000. (Initial report superseded by DCS-TR-428).

[55] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 43–55, October 2001.

[56] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in Java software. In *International Conference on Software Engineering*, pages 210–220, May 2003.

[57] E. Ruf. Effective synchronization removal for Java. In *Conference on Programming Language Design and Implementation*, pages 208–218, 2000.

[58] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *International Conference on Compiler Construction*, LNCS 2622, pages 126–137, 2003.

[59] B. G. Ryder, W. Landi, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems*, 23(2):105–186, March 2001.

[60] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Static Analysis Symposium*, LNCS 1302, pages 16–34, 1997.

[61] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[62] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[63] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[64] M. Streckenbach and G. Snelting. Points-to for Java: A general framework and an empirical comparison. Technical report, U. Passau, September 2000.

[65] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Symposium on Principles of Programming Languages*, pages 81–95, 2000.

[66] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 264–280, 2000.

[67] K.-C Tai and J. F. Daniels. Interclass test order for object-oriented software. *Journal of Object-Oriented Programming*, 12(4):18–25, 1999.

[68] F. Tip, C. Laffra, P. Sweeney, and D. Streeter. Practical experience with an application extractor for Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 292–305, 1999.

[69] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 281–293, 2000.

[70] P. Tonella and A. Potrich. Reverse engineering of the UML class diagram from c++ code in the presence of weakly typed containers. In *International Conference on Software Maintenance*, pages 376–385, 2001.

[71] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction*, LNCS 1781, pages 18–34, 2000.

[72] J. Whaley and M. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Static Analysis Symposium*, pages 180–195, 2002.

[73] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 187–206, 1999.

[74] L. White and H.K.N. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *International Conference on Software Maintenance*, pages 262–271, 1992.

[75] L. White, V. Narayanswamy, T. Friedman, M. Kirschenbaum, P. Piwowarski, and M. Oha. Test manager: A regression testing tool. In *International Conference on Software Maintenance*, pages 338–347, 1993.

Vita

Ana Milanova

1997	B.A. in Computer Science, American University in Bulgaria.
1999	M.S. in Computer Science, Rutgers, The State University of New Jersey.
1997–1999	Teaching Assistant, Department of Computer Science, Rutgers, The State University of New Jersey.
1999–2001	Research Assistant, Department of Computer Science, Rutgers, The State University of New Jersey.
2001–2002	On maternity leave, Tucson, Arizona.
2002–2003	Research Assistant, Department of Computer Science, Rutgers, The State University of New Jersey.
2001	Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2001.
2002	Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the International Symposium on Software Testing and Analysis*, July 2002.
2002	Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Constructing precise object relation diagrams. In *Proceedings of the International Conference on Software Maintenance*, October 2002.
2002	Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise call graph construction in the presence of function pointers. In *Proceedings of the International Workshop on Source Code Analysis and Manipulation*, October 2002.
2003	Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise call graphs for C programs with function pointers. To appear in *Special Issue of the Journal of Automated Software Engineering*, 2004.
2003	Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Fragment class analysis for testing of polymorphism in Java Software. In *Proceedings of the International Conference on Software Engineering*, May 2003.