STATIC ANALYSES FOR JAVA IN THE PRESENCE OF DISTRIBUTED COMPONENTS AND LARGE LIBRARIES

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of The Ohio State University

By

Mariana L. Sharp, B.S., M.S.

* * * * *

The Ohio State University

2008

Dissertation Committee:

Atanas Rountev, Adviser

Neelam Soundarajan

Paul A. G. Sivilotti

Approved by

Adviser
Graduate Program in
Computer Science and
Engineering

ABSTRACT

Modern Java applications present significant challenges for existing algorithms for static compile-time analysis. Two such challenges are large code size, and applications that are distributed across multiple machines. Existing analysis algorithms are not suitable to deal with these challenges. One reason is that analyses are typically designed to operate on whole homogeneous programs. For applications built with libraries, the library code is analyzed together with the application code. This leads to scalability problems when the analysis algorithms are used on large-scale Java software built with reusable library components. Moreover, Java applications using component models such as RMI (Remote Method Invocation) have complex semantics which are not modeled by existing analyses.

In this work we propose several analysis techniques for modern Java software. First, we define a theoretical model for points-to analysis of distributed RMI-based Java applications. We propose an algorithm for computing points-to information for such applications. This algorithm is implemented and evaluated experimentally on a set of distributed Java programs. Second, we define an incremental approach for type analysis and dependence analysis of large-scale Java software built with reusable library components. Our approach employs precomputed library summary information. The library code is analyzed independently of any client code, using a summary-generation analysis. The resulting summary is reusable for subsequent analysis of any client component. The solution for the client code, computed using the summary, is the same as the solution what would have been computed by a whole-program analysis. Our experimental studies indicate that the cost of whole-program type analysis and dependence analysis can be reduced dramatically by the proposed approach.

This work presents novel advances towards solving two important analysis problems for Java software: analysis in the presence of large libraries and analysis in the presence of RMI calls. We also provide theoretical foundations and experimental insights for future work on static analyses for Java software built with large reusable components and with RMI-based middleware platforms such as Enterprise JavaBeans. As a result, our work brings static analysis techniques a step closer to practical use in real-world software tools for industrial Java applications.

To my little girl, and to my Rich

ACKNOWLEDGMENTS

I would like to thank my adviser Prof. Nasko Rountev for providing support and guidance. Without his help and constant feedback I would have never known where to go or what to look for. Under his patient guidance I learned many things that helped my professional and personal growth.

I am grateful to Prof. Paul Sivilotti for teaching distributed computing in such great and entertaining lectures. I would also like to thank Prof. Neelam Soundarajan, for providing valuable feedback on my writing in his seminar. They have taught me that theory is not as hard as it seems. I would like to thank Prof. Thomas Marlowe from Seton Hall University for providing comments on earlier versions of this thesis.

Thanks to Jason Sawin for sharing interesting ideas and making the work on the TACLE project entertaining. To Laura Stoia, my friend and fellow graduate student, thanks for being by my side during all the years of graduate school.

I am grateful to my parents for supporting me in my plans, even when they did not like me leaving the country. Finally I would like to thank my husband Rich for the enormous efforts he made to help raise our baby. His support and humor got me through a lot during the years of graduate school. I look forward to our future life together, wherever it may be.

VITA

August 2006	M.S. Computer Science & Engineering, The Ohio State University
June 1999	B.S. Computer Science, University of Bucharest
September 2002 – present	Graduate Research/Teaching Associate, The Ohio State University
March 2000 – August 2002	Software Developer, Cornersoft Tech, Bucharest
December 1999 – March 2000	Software Developer, Ifsoft, Bucharest
February 2, 1977	Born – Constanta, Romania

PUBLICATIONS

Research Publications

J. Sawin, M. Sharp and A. Rountev Generating Run-Time Progress Reports for a Points-to Analysis in Eclipse In *Eclipse Technology Exchange Workshop at OOPSLA (ETX'06)*, Oct 2006.

M. Sharp and A. Rountev Static Analysis of Object References in RMI-based Java Software In *IEEE Transactions on Software Engineering (TSE)*, pages 664–681, Sep 2006.

M. Sharp, J. Sawin and A. Rountev Building a Whole-Program Type Analysis in Eclipse In *Eclipse Technology Exchange Workshop at OOPSLA (ETX'05)*, Oct 2005.

M. Sharp and A. Rountev Static Analysis of Object References in RMI-based Java Software In *IEEE International Conference on Software Maintenance (ICSM'05)*, pages 101–110, Sep 2005.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

Software Engineering	Prof. Atanas Rountev
	Prof. Neelam Soundarajan
	Prof. Paul A. G. Sivilotti

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Table **Page**

CHAPTER 1

INTRODUCTION

Static program analysis is a widely used approach for inferring properties of program behavior based on the structure and semantics of program source code. Static analysis techniques play an important role in tools for performance optimization, program understanding and maintenance, software testing, and verification of program properties. Our work considers a number of *interprocedural* static analyses which model the interactions among multiple procedures or methods due to calls and shared memory locations. In particular, we focus on four interprocedural analyses that play a foundational role in various software tools: points-to analysis, type analysis, side-effect analysis, and dependence analysis (outlined in Section 1.1). Our work addresses several challenges for adapting existing algorithms for these analyses to handle features of modern Java software, such as the existence of large library components, and distribution across multiple Java virtual machines (Section 1.2).

1.1 Static Analyses Considered in Our Work

Points-to analysis

The analysis of points-to relationships is a research problem that has received a lot of attention in the last fifteen years, due to the important role it plays as an enabling technology for a variety of other static analyses. As such analysis is in general undecidable [32], a number of approximation algorithms have been proposed, with different trade-offs between analysis cost and the precision of the results.

There are several algorithmic dimensions that affect the cost/precision trade-offs of points-to analyses. The most important dimensions are the following:

- *Flow sensitivity.* Flow-insensitive analyses do not take into account the flow of control within a procedure (or a method), and compute a single set of points-to relationships for the entire procedure. In contrast, flow-sensitive analyses compute a separate solution at each program point inside a procedure. Theoretically, flow-sensitive analyses are more expensive and more precise than their flow-insensitive counterparts.

- *Context sensitivity.* Context-insensitive analyses do not attempt to distinguish among the different invocation contexts of a procedure. Context-sensitive analyses employ some abstraction of the calling context; as a result, such analyses are potentially more precise and more expensive than context-insensitive ones. In parameter-based context-sensitive analyses, calling context is modeled by using some abstraction of the values of the actual parameters at a call site, while call-chain-based context-sensitive analyses represent the context with a vector of call sites for the procedures that are currently active on the run-time call stack.

- *Representation for dynamically-allocated objects.* There are various possible representations for entities that are allocated dynamically, for example, due to `malloc` calls in C or `new` expressions in C++ and Java. For analysis of

C, typically a separate abstract object (i.e., an analysis abstraction of a set of run-time objects) is used for each malloc call site. For analysis of object-oriented programs, the two most popular schemes are to use (1) one abstract object per class, or (2) one abstract object per new expression. More precise versions of these schemes have also been proposed: for example, by introducing context-sensitive versions of abstract objects.

- *Field sensitivity.* A field-sensitive analysis computes a separate points-to solution for each field (e.g., object field in C++/Java or structure field in C/C++) of each abstract object. A field-insensitive analysis computes a points-to solution for the entire abstract object, without distinguishing the values of individual fields. Some points-to analyses for object-oriented languages are field-based, meaning that a separate solution is computed for each declared field without distinguishing the incarnations of this field across different abstract objects.

- *Directionality.* This dimension typically applies to flow-insensitive analyses. An equality-based analysis treats an assignment p = q as representing bi-directional flow of values from p to q and from q to p, that is, the points-to solutions for p and q are considered to be the same. A subset-based analysis treats such an assignment as unidirectional flow of values from q to p; as a result, the points-to solution for q is a subset of the points-to solution for p.

- *Call graph construction.* The calling structure of the program could depend on points-to relationships, for example, due to function pointer calls in C and polymorphic calls in C++ and Java. Some analyses use a precomputed conservative call graph, while others compute the call graph on the fly during the analysis, as points-to relationships are discovered.

In Chapter 2, we describe a specific subset-based, context-insensitive, flow-insensitive, field-sensitive points-to analysis with on-the-fly call graph construction in the context of RMI applications.

Type analysis

Type analysis computes a static approximation of the set of types for the run-time values of program variables and expressions. In the context of Java software, type analysis can be thought of a form of points-to analysis in which dynamically allocated objects are represented with one abstract object per class. Chapter 3 presents a specific flow-insensitive, context-insensitive, subset-based, field-based type analysis with on-the-fly call graph construction, and considers the problem of reducing the cost of such analysis for Java applications built with large library components.

Side-effect analysis

Points-to analysis is a prerequisite for a variety of other analyses, for example side-effect analysis. Side-effect analysis (or MOD analysis) determines the memory locations that may be modified by the execution of a program statement. The information computed by this analysis has a wide variety of uses in compilers, such as code motion and partial redundancy elimination, and in software tools. Because the output of a points-to analysis is required as an input for side-effect analysis, the precision of this output affects the MOD solution. Thus the dimensions presented above for the points-to analysis space directly affect the outcome of the side-effect analysis in terms of cost and precision. In Chapter 2 we present a side-effect analysis for RMI applications based on a flow- and context-insensitive, field-sensitive, subset-based points-to analysis.

Dependence analysis

In dependence analysis, points-to information is used typically to create the call graph and to resolve memory reads and memory writes through pointers. If dependence analysis is field-based, there is no need to distinguish accesses through pointers, and only call graph information is needed. Thus, a type analysis solution is sufficient input to the dependence analysis, in order to create the program call graph in the presence of polymorphism. Chapter 4 presents a field-based flow- and context-sensitive dependence analysis (based on an earlier field-based subset-based type analysis) and addresses the problem of reducing analysis cost for applications built with large libraries.

1.2 Two Challenges for Analysis of Modern Java Software

This section outlines two features of modern Java applications what present serious challenges for existing approaches for static analysis.

Distributed Java applications. Since its introduction in 1995, Java has been recognized as an excellent platform for creating enterprise solutions. Of particular interest to our work is the use of Java for developing distributed server-side applications.

The Java distributed object protocol is built on the same basic architecture as general remote objects protocols, which are designed to make an object on one computer appear as if it is residing on a different computer. The distributed object architecture is based on a network communication layer that is quite simple. Essentially there are three parts to this architecture: the business object, the skeleton and the stub. The stub and skeleton are responsible for making the object located on the server machine appear as if it is running locally on the client machine. This is accomplished through some kind of *remote method invocation* (RMI) protocol. The RMI protocol is used to call methods over the network. CORBA, Java RMI, and Microsoft .NET all use their own RMI protocols. Java RMI is the Java language version of an RMI protocol and it is used as the underlying communication mechanism for other more complex architectures, such as Enterprise Java Beans (EJB).

Large-scale applications. Java code characteristically consists of a large number of classes and methods, the majority of them being grouped in JAR archives and representing libraries (either the standard library produced by Sun or third party libraries that provide specific functionality). Even applications that seem simple can trigger the use of hundreds of classes because they call methods that are defined in these external archives.

The majority of library classes are designed with reusability in mind. This is clearly the case for Java standard library classes that are involved in *any* Java application. This is usually also true about other library classes, including classes provided by third-party vendors. The result is an architecture that can be viewed as layered, because of the way classes are grouped into packages that do not necessarily have knowledge of each other. Such features present significant challenges to existing algorithms for static compile-time analysis.

Two such challenges that result directly from the features presented above are:

1. Traditional algorithms are designed for non-distributed software and cannot be used for applications that run on multiple machines. For example, distributed

Java applications using RMI have complex semantics which existing static analysis are not capable of modeling.

2. Existing analysis algorithms are typically designed to operate on whole homogeneous programs, starting from scratch at each analysis execution. For Java applications built with large libraries, the library code is analyzed together with the application code as part of the whole program. This creates potential scalability problems with respect to analysis time and memory usage, which could limit the practical use of these analyses for real-world Java applications.

1.3 Contributions

We consider the important categories of static analyses outlined in Section 1.1, and show how points-to analysis and side-effect analysis can be successfully generalized to handle RMI Java applications, with practical analysis cost (Chapter 2) and how analysis cost can be reduced for applications built with large libraries, both for type analysis (Chapter 3) and for dependence analysis (Chapter 4). The specific contributions of this work are as follows:

- We define a theoretical model for the analysis of distributed Java applications. We present an extension of points-to analysis for RMI-based applications, and define the RMI-specific generalization of side-effect analysis. We also evaluate experimentally these techniques on a collection of RMI programs.

- We define an incremental approach for analyzing Java applications built with reusable components. We use precomputed summaries of these components as input to the analysis of the client code. Summary generation algorithms for type analysis and dependence analysis are defined, based on the general theoretical framework of IDE dataflow analysis [68]. An experimental evaluation of the running time and memory usage of the analyses on a collection of Java programs indicates that significant cost reductions can be achieved with no loss of precision.

CHAPTER 2

STATIC ANALYSIS OF OBJECT REFERENCES IN RMI-BASED JAVA SOFTWARE

2.1 Introduction

Java Remote Method Invocation (RMI) is an object model for developing distributed applications in Java [44]. Using RMI, objects in one Java virtual machine (JVM) can invoke methods on objects in other JVMs. RMI provides powerful features such as object references that cross JVM boundaries, remote invocations that can use entire object graphs as parameters, and distributed garbage collection. RMI can either be used as a stand-alone middleware platform, or as the foundation for more advanced architectures. For example, both Enterprise JavaBeans and Jini are based on RMI and also provide additional middleware services.

Distributed applications play an important role in various commercial, scientific, and engineering domains. The development of such applications poses numerous problems related to software correctness, performance, and maintainability. For RMI applications in particular, some approaches have been investigated for program understanding, performance optimizations, and software testing (e.g., [20,29,42,48–50,82]). However, at present there is no work on establishing *systematic foundations for static* *analysis* of RMI applications. The goal of our work is to take a significant step towards defining such foundations; an earlier description of this investigation appeared in [70, 71].

The target of the work presented in this chapter is *points-to analysis*. Such analysis determines the objects to which locals, formals, and fields may point. This information has a wide range of uses in other static analyses; in turn, the results of these analyses are used in a variety of program understanding applications, testing approaches, software verification techniques, and performance optimizations. There has been a large body of work on points-to analysis; most of this work is summarized in [32, 67]. A brief overview of existing results on points-to analysis is presented in Chapter 5. However, these existing analyses cannot be applied directly to RMI-based distributed Java applications. Thus, the builders of such applications cannot take advantage of a large number of well-known static analyses (points-to analyses as well as other popular analyses that require points-to information).

Theoretical Model. The first goal in this work is to establish the foundations for points-to analysis of RMI-based Java applications. We define formally a particular style of points-to analysis: flow- and context-insensitive subset-based analysis (i.e., Andersen-style analysis [2]). Our approach could easily be extended to flow- and context-sensitive points-to analyses, and to analyses that are not subset-based. Such extensions are well understood for non-distributed Java programs (e.g., [23, 47]) and there are no conceptual difficulties in defining such extensions for our analysis.

The importance of these foundations is twofold. First, they provide a basis for defining a wide range of points-to analyses for RMI applications, based on the large number of such analyses for non-distributed programs. Second, they enable work on RMI-based extensions of other popular static analyses (e.g., dependence analyses, side-effect analyses, program slicing, change impact analyses, etc.).

Analysis Algorithm. The second goal is to define an algorithm for implementing the points-to analysis. The algorithm is a generalization of an approach by Lhoták and Hendren [37] for non-distributed Java programs. We introduce new techniques that allow the analysis to represent the flow of remote object references, the effects of remote invocations, and the remote propagation of object graphs through serialization. Furthermore, we present an approach for efficient modeling of the code in the standard Java libraries; our experiments indicate that this approach is essential for reducing the running time of the analysis.

Static Analyses for Program Understanding. The third goal of this work is to describe three uses of points-to analysis for the purpose of understanding RMI applications. First we discuss the use of the resulted call graph to answer questions related to the inter-method and inter-component flow of control. Second, we outline the use of points-to information to identify write-read dependencies due to remote calls. In particular, we consider *inter-component* dependencies, in which components running in two different JVMs potentially access the same memory location. Third, we discuss the use of the points-to analysis to identify opportunities for improving the analyzed program by reducing the cost of serialization at remote calls [82].

Analysis Implementation. The fourth goal of this work is to implement and evaluate the points-to analysis. We present a preliminary experimental study on a set of 12 RMI applications. Our initial results suggest that the analysis could be a good candidate for a general-purpose points-to analysis of RMI-based programs.

```
—– Event —–
class Event implements Serializable {
     public Date date() { return on; }
     public String description() { return DPs; }
     public Event(String a) { des = a; on = new Date(); }
     private Date on; private String des; }
—– Event Listener —–
interface Listener extends Remote {
    public void occurred(Event b); }
—– Event Channel —–
interface Channel extends Remote {
     public void add(Listener c);
     public void announce(Event d); }
```

Figure 2.1: Running example, part 1

2.2 Overview of Java RMI

The input to the points-to analysis contains the code for several components $C_1, C_2, \ldots, C_k$. The set of components will be denoted by $\mathcal{C}$. For each component $C_i \in \mathcal{C}$, the analysis takes as input a set $\mathit{cls}(C_i) = \{X_1, \ldots, X_{n_i}\}$ of Java classes. ("Classes" will refer to both Java classes and Java interfaces.) Each component is executed in a separate JVM, typically on a different physical machine. Set $\mathit{cls}(C_i)$ is the complete set of classes that may be loaded at run time in the JVM that executes component C_i. Note that an implementation of the RMI mechanism requires additional helper classes that are generated automatically from classes in $\mathit{cls}(C_i)$. For example, in the default implementation of RMI by Sun, the `rmic` compiler produces a variety of stub classes that implement the details of remote invocations. Such classes are not part of the analysis input.

For any two components C_i and C_j, sets $\mathit{cls}(C_i)$ and $\mathit{cls}(C_j)$ are not necessarily disjoint: it is possible for the same class to be loaded in the two virtual machines that

```
—– Event Channel Implementation: Component C1 —–
class MyChannel implements Channel extends UnicastRemoteObject {
 private Listener[] all; private int num;
 public MyChannel() {
  Listener[] arr = new Listener[10]; all = arr; num = 0; }
 public void add(Listener c) { all[num++] = c; }
 public void announce(Event d) {
  for(int i=0; i<num; i++) all[i].occurred(d); }
 public static void main(String[] args) {
  String channel_id = args[0]; Channel e = new MyChannel();
  Naming.bind(channel_id,e); } }
—– Event Listener Implementation: Component C2 —–
class MyListener implements Listener extends UnicastRemoteObject {
 public void occurred(Event b) {...}
 public static void main(String[] args) {
  String channel_id = args[0];
  Channel f = (Channel) Naming.lookup(channel_id);
  Listener g = new MyListener(); f.add(g);
  g = new MyListener(); f.add(g); } }
—– Event Source Implementation: Component C3 —–
class EventSource {
 public static void main(String[] args) {
  String channel_id = args[0];
  Channel h = (Channel) Naming.lookup(channel_id);
  Event k = new Event("abc"); h.announce(k); } }
```

Figure 2.2: Running example, part 2

execute C_i and C_j. One example are the classes from the standard Java libraries. We assume that the same version of the libraries is loaded in each JVM; thus, all library classes are included implicitly in $cls(C_i)$ for all $C_i \in \mathcal{C}$.

Figures 2.1 and 2.2 show the example used in the rest of the chapter; this example is based on a similar example from [22]. For simplicity, we exclude error-handling code (e.g., code related to exceptions thrown by remote invocations). The example contains events, listeners for these events, channels along which events are announced to the listeners, and event sources that create the events and send them to the channels. We consider the following configuration of components:

$$cls(C_1) = \{\texttt{Event}, \texttt{Listener}, \texttt{Channel}, \texttt{MyChannel}\}$$
$$cls(C_2) = \{\texttt{Event}, \texttt{Listener}, \texttt{Channel}, \texttt{MyListener}\}$$
$$cls(C_3) = \{\texttt{Event}, \texttt{Listener}, \texttt{Channel}, \texttt{EventSource}\}$$

In C_1, `MyChannel.main` creates an instance of remote class `MyChannel` and registers it with a naming service. (The naming service will be discussed shortly.) In C_2, `MyListener.main` uses the naming service to obtain a reference to the remote channel object, and then registers with the channel two remote listener objects. Similarly, in C_3, `EventSource.main` obtains a reference to the remote channel object and then announces an event on the channel. In `MyChannel.announce`, the channel object dispatches the event to the registered remote listeners.

2.2.1 Remote Objects, References, and Calls

A *remote class* implements the interface `java.rmi.Remote`. This is a marker interface that does not contain any methods or fields. A *remote object* is any instance of a remote class. Class `java.rmi.server.UnicastRemoteObject`, which implements `Remote`, provides default support for point-to-point object references using TCP. The simplest mechanism for creating remote classes is to subclass `UnicastRemoteObject`. Other mechanisms are also possible [44], but they are conceptually similar and are beyond the scope of our work.

A *remote reference* represents a connection between two different JVMs. Similarly to an ordinary (non-remote) object reference, a remote reference is a pointer to an object. The notion of a remote reference is an abstraction: in reality, a component has a reference to a stub object in its own JVM. Typically the existence of these stub objects is ignored, and instead RMI programming uses the abstraction of a reference pointing directly to the remote object. An invocation through a remote reference is a *remote invocation*.

Remote references can be created in several ways. For example, a remote invocation can take as an actual parameter an ordinary reference to a locally-created remote object o. As a result of the call, the remotely-invoked method takes as formal parameter a remote reference to o. Another mechanism for obtaining remote references is the use of some *naming service*. The calls to `java.rmi.Naming` in the running example illustrate such use. A naming service is a separate component whose purpose is to allow registration and lookup of remote objects. Sun's RMI implementation provides a default naming service referred to as the *RMI registry*. A call `Naming.bind(name,x)` inserts in the registry a reference to the remote object o referred to by `x`, under the given string name. In the running example, the two invocations `Naming.lookup(channel_id)` are used to initialize local variables `f` and `h` with remote references to the remote object of class `MyChannel`.

While the RMI registry provides a simple naming service, in general there could be other mechanisms for establishing initial "bootstrapping" remote references between two components [44]. Here by "bootstrapping" we mean references that are created with the help of some external mechanism (e.g., a naming service) in order to establish initial connections between components. To model such initial references, we assume that the analysis input contains information about the variables through which such references are created. For each pair of components $(C_i, C_j) \in \mathcal{C} \times \mathcal{C}$, the analysis input contains a set $I_{i \to j}$ of pairs of local variables. Each pair (v_1, v_2) represents a use of the external mechanism which results in creating remote references from v_2 in C_j to all remote objects pointed-to by v_1 in C_i. For our example, $I_{1\to2} = \{(\mathtt{e}, \mathtt{f})\}$ and $I_{1\to3} =$

$\{(\mathtt{e},\mathtt{h})\}$. Sets $I_{i\rightarrow j}$ depend on the specific mechanism used by the application. It may be possible to construct these sets automatically in some simpler cases (e.g., when using the default RMI registry). However, since in general the external mechanism for creating initial remote references could be application-specific, programmer input may be required to obtain the information in $I_{i\rightarrow j}$.

2.2.2 Call-by-Copy through Serialization

When actual parameters of a remote call are references to non-remote objects o_i, the parameter passing mechanism used is call-by-deep-copy. Objects o_i together with all other objects reachable from them are subject to serialization. This process encodes the object graph starting from o_i and recreates it in the target JVM. For example, consider the call to `announce` in `EventSource.main`. In this call the actual parameter is a (non-remote) reference to an instance o of class `Event`. The class is serializable because it implements interface `java.io.Serializable`. Fields `on` and `des` of o refer to serializable objects. Information about o and the two associated instances of `Date` and `String` is sent across the network. The "mirror image" of this object graph is created in C_1, and formal `d` in `MyChannel.announce` points to the copy of o. This process does *not* invoke the constructor of `Event` in C_1 on the copy of o. The two calls to `occurred` trigger this process again, and in the JVM for C_2 the object graph is recreated twice. Our analysis assumes that objects are serialized using the default serialization mechanism [45], and the application does not use custom serialization methods (e.g., methods such as `writeObject`); this assumption is checked by our implementation.

2.3 Points-to Analysis

This section defines the theoretical foundations for points-to analysis of RMI-based Java applications. The proposed analysis is subset-based, flow- and context-insensitive, but it should be straightforward to introduce flow sensitivity and various forms of context sensitivity.

2.3.1 Variables, Objects, and Points-to Graphs

The analysis can be defined in terms of several sets. Let Cls be the union of all sets of classes $\mathit{cls}(C_i)$ for all components C_i. We will denote by L the set of all local variables, formal parameters, and implicit parameters `this` in Cls. Similarly, let F and SF be the sets of all instance fields and static fields in Cls, respectively. Finally, let S be the set of all allocation expressions of the form `new X(..)` in Cls.

The analysis is defined in terms of a set V of *variable names* for reference variables, and a set O of *object names* for run-time objects. Figure 2.3 shows some of these names for the running example. The set V of variable names is a subset of $(L \cup \mathit{SF}) \times \mathcal{C}$. A pair $(v, C_i) \in V$ represents a local variable, a formal parameter, or a static field v in some class from $\mathit{cls}(C_i)$ such that v exists in the JVM executing C_i. The variable names will be denoted by v^i, where the superscript corresponds to the component. For the same $v \in L \cup \mathit{SF}$ there may be multiple $v^i \in V$, each one corresponding to a different C_i.

There are two categories of object names $o \in O$. First, $o = (s, C_i) \in S \times \mathcal{C}$ corresponds to run-time objects that are created by object allocation site s when this site is executed in the JVM for component C_i. Each such object is in the address space of that same JVM. Typically we will use s^i to denote such an object name; as

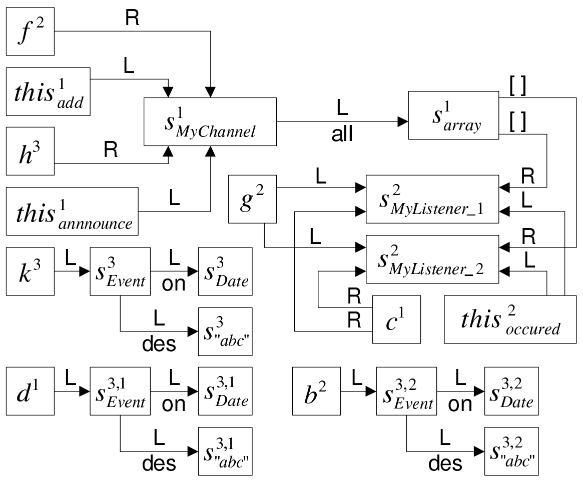


Figure 2.3: Partial points-to graph for the running example

with variable names, the superscript indicates the corresponding component. Each s^i is labeled as remote or non-remote, depending on whether it is an instance of a remote class.

Remote calls can create copies of serializable objects. We use object names $o = (s, C_i, C_j) \in S \times \mathcal{C} \times \mathcal{C}$ to represent such "copy objects". The names will typically be denoted by $s^{i,j}$. Such a name corresponds to a run-time object which exists in the JVM for component C_j and was created as a (transitive) copy of a "normal" object which was created in the JVM for C_i by allocation site s. For example, let s_{Date} be the allocation site `new Date()` in the constructor of `Event` in the running example. Name s^3_{Date} denotes the instance of `Date` which is created in C_3. Due to the remote call to `announce` from C_3 to C_1, a copy of that `Date` object is created in C_1; the name representing this copy object will be $s^{3,1}_{Date}$. The remote calls to `occurred` from C_1 to C_2 create in C_2 two run-time copies of the copy object from C_1. Both objects are transitive copies of the original object from C_3, and are represented by object name $s^{3,2}_{Date}$. Due to the properties of RMI, names $s^{i,j}$ can correspond only to non-remote objects.

The analysis builds a *points-to graph* in which the edges represent points-to relationships. An edge $(v^i, o) \in V \times O$ shows that a variable represented by v^i may point to an object represented by o. An edge $(o_1, f, o_2) \in O \times F \times O$ shows that some object represented by o_1 may store in its f field a reference to an object represented by o_2. An edge (v^i, o) could be either a *remote edge*, denoted by $(v^i, o)_R$, or a *local edge*, denoted by $(v^i, o)_L$. The same subscripts will also be applied to edges (o_1, f, o_2).

For $(v^i, o)_L$ both the variable and the target object must belong to the same JVM. Thus, such edges are either of the form $(v^i, s^i)_L$ or $(v^i, s^{k,i})_L$. Note that s^i could be a remote object (i.e., an instance of a class which implements `Remote`), but the reference to it is still an ordinary local reference. Edge $(v^i, s^j)_R$ represents a points-to relationship through a remote reference, and s^j is always a remote object.[1] Since copy objects created due to serialization cannot be remote, it is not possible to have an edge $(v^i, s^{k,j})_R$. For $(o_1, f, o_2)_L$ the two objects belong to the same JVM; either one (or both) could be a copy object $s^{k,i}$ instead of an ordinary object s^i. For $(o_1, f, o_2)_R$ object o_2 is always a remote object. Figure 2.3 shows several of the points-to edges for the running example. Edges labeled with `[]` represent points-to relationships for array elements.

[1]It is possible to have $i = j$ and the reference to be remote at the same time. For example, if C_i creates a remote object, registers it with a naming service, and then immediately looks it up, the component will obtain a remote reference to the object. Calls through this reference will be remote calls that are handled by the RMI infrastructure.

2.3.2 Effects of Program Statements

For brevity, we discuss only the following statements (our implementation handles all other kinds of statements):

- Direct assignment: $\texttt{v}_1$ `=` $\texttt{v}_2$
- Instance field write: $\texttt{v}_1\texttt{.f}$ `=` $\texttt{v}_2$
- Instance field read: $\texttt{v}_1$ `=` $\texttt{v}_2\texttt{.f}$
- Static field write: `X.f = v`
- Static field read: `v = X.f`
- Object creation: `v = new X`
- Static invocation: `w =` $\texttt{X.m(v}_1\texttt{,...,v}_k\texttt{)}$
- Instance invocation: `w =` $\texttt{v}_0\texttt{.m(v}_1\texttt{,...,v}_k\texttt{)}$

In the above statements, $\texttt{v}_i \in L$ denotes a local variable or a formal parameter (including `this`).

The analysis constructs a points-to graph G for the entire application, as well as component-specific sets of reachable methods $Reach_i$ for all $C_i \in \mathcal{C}$. In the beginning, G is empty and each $Reach_i$ contains the main method of the corresponding C_i.[2] For each statement that appears in some method from $Reach_i$ for some i, the analysis adds to G nodes and edges that represent the effects of the statement, and updates all affected sets $Reach_j$.

[2]Actually, the initialization of $Reach_i$ should also include all library methods that are executed at JVM startup. Furthermore, during the analysis $Reach_i$ should be updated with static initializers, finalizers, and `run` methods of threads. Our implementation handles these issues.

The rules for handling different statements are represented as function definitions of the form $f(G) = G'$, where G and G' are points-to graphs. The first rule **R1** considers the references that are created with the help of an external mechanism such as a naming service:

$$\begin{array}{l}\textbf{R1}\ \ \text{for each } (v^i, w^j) \in I_{i\to j} \text{ such that } v^i \text{ is a local in some} \\ m' \in \mathit{Reach}_i \text{ and } w^j \text{ is a local in some } m'' \in \mathit{Reach}_j : \\ f(G) = G \cup \{\, (w^j, o)_R \,|\, (v^i, o)_x \in G \wedge o \text{ is remote} \,\}\end{array}$$

Points-to edge $(v^i, o)_x$ could be either local or remote: the object exported by component C_i is either created locally, in which case the edge is $(v^i, o)_L$, or is obtained from some other component, in which case the edge is $(v^i, o)_R$.

Suppose the statement under consideration occurs in some method from Reach_i in component C_i. For an assignment $v_1 = v_2$, we use the following rule **R2**:

$$\textbf{R2}\ \ \text{for } v_1 = v_2 : f(G) = G \cup \{\, (v_1^i, o)_x \,|\, (v_2^i, o)_x \in G \,\}$$

The kind x of the new edge is the same as the kind of the old one ($x \in \{L, R\}$). The sources of the point-to edges are the component-specific copies v_1^i and v_2^i of v_1 and v_2.

In the following rules, *fld* represents an instance field for an object.

$$\begin{array}{l}\textbf{R3}\ \ \text{for } v_1 = v_2.\mathit{fld} : f(G) = G\ \cup \{\, (v_1^i, o_2)_x \,|\, (v_2^i, o_1)_L \in G \wedge (o_1, \mathit{fld}, o_2)_x \in G \,\} \\ \textbf{R4}\ \ \text{for } v_1.\mathit{fld} = v_2 : f(G) = G\ \cup \{\, (o_1, \mathit{fld}, o_2)_x \,|\, (v_1^i, o_1)_L \in G \wedge (v_2^i, o_2)_x \in G \,\}\end{array}$$

In reading and writing of object fields, only local points-to edges are considered because fields of remote objects are not accessible through remote references.

In the following rules, *fld* represents a static field of class X.

$$\begin{array}{l}\textbf{R5}\ \ \text{for } v = X.\mathit{fld} : f(G) = G\ \cup \{\, (v^i, o)_x \,|\, (X.\mathit{fld}^i, o)_x \in G \,\} \\ \textbf{R6}\ \ \text{for } X.\mathit{fld} = v : f(G) = G\ \cup \{\ (X.\mathit{fld}^i, o)_x \,|\, (v^i, o)_x \in G\ \}\end{array}$$

Static fields are treated similarly to local variables. Hence, these rules are essentially the same as the rules for $v_1 = v_2$, and they use the component-specific copy $X.\mathit{fld}^i$ of static field $X.\mathit{fld}$.

$$\textbf{R7}\ \ \text{for } v = \mathit{new}\ X : f(G) = G \cup \{\, (v^i, s^i)_L \,\}$$

Here $s \in S$ is the allocation site corresponding to the `new` expression. Even if the newly created object is remote (i.e., an instance of a class that implements `Remote`), the reference to it is an ordinary local reference.

$$\begin{array}{l}\textbf{R8}\ \ \text{for } w = X.m(v_1, \ldots, v_k) : f(G) = G \cup \\ \{\, (p_t^i, o)_x \mid (v_t^i, o)_x \in G \wedge 1 \le t \le k \,\} \cup \{\, (w^i, o)_x \mid (\mathit{ret}^i, o)_x \in G \,\}\end{array}$$

In this rule p_t are the corresponding formals of the called static method $X.m$. We use *ret* to denote a special artificial local in $X.m$ which is assigned all and only return values of the method; static analyses often introduce such helper variables for convenience. The effects of parameter passing and return values are essentially the same as in rule **R2**.

$$\begin{array}{l}\textbf{R9}\ \ \text{for } w = v_0.m(v_1, \ldots, v_k) : f(G) = G \cup \\ \{\, \mathit{ResolveLocal}(G, m, o, v_1^i, \ldots, v_k^i, w^i) \mid (v_0^i, o)_L \in G \,\} \cup \\ \{\, \mathit{ResolveRemote}(G, m, s^j, v_1^i, \ldots, v_k^i, w^i) \mid (v_0^i, s^j)_R \in G \,\}\end{array}$$

For calls made through local references we have

```
$\mathit{ResolveLocal}(G, m, o, v_1^i, \ldots, v_k^i, w^i)$
  let $m'(p_0, p_1, \ldots, p_k, \mathit{ret})$ be the result of $\mathit{dispatch}(o, m)$
  add $m'$ to $\mathit{Reach}_i$
  return $\{\, (p_0^i, o)_L \,\} \cup$
         $\{\, (p_t^i, o')_x \mid (v_t^i, o')_x \in G \wedge 1 \le t \le k \,\} \cup$
         $\{\, (w^i, o')_x \mid (\mathit{ret}^i, o')_x \in G \,\}$
```

The run-time target method m' is determined based on the type of o and on the compile-time target m, using helper function *dispatch* which encodes the rules for run-time virtual dispatch. The implicit formal `this` in m' is represented by p_0, and the explicit formals are $p_1, \ldots, p_k$.

For remote invocations from component C_i to a remote object s^j in component C_j, we have

```
$ResolveRemote(G, m, s^j, v_1^i, \ldots, v_k^i, w^i)$
    let $m'(p_0, p_1, \ldots, p_k, ret)$ be the result of $dispatch(s^j, m)$
    add $m'$ to $Reach_j$
    return $\{\, (p_0^j, s^j)_L \,\} \cup$
            $\{\, (p_t^j, o')_R \mid (v_t^i, o')_x \in G \wedge 1 \leq t \leq k \wedge o'$ is remote$\} \cup$
            $\{\, (w^i, o')_R \mid (ret^j, o')_x \in G \wedge o'$ is remote$\,\} \cup$
            $ResolveSerialization(G, v_1^i, \ldots, v_k^i, p_1^j, \ldots, p_k^j) \cup$
            $ResolveSerialization(G, ret^j, w^i)$
```

The invoked remote method m' in component C_j is determined based on the same rules for virtual dispatch that are used for ordinary non-remote calls [44]. The invocation creates a local points-to edge from `this` in m' to the remote object s^j. For actual parameters v_t^i that point to remote objects o', remote references to o' are created for the corresponding formals p_t^j of m'. Note that edge (v_t^i, o') could be either local or remote. If the return value of m' is a (local or remote) reference to a remote object o', the left-hand-side variable w^i at the call site starts pointing remotely to o'.

Functions *ResolveLocal* and *ResolveRemote* can be easily augmented to construct the call (multi)graph of the application. The nodes in the call graph are pairs (m, i), where method m belongs to $Reach_i$. The edges correspond to call statements: if statement st in method m in component C_i invokes method n in component C_j ($i = j$ or $i \neq j$), the call graph contains an edge from (m, i) to (n, j), labeled with st. Our implementation builds the call graph on the fly, during the analysis.

2.3.3 Modeling of Non-Remote Parameters

Function *ResolveSerialization* models parameter passing for non-remote actual parameters. Recall that for each object name s^i which represents non-remote serializable run-time objects created by allocation site s in component C_i, the analysis defines a set of object names $s^{i,j}$ for copy objects, one for each component C_j. For convenience, for each component C_j we define the following map μ_j:

- $\mu_j(s^i) = s^{i,j}$ when s^i is a non-remote serializable object created in some C_i
- $\mu_j(s^{k,i}) = s^{k,j}$ when $s^{k,i}$ is an object created in some C_i as a deserialized copy of ordinary object s^k
- $\mu_j(s^i) = s^i$, when s^i is a remote object in some C_i

Given an object name o which represents run-time objects in some component C_i, object name $\mu_j(o)$ represents the corresponding run-time objects in C_j.

The effects of a remote call $v_0.m(v_1, \ldots, v_k)$ on non-remote parameters are as follows. The object graph reachable from $v_1, \ldots, v_k$ is traversed according to the rules described below. All traversed non-remote serializable objects are serialized and recreated in the target component. This process can be described by defining a subgraph *Copied*:

- If $(v^i_t, o)_L \in G$ and o is a non-remote serializable object, then $(v^i_t, o)_L \in \mathit{Copied}$
- If $o \in \mathit{Copied} \wedge (o, \mathit{fld}, o')_L \in G$, where *fld* is a non-transient field and o and o' are non-remote serializable objects, then $(o, \mathit{fld}, o')_L \in \mathit{Copied}$
- If $o \in \mathit{Copied} \wedge (o, \mathit{fld}, o')_x \in G$, where *fld* is a non-transient field, o is a non-remote serializable object, and o' is a remote object, then $(o, \mathit{fld}, o')_x \in \mathit{Copied}$
- *Copied* is the smallest set with these properties

If a field is declared as transient, its value is not subjected to further serialization. If a non-transient field points to a remote object (either locally or remotely; $x \in \{L, R\}$), the traversal stops and the remote object is not serialized. However, if the field points to a non-remote object, serialization is attempted; if the pointed-to object is not serializable, an exception is thrown. The definition of *Copied* leads to

$$\begin{array}{l}
\mathit{ResolveSerialization}(G, v^i_1, \ldots, v^i_k, p^j_1, \ldots, p^j_k) = \\
\{\, (p^j_t, \mu_j(o))_L \mid (v^i_t, o)_L \in \mathit{Copied} \,\wedge\, o \text{ is n.r.s.} \,\} \,\cup \\
\{\, (\mu_j(o), \mathit{fld}, \mu_j(o'))_L \mid (o, \mathit{fld}, o')_L \in \mathit{Copied} \,\wedge\, o, o' \text{ are n.r.s.} \,\} \,\cup \\
\{\, (\mu_j(o), \mathit{fld}, \mu_j(o'))_R \mid (o, \mathit{fld}, o')_x \in \mathit{Copied} \,\wedge\, o \text{ is n.r.s.} \,\wedge\, o' \text{ is remote} \,\}
\end{array}$$

Here "n.r.s." stands for "non-remote but serializable". The serialization mechanism initializes a copy object (i.e., a deserialized object) not by invoking a constructor of its class, but rather by invoking the no-arguments constructor of the "lowest" non-serializable superclass. It is easy to add this invocation to the rules, and for simplicity we omit this detail from the presentation.

2.4 Analysis Algorithm

This section describes an algorithm for implementing the points-to analysis. Our approach is based on techniques proposed by Lhoták and Hendren [37] for analysis of non-distributed Java applications. We define several extensions and generalizations of their approach, in order to enable analysis of distributed programs.

2.4.1 Pointer Assignment Graph

The analysis uses a data structure referred to as a *pointer assignment graph* (PAG). Nodes in this graph represent memory locations or expressions that refer to such locations. For a name $v^i \in V$, there is a PAG node $\mathit{node}(v^i)$ corresponding to this name. There are also PAG nodes of the form $\mathit{node}(v^i.\mathit{fld})$ for each instance field fld accessed through v^i. Similarly, for each object name $o \in O$, there are PAG nodes $\mathit{node}(o)$ and $\mathit{node}(o.\mathit{fld})$.

$$\frac{v = new\ X}{node(s^i) \longrightarrow node(v^i)} \qquad \frac{v_1 = v_2}{node(v_2^i) \longrightarrow node(v_1^i)}$$

$$\frac{v_1 = v_2.fld}{node(v_2^i.fld) \longrightarrow node(v_1^i)} \qquad \frac{v_1.fld = v_2}{node(v_2^i) \longrightarrow node(v_1^i.fld)}$$

$$\frac{v = X.fld}{node(X.fld^i) \longrightarrow node(v^i)} \qquad \frac{X.fld = v}{node(v^i) \longrightarrow node(X.fld^i)}$$

Figure 2.4: Creation of PAG edges

The edges of the graph represent the flow of information between the nodes. For example, if a statement $v_1 = v_2$ belongs to some method from $Reach_i$, the PAG contains an edge $node(v_2^i) \longrightarrow node(v_1^i)$. Some of the rules for creating PAG edges are shown in Figure 2.4. Each rule $\frac{x}{y}$ should be read as "if statement x belongs to some method from $Reach_i$, then edge y is in the PAG". Whenever the analysis adds a method to $Reach_i$ (as described later), the statements in the body of that method are processed and the corresponding PAG edges are created.

A remote edge $node(v^i) \xrightarrow{remote} node(w^j)$ shows that there is a flow of values from a variable v in component C_i to a variable w in component C_j. The following rule for adding remote edges considers variables that are used to create remote references through an external mechanism such as a naming service:

$$\frac{(v^i, w^j) \in I_{i \to j}}{node(v^i) \xrightarrow{remote} node(w^j)}$$

Whenever the methods that contain locals v and w become reachable (in C_i and C_j, respectively) the new edge is added to the PAG. Two examples of such edges are given in the partial PAG shown in Figure 2.7. Edge $node(e^1) \xrightarrow{remote} node(f^2)$ shows

$$\frac{node(s^i) \longrightarrow node(v^i)}{s^i \in Pt_L(v^i)} \qquad \frac{\begin{array}{c} node(v^i) \longrightarrow node(w^i) \\ o \in Pt_x(v^i) \end{array}}{o \in Pt_x(w^i)}$$

$$\frac{\begin{array}{c} node(v^i) \longrightarrow node(w^i.\mathit{fld}) \\ o \in Pt_x(v^i) \\ o' \in Pt_L(w^i) \end{array}}{o \in Pt_x(o'.\mathit{fld})} \qquad \frac{\begin{array}{c} node(v^i.\mathit{fld}) \longrightarrow node(w^i) \\ o \in Pt_L(v^i) \\ o' \in Pt_x(o.\mathit{fld}) \end{array}}{o' \in Pt_x(w^i)}$$

$$\frac{\begin{array}{c} node(v^i) \longrightarrow node(X.\mathit{fld}^i) \\ o \in Pt_x(v^i) \end{array}}{o \in Pt_x(X.\mathit{fld}^i)} \qquad \frac{\begin{array}{c} node(X.\mathit{fld}^i) \longrightarrow node(v^i) \\ o \in Pt_x(X.\mathit{fld}^i) \end{array}}{o \in Pt_x(v^i)}$$

$$\frac{\begin{array}{c} node(v^i) \xrightarrow{remote} node(w^j) \\ o \in Pt_x(v^i) \\ o \text{ is remote} \end{array}}{o \in Pt_R(w^j)} \qquad \frac{\begin{array}{c} node(o.\mathit{fld}) \xrightarrow{remote} node(\mu_j(o).\mathit{fld}) \\ o' \in Pt_x(o.\mathit{fld}) \\ o' \text{ is remote} \end{array}}{o' \in Pt_R(\mu_j(o).\mathit{fld})}$$

Figure 2.5: Propagation of points-to information

that `f` in `MyListener.main` in component C_2 may be assigned remote references to the remote objects pointed-to by `e` in `MyChannel.main` in C_1. Similarly, edge $node(e^1) \xrightarrow{remote} node(h^3)$ is created for `h` in `EventSource.main` in C_3.

Values can also flow between components through remote calls. Remote edges are used to represent the flow of values from actual to formal parameters at such calls. For example, the remote calls to `add` in `MyListener.main` use `g` as an actual parameter. In Figure 2.7, there is a remote edge from g^2 to the corresponding formal parameter c^1 in `add`. Similarly, the remote call to `announce` in `EventSource.main` uses `k` as an actual parameter, and there is an edge from k^3 to formal d^1. The rules for creating and using such PAG edges will be described shortly.

2.4.2 Points-to Sets

PAG edges are used to propagate information about points-to relationships involving the nodes. For each PAG node $node(v^i)$ we define two points-to sets $Pt_L(v^i)$ and $Pt_R(v^i)$ representing the local and remote points-to relationships for v^i. The notation Pt_x is used to represent either one of these sets. If Pt_x occurs in both parts of a rule, x refers to the same element of $\{L, R\}$ in both parts.

The first six rules in Figure 2.5 correspond to **R2** through **R7** from Section 2.3. For example, in Figure 2.1, consider the creation of a new instance of `MyChannel` in `MyChannel.main` in C_1. The analysis uses object name $s^1_{MyChannel}$ to represent this allocation site. The rules in Figure 2.4 and Figure 2.5 result in adding this object name to $Pt_L(e^1)$. In Figure 2.7, this points-to set is shown under the PAG node representing e^1. As a result of the invocation of the constructor of `MyChannel`, Pt_L for `this` inside the constructor also contains this name. The array creation expression inside the constructor is represented by object name $s^1_{ListenerArray}$, and this name is added to $Pt_L(arr^1)$. Because of the assignment `this.all = arr` and the corresponding edge $node(arr^1) \longrightarrow node(this^1.all)$, the third rule from Figure 2.5 is applied. As a result, a PAG node $s^1_{MyChannel}.all$ is created and $s^1_{ListenerArray}$ is added to its local points-to set, as shown in Figure 2.7.

The last two rules in Figure 2.5 consider PAG edges that are labeled as "remote", and therefore represent the creation of remote references in the target component C_j. For a remote edge $node(v^i) \xrightarrow{remote} node(w^j)$, the algorithm considers all objects $o \in Pt_L(v^i) \cup Pt_R(v^i)$. If o is a remote object, it is added to $Pt_R(w^j)$. Remote propagation of values can also occur due to non-remote serializable objects. An edge $node(o.fld) \xrightarrow{remote} node(\mu_j(o).fld)$ represents the flow of values from an object o in component C_i to its corresponding copy object $\mu_j(o)$ in C_j; the definition of map μ_j was given in Section 2.3.3. As described below, such edges are created for non-remote serializable objects o and their non-transient fields *fld* to represent remote calls in which actual parameters are references to non-remote objects.

2.4.3 Processing of Calls

For a static call $w = X.m(v_1, \ldots, v_k)$, let $m(p_1, \ldots, p_k, \mathit{ret})$ be the invoked static method. If the call occurs in some method in Reach_i, the analysis adds m to the set of reachable methods and creates the corresponding PAG edges, as described by the following rule:

$$\frac{}{\begin{array}{c} m \in \mathit{Reach}_i \\ \mathit{node}(v_t^i) \longrightarrow \mathit{node}(p_t^i) \\ \mathit{node}(\mathit{ret}^i) \longrightarrow \mathit{node}(w^i) \end{array}}$$

The new PAG edges are ordinary non-remote edges because static calls take place in the same component.

The first two rules in Figure 2.6 describe the processing of instance calls. These rules are applied for a call statement $w = v_0.m(v_1, \ldots, v_k)$ whose enclosing method belongs to Reach_i. As a result of processing the statement, new edges are added to the PAG and the called method is added to the set of reachable methods for the appropriate component. Subsequent additions to the points-to sets of v_0^i also trigger applications of the rules. In the case of a remote call, PAG edges labeled as "remote" are created from actual to formal parameters; the fifth rule in Figure 2.5 is subsequently used to propagate information along such edges. A similar PAG edge is created to represent the effects of return values from remote calls. For example, consider the calls `f.add(g)` in `MyListener.main`. Since the remote points-to set of f^2 contains $s^1_{\mathit{MyChannel}}$, a remote PAG edge is added from actual parameter g^2 to

$$\frac{\begin{array}{c} o \in Pt_L(v_0^i) \\ dispatch(o, m) \text{ produces} \\ m'(p_0, p_1, \ldots, p_k, ret) \end{array}}{\begin{array}{c} m' \in Reach_i \\ o \in Pt_L(p_0^i) \\ node(v_t^i) \longrightarrow node(p_t^i) \\ node(ret^i) \longrightarrow node(w^i) \end{array}} \qquad \frac{\begin{array}{c} s^j \in Pt_R(v_0^i) \\ dispatch(s^j, m) \text{ produces} \\ m'(p_0, p_1, \ldots, p_k, ret) \end{array}}{\begin{array}{c} m' \in Reach_j \\ s^j \in Pt_L(p_0^j) \\ node(v_t^i) \xrightarrow{remote} node(p_t^j) \\ node(ret^j) \xrightarrow{remote} node(w^i) \end{array}}$$

$$\frac{\begin{array}{c} node(v^i) \xrightarrow{remote} node(w^j) \\ o \in Pt_L(v^i) \\ o \text{ is non-remote serializable} \\ \mathit{fld} \text{ is not transient} \end{array}}{\begin{array}{c} \mu_j(o) \in Pt_L(w^j) \\ node(o.\mathit{fld}) \xrightarrow{remote} node(\mu_j(o).\mathit{fld}) \end{array}} \qquad \frac{\begin{array}{c} node(o.\mathit{fld}) \xrightarrow{remote} node(\mu_j(o).\mathit{fld}) \\ o' \in Pt_L(o.\mathit{fld}) \\ o' \text{ is non-remote serializable} \\ \mathit{fld2} \text{ is not transient} \end{array}}{\begin{array}{c} \mu_j(o') \in Pt_L(\mu_j(o).\mathit{fld}) \\ node(o'.\mathit{fld2}) \xrightarrow{remote} node(\mu_j(o').\mathit{fld2}) \end{array}}$$

Figure 2.6: Processing of calls

formal c^1. The propagation of the points-to set of the actual parameter along this edge (using the fifth rule in Figure 2.5) adds object names $s^2_{MyListener1}$ and $s^2_{MyListener2}$ to $Pt_R(c^1)$, indicating that `c` in component C_1 may contain remote references to the two remote instances of `MyListener` created by C_2. The subsequent processing of the body of `add` propagates these two names to the remote points-to set of $s^1_{ListenerArray}.element$. This node represents the elements of the array; essentially, we treat the array as an instance of an artificial class which has a single field `element` pointing to all elements of the array.

The remote PAG edges created at remote calls also model the effects of object serialization for non-remote actual parameters. To illustrate this process, consider the call to `announce` in `EventSource.main`. The remote PAG edge from actual parameter h^3 to formal d^1 is used to propagate the non-remote serializable s^3_{Event} to C_1. As a result, a copy object $s^{3,1}_{Event}$ is created in C_1 based on the third rule in Figure 2.6. This copy object is added to the local points-to set of d^1. The original object s^3_{Event} has fields `on` and `des` that point to two serializable objects: s^3_{Date} and $s^3_{"abc"}$. The analysis creates remote PAG edges from $s^3_{Event}.des$ to $s^{3,1}_{Event}.des$ and from $s^3_{Event}.on$ to $s^{3,1}_{Event}.on$. The last rule in Figure 2.6 is applied to these two edges, and as a result copy objects $s^{3,1}_{Date}$ and $s^{3,1}_{"abc"}$ are created and added to the appropriate points-to sets in C_1. The subsequent call to `occurred` from C_1 to C_2 creates additional remote PAG edges, and new copy objects $s^{3,2}_{Event}$, $s^{3,2}_{Date}$, and $s^{3,2}_{"abc"}$ are created and propagated to points-to sets in C_2. In the general case, this iterative process is equivalent to function *ResolveSerialization* from Section 2.3.2.

2.4.4 Worklist Algorithm

The analysis can be implemented using a worklist algorithm which is a generalization of an algorithm for the non-distributed case [36, 37]. Several new techniques need to be introduced to this existing algorithm in order to handle remote references, remote calls, serialization, and on-the-fly call graph construction. The elements of the worklist are PAG nodes whose local or remote points-to sets have changed. When a worklist element is processed, new elements are added to the points-to sets of other PAG nodes, as defined by the rules presented above. The propagation can also result in (1) finding new reachable methods, (2) creating new PAG edges for actual-formal parameter pairs and for method return values, and (3) creating new remote PAG edges to represent the effects of serialization. New reachable methods and new PAG

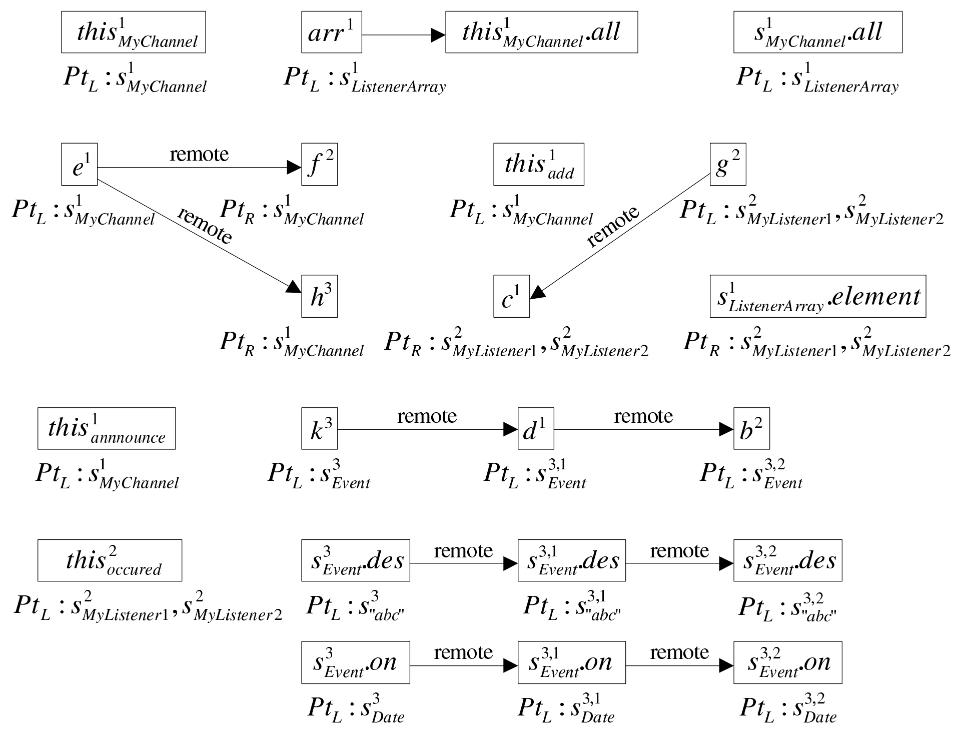


Figure 2.7: Sample PAG nodes, edges, and points-to sets.

edges trigger additional propagation. The process continues until no additional information could be inferred using the rules defined above. Additional details about the analysis algorithm are provided in the appendix at the end of the thesis.

2.5 Handling of the Standard Java Libraries

The standard Java libraries are implicitly added to the set of classes $cls(C_i)$ for each component. Based on the analysis definition presented earlier, library variables and objects will have multiple copies. For example, if a library method m has a local variable v, the points-to analysis will use multiple copies of v—that is, a separate name v^i for each component C_i. Object names are treated similarly. Our experiments with this approach showed that the majority of analysis time is spent on processing the relevant code from the libraries. Even when the size of the non-library code is small, the necessary conservative treatment of various features from the libraries (e.g., JVM startup, initialization of static fields, dynamic class loading and reflection, finalizers, etc.) requires the analysis to consider a large number of library methods as reachable. The replication of library variables and objects results in significant running time for the analysis.

To reduce running time, we designed and implemented an alternative technique for handling the standard libraries. The basic idea is to create only one replica of a library entity. For a variable v, we use a single name v^{lib} instead of multiple names v^i. For an object allocation site s, there is a single object name s^{lib}. The analysis also maintains a set of reachable methods $Reach_{lib}$, and library methods are added to this set rather than to the component sets $Reach_i$.

The rules for PAG construction can be modified in a corresponding manner. For example, if an assignment $v_1 = v_2$ is in the body of a reachable library method $m \in Reach_{lib}$, the analysis creates PAG edge $node(v_2^{lib}) \longrightarrow node(v_1^{lib})$. As another example, consider an assignment $v = X.fld$ from a static field to a local variable v. If the assignment appears in a reachable library method, edge $node(X.fld^{lib}) \longrightarrow node(v^{lib})$ should be created. On the other hand, if the assignment is discovered in some reachable non-library method $m^i \in Reach_i$, one of the following edges should be created: $node(X.fld^{lib}) \longrightarrow node(v^i)$ if X is a library class, or $node(X.fld^i) \longrightarrow node(v^i)$ otherwise. As another example, whenever a non-library method calls a library method, the actual-to-formal PAG edges are of the form $node(v^i) \longrightarrow node(p^{lib})$. Likewise, edges $node(ret^{lib}) \longrightarrow node(w^i)$ are created to represent the flow of return values.

Similar treatment is necessary for a callback from a library method to a non-library one.

The propagation of points-to sets along PAG edges follows the rules from Section 2.4. However, it is possible to filter out some of the objects that are being propagated, in cases when the component labels do not match. For example, consider some $o \in Pt_L(v^{lib})$ and an edge $node(v^{lib}) \longrightarrow node(w^i)$. If o is a non-library object s^j, it is propagated to the points-to set of w^i only if $i = j$. More generally, filtering can be used to ensure that the local points-to set of any v^i or $s^i.\mathit{fld}$ does not contain any non-library objects s_2^j for which $i \neq j$. Note that this filtering cannot be applied to remote points-to sets, because the elements of these sets could be non-library objects created in arbitrary components.

After the completion of the analysis, the local points-to sets for non-library variables and objects are processed to replace names s^{lib}. For example, if $Pt_L(v^i)$ contains s^{lib}, this object name can stand only for objects created in component C_i; thus, s^{lib} can be replaced by s^i. Note that such a replacement cannot be performed for $Pt_R(v^i)$, because in this case s^{lib} represents objects in any component, and not necessarily objects in C_i.

The full-replication approach from Section 2.3 and the zero-replication approach from above are the two endpoints of the design spectrum for handling of the standard libraries. Since the degree of replication has a direct effect on both analysis cost and analysis precision, future investigations should be performed in order to understand thoroughly this entire spectrum of cost-precision trade-offs.

2.6 Analyses for Program Understanding

Points-to information is a frequently required "enabler" for a wide range of other techniques. This section discusses briefly three specific uses of the points-to analysis for the purposes of program understanding of RMI-based applications. Of course, many other uses are possible (e.g., for program slicing, change impact analysis, etc.).

2.6.1 Call Graph

As discussed Section 2.3.2, the analysis performs on-the-fly call graph construction. The resulting graph can serve as the starting point for many other static analyses. The call graph can also be used to answer questions such as "Given a call statement st in component C_i, which methods in other components may be invoked by st, directly or transitively?". This and similar questions can enhance the understanding of the inter-method and inter-component flow of control, especially when combined with GUI-based browsing tools that display graphically the relevant parts of the call graph.

2.6.2 Data Dependencies

Consider a component C_i and some object s^i created in this component. A statement st_1^i in C_i could potentially read or write some field of s^i (either directly or transitively through its callees). Now consider a call site st_2^j in some other component C_j, and suppose st_2^j invokes some remote method from C_i. Due to the remote call, the execution of st_2^j could (transitively) read or write some field of object s^i. Thus, it is possible to have a read-write or write-read dependence between st_1^i and

st_2^j. The pair (st_1^i, st_2^j) represents a potential *inter-component data dependence* between C_i and the caller component C_j. Furthermore, consider another call site st_3^k in a third component C_k, and suppose st_3^k invokes some remote method from C_i. It is possible to have a dependence between st_2^j and st_3^k due to some field of s^i. In this case the inter-component dependence is between C_j and C_k, but the memory responsible for the dependence is in the JVM for C_i.

We have defined and implemented an algorithm which, for a given component C_i, computes all pairs (st_1^i, st_2^j) and (st_2^j, st_3^k) that correspond to potential data dependencies, as defined above. To illustrate the algorithm, consider component C_1 from Figure 2.1. The call `h.announce(k)` in `main` in C_3 creates a copy object $s^{3,1}_{Event}$ in C_1 (based on s^3_{Event} in C_3) and initializes its fields *on* and *des* with copy objects $s^{3,1}_{Date}$ and $s^{3,1}_{\text{"abc"}}$. Consider now `all[i].occurred(d)` in `announce` in C_1. Since the local points-to set of actual parameter d^1 contains $s^{3,1}_{Event}$, we can determine that due to the serialization, the values of fields $s^{3,1}_{Event}.on$ and $s^{3,1}_{Event}.des$ are read. Thus, there is a write-read dependence between the call to `announce` in C_3 and the call to `occurred` in C_1, due to memory locations $s^{3,1}_{Event}.on$ and $s^{3,1}_{Event}.des$. As another example, consider `f.add(g)` in `main` in C_2 and `h.announce(k)` in `main` in C_3. The calls to `add` results in a modification of $s^1_{MyChannel}.num$, due to `num++`. Since `announce` reads the value of $s^1_{MyChannel}.num$, there is a dependence between `f.add(g)` in C_2 and `h.announce(k)` in C_3.

The computation of such dependencies starts by examining the local points-to set at reads and writes of expressions $v.fld$. For each statement $st^i \in Reach_i$ of the form $v_1.fld = v_2$, the analysis defines a set $Mod(st^i) = \{o.fld \mid o \in Pt_L(v_1^i)\}$. Similarly, for $v_1 = v_2.fld$ we have $Use(st^i) = \{o.fld \mid o \in Pt_L(v_2^i)\}$. The reads and writes of static fields are processed in a similar fashion. The analysis also needs to take into account the reads and writes performed during object serialization and deserialization. For an instance call $w = v_0.m(v_1, \ldots, v_k)$ where $Pt_R(v_0^i) \neq \emptyset$, set $\mathit{Use}(st^i)$ should include $\{o.\mathit{fld} \mid (o, \mathit{fld}, o')_x \in \mathit{Copied} \wedge x \in \{L, R\}\}$, where *Copied* is defined in Section 2.3.3. In other words, any object field that is examined during the serialization process should be part of the *Use* set of the corresponding call. Note that we include both $o.\mathit{fld}$ that point to non-remote objects and $o.\mathit{fld}$ that point to remote objects. For the latter, even though serialization is not applied (the remote object in not serialized), the object field is still examined by the serialization mechanism. In addition, the remote call initializes the deserialized objects in the callee component C_j; therefore, $\mathit{Mod}(st^i)$ should include $\{\mu_j(o).\mathit{fld} \mid o.\mathit{fld} \in \mathit{Use}(st^i)\}$.

After the initial *Mod* and *Use* sets are computed as described above, the analysis performs iterative backward propagation of this information along the call graph edges, from callees to callers. This propagation computes the transitive read/write relationships due to method calls. Given the final solution, the intersections of sets *Mod* and *Use* for pairs of statements can be used to identify potential data dependencies.

2.6.3 Customized Serialization

One of the performance bottlenecks for RMI is the serialization and deserialization of non-remote actual parameters [42,49]. Several optimizations can be used to reduce this cost. For example, if the types of the serialized objects are unique and known in advance, specialized serialization code can be created rather than using the more expensive default serialization mechanism. As another example, if the object graph that will be serialized is always acyclic, a cheaper version of the serialization algorithm can be used, as opposed to the general version which must detect cycles. Such techniques have been shown to be quite effective in reducing the cost of serialization in RMI applications [82]. By analyzing the structure of the points-to graph produced by our analysis, it is straightforward to expose these optimization opportunities to a programmer. This information enables the introduction of customized serialization, either manually (through methods `writeObject` and `readObject` [45]), or automatically with the help of an optimization tool.

2.6.4 Other Potential Uses

Testing of distributed Java applications can be based on adequacy criteria that consider the coverage of start-to-end scenarios [5]; the corresponding execution paths can be automatically constructed (and monitored at run time) based on the call graph. As another example, the call graph and the data dependencies may be useful for static analyses that attempt to identify potential deadlocks and race conditions in RMI-based Java software.

2.7 Experimental Study

We implemented the points-to analysis algorithm using the Soot framework [81], version 2.1, and the Spark component of Soot which implements the points-to analysis techniques from [37] for non-distributed Java programs. The analysis in Spark uses state-of-the-art analysis techniques and provides the basis for our own implementation, including the handling of issues such as JVM startup, native methods, reflection, etc. The analysis was executed on a 2.8GHz Pentium4 PC with 3GB of memory,[3] using Sun's HotSpot Client JVM 1.4.2 for Windows, with maximum JVM heap size 1.5GB (option `Xmx`). The experiments were performed on the set of RMI-based Java applications listed in Table 2.1. The applications were obtained from publicly available projects and books, and represent a variety of domains.[4] For example, `auction` implements an auctioning system: clients connect to a server and place bids for items. As another example, `jodl` uses a JOb Dispatching Library to dispatch and execute tasks on different network nodes. For the applications that were GUI-based, we created and used equivalent non-GUI versions; this was done to avoid polluting the measurements of analysis running time with the time necessary to analyze the Java GUI libraries. As discussed below, the time for library analysis is the dominant factor in the points-to analysis running time; however, the library functionality is typically irrelevant to the distributed behavior of the application, which is implemented by the non-library user-level code.

We ran two different versions of the points-to analysis. The *original version* uses the algorithm from Section 2.4 and creates replicated versions of library variables and objects. The *approximate version* creates non-replicated versions for library entities (using the approximation techniques from Section 2.5) in order to reduce analysis running time, possibly at the expense of some loss of precision. Both versions employed an optimization technique for the propagation of exception objects. Since the points-to analysis is flow- and context-insensitive, it does not track precisely the flow of exception objects, and directly propagates each such object to the points-to sets of all appropriate type-compatible variables and object fields. (This conservative

[3]For other configurations, see Table 2.2

[4]We want to thank Prof. Lionel Briand from Carleton University for providing the source code for `library`.

(1) Application	(2) $\#C$	(3) Classes			(4) Orig		(5) Appr		(6) Time (sec)		
		Usr	Orig	Appr	Usr	All	Usr	All	Orig	Appr	NoDst
`filesrv`	2	7	2513	1258	14	14397	14	7209	1126.3	339.0	310.3
`stocks`	2	8	2515	1263	20	14443	21	7291	1127.7	340.2	312.3
`rmttask`	2	9	2514	1261	12	14390	12	7204	1140.3	335.9	319.9
`channel`	3	11	3767	1261	18	21587	18	7210	2368.6	336.1	308.2
`bank`	2	14	2518	1265	21	14402	21	7216	1135.0	341.5	323.7
`auction`	2	17	2524	1262	62	14475	62	7321	1164.3	343.5	309.3
`jodl`	2	29	2551	1289	125	14630	125	7387	1190.6	358.1	330.9
`jenut`	2	33	2546	1292	68	14533	68	7341	1177.6	353.9	318.7
`translator`	2	38	2526	1275	85	14454	85	7299	1206.7	368.7	329.9
`database`	2	67	2583	1304	62	14629	62	7400	1237.3	382.2	325.0
`ssl`	2	67	2587	1306	62	14642	62	7405	1227.4	381.7	377.5
`library`	3	69	3823	1317	414	22011	414	7637	2549.1	404.6	371.8

Table 2.1: Subject programs, reachable methods, and running times

treatment of exceptions is standard for subset-based points-to analysis for Java.) To reduce the cost of exception-related propagation, both versions of the analysis did not replicate exception objects on per-component basis, but rather used a single component-insensitive object name per exception object, in a manner similar to the approach from Section 2.5. It is easy to show that this optimization technique does not affect the precision of the analysis solution.

Column (2) in Table 2.1 shows the number of components C_i in each application. Column (3) contains three measurements related to the number of classes involved in the analysis. Column "Usr" represents the sum of the sizes of $cls(C_i)$, excluding library classes. Subcolumns "Orig" and "Appr" show the total numbers of classes that contain at least one reachable method in the original and the approximate version of the algorithm. The total numbers of classes for the two versions are displayed in order to give an idea on the amount of work done by these algorithms. Note that the application classes represent only a small percentage of the classes being analyzed. Column (4) describes the number of reachable methods processed by the original versions of the analysis. Column "Usr" shows the number of call graph nodes for non-library methods. Column "All" also includes number of call graph nodes for library methods, using the standard Java libraries from Sun's J2SDK 1.4.2 distribution for Windows. Similarly, column (5) and its two subcolumns describe the number of reachable methods for the approximate version. Not surprisingly, the size of the call graph is significantly larger for the original version because multiple component-specific call graph nodes m^i may correspond to the same library method m. On the other hand, the approximate version has a single call graph node m^{lib} for a reachable library method m.

Column (6) in Table 2.1 shows the running time of the two versions of the analysis[5] and the running time for the original Soot implementation of points-to analysis applied for "artificial" versions of the programs, where the RMI calls were manually replaced by ordinary calls. The times for the approximate version of our algorithm do not differ much from the running times obtained with the original Soot implementation, which shows that the extra processing overhead required for analyzing distributed RMI applications is not significant. The number of methods in columns "All", as well as the number of classes displayed in column (3), are indications of the amount of work that the analysis needs to perform, since the body of each reachable method must be processed in order to create PAG edges and to populate points-to sets. Clearly, the majority of analysis time is spent on processing the relevant code from the standard libraries. The special handling of library variables and objects in the approximate version reduces significantly the cost of the analysis. As a rough

[5]These measurements differ from the ones in [70] due to some modifications and enhancements of our implementation.

estimate of running time, the cost of the analysis is around 0.05 seconds per analyzed method.

The overall analysis time can be reduced further if the libraries are analyzed once and the computed information is reused every time an application is analyzed. Similar approaches have already been developed for points-to analysis for C (e.g., [63]), but it remains to be seen whether they can be successfully adapted to Java. This issue is not specific to analysis of RMI applications: in the static analysis community, it is well known that the cost of points-to analysis for non-distributed Java programs is dominated by the cost of processing and analyzing the relevant code from the standard Java libraries.

We gathered measurements of running time and heap size of the approximate version of our algorithm run on two different system configurations, in order to show that various system constraints do not affect the cost of the analysis. The results are presented in the Table 2.2. Both configurations are the 2.8GHz Pentium 4 presented in the beginning of this section. The first configuration has 3GB of memory and runs the JVM with the heap size of 800MB. The second has 1GB of memory and runs the JVM with 300MB heap size. The two configurations yield almost identical results in memory usage and very similar time measurements.

To gain more insight into the points-to analysis solution, we gathered a variety of measurements, as summarized in Tables 2.3 and 2.4. First, for each local variable v in a reachable non-library method m^i, we considered the sizes of $Pt_L(v^i)$ and $Pt_R(v^i)$. The average sizes of the local points-to sets are shown in sub-columns Pt_L in columns (2) and (3). Similarly, sub-columns Pt_R show the average sizes of the non-empty remote points-to sets. All of these averages exclude variables of exception types—as

(1) Application	(2) Xmx800m, 3GB		(3) Xmx300m, 1GB	
	Time(sec)	Heap(MB)	Time(sec)	Heap(MB)
`filesrv`	337.6	160.6	347.5	160.6
`stocks`	328.8	163.0	364.1	163.0
`rmttask`	336.5	160.4	347.0	160.3
`channel`	331.0	161.7	344.5	161.7
`bank`	341.5	161.0	351.2	160.9
`auction`	338.1	162.0	361.4	162.0
`jodl`	360.9	167.3	375.4	167.3
`jenut`	365.8	166.0	413.4	166.0
`translator`	364.9	160.7	380.6	160.9
`database`	380.9	166.2	396.4	166.3
`ssl`	382.0	166.8	401.0	166.8
`library`	391.0	170.7	406.8	170.7

Table 2.2: Time and memory measurements for other configurations

described above, the analysis propagates exception objects very conservatively, and the average points-to set sizes become artificially large if exception-typed variables are included in the metric. The results in columns (2) and (3) indicate that the approximate handling of the standard libraries has some impact on the precision of the analysis solution, but this impact does not appear to be particularly significant.

The rest of Table 2.3 and Table 2.4 contain additional measurements based on the points-to solution. Each of these measurements was obtained first with the original version of the analysis, and then again with the approximate version. However, in all cases the results were the same; thus, columns (4)–(10) apply to both versions. Consider an expression $v.m(\ldots)$ in some non-library method $m' \in \mathit{Reach}_i$. Column (4) shows the total number of such call sites for all components; if a call site occurs in multiple components, it is counted multiple times. Column (5) contains the number of remote call sites—that is, sites for which $Pt_R(v^i)$ was not empty. Most programs

(1) Application	(2) Original		(3) Approx		(4) Calls	(5) Emit	(6) RmtTgt
	Pt_L	Pt_R	Pt_L	Pt_R			
`filesrv`	7.3	1.0	7.5	1.0	8	5	1.0
`stocks`	87.8	1.0	88.0	1.0	12	2	1.0
`rmttask`	7.1	1.0	7.2	1.0	9	2	1.0
`channel`	5.2	1.3	5.4	1.3	11	4	1.0
`bank`	21.2	1.25	21.4	1.25	15	9	1.0
`auction`	43.4	1.0	44.3	1.0	75	5	1.0
`jodl`	121.2	1.0	121.9	1.0	158	13	1.0
`jenut`	77.3	1.23	78.3	1.72	99	36	1.0
`translator`	75.1	2.0	75.7	2.0	69	1	2.0
`database`	31.9	1.0	35.4	1.0	33	8	1.0
`ssl`	31.0	1.0	34.4	1.0	33	8	1.0
`library`	66.6	1.67	73.0	2.46	900	26	1.0

Table 2.3: Analysis precision, part 1

have multiple remote call sites, which indicates that there may be several different kinds of remote interactions between application components.

For each site from (5), we computed the number of distinct remote methods that were potentially invoked by the site. More precisely, consider $v.m(\ldots)$ in some method $m' \in \mathit{Reach}_i$. For each $s^j \in Pt_R(v^i)$, let m'' be the target method for receiver s^j. For each remote call site we computed the number of distinct targets m'' based on $Pt_R(v^i)$. Column (6) shows the average number of remote target methods over the call sites from (5). For all applications except one, the analysis resolved each remote call site to a unique target method. Since 1.0 is a lower bound for this metric, these results show that the call graphs contain precise information about the targets of remote calls.

For each remote call site, we also examined the points-to solution and determined whether there is any flow of remote references due to parameter passing. Such flow may occur when there exists an actual parameter v for which $Pt_L(v^i)$ or $Pt_R(v^i)$

(1) Application	(7) RmtRef Parm	Ret	(8) Serial	(9) OpType	(10) OpCycle
`filesrv`	0	0	0	0	0
`stocks`	1	0	2	2	2
`rmttask`	0	0	1	1	1
`channel`	2	0	4	4	4
`bank`	0	1	2	2	2
`auction`	2	0	4	4	4
`jodl`	0	0	2	2	2
`jenut`	11	9	20	20	20
`translator`	0	0	1	1	1
`database`	0	4	2	2	2
`ssl`	0	4	2	2	2
`library`	0	0	26	26	26

Table 2.4: Analysis precision, part 2

contains a remote object. Column (7), subcolumn "Param" shows the number of remote call sites at which remote references may be created in the callee due to actual parameters in the caller. Remote references also may flow as return values in the case when $Pt_L(ret^j)$ or $Pt_R(ret^j)$ contains a remote object; here ret^j denotes the artificial variable that contains the return values of the called remote method. Column (7), subcolumn "Ret" contains the number of remote call sites at which remote references may be created in the caller due to the return value from the callee. The measurements indicate that it is not unusual for RMI applications to create additional remote references at remote calls, either in the callee (through parameter passing) or in the caller (through return values). Thus, any points-to analysis needs to include techniques for handling such flow of remote references. Any subsequent analysis (e.g., change impact analysis) must also take into account this flow, based on the output of the points-to analysis.

We also considered $Pt_L(v^i)$ for an actual parameter v at a remote call site to determine whether serialization for non-remote parameters may occur at the site. Column (8) shows the number of sites from (5) for which serialization may occur due to actual parameters that point to non-remote serializable objects. These results indicate that RMI applications often take advantage of the ability to use serializable objects (and more generally, serializable object graphs) as parameters of remote calls. A points-to analysis cannot expect that the non-remote actual parameters at remote call sites are always of primitive types, and therefore the analysis must model in a general manner the possible effects of serialization. Our analysis handles this issue by introducing special PAG edges connecting the original object with its deserialized copy (Section 2.4).

The last two columns consider the remote call sites at which serialization may occur (i.e., the sites from column 8). As described in Section 2.6, points-to information can be used to provide a programmer with information about call sites at which the types of the serialized objects are unique and known in advance, or the object graph that will be serialized is always acyclic. Customized serialization at such call sites can improve the performance of the application. For each site from (8) we determined whether the type-based optimization was possible; the number of optimizable sites is shown in column (9). Similarly, for each site from (8) we determined the shape of the serialized object graph; column (10) shows the number of sites with acyclic graphs. For our subject applications, both optimizations were possible at all remote calls at which serialization is performed.

Conclusions. (1) The analysis appears to achieve high precision when modeling the semantics of remote calls. (2) The analysis suffers from the same problem exhibited by subset-based points-to analysis for non-distributed Java programs: most of the running time is spent in the standard libraries. (3) The approximate handling of the libraries can be used to reduce the running time significantly, without major reduction in analysis precision. We believe that in the current state of the art, the approximate version of the analysis is a viable choice for a relatively precise and practical points-to analysis of RMI-based Java applications.

CHAPTER 3

TYPE ANALYSIS IN THE PRESENCE OF LARGE LIBRARIES

3.1 Introduction

Interprocedural dataflow analysis is a widely-used form of static program analysis, which takes as input a program and produces information about the behavior of that program. Dataflow analysis techniques play an important role in tools for performance optimization, program understanding and maintenance, software testing, and verification of program properties.

Whole-program dataflow analysis takes as input the source code of an entire program and analyzes it as a single unit. Using whole-program analysis for modern software that is built on large libraries presents serious challenges, because traditional analysis techniques do not scale well to the size of such applications.

In applications built with library modules, the libraries typically represent a large part of the code. For example, for the whole-program type analysis described in this chapter, and the dependence analysis described in Chapter 4, roughly 93% of the methods in our benchmark programs are in the standard Java libraries. An application built with libraries can be viewed as a user component built on top of a much larger library component.

We consider a summary-based analysis [56,58] as a solution for analyzing programs built with large libraries. The code of the input program is split into a user component and a library component. The summary-based analysis processes the source code of the user component, given a precomputed summary of the library component. The technique has two steps: the first step is computing the information about the library, and the second is running the analysis that uses this information. The first step creates a summary and writes it to a file on disk. The summary contains information about the library that is relevant from the point of view of the specific analysis and is reusable for any future user code. In the second step the summary information is read from disk and used in the actual analysis of the user code. The summary-based analysis comes with no loss of precision compared to the whole-program analysis. This is because the summary contains all the information that would be obtained by analyzing the library in the whole-program version.

The method of using precomputed summaries in the analysis presents serious challenges. One challenge is the presence of information that is local to each method in the library. Although this information is necessary when the summary of the library is built, it is not needed in the analysis of a user component and should not be included in the summary. Another challenge in building the summary is the presence of polymorphic calls in the library methods. Their behavior depends on the way the client component uses the library component. For example, these calls can produce callbacks to the user code, which is not available at the time when the summary is created.

The main goal of this chapter and the next one is to define a summary-based analysis that deals with these challenges. In this chapter we consider type analysis, and in Chapter 4 we study dependence analysis. These two important kinds of analysis are examples of dataflow problems. The following items detail the content of this chapter.

- We describe whole-program type analysis, one of the two analyses we use as the basis for our summary-based approach. We formulate it as an interprocedural distributive environment (IDE) problem [68]. The IDE framework describes a large class of dataflow problems, in which dataflow functions have compact representation with efficient operations of functional meet and functional composition. Based on this formulation, we restate the analysis in terms of graph representations and operations.

- We define the summary generation analysis, and describe how we deal with the challenges posed by creating a summary that can be used effectively by a summary-based type analysis. One challenge is to represent in the summary all the information necessary for the analysis of a user component. For example, for type analysis, the summary of the library should contain enough information such that a complete call graph of the application is built, starting from the entry methods in the user code and including the library methods. In order to do this, the calls in the library methods are not abstracted away but left "untouched" in the summary. Another challenge is to have a compact summary. We represent in the summary only the information about the library that is relevant to a user component. In order to do this we eliminate information that is local to the library methods and not necessary for analyzing the user code.

- We evaluate the summary-based type analysis by implementing an algorithm for summary generation, and using the resulting summary in the analysis of 20 real-world Java programs. The results are compared with the results of running whole-program type analysis on the same benchmarks. Our experiments show that the summary-based approach provides a significant reduction in time and memory usage without affecting the analysis precision.

3.2 IDE Dataflow Problems

In *interprocedural distributive environment* (IDE) dataflow problems [68], the dataflow facts are maps ("environments") from some finite set of symbols D to lattice elements from a finite-height meet semi-lattice L. Semantic effects are represented by distributive environment transformers. The IDE class is a general category of dataflow problems, examples of which are copy-constant propagation and linear-constant propagation [68], object naming analysis [57], 0-CFA type analysis for Java [23, 30, 78], and all IFDS (interprocedural, finite, distributive, subset) problems [52] such as reaching definitions, available expressions, live variables, truly-live variables, possibly-uninitialized variables, flow-sensitive side-effects [7], some forms of may-alias and must-alias analysis [53], and interprocedural slicing [33].

3.2.1 Interprocedural Control-Flow Graph

As usual in interprocedural dataflow analysis [69], a program is represented by an interprocedural control-flow graph (ICFG) $G = (N, E)$. Graph G contains the control-flow graphs (CFGs) for the individual procedures.[6] Nodes $n \in N$ correspond to statements, and intraprocedural edges $e \in E$ represent flow of control within the

[6]We will use "procedure" to refer to procedures and methods.

same procedure. The CFG for a procedure p has an artificial start node $start_p$ and an artificial exit node $exit_p$. The start node $start_{main}$ of the main procedure is the entry point of the program. Each call is represented by two nodes: a *call-site* node and a *return-site* node. There is an interprocedural edge $e \in E$ from a call-site node to the start node of the invoked procedure p; there is also a corresponding edge $e \in E$ from $exit_p$ to the return-site node. Dataflow functions are associated with these edges to represent the effects of parameter passing and return values.

A path in G is a sequence of edges $q = [e_1, \ldots, e_k]$ such that the target of e_i is the same as the source of e_{i+1}. The dataflow function associated with q is the composition of the edge functions: $f_q = f_{e_k} \circ \ldots \circ f_{e_1}$. Not all ICFG paths represent possible executions. A *valid* path has interprocedural edges that are properly matched: each (exit,return-site) edge is matched correctly with the last unmatched (call-site,start) edge on the path [52, 68, 69].

3.2.2 Environments and Transformers

The definition of an IDE problem is based on the notions of environments and environment transformers. An *environment* is a map $D \to L$ where D is a finite set of symbols and L is a finite-height meet semi-lattice with a top element $\top$ and a meet operator $\wedge$. Let $Env(D, L)$ be the set of all environments for a given pair (D, L). The following operations are defined on $Env(D, L)$:

- The meet operator $\wedge$ extended to environments is defined as $env_1 \wedge env_2 = \lambda d.(env_1(d) \wedge env_2(d))$
- The top element in $Env(D, L)$, denoted by Ω, is $\lambda d.\top$

- For an environment $env \in Env(D, L)$, $d \in D$ and $l \in L$, $env[d \mapsto l]$ denotes an environment in which each symbol d' is mapped to the value $env(d')$, except for d which is mapped to l.

Functions of the form

$$t : Env(D, L) \rightarrow Env(D, L)$$

are environment transformers (i.e., t maps an environment to another environment). Note that environment transformers can be generalized to allow the set of symbols to change in predictable ways (particularly at calls, returns, and exceptions, to account for scope changes). To simplify the discussion, we consider only environments on a fixed set of symbols; in this approach, unused symbols can be thought of as being mapped to $\top$. An environment transformer t is *distributive*, denoted by

$$t : Env(D, L) \xrightarrow{d} Env(D, L)$$

if for every $env_1, env_2, \ldots, env_n \in Env(D, L)$ and every $d \in D$, we have

$$(t(\bigwedge_i env_i))(d) = \bigwedge_i (t(env_i))(d)$$

An instance of an *IDE problem* is defined as a tuple (G, D, L, M) where G is the program's ICFG, D and L are sets as defined above, and

$$M : E \rightarrow (Env(D, L) \xrightarrow{d} Env(D, L))$$

is a mapping that associates distributive environment transformers with the edges of G.

3.2.3 MVP Solution for an IDE Problem

For an IDE problem instance we can define the *meet-over-all-valid-paths* solution for a given node $n \in N$ as follows:

$$MVP_n = \bigwedge_q M(q)(\Omega)$$

where q is a valid path from $start_{main}$ to n. In this definition, M is extended to paths by composition:

$$M([e_1, e_2, \ldots, e_j]) = M(e_j) \circ \ldots \circ M(e_2) \circ M(e_1)$$

3.3 Whole-Program Type Analysis

This section describes the *whole-program* type analysis used as the basis of our approach for *summary-based* type analysis. The analysis is subset-based, flow- and context-insensitive, with on-the-fly call graph construction. The proposed theoretical approach is also applicable to a range of other similar analyses (e.g., field-based points-to analysis [37]).

We first define the analysis in terms of dataflow lattices and functions. Next, we show that the analysis problem is an IDE problem. Based on this formulation, we restate the analysis in terms of graph representations and operations.

3.3.1 Variables, Object Types, and Graphs

The analysis is defined in terms of several sets that characterize the input program. Let *Cls* be the set of all classes and interfaces in the program. We will denote by *LP* the set of all local variables and formal parameters, including the implicit formal parameters `this`. Similarly, let *IF* and *SF* be the sets of all instance fields and static fields in *Cls*, respectively. Since the analysis is field-based (i.e., it treats an expression `x.f` as simply `f`), the treatment of arrays is also field-based: the elements of all arrays are represented by a single analysis element, denoted by *arr_elem*. We consider *arr_elem* to be an element of set *IF*, since an array element access expression `x[]` is conceptually similar to an instance field access expression `x.f`, where `[]` can be thought of as an artificial field of an artificial class `Array`.

The solution of the analysis is defined in terms of a set V of *variable names* for reference variables, and a set T of *type names* for the types of run-time objects. Set V is a subset of $LP \cup IF \cup SF$. A special element $excp \in V$ is a placeholder name for all variables of type `java.lang.Throwable` that appear in the program. Set T represents the types of all allocation expressions of the form `new X(..)` and `new X[..]`. The analysis builds a graph in which the edges represent type relationships. An edge $(v, t) \in V \times T$ shows that a variable represented by v may point to an object of type t.

3.3.2 Input Language

Consider the following grammar with starting non-terminal $\langle AssignStmt \rangle$. This grammar defines a Java-like language, similar to a subset of the Jimple intermediate representation provided by the Soot bytecode analysis framework [81]. This grammar captures the subset of Jimple that is relevant for type analysis. Note that irrelevant features such as numeric constants, arithmetic expressions, etc. are not represented. Thus, all program variables and expressions are of reference types (i.e., class, interface, and array types in Java). We also assume that any program in this language is well-typed according to the compile-time typing rules of Java.

;; — Constants

$\langle \textit{StringConstant} \rangle$ ::= `"abc"` ;; string literal

;; — Variables

$\langle \textit{Local} \rangle$ ::= `id`

$\langle \textit{IdentityRef} \rangle$ ::= $\langle \textit{ThisRef} \rangle$ | $\langle \textit{ParameterRef} \rangle$ | $\langle \textit{CaughtExceptionRef} \rangle$

$\langle \textit{ThisRef} \rangle$ ::= `this` ;; for instance methods

$\langle \textit{ParameterRef} \rangle$ ::= $\texttt{parameter}_i$;; i-th formal parameter with $i \geq 1$

$\langle \textit{CaughtExceptionRef} \rangle$ ::= `caughtexception` ;; artificial var for a catch clause

;; — Expressions

$\langle \textit{Expr} \rangle$ ::= $\langle \textit{AnyNewExpr} \rangle$;; creation of an object

| $\langle \textit{InvokeExpr} \rangle$;; method call

| $\langle \textit{FieldRef} \rangle$;; field access

| $\langle \textit{ArrayRef} \rangle$;; array element access

| $\langle \textit{CastExpr} \rangle$;; casting

;; — Object Creation Expressions

$\langle \textit{AnyNewExpr} \rangle$ ::= $\langle \textit{NewExpr} \rangle$ | $\langle \textit{NewArrayExpr} \rangle$

$\langle \textit{NewExpr} \rangle$ ::= `new` $\langle \textit{Type} \rangle$;; create instance of the corresponding class

$\langle \textit{NewArrayExpr} \rangle$ ::= `newarray` $\langle \textit{Type} \rangle$;; create array of specified type

;; — Method Call Expressions

$\langle \textit{InvokeExpr} \rangle$ ::= $\langle \textit{StaticInvokeExpr} \rangle$;; call to a static method

| $\langle \textit{InstanceInvokeExpr} \rangle$;; call to an instance method

$\langle \textit{StaticInvokeExpr} \rangle$ ::= $\langle \textit{Method} \rangle$ `(` $\langle \textit{Parameter} \rangle_1, \ldots$ `)` ;; call to a static method

$\langle \textit{Parameter} \rangle$ ::= $\langle \textit{Local} \rangle$ | $\langle \textit{StringConstant} \rangle$

$\langle \textit{InstanceInvokeExpr} \rangle$ ::= $\langle \textit{SpecialInvokeExpr} \rangle$;; call without dynamic dispatch

| ⟨*VirtualInvokeExpr*⟩ ;; call with dynamic dispatch

⟨*SpecialInvokeExpr*⟩ ::= ⟨*Local*⟩ `.` ⟨*Method*⟩ `(` ⟨*Parameter*⟩$_1, \ldots$ `)`

⟨*VirtualInvokeExpr*⟩ ::= ⟨*Local*⟩ `.` ⟨*Method*⟩ `(` ⟨*Parameter*⟩$_1, \ldots$ `)`

;; the method is a compile-time target

;; — Other Expressions

⟨*FieldRef*⟩ ::= ⟨*StaticFieldRef*⟩ | ⟨*InstanceFieldRef*⟩

⟨*StaticFieldRef*⟩ ::= `id`

⟨*InstanceFieldRef*⟩ ::= ⟨*Local*⟩ `.` `id`

⟨*ArrayRef*⟩ ::= ⟨*Local*⟩`[x]` ;; `x` is an integer index into the array

⟨*CastExpr*⟩ ::= `cast` ⟨*Local*⟩ `to` ⟨*Type*⟩ | `cast` ⟨*StringConstant*⟩ `to` ⟨*Type*⟩

;; — Statements

⟨*Stmt*⟩ ::= ⟨*ReturnStmt*⟩ | ⟨*ThrowStmt*⟩ | ⟨*InvokeStmt*⟩ | ⟨*DefinitionStmt*⟩

⟨*ReturnStmt*⟩ ::= `return` ⟨*Local*⟩ | `return` ⟨*StringConstant*⟩ ;; return a value

⟨*ThrowStmt*⟩ ::= `throw` ⟨*Local*⟩ ;; throw exception object referenced by local

⟨*InvokeStmt*⟩ ::= ⟨*InvokeExpr*⟩ ;; call without return value

;; — Assignment Statements

⟨*DefinitionStmt*⟩ ::= ⟨*IdentityStmt*⟩ | ⟨*AssignStmt*⟩

⟨*IdentityStmt*⟩ ::= ⟨*Local*⟩ `:=` ⟨*IdentityRef*⟩

⟨*AssignStmt*⟩ ::= ⟨*Local*⟩ `:=` ⟨*SimpleRhs*⟩

| ⟨*Local*⟩ `:=` ⟨*Expr*⟩

| ⟨*ArrayRef*⟩ `:=` ⟨*SimpleRhs*⟩

| ⟨*FieldRef*⟩ `:=` ⟨*SimpleRhs*⟩

⟨*SimpleRhs*⟩ ::= ⟨*Local*⟩ | ⟨*StringConstant*⟩

The statement types of the language are:

1. `l = r`
2. `l = "abc"`
3. `l = new X`
4. `l = newarray X`
5. `l = X.fld`
6. `l = r.fld`
7. `l = r[i]`
8. `l[i] = r`
9. `l[i] = "abc"`
10. `X.fld = r`
11. `X.fld = "abc"`
12. `l.fld = r`
13. `l.fld = "abc"`
14. `l = staticinvoke method(r1, r2, ...)`
15. `l = specialinvoke r0.method(r1, r2, ...)`
16. `l = virtualinvoke r0.method(r1, r2, ...)`
17. `l = cast r to X`
18. `l = cast "abc" to X`
19. `l = caughtexception`
20. `return l`
21. `return "abc"`
22. `throw l`
23. `staticinvoke method(l1, l2, ...)`
24. `specialinvoke l0.method(l1, l2, ...)`
25. `virtualinvoke l0.method(l1, l2, ...)`

```
public class MyClass {

   private java.lang.String[] names;

   java.lang.String replaceName(java.lang.String) {
      MyClass r0;
      java.lang.String r1, r3;
      java.lang.String[] r2;

      r0 := this;
      r1 := param0;
      r2 := r0.names;
      r2[0] := r1;
      r3 := r1;
      return r3;
   }
}
```

Figure 3.1: Sample method `replaceName`

Figure 3.1 shows a sample method whose statements are based on the language described earlier. This method will be used as a running example in the rest of this section.

3.3.3 Dataflow Effects of Statements

The type analysis can be formulated as a dataflow problem with a lattice $2^{V\times T}$, partial order $\supseteq$, meet $\cup$, top element $\emptyset$, and bottom element $V \times T$. Thus, a lattice element is a graph $G \subseteq V \times T$. The semantics of an elementary statement can be represented as a dataflow function $f : 2^{V\times T} \rightarrow 2^{V\times T}$. Since we consider a flow-insensitive analysis, the dataflow functions do not perform "kills" — that is, $f(G) \supseteq G$ for any graph G. The functions for the statement types that have only an intraprocedural effect are as follows:

- `l = r`: $f(G) = G \cup \{\,(l,t) \mid (r,t) \in G\,\}$
- `l = "abc"`: $f(G) = G \cup \{\,(l, \mathit{java.lang.String})\,\}$
- `l = new X`: $f(G) = G \cup \{\,(l,t)\,\}$, where $t \in T$ is the type for the `new` expression
- `l = newarray X`: $f(G) = G \cup \{\,(l,t)\,\}$, where $t \in T$ is the type for the `newarray` expression
- `l = X.fld`: $f(G) = G \cup \{\,(l,t) \mid (\mathit{fld},t) \in G\,\}$

  Static fields are treated similarly to local variables and this rule is essentially the same as the rule for `l = r`.
- `l = r.fld`: $f(G) = G \cup \{\,(l,t) \mid (\mathit{fld},t) \in G\,\}$

  Since the analysis is field-based [37], it treats `r.fld` as `fld`.
- `l = r[i]`: $f(G) = G \cup \{\,(l,t) \mid (\mathit{arr\_elem},t) \in G\,\}$
- `l[i] = r`: $f(G) = G \cup \{\,(\mathit{arr\_elem},t) \mid (r,t) \in G\,\}$
- `l[i] = "abc"`: $f(G) = G \cup \{\,(\mathit{arr\_elem}, \mathit{java.lang.String})\,\}$
- `X.fld = r`: $f(G) = G \cup \{\,(\mathit{fld},t) \mid (r,t) \in G\,\}$
- `X.fld = "abc"`: $f(G) = G \cup \{\,(\mathit{fld}, \mathit{java.lang.String})\,\}$
- `l.fld = r`: $f(G) = G \cup \{\,(\mathit{fld},t) \mid (r,t) \in G\,\}$
- `l.fld = "abc"`: $f(G) = G \cup \{\,(\mathit{fld}, \mathit{java.lang.String})\,\}$
- `throw l`: $f(G) = G \cup \{\,(\mathit{excp},t) \mid (l,t) \in G\,\}$
- `l = caughtexception`: $f(G) = G \cup \{\,(l,t) \mid (\mathit{excp},t) \in G\,\}$

- `l = cast r to X`: $f(G) = G \cup \{\, (l,t) \,|\, (r,t) \in G \,\}$

  For simplicity, we ignore casting effects. A more precise definition of the dataflow function would be $f(G) = G \cup \{\, (l,t) \,|\, (r,t) \in G \wedge t \textit{ subtypeof } X \,\}$

- `l = cast "abc" to X`: $f(G) = G \cup \{\, (l, \textit{java.lang.String}) \,\}$

All statements presented above can be grouped in two categories:

- Assignments `l = type_of(alloc)`, in which the right-hand side of the assignment is an allocation site or a string literal. The dataflow function has the form $f(G) = G \cup \{(l,t)\}$ where $l \in V$ and $t \in T$

- Assignments `l = r`, with dataflow functions $f(G) = G \cup \{(l,t)|(r,t) \in G\}$ where $l, r \in V$ and $t \in T$

The functions for statement types that contain calls are described similarly with assignments. In the following formulations, $\textit{parm}^i_{\textit{method}}$ represents the i^{th} formal parameter of the method *method*. For a non-static method, $\textit{this}_{\textit{method}}$ is the implicit formal `this`. For a method that has non-void return type, we use $\textit{ret}_{\textit{method}}$ to denote a special artificial variable which is assigned all return values of the method.

- `l = staticinvoke method(r1, r2, ...)`:

$$f(G) = G \cup \bigcup_{i=1,2,\ldots} \{\, (\textit{parm}^i_{\textit{method}}, t) \,|\, (r_i, t) \in G \,\} \cup \{\, (l,t) \,|\, (\textit{ret}_{\textit{method}}, t) \in G \,\}$$

- `l = specialinvoke r0.method(r1, r2, ...)`

$$\begin{aligned} f(G) = G \;&\cup\; \{\, (\textit{this}_{\textit{method}}, t) \,|\, (r_0, t) \in G \,\} \\ &\cup\; \bigcup_{i=1,2,\ldots} \{\, (\textit{parm}^i_{\textit{method}}, t) \,|\, (r_i, t) \in G \,\} \\ &\cup\; \{\, (l,t) \,|\, (\textit{ret}_{\textit{method}}, t) \in G \,\} \end{aligned}$$

- `l = virtualinvoke r0.method(r1, r2, ...)`

$$\begin{aligned} f(G) = G \;&\cup\; \{\, (\mathit{this}_{\mathit{rmethod}}, t) \,|\, (r_0, t) \in G \,\} \\ &\cup \bigcup_{i=1,2,\ldots} \{\, (\mathit{parm}^i_{\mathit{rmethod}}, t) \,|\, (r_i, t) \in G \,\} \\ &\cup\; \{\, (l, t) \,|\, (\mathit{ret}_{\mathit{rmethod}}, t) \in G \,\} \end{aligned}$$

  for each run-time target method *rmethod*.

- `staticinvoke method(l1, l2, ...)`

$$f(G) = G \cup \bigcup_{i=1,2,\ldots} \{\, (\mathit{parm}^i_{\mathit{method}}, t) \,|\, (l_i, t) \in G \,\}$$

- `specialinvoke l0.method(l1, l2, ...)`

$$\begin{aligned} f(G) = G \;&\cup\; \{\, (\mathit{this}_{\mathit{method}}, t) \,|\, (l_0, t) \in G \,\} \\ &\cup \bigcup_{i=1,2,\ldots} \{\, (\mathit{parm}^i_{\mathit{method}}, t) \,|\, (l_i, t) \in G \,\} \end{aligned}$$

- `virtualinvoke l0.method(l1, l2, ...)`

$$\begin{aligned} f(G) = G \;&\cup\; \{\, (\mathit{this}_{\mathit{rmethod}}, t) \,|\, (l_0, t) \in G \,\} \\ &\cup \bigcup_{i=1,2,\ldots} \{\, (\mathit{parm}^i_{\mathit{rmethod}}, t) \,|\, (l_i, t) \in G \,\} \end{aligned}$$

  for each run-time target method *rmethod*.

The solution computed by the type analysis is a graph (i.e., a lattice element) G_{sol} such that

- $f(G_{sol}) = G_{sol}$ for each dataflow function f corresponding to some program statement

- For any G with this fixed-point property, $G_{sol} \subseteq G$

G_{sol} is computed using the call information provided by the call graph of the input program. In our implementation, the call graph is built on the fly while the analysis runs. The details of the on-the-fly call graph construction are discussed later in Section 3.3.7.

3.3.4 Type Analysis as an IDE Problem

For the type analysis described above, an environment is a map $D \to L$, where:

- D is the set of local variables and fields that appear in the program — that is, $D = V = LP \cup IF \cup SF \cup \{excp\}$.
- L is the power set of the set of object types: $L = 2^T$. L is a lattice with partial order $\supseteq$, meet $\cup$, top element $\emptyset$, and bottom element T.[7]

An environment $V \to L$ associates a set of object types (i.e., a *type set*) with a local variable, a formal parameter, or a field. The top element Ω in $Env(V, L)$ assigns an empty type set to each variable name $v \in V$: $\Omega = \lambda v.\emptyset$.

To complete the definition of an IDE dataflow problem, we need to define the mapping

$$M : E \to (Env(V, L) \xrightarrow{d} Env(V, L))$$

that assigns an environment transformer to each edge in the ICFG. At this point we consider only the *intra*procedural case, and we can assume that the program contains only one procedure (i.e., *main*) and the ICFG is the CFG of that procedure. Furthermore, since the analysis is flow-insensitive, we can treat the CFG as having

[7]Note that this lattice is different from the lattice $2^{V \times T}$ used earlier to define the type analysis.

a switch-in-loop structure, as defined in [59]. The only edges that have non-identity transformers are edges coming out of assignment statements.

- `l = alloc`: The environment transformer is $M(e) = \lambda env.env[\, l \mapsto env(l) \cup \{t\}\,]$, where $t \in T$ is the type of the allocated object
- `l = r`: The environment transformer in this case is $M(e) = \lambda env.env[\, l \mapsto env(l) \cup env(r)\,]$

Let F be the set of transformers of these two types. Let F' be the closure of F under composition and meet. We define a point-wise representation for the elements in F', that can be written in a unique canonical form.

The canonical form of a general transformer in F' is given by:

$$f = \lambda env.env \left[l_i \mapsto env(l_i) \cup T_i \cup \bigcup_{1 \le j \le k_i} env(r_{ij}) \right],\ i = 1 \ldots |V|$$

where the elements $l_i \in V$ are distinct. $T_i \subseteq T$ is a set of types that are added to the type set for l_i. Variables $r_{i,j} \in V$ for $1 \le j \le k_i$ are sources of additional flow of types to l_i (e.g., through assignments and parameter passing). For a specific l_i, set T_i could be empty, or k_i could be 0, or both. It can proved by induction that any transformer in F' can be written in the canonical form.

We have proved that these environment transformers are distributive over $\cup$. Therefore, this is an IDE dataflow problem. We have also proved that the meet-over-all-valid paths solution for the IDE formulation is exactly the same as the type analysis solution G_{sol} defined earlier. More precisely, for any CFG node n, the solution MVP_n (which is an environment) is the same as G_{sol}.

3.3.5 Graph Representations of Environment Transformers

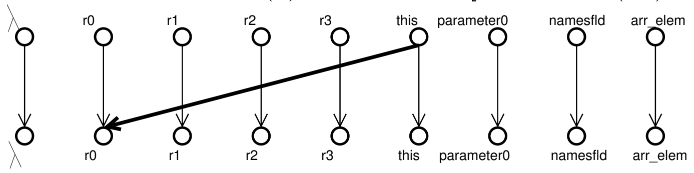
In [68], Sagiv et al. define the notion of *point-wise representation* for environment transformers. In the case of the transformers for type analysis, the unique point-wise representation for a transformer is a labeled directed graph with $2(|V|+1)$ nodes. The graph is bipartite: the vertices are decomposed into two disjoint partitions, each of size $|V|+1$, such that vertices within the same partition are not connected by edges. In each partition, $|V|$ nodes are labeled with variable names from V (one node per variable name), and one node is labeled with a special symbol Λ.

The edges in the graph are labeled with functions $L \rightarrow L$. For the specific type analysis described above, there are only three kinds of edge labels: (1) the identity function $id = \lambda x.x$, (2) the constant function $\lambda x.T_i$ for some set of object types $T_i \subseteq T$, and (3) the function $\lambda x.\emptyset$. For a general transformer in the canonical form shown in the previous section, the set of edges can be defined as follows:

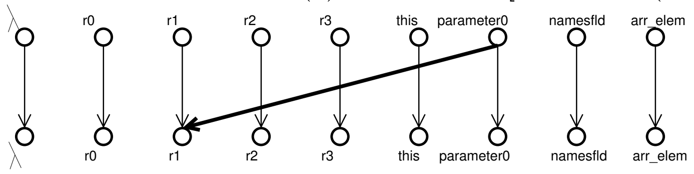
- *type 1*: $\Lambda \rightarrow l_i$, labeled with $\lambda x.T_i$
- *type 2*: $r^{i,j} \rightarrow l_i$, labeled with id, for $j = 1, \ldots, k_i$
- *type 3*: $l \rightarrow l$ for all $l \in V$, and $\Lambda \rightarrow \Lambda$, labeled with id
- *type 4*: all other edges $r \rightarrow l$ labeled with $\lambda x.\emptyset$.

Intuitively, edges of type 1 show that regardless of the current type sets, the type set of l_i should be updated with the object types in T_i. Edges of type 2 indicate that the current type set of $r^{i,j}$ should be added to the type set of l_i. Edges $l \rightarrow l$ show that the type sets never shrink. Edge $\Lambda \rightarrow \Lambda$ is needed for technical reasons.

1. `r0 := this;` $M(e) = \lambda env.env[\,r0 \mapsto env(r0) \cup env(this)\,]$



2. `r1 := param0;` $M(e) = \lambda env.env[\,r1 \mapsto env(r1) \cup env(param0)\,]$



3. `r2 := r0.names;` $M(e) = \lambda env.env[\,r2 \mapsto env(r2) \cup env(namesfld)\,]$

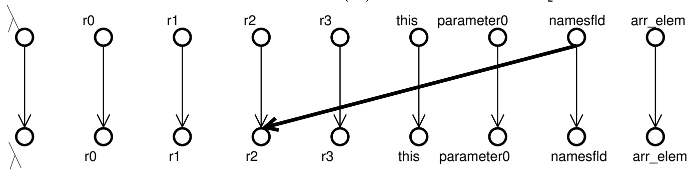


Figure 3.2: Environment transformers and graph representations for the running example (part 1)

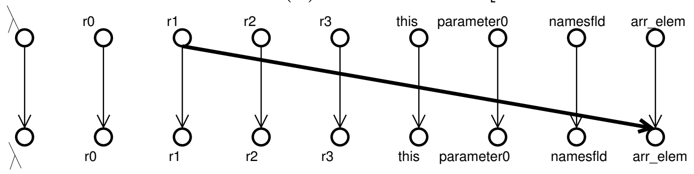
Finally, edges labeled with $\lambda x.\emptyset$ show that l does not depend on r. These edges can be omitted from the graph.

Figures 3.2 and 3.3 show the environment transformers and their graph representations of our running example. The thick lines in the figures highlight the effect of the statements on the variables. The labels for the edges in all the graphs are id.

3.3.6 Transformer Meet and Composition

Consider two transformers $f_1, f_2 \in F'$ where, as described in Section 3.3.4, F' is the closure of the set of transformers that correspond to assignments. Let $f_3 = f_1 \cup f_2$ and $f_4 = f_2 \circ f_1$.

4. `r2[0] := r1;` $M(e) = \lambda env.env[\,arr\_elem \mapsto env(arr\_elem) \cup env(r1)\,]$



5. `r3 := r1;` $M(e) = \lambda env.env[\,r3 \mapsto env(r3) \cup env(r1)\,]$

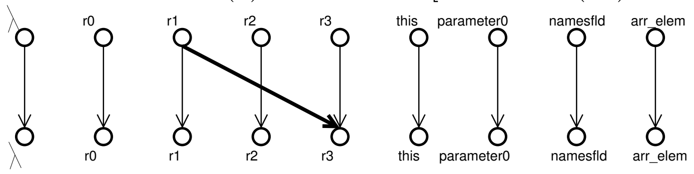


Figure 3.3: Environment transformers and graph representations for the running example (part 2)

The edges in the graph representation of f_3 are as follows:

- *type 1*: $\Lambda \to l_i$, labeled with $\lambda x.O_i^3$, where f_1 and/or f_2 have an edge $\Lambda \to l_i$. If only one of them has such an edge, the label on f_3's edge is the same. If both have such edges, with labels $\lambda x.O_i^1$ and $\lambda x.O_i^2$, then $O_i^3 = O_i^1 \cup O_i^2$.
- *type 2*: for any $l_m \to l_i$, labeled with *id*, occurring in f_1 or f_2 (or both), the edge is also in f_3.
- *type 3*: $l \to l$ and $\Lambda \to \Lambda$, labeled with *id*

The edges in the graph representation of f_4 are as follows:

- *type 1*: $\Lambda \to l_i$, labeled with $\lambda x.O_i^4$. This label satisfies the following properties:

  1. if f_1 has $\Lambda \to l_i$ with $\lambda x.O_i^1$, then $O_i^1 \subseteq O_i^4$
  2. if f_2 has $\Lambda \to l_i$ with $\lambda x.O_i^2$, then $O_i^2 \subseteq O_i^4$

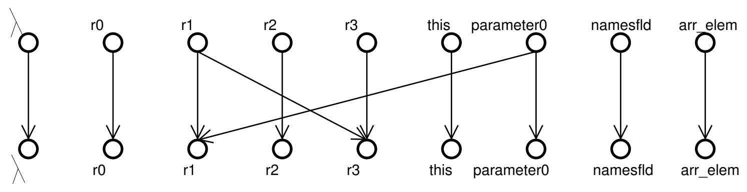


Figure 3.4: Representation of the union of the transformers for statements 2 and 5

3. if f_1 has $\Lambda \rightarrow l_m$ with $\lambda x.O^1_m$, and f_2 has $l_m \rightarrow l_i$ with id, then $O^1_m \subseteq O^4_i$
4. O^4_i is the smallest set with these properties

If a type is added to the type set of l_i by either transformer, this type should also be added to this set in f_4. In addition, if the first transformer adds a type to the type set of l_m, and the second transformer copies this type set to the type set of l_i, the composed transformer f_4 should directly add the type to the type set of l_i.

- *type 2*: $r^{i,j} \rightarrow l_i$, labeled with id:

  1. if f_1 or f_2 has $l_m \rightarrow l_i$ with id, then f_4 also has this edge
  2. if f_1 has $l_p \rightarrow l_m$ and f_2 has $l_m \rightarrow l_i$, both with id, then f_4 has $l_p \rightarrow l_i$ with id

- *type 3*: $l \rightarrow l$ and $\Lambda \rightarrow \Lambda$, labeled with id

Figure 3.4 shows the graph representation of the union of transformers corresponding statements 2 and 5 from the running example. The union of the two transformers is given by:

$$f_\cup = \lambda env.(env[\,r3 \mapsto env(r3) \cup env(r1)\,])[\,r1 \mapsto env(r1) \cup env(param0)\,]$$

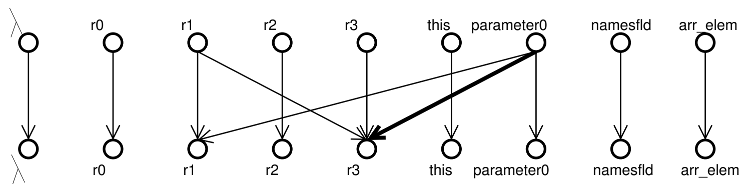


Figure 3.5: Representation of the composition of the transformers for statements 2 and 5

The composition of transformers corresponding statements 5 and 2 from the running example ($(f_5 \circ f_2)(x) = f_5(f_2(x))$ for any x) is given by the following equation:

$$f_\circ = \lambda env.(env[\,r3 \mapsto env(r3) \cup env(r1) \cup env(param0)\,])[\,r1 \mapsto env(r1) \cup env(param0)\,]$$

Figure 3.5 shows its graph representation. The thicker edge represents a transitive relation obtained by composition. Intuitively, it finds a path through which the objects pointed to by `param0` flow to `r3`.

3.3.7 Graph-Based Algorithm

Our goal is to compute G_{sol}. To achieve this, we compute the meet-over-all-valid-paths solution for the IDE formulation, using the graph-based algorithm from [68]. The key idea of this algorithm is to represent environment transformers as graphs, and to use *composition* and *meet* of transformers as graph operations, as described in the previous section.

At method level, the graph-based algorithm uses a worklist to propagate object types to the variables and fields present in the body of that method. The iterations on the worklist stop once all the object types have been propagated. Figure 3.6 shows the meet-over-all-valid-paths solution for the method of our running example. The thick lines show edges added to the graph by composition. They represent the flow

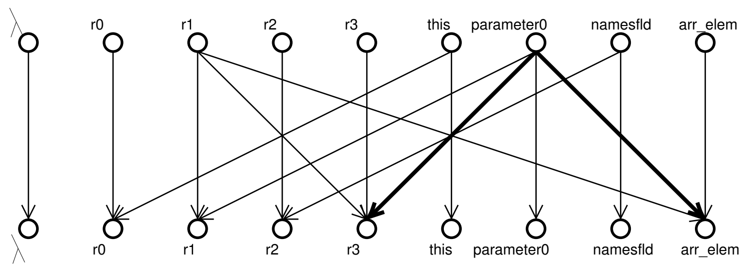


Figure 3.6: Graph representation of the MVP solution for the running example

of data through a transitive path from a variable to another (not through just one statement).

In the methods that contain calls, the algorithm propagates object types to the call targets indicated by a call graph. In our implementation, the call graph of the analyzed program is built on the fly while the analysis runs. The run-time target methods for a call site are found using virtual dispatch, based on the types of the objects to which the receiver points to. Once these run-time targets are determined, the new call graph edges are created, and the object types propagate to the target method according to the dataflow function that describes the effect of the call. If the method has formals, the types of the corresponding actuals flow to them. If the call is not static, the object types flow from the receiver in the source method to `this` of the target method. Also, if the method has non-void return type, types are propagated from the return of the target method to the local that receives the result of the call in the source method.

The algorithm is based on a worklist for the whole program. The worklist contains the variables and fields for which the type sets have object types that need to be propagated. The iterations stop when the worklist is empty, which means that all the object types have been propagated and a fixed point has been reached for the whole program.

3.4 Summary Generation Analysis

The summary generation analysis takes as input the code of the library classes and produces a complete and compact representation that is similar to the graph representation of dataflow functions described in the previous section. The summary is complete in the sense that it contains all the information needed to build the call graph and perform type analysis on any user component that uses this library. The summary is also compact, because it stores only the information about the library that is relevant to a type analysis.

3.4.1 Summary Representation

The flow of data and the type relationships inside a method are represented in the summary as a dataflow graph. The nodes of the graph represent object types or variables, and they will be referred to as *object type nodes* and *variable nodes*, respectively.

In general, object type nodes store the types of objects created with `new` expressions, or the type `java.lang.String` for string constants. A special kind of object type nodes are the ones that represent types of arrays and multi-dimensional arrays. These nodes will be referred to as *array type nodes* and correspond to `new` expressions that involve array types.

Variable nodes also are of different kinds. Variables declared in a method are represented by *local nodes.* Formal parameters of methods are represented as *formal nodes.* For non-static methods, `this` is treated as an artificial formal parameter and also represented by a formal node. The methods whose return type is not void have an additional *return node* that represents the return value of the method.

Other types of variable are the *exception node*, the *array element node*, and the *field nodes.* They are generally called *global nodes*, in order to distinguish them from the other type of variable nodes that represent entities local to a method. The exception node is unique for the program and represents the destination of all `throw` statements and the source of all `catch` statements in the program. The field nodes are variable nodes that represent fields. There is one field node for each static or instance field declared in a class. There is also a unique array element node for the program and it represents all array elements of reference type.

The edges of the graph are of two kinds: *var-to-var edges* that connect two variable nodes and *type-to-var edges* that connect object type nodes to variable nodes. A type-to-var edge represents a type relationship between the object type node and the variable node it connects. For example, if an object type node t is connected to a variable node v, then the type set of the variable represented by v contains the object type represented by t. A var-to-var edge between two variable nodes shows there is a flow of object types from the source node to the destination. This translates into an inclusion relation between the type set at the source and the type set at the destination. For example, if variable node v_1 is connected to variable node v_2, then the type sets of variables represented by v_2 include the type sets of variables represented by v_1.

We extended the method in our running example to contain call statements and the creation of an object (a string constant), in order to illustrate the technique for building the summary. Figure 3.7 shows the body of method `replaceName`, and

```
java.lang.String replaceName(java.lang.String) {
   r0 := this;
   r1 := param0;
   r2 := r0.names;
   r2[0] := r1;
   r3 := r1;
   r4 := new java.lang.String;
   specialinvoke r4.<java.lang.String:
      void <init>(java.lang.String)>("New name:");
   r5 := r4;
   r6 := virtualinvoke r5.<java.lang.String:
      java.lang.String concat(java.lang.String)>(r3);
   return r6;
}
```

Figure 3.7: Sample method `replaceName`

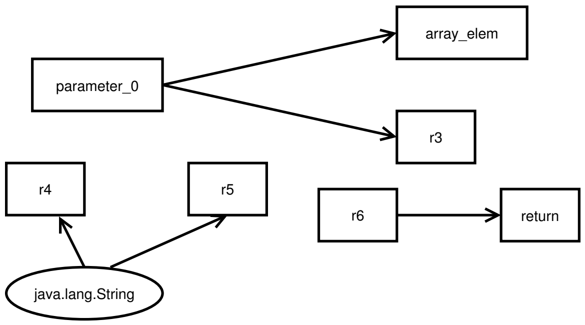
Figure 3.8 shows the summary information that is computed for this method. The summary consists of (1) the dataflow graph of the method, and (2) the information about the call sites occurring in the method, stored in "Pending calls".

In the rest of this section the running example will be used to show step by step how the summary is built.

3.4.2 Summary Generation

Generating the summary is done in three phases:

- In the first phase dataflow functions are computed for all methods in the library. These functions represent only intraprocedural type relationships.
- The second phase computes the closure of each dataflow function under functional composition and functional meet. The resulted representation shows type relationships that exist due to transitivity.



Pending calls:
```
   specialinvoke r4.<java.lang.String:
      void <init>(java.lang.String)>("New name:");

   r6 := virtualinvoke r5.<java.lang.String:
      java.lang.String concat(java.lang.String)>(r3);
```

Figure 3.8: Summary information for method `replaceName`

- In the third phase the dataflow functions are minimized. Edges of the dataflow graph are eliminated if they are not needed in the summary.

Computing initial dataflow functions

The dataflow graph of a method is built by traversing the statements in the body and adding graph edges based on the dataflow function of each statement. The graph describes only intraprocedural type relationships, therefore only the statements that do not contain calls are considered. All call statements are left in a "pending" state in which they are left unresolved until the future summary-based analysis in the presence of a specific client component. This is done because, in general, it is impossible to determine precisely which methods are the potential targets of call sites without having any information about the client component. For brevity, we will refer to call statements as *pending calls*.

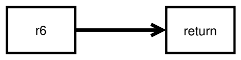


Figure 3.9: Var-to-var edge created for the return statement

Since the dataflow graph describes type relationships, the statements that do not affect any type sets can be omitted from our discussion. These statements are the returns for methods that return void or primitive types, goto statements, if and switch statements and statements related to synchronization. The remaining kinds of statements that are interesting from the point of view of the type analysis are return statements (for methods whose return type is not void), throw statements and definition statements.

For methods that return reference or array types, the return statement corresponds to a var-to-var edge in the graph. This edge connects the node representing the local being returned to the return node of the method. An exception from this rule is when the method returns a string constant instead of a local. In this case a type-to-var edge is created instead of a var-to-var edge, and it connects a string type node to the return node of the method. Figure 3.9 shows the var-to-var edge that represents the return statement of the running example.

A throw statement triggers the creation of a var-to-var edge from the local node representing the exception that is being thrown to the exception node. Var-to-var edges further connect this unique exception node to local nodes that represent destinations of catch clauses.

Recall from Section 3.3.2 that a definition statement can be either an identity statement or an assignment statement. In the case of an identity statement, which

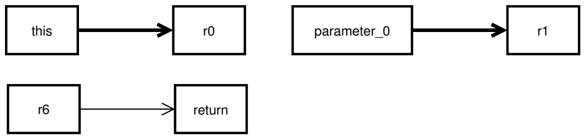


Figure 3.10: Var-to-var edges added for identity statements

reads the value of `this` or of a formal parameter of reference type, a var-to-var edge connects the corresponding formal node to the variable node that represents the local which is assigned to. Figure 3.10 shows the dataflow graph of the running example updated with the var-to-var edges created for identity statements (in thick lines).

Var-to-var edges are also created for most of the cases of assignment statements, namely the statements that assign a local variable to another local variable, to a field or to an array element, and the statements that assign a field or an array element to a local variable. The kinds of nodes created for these edges depend on the variables involved in the assignment. As described in the beginning of the section, the local variables are represented by local nodes, the field accesses are represented by field nodes and the array element accesses are represented by the unique array element node. In our example, assignment statements trigger the creation of four more edges added to the dataflow graph. They are marked with thick line in figure 3.11.

Type-to-var edges are created for the assignments that have an object creation in the right hand side. The expression that creates the object can be any `new` expression, including the creation of an array or a multi-dimensional array, or just a string constant. If it is a `new` expression with a reference type or a string constant, it is represented in the dataflow graph by a object type node. If it is the creation of an array, it is represented by an array node. In our example an object type node

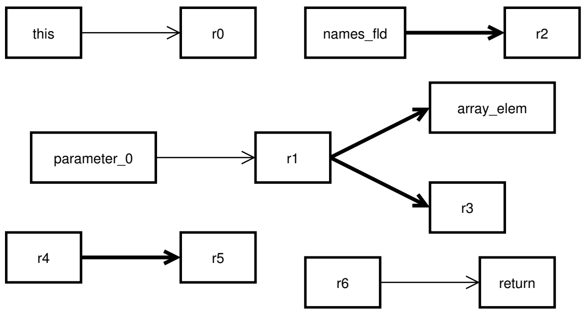


Figure 3.11: Var-to-var edges added for assignment statements

`java.lang.String` is created to represent the string constant in the assignment statement `r4 := new java.lang.String;`. The new node is connected with a type-to-var edge to the variable node $r4$. Figure 3.12 shows the result of representing the assignment to $r4$.

Recall from Section 3.3.5 that the dataflow functions can be represented as graphs, and their meet and composition can be expressed as graph operations. The initial dataflow function is the meet of all dataflow functions for the individual statements. Strictly speaking, the graph representation from Section 3.3.5 uses edges $\Lambda \to l_i$, where the edge labels are of the form $\lambda x.T_i$. In the graph representation described here, such edges are "separated" into individual edges $t_j \to l_i$ for each type $t_j \in T_i$. It is clear that this representation is equivalent to the one from Section 3.3.5, and the graph union operation described here is equivalent to the graph operation for functional meet described in Section 3.3.6.

Computing the transitive closure of the dataflow graph

The graph representation built in the first step captures a few but not all of the type relationships within the body of the method. Some type relationships are

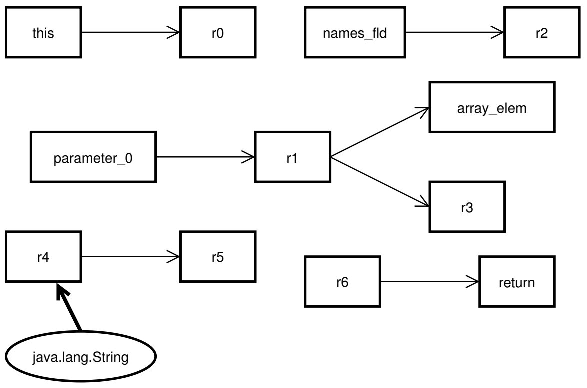


Figure 3.12: Type-to-var edge added for object creation

transitively created by multiple edges in the graph. For example, there is a type-to-var edge between the object type node `java.lang.String` and the variable node $r4$, and a var-to-var edge between $r4$ and the variable node $r5$. By considering these two edges we can determine that `java.lang.String` should be included in the type set of the variable $r5$, but there is no type-to-var edge in the graph to express this relationship directly. The complete set of type relationships can be obtained by computing the *transitive closure* of the dataflow graph. Recall from Section 3.3.6 that the composition of dataflow functions can be represented as a graph operation. It is easy to see that this graph operation is equivalent to adding "one-hop" transitive edges to the graph described here. A complete transitive closure of this graph is equivalent to computing the closure, under functional composition, of the initial dataflow function (which, as discussed above, is the meet of the functions for individual statements).

The algorithm for computing the closure of the dataflow graph has two stages. In the first stage, the closure is computed only for the var-to-var edges. In a first

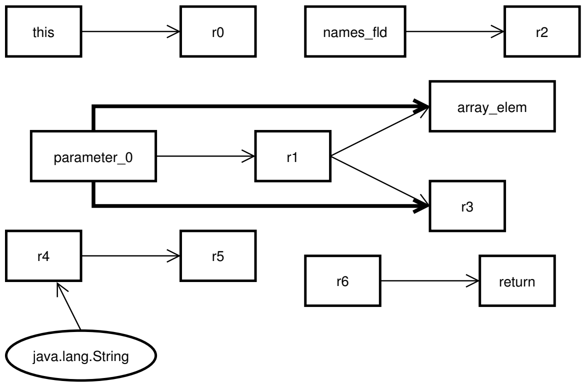


Figure 3.13: Var-to-var edges added to the closure

iteration, all the var-to-var edges from the dataflow graph of the method are considered. For each such edge, new var-to-var edges are created between the source node and the successors of the destination node, such that the successors of the destination node become direct successors of the source node. The procedure is reiterated for the source nodes whose successor sets have changed during the last iteration. The iterations stop once a fixed point is reached (no new edges are created).

The second stage of the algorithm completes the closure over type-to-var edges. For each such edge, new type-to-var edges are created to connect the object node that represents the source to the successors of the destination. In our example, an edge is added between the object type node `java.lang.String` and the successor of $r4$, which is the variable node $r5$. Figure 3.14 shows this new edge.

Minimizing the dataflow functions

The final step of the summary generation analysis is making the summary as compact as possible. This is done by eliminating unnecessary edges from the dataflow graphs. Only the edges that are relevant to the type analysis of any user component

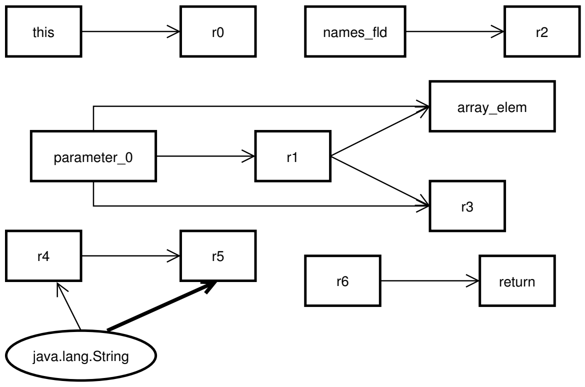


Figure 3.14: Type-to-var edge added to the closure

are kept. During a type analysis, such edges may be involved in interprocedural propagation of types. Var-to-var edges with this property have the source in a node which is a data entry point of the current method, and the destination in a node which is a data exit point of the same method. Type-to-var edges that propagate object types outside a method have the destination in a node that represents a data exit point of the current method.

A node represents a *data entry point* in the method if it is one of the following: (1) a formal parameter node of the method, (2) a global node, or (3) a local node that is assigned the return value of a call occurring in the method. A node that is a *data exit point* of a method is one of the following: (1) the return node of the method, (2) a global node, or (3) a local that represents an actual parameter or the receiver of a call occurring in the method.

The minimization algorithm iterates through all the edges of a dataflow graph of a method and checks if they are relevant and need to be kept in the summary. Checking if a var-to-var edge is relevant means checking if its source is an entry point, and if its

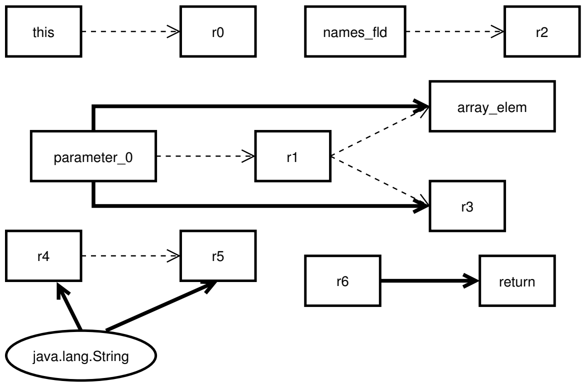


Figure 3.15: Dataflow graph of the running example before minimization

destination is an exit point. Checking if a type-to-var edge is relevant means checking if its destination is an exit point. The edges that are found redundant are deleted. Figure 3.15 shows the dataflow graph of the running example with the redundant edges marked by dotted lines. The final version of the summary information for the running example is shown in Figure 3.8.

3.5 Experimental Study

The goal of this experimental study is to investigate the following questions: (1) what is the cost of creating the summary, (2) what are the effects of the closure and minimization operations on the summary, (3) what is the size of the summary, and (4) how does a summary-based type analysis compare with its whole-program counterpart.

All experiments were performed on a Dell PowerEdge 1950 server with four Intel Xeon 2.8GHz CPUs (but our implementation is single-threaded) and 8GB of memory, running Red Hat Enterprise Linux AS release 4 (Nahant Update 4). The Java virtual machine used in the experiments was version 1.5.0_08-b03, deployed with command line flags -Xmx1536m. The -Xmx option sets the maximum heap size for the Java virtual machine to 1.5 GB.

All time measurements were obtained using `System.currentTimeMillis()`. Memory usage was obtained by taking the difference of the return values of:

- `Runtime.getRuntime().totalMemory()`, which reports the total amount of memory currently available in the JVM for current and future objects, and
- `Runtime.getRuntime().freeMemory()`, which reports an approximation to the total amount of memory currently available for future allocated objects in the JVM.

The time and memory measurements shown later are the average values out of five runs.

3.5.1 Generating the summary

We implemented the algorithm for summary generation using the Soot framework [81]. The summary information generated with our algorithm contains the following data structures: *dataflow functions* that encode intraprocedural type relationships, and *pending calls* that store information about calls.

In our experiment, the input of the summary generation were the classes from Java standard libraries from J2SE 1.4.2, which contain all classes in packages java., javax, com., COM., org., and sun. They contain a total of 10238 classes, 77190 methods and 1496003 statements. The output of the summary generation was a file that stores the summary representation of these classes, methods, and the corresponding dataflow functions.

Building the initial dataflow functions (without applying closure or minimization) for all the library methods resulted in graphs with a total of 498740 edges, which represents 33.34% of the number of original statements in the code. Computing the closure of the initial dataflow functions increased the number of edges to 698924, which represents 140.14% of the number before closure. The new dataflow functions were further minimized, which reduced the total number of edges to 313596, or 62.87% of the number of edges before closure.

Running the summary generation was completed in 16 minutes and 47 seconds, with a memory usage of 430.2MB. The final dataflow functions and the pending calls were stored on disk. The size of the resulting file was small, 12.2 MB, out of which the dataflow functions (including the pending calls) took 90.5% of the space. The rest of the space was used for storing meta information about classes (2.4%) and methods (7.1%). Out of the dataflow function space, the representation of pending calls took 51%, and the representation of dataflow edges took 12%, the rest being used for storing the representation of the dataflow nodes. Information on how the disk space is distributed may be useful for future work on creating a compact summary file.

The resulting summary file was used to evaluate experimentally the summary-based technique for type analysis.

Conclusions. (1) Summary generation for the standard Java libraries has practical cost with regard to time and memory. (2) The minimization technique significantly decreases the number of edges in the dataflow functions. (3) The size on disk is quite reasonable, even with our rather simplistic and non-compressed representation.

(1) Application	(2) User methods	(3) All methods	(4) Library methods (%)
RabbIT2	176	4978	96.46
compress	69	3855	98.21
db	66	3857	98.28
fractal	187	6037	96.90
jack	318	4114	92.27
javac	1155	5046	77.11
javacup-0.10j	317	4139	92.34
jb-6.1	140	3960	96.46
jess	465	4280	89.13
jflex-1.4.1	499	6416	92.22
jlex-1.2.6	131	3953	96.68
jtar-1.21	219	6341	96.54
mindterm-1.1.5	571	6730	91.51
mpegaudio	259	4050	93.60
muffin-0.9.3a	667	7806	91.45
raytrace	190	3988	95.23
sablecc-2.18.2	1525	5389	71.70
socksecho	168	6301	97.33
socksproxy	99	4142	97.60
violet	209	9067	97.69

Table 3.1: Subject programs, and number of reachable methods

3.5.2 Summary-based analysis

This section describes the experiments we ran in order to evaluate the performance of the summary-based technique. We implemented the summary-based type analysis with the Soot 2.2.2 framework, using the type analysis provided by the Spark module. We modified Spark such that it read information about a library class from the summary file instead of processing the corresponding class file. In the whole-program analysis (the original version), each class file is read and a Jimple representation is created for the class. In the summary-based version, this approach is still in place for the classes that belong to the user component of the analyzed program. For the rest of the classes, a representation is obtained form the summary.

We ran our implementation on a collection of 20 Java programs. Our benchmarks are presented in Table 3.1, together with some measurements reported by the type analysis indicating their size. Column "User methods" shows the number of user methods that are reachable from the main method of each program. Column "All methods" shows the total number of reachable methods, including the ones in the library classes. Column "Library methods" displays the percentage of reachable library methods out of all reachable methods for each program. The numbers show that library methods are the majority of the code being analyzed, and indicate that a large part of the processing time will be spent analyzing library code. This provides a strong motivation to use a summary-based method to optimize the running time of the type analysis.

For each benchmark, we ran the whole-program type analysis and the summary-based type analysis, and we compared the running time and memory usage for the two analyses. Table 3.2 shows the measurements for each program. Column "Time" shows the running times in seconds for the whole-program analysis and the summary-based analysis, and displays the reduction of running time in the summary-based version. Column "Memory" shows the memory usage for the two analyses, and displays the reduction of memory usage of the summary-based version. Clearly, using library information from a precomputed summary results in significant savings compared to the whole-program analysis.

We further investigated the benefits of the summary-based technique by running the analysis with the `-prejimplify` option turned on. Pre-jimplification for the whole-program analysis means that the Jimple representation for the user classes library classes is already created and available in memory when the type analysis

(1) Application	(2) Time (sec)			(3) Memory (MB)		
	Regular	Summary	Reduced(%)	Regular	Summary	Reduced(%)
`Rabbit2`	49.21	13.23	73.12	76.74	36.73	52.14
`compress`	41.87	12.04	71.23	61.89	29.12	52.95
`db`	40.59	12.47	69.28	62.12	29.20	53.00
`fractal`	55.78	15.14	72.87	91.71	43.34	52.74
`jack`	49.66	21.42	56.86	68.52	35.78	47.79
`javac`	52.52	23.59	55.08	84.49	51.23	39.36
`javacup-0.10j`	46.09	17.63	61.75	67.41	35.77	46.93
`jb-6.1`	39.60	11.61	70.69	63.32	29.98	52.66
`jess`	45.11	16.60	63.21	68.38	36.06	47.26
`jflex-1.4.1`	66.29	24.65	62.81	101.00	52.80	47.73
`jlex-1.2.6`	42.21	15.01	64.44	65.15	32.22	50.55
`jtar-1.21`	59.92	16.94	71.73	97.36	47.23	51.48
`mindterm-1.1.5`	68.94	26.81	61.11	106.10	55.80	47.41
`mpegaudio`	50.57	22.69	55.14	68.70	36.35	47.09
`muffin-0.9.3a`	72.41	24.83	65.71	119.11	61.62	48.26
`raytrace`	43.47	15.13	65.19	64.23	31.40	51.12
`sablecc-2.18.2`	52.25	22.11	57.68	83.89	51.58	38.51
`socksecho`	60.26	15.95	73.54	94.79	45.05	52.47
`socksproxy`	42.99	11.59	73.04	65.31	31.02	52.50
`violet`	81.28	20.56	74.71	135.04	64.88	51.96

Table 3.2: Comparison of the whole-program analysis and the summary-based analysis running times and memory usage

starts. In the summary-based analysis, before the type analysis algorithm is executed, pre-jimplification is applied to the user code, and the information about the library classes is read from the summary file. The results of this experiments are shown in Table 3.3. In this table, column "Time" shows the running times in seconds for the whole-program analysis and the summary-based analysis. Column "Memory" shows the memory usage for the two analyses.

When the Jimple representation already exists for library classes, the summary-based approach achieves saving in running time ranging from 0.9% to 13.06% and savings in memory usage ranging from 9.13% to 14.98%. These reductions are relatively small, and they indicate that the main reason for the overall reductions shown

(1) Application	(2) Time (sec)			(3) Memory (MB)		
	Regular	Summary	Reduced(%)	Regular	Summary	Reduced(%)
`Rabbit2`	7.89	7.53	4.52	56.52	49.86	11.79
`compress`	6.69	6.56	1.85	48.94	42.88	12.39
`db`	6.77	6.27	7.31	49.07	43.06	12.25
`fractal`	9.31	8.53	8.35	63.70	55.54	12.81
`jack`	7.11	6.74	5.31	51.73	45.65	11.76
`javac`	8.76	8.68	0.90	61.40	55.79	9.13
`javacup-0.10j`	7.31	6.96	4.80	52.33	47.16	9.89
`jb-6.1`	6.86	6.27	8.62	50.01	43.85	12.30
`jess`	7.71	7.40	4.06	53.01	46.95	11.42
`jflex-1.4.1`	9.95	9.60	3.52	68.97	60.36	12.48
`jlex-1.2.6`	6.96	6.64	4.64	50.30	44.49	11.56
`jtar-1.21`	9.51	8.94	6.00	66.64	58.29	12.52
`mindterm-1.1.5`	9.86	9.22	6.50	69.73	61.15	12.31
`mpegaudio`	7.22	6.56	9.17	50.83	44.99	11.49
`muffin-0.9.3a`	11.87	11.06	6.77	79.53	68.54	13.82
`raytrace`	7.02	6.75	3.76	50.12	44.00	12.21
`sablecc-2.18.2`	9.64	8.78	8.85	63.00	57.36	12.84
`socksecho`	9.52	8.77	7.78	65.46	57.05	12.84
`socksproxy`	6.91	6.70	3.03	50.80	44.78	11.84
`violet`	14.10	13.06	13.06	88.50	75.24	14.98

Table 3.3: Comparison of the whole-program analysis and the summary-based analysis running times and memory usage. The analyses use pre-jimplification

in Table 3.2 (randing from 55.08% to 74.71% for time and 39.36% to 53.00% for memory) are due primarily to the savings from avoiding the cost of building the intermediate representation for the library code. One potential direction for future work is to consider more aggressive summary generation in which some interprocedural propagation is performed (recalls that the current summary generation approach is purely intraprocedural).

Conclusions. (1) Summary-based type analysis can achieve significant savings of running time and memory usage, compared to its whole-program counterpart; for example, for all experimental subjects, the running time reduction was at least 55%, with average running time reduction of 70%. (2) Most savings come from avoiding the cost of reading the library code and building its intermediate representation. (3) Interprocedural techniques for summary generation should be investigated in future work, in order to eliminate some pending calls and to achieve some degree of type propagation across method boundaries.

CHAPTER 4

DEPENDENCE ANALYSIS IN THE PRESENCE OF LARGE LIBRARIES

4.1 Introduction

Analysis of dependencies between source code entities plays a fundamental role in various tools for software understanding, transformation, optimization, and verification. Program representations such as the program dependence graph (PDG) [16] explicitly show dependencies due to the flow of control and data in the analyzed program. Typically, *control dependencies* capture the relationship between conditional expressions and the statements guarded by them, while *data dependencies* represent the flow of values from one statement to another due to writes and reads of shared memory locations.

This chapter considers a particular form of whole-program interprocedural analysis of data dependencies. This analysis is based on the work by Horwitz et al. [33] on interprocedural slicing algorithms using a program representation referred to as the *system dependence graph* (SDG), which generalized the PDG by representing multiple procedures and their interactions. A key component in the construction of the SDG is the interprocedural analysis of transitive dependencies due to procedure calls; such dependencies are represented as "summary edges" in the SDG. The flow- and context-sensitive analysis of transitive dependencies can be formulated in the IDE dataflow analysis framework [53, 68]. The specific version of this analysis considered in our work is based on the following three restrictions: (1) control dependencies are not considered, (2) dependencies due to static or instance fields are not considered, and (3) exceptional flow of control is not considered. Generalizing our approach to remove the first two restrictions is conceptually simple, and does not introduce any significant new issues. The handling of exceptions could be done, for example, using the approach from [72].

In the whole-program dependence analysis considered in this chapter, the solution for each method m is a set of formal parameters of m on which the return value of m may have (transitive) data dependencies. The dependence information is computed in three phases:

- In the first phase, an intraprocedural def-use analysis is used, based on the classical reaching definitions analysis.

- In the second phase, intraprocedural dependence information is computed, based on the dependencies from the first phase and their transitive closure.

- In the third phase, interprocedural dependencies are determined based on the solution from the second phase.

We first present the whole-program version of the analysis, and then describe a summary generation approach and a subsequent summary-based analysis. The input language is the same as in Section 3.3.2, and we reuse the notation from Section 3.3.1 in the analysis description.

4.2 Whole-Program Dependence Analysis

The whole-program analysis described in this section is the starting point for our summary-based dependence analysis.

4.2.1 Phase 1: Intraprocedural Def-Use Analysis

Intraprocedural reaching definitions analysis [1] is the first step in calculating dependence information. This analysis computes a set of reaching definitions for each statement inside a method body. Let the set of all definition statements in a method be denoted by *Defs*. These statements correspond to non-terminal ⟨*DefinitionStmt*⟩ from the grammar in Section 3.3.2, in cases where the left-hand side of the definition statement is a local variable. Since we do not consider dependencies due to fields, definitions with left-hand side expression ⟨*FieldRef*⟩ (or the related ⟨*ArrayRef*⟩) are ignored. We also assume that each non-void method has exactly one return statement. If a method has more than one return statement, we consider the equivalent version in which each return statement is replaced by an assignment to a unique artificial return variable of the method, and then the flow of control is directed to the statement that returns this variable. The newly created assignments are treated like regular definitions. The CFGs used in our implementation are built in this manner.

For a local variable l, Defs_l represents the set of all definitions of l. The lattice L for the problem is the power set of *Defs*, with partial order $\supseteq$, meet operation $\cup$, top element $\top = \emptyset$, and bottom element $\bot = \mathit{Defs}$. Let l be a local variable and $d \in \mathit{Defs}_l$ be a definition that assigns a value to l. The dataflow function for any CFG edge (d, s) is $f_d(X) = (X - \mathit{Defs}_l) \cup \{d\}$, where $X \subseteq \mathit{Defs}$. Function f_d shows that all definitions of l are killed and replaced with d in the set of reaching definitions for s. For all CFG edges starting in statements that are not definitions of local variables, the dataflow function is id. A standard fixed-point algorithm can be used to compute the set of all definitions reaching a CFG node n. It is straightforward to state the reaching definitions problem as an IDE problem [68]; for brevity, we omit this formulation.

The intraprocedural def-use analysis calculates direct dependencies between the statements of a method without considering the effects of calls. Consider a statement s_1 which is a definition of a local variable l, and a statement s_2 at which the value of l is read. If s_1 belongs to the set of reaching definitions for s_2, the def-use analysis solution will contain the triple (s_1, s_2, l). A use of l is one of the following statements:

- $\langle$*ReturnStmt*$\rangle$ when the argument is l.
- $\langle$*InvokeStmt*$\rangle$, when any actual argument of the call is l. If the call is an $\langle$*InstanceInvokeExpr*$\rangle$ and l is a reference to the receiver object, this is also a use of l.
- $\langle$*AssignStmt*$\rangle$, when l appears on the right-hand side of the assignment.

4.2.2 Phase 2: Intraprocedural Dependence Analysis

Recall that dependence analysis solution for each method m is a set of formal parameters of m on which the return value of m may have transitive data dependencies. In phase 2 of the analysis, such relationships are computed purely intraprocedurally, without taking into account the effects of procedure calls. Interprocedural propagation is performed during phase 3.

Intraprocedural Dependence Analysis as an IDE problem: Simplified Case

To make the description of phase 2 easier to read, we will separate the explanation into two parts. In the first part, we consider a simplified case in which the method under analysis does not contain calls. In the second part, we discuss the more general case in which calls are present in the analyzed method.

In the absence of calls, the analysis solution for each CFG node n is a set of pairs (l, s) where l is a local variable and s is an $\langle \mathit{IdentityStmt} \rangle$ representing an assignment from a formal parameter to some local variable. These identity statements are the only statements at which formal parameters occur (Section 3.3.2). A pair (l, s) at n shows that the value of l used inside n is directly or transitively dependent on the formal parameter appearing on the right-hand side of s. The only variables l for which pairs (l, s) exist in n's solution are variables used inside n — this constraint is enforced by the dataflow functions described below. Given an analysis solution, the final dependence information can be obtained by considering all pairs (l, s) computed for the return statement of the CFG.

Instead of performing the analysis on the method's CFG, it can be performed on a "reduced" CFG in which edges correspond to the output of the def-use analysis. Recall that this output contains a set of triples (s_1, s_2, l). Each such triple corresponds to one edge in the reduced CFG. Figure 4.1 shows the reduced CFG of the sample method `replaceName` from in Figure 3.7. The dataflow function associated with an edge (s_1, s_2) can be defined based on the structure of statement s_1. Each such function takes as input a set of variable-statement pairs and produces a new set of variable-statement pairs. The cases to be considered for s_1 are as follows:

- `l = formal`: $f(X) = \{ (l, s_1) \}$ if statement s_1 is an identity statement

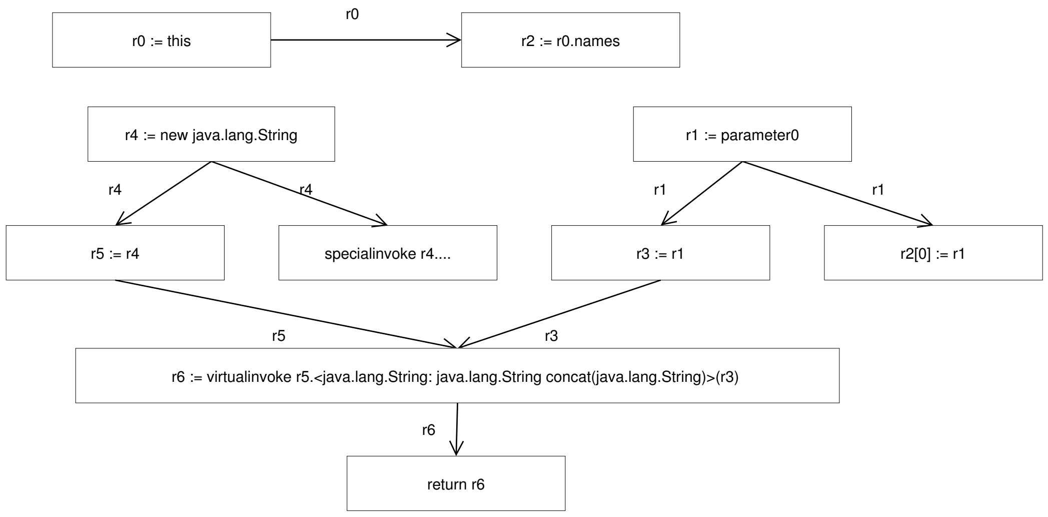

r0 := this
r0
r2 := r0.names
r4 := new java.lang.String
r1 := parameter0
r4
r4
r1
r1
r5 := r4
specialinvoke r4....
r3 := r1
r2[0] := r1
r5
r3
r6 := virtualinvoke r5.<java.lang.String: java.lang.String concat(java.lang.String)>(r3)
r6
return r6

Figure 4.1: Reduced CFG for the running example

- `l = r`: $f(X) = \{\, (l, s) \mid (r, s) \in X \,\}$ if s_1 is not an identity statement.
- `l = "abc"`: $f(X) = \emptyset$
- `l = new X`: $f(X) = \emptyset$
- `l = newarray X`: $f(X) = \emptyset$
- `l = X.fld`: $f(X) = \emptyset$
- `l = r.fld`: $f(X) = \{\, (l, s) \mid (r, s) \in X \,\}$
- `l = r[i]`: $f(X) = \{\, (l, s) \mid (r, s) \in X \vee (i, s) \in X \,\}$
- `l[i] = r`: $f(X) = \emptyset$
- `l[i] = "abc"`: $f(X) = \emptyset$
- `X.fld = r`: $f(X) = \emptyset$

- `X.fld = "abc"`: $f(X) = \emptyset$
- `l.fld = r`: $f(X) = \emptyset$
- `l.fld = "abc"`: $f(X) = \emptyset$
- `throw l`: $f(X) = \emptyset$
- `l = caughtexception`: $f(X) = \emptyset$
- `l = cast r to X`: $f(X) = \{\, (l, s) \,|\, (r, s) \in X \,\}$
- `l = cast "abc" to X`: $f(X) = \emptyset$

The dataflow functions can be written in one of the following general forms:

- Identity statement `l = formal`: $f(X) = \{\, (l, \texttt{l = formal}) \,\}$
- Other relevant assignments `l = no_call_expr(r1,r2,...)`:

  $f(X) = \{\, (l, s) \mid \exists i \ s.t. \ (r_i, s) \in X \,\}$
- All other statements: $f(X) = \emptyset$

In the equations shown above, `no_call_expr(r1,r2,...)` represents a generic expression that does not contain method calls, and uses several variables.

To state this analysis in the IDE framework, we define an environment as a map $V \rightarrow 2^I$, where V is the set of all local variables in the method, and $I \subseteq \mathit{Defs}$ is the set of all identity statements. An environment associates a variable with the identity statements that read the formal parameters on which the variable is transitively dependent. The top element Ω in $Env(V, 2^I)$ assigns an empty set to each variable. We also define the mapping

$$M : E \rightarrow (Env(V, 2^I) \xrightarrow{d} Env(V, 2^I))$$

that assigns an environment transformer to each edge in the reduced CFG. Below are the environment transformers for all CFG edges:

- For an edge e that comes out of an identity statement s of the form `l = formal`:

  $M(e) = \lambda env.\Omega[l \mapsto \{s\}]$

- For an edge e that comes out of an assignment `l = no_call_expr(r1,r2,...)`:

  $M(e) = \lambda env.\Omega[l \mapsto env(r_1) \cup env(r_2) \cup \ldots]$

- For all other edges e: $M(e) = \lambda env.\Omega$

These environment transformers are distributive, and therefore this formulation of dependence analysis is an IDE problem. The solution computed for a CFG node n can be obtained as the meet-over-all-paths (MOP) solution, constructed through transformer composition and meet. For the single statement `return l` in the method, the MOP solution is of the form $\lambda env.\Omega[l \mapsto \{s_1, s_2, \ldots\}]$ where s_i are identity statements. This solution defines all transitive data dependencies between the return value of the procedure and its formal parameters.

Intraprocedural Dependence Analysis as an IDE problem: General Case

Consider the more general case in which calls are present in the analyzed method. In a purely intraprocedural analysis, such calls are left in "pending state" and their effects are not modeled by the analysis. In this case, in addition to dependencies on the formals used at identity statements, dependencies also exist on the return values of such pending calls. Thus, the analysis solution for each CFG node n is a set of pairs (l, s) where l is a local variable and s is either an identity statement or a pending call. A pair (l, s) at n, where s is an identity statement, still shows that the value of l used inside n is directly or transitively dependent on the formal parameter appearing on the right-hand side of s. A pair (l, s) at n, where s a pending call, shows that the value of l used inside n is directly or transitively dependent on the return value of s. As before, the only variables l for which pairs (l, s) exist in n's solution are variables used inside n.

For each edge $e = (s_1, s_2)$ in the reduced CFG, the set of cases to be considered for s_1 also includes the following:

- `l = staticinvoke method(r1,r2,...)`: $M(e) = \lambda env.\Omega[l \mapsto \{s_1\}]$
- `l = specialinvoke/virtualinvoke r0.method(r1,r2,...)`: $M(e) = \lambda env.\Omega[l \mapsto \{s_1\}]$

Note that in both cases the dependencies of l due to r_i are "lost", and interprocedural analysis is needed to fully track all such dependencies. With this generalization, a MOP solution can still be computed for each CFG node. For any statement `return l`, the MOP solution is $\lambda env.\Omega[l \mapsto \{s_1, s_2, \ldots\}]$ where s_i are identity statements or pending calls. For any pending call `l = staticinvoke method(r1,r2,...)`, the MOP solution is $\lambda env.\Omega[r_1 \mapsto \{s_1^1, s_2^1, \ldots\}][r_2 \mapsto \{s_1^2, s_2^2, \ldots\}]\ldots$ where s_i^j are identity statements or pending calls. The MOP solution for an instance call has a similar form. The set of MOP transformers for the return statement and for all pending calls is the final output of the intraprocedural dependence analysis.

4.2.3 Phase 3: Interprocedural Dependence Analysis

Once phase 2 has performed complete intraprocedural propagation of information, phase 3 considers interprocedural dependencies. The dependence information of a method that contains calls is computed using the dataflow functions of its call statements. The dataflow function of a call is obtained using the dependence information of all the methods that are potential run-time targets of the call. Therefore, in order to define the dataflow function for a call, we need to have the dependence information of all call targets, i.e. we need to know which parameters affect the return value for each target.

The dependence information of all methods in the program, including those that contain calls, is obtained by iterating over all methods in the call graph, starting with the ones that do not contain any calls. The dependencies for methods with no calls are computed using the technique outlined in the previous section. These dependencies are then used to compute the dependencies of the methods' callers, and so on, until the dependence information for all methods has been computed. The algorithm is presented in detail in Section 4.2.4. The output of this algorithm is the meet-over-all-valid-paths solution (for interprocedurally valid paths with properly matched calls and returns) for each return statement in the program. Thus, for any statement `return l`, the solution is of the form $\lambda env.\Omega[l \mapsto \{s_1, s_2, \ldots\}]$ where s_i are identity statements in the same method. These solutions define all transitive data dependencies between the return value of the method and its formal parameters, including dependencies due to calls.

The dataflow functions for call statements are shown below; these functions are associated with the CFG edges coming out of such statements.

- `l = staticinvoke method(r1,r2,...)`:

$$f(X) = \bigcup_{i=1,2,\ldots} \{ (l,s) \,|\, (r_i,s) \in X \,\wedge\, (\mathit{formal}^i_{\mathit{method}}, \mathit{ret}_{\mathit{method}}) \in S_{\mathit{method}} \}$$

- `l = specialinvoke r0.method(r1,r2,...)`

$$f(X) = \bigcup_{i=0,1,\ldots} \{ (l,s) \,|\, (r_i,s) \in X \,\wedge\, (\mathit{formal}^i_{\mathit{method}}, \mathit{ret}_{\mathit{method}}) \in S_{\mathit{method}} \}$$

- `l = virtualinvoke r0.method(r1,r2,...)`

$$f(X) = \bigcup\nolimits_{i=0,1,\ldots} \{ (l,s) \,|\, (r_i,s) \in X \; \wedge \exists \mathit{rmethod}\ s.t.\ (\mathit{formal}^i_{\mathit{rmethod}}, \mathit{ret}_{\mathit{rmethod}}) \in S_{\mathit{rmethod}} \}$$

  where *rmethod* is a run-time target method of the call.

Here $(\mathit{formal}^i_{\mathit{method}}, \mathit{ret}_{\mathit{method}}) \in S_{\mathit{method}}$ shows that the return value of the method may depend transitively on its i-th formal parameter — in other words, for the solution S_{method} for the target method, and for the return statement *ret* of the form `return l` in that method, the environment transformer S_{ret} for *ret* has $S_{\mathit{ret}}(l)$ which contains the identity statements that reads the value of $\mathit{formal}^i_{\mathit{method}}$.

Figure 4.2 shows the reduced CFG for the sample method. Edges are labeled with environment transformers. For the virtual call whose return value is assigned to `r6`, in this example we have assumed that there are target methods for which the return value depends both on the formal parameter `r3` and on the receiver object pointed-to by `r5`. Thus, the transformer corresponding to outgoing edge represents the dependence of `r6` on `r3` and `r5`. In the final solution computed by the interprocedural analysis, the return value of the method depends on the value of formal parameter `parameter0`, but not on the value of `this`.

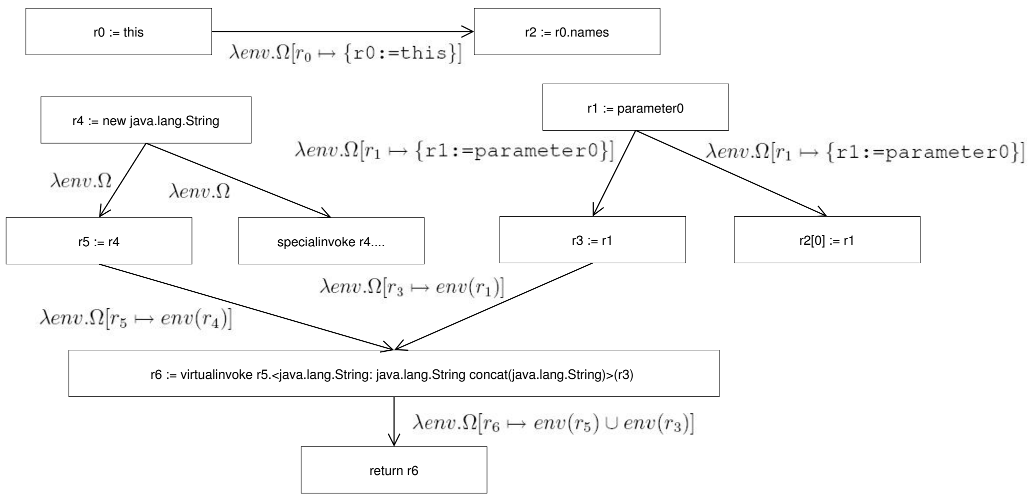

r0 := this
$\lambda env.\Omega[r_0 \mapsto \{\texttt{r0:=this}\}]$
r2 := r0.names
r4 := new java.lang.String
r1 := parameter0
$\lambda env.\Omega[r_1 \mapsto \{\texttt{r1:=parameter0}\}]$
$\lambda env.\Omega[r_1 \mapsto \{\texttt{r1:=parameter0}\}]$
$\lambda env.\Omega$
$\lambda env.\Omega$
r5 := r4
specialinvoke r4....
r3 := r1
r2[0] := r1
$\lambda env.\Omega[r_3 \mapsto env(r_1)]$
$\lambda env.\Omega[r_5 \mapsto env(r_4)]$
r6 := virtualinvoke r5.<java.lang.String: java.lang.String concat(java.lang.String)>(r3)
$\lambda env.\Omega[r_6 \mapsto env(r_5) \cup env(r_3)]$
return r6

Figure 4.2: Reduced CFG for the running example, annotated with environment transformers for graph edges

4.2.4 Dependence Analysis Algorithm

The algorithm for computing dependencies has three steps, which follow closely the phases described earlier. In the first phase, the reduced CFG for each method is built using the result of a reaching definitions analysis. An empty dependence set is created for each local variable.

In the second phase, we compute the intraprocedural dependencies for each method of the program. The effects of calls are not considered. Using a worklist, identity statements and pending calls are propagated to dependence sets along CFG paths that do not contain calls. At the end of this phase, if a method has a return value, the dependence set of the return variable may contain some of the identity statements that affect this value. For methods that do not contain assignments with calls, this set is complete by the end of the phase. If there are assignments with calls, these assignments are added to the dependence sets of affected local variables; such locals are either the return variable, or some actual parameter of a pending call.

The third phase computes the dependencies that are due to calls. We perform bottom-up traversal of the call graph, and for each call site we inline the callee's dependence information into the caller's dependence information. This is done in two steps:

1. The strongly connected components (SCCs) are computed for the call graph.
2. The SCCs are visited in reverse-topological sort order of the SCC-DAG. During this traversal, after a method is processed, we have computed for it the complete set of formal parameters that affect the return value. This set could be empty. The dependence set of a callee is used later to compute the dependence set of a caller.

When processing an SCC, there are three possible cases:

1. The SCC contains a single method M, and this method is not self-recursive (it is not a target for any of its own calls). In this case, we compute the set of dependencies for M using the already-computed sets of all of M's callee methods. For a call that appears in an assignment in M, the set of target methods is iterated. For each target t, there is a set of parameters I_t that affect the return value of t. For each parameter in I_t, the dependence set of its corresponding actual in M is propagated forward from the call. New dependence information may be added due to calls, and therefore the dependence sets in M need to be propagated intraprocedurally again. This gives us the set I_M for

```
java.lang.Object selfRecursive() {
   r0 := this;
   ...
   r8 := r7;
   r9 := virtualinvoke r8.<MyClass:
      java.lang.Object selfRecursive()>;
   r10 := r9;
   ...
   return r30;
}
```

Figure 4.3: Code fragment from method `selfRecursive`

M. Essentially, we use the I_t sets for M's callees to jump over calls, and to introduce the transitive dependence effects of these calls.

2. The SCC contains a single method M, and this method is self-recursive. The treatment of calls is similar to the previous case when the targets are different from the current method. However, there are new kinds of dependencies due to recursion. When the target of a call is the same method as the caller, the dependence information of the callee is not complete. We solve this issue by creating new edges in the reduced CFG. For each edge (d, c) labeled with l, that starts in a definition d and ends in a call c, there will be a new edge between d and the identity statement for the formal corresponding to the actual l in c. The new edge is labeled with l. For each edge (c, s) labeled with l, that starts in a call c and ends in a statement s, there will be a new edge from the return statement of the method to s. The new edge is labeled with the return variable of the method. An example of a self-recursive method is shown in Figure 4.3 and its reduced CFG is shown in Figure 4.4. The thick lines represent new

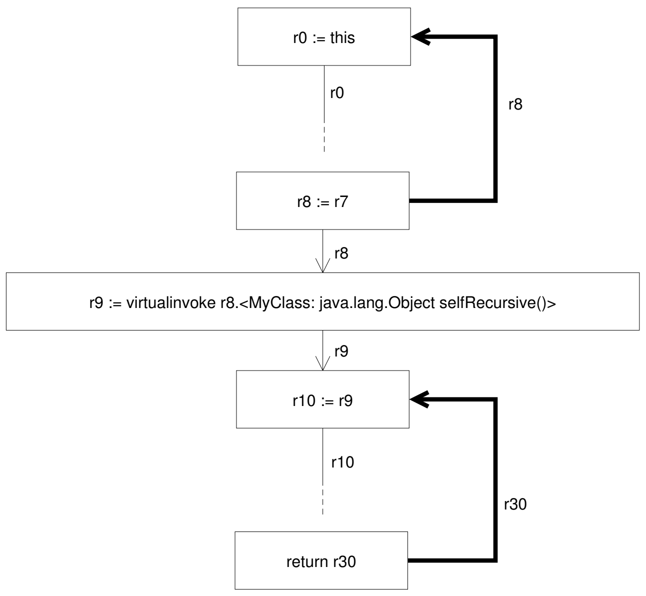

r0 := this
r0
r8
r8 := r7
r8
r9 := virtualinvoke r8.<MyClass: java.lang.Object selfRecursive()>
r9
r10 := r9
r10
r30
return r30

Figure 4.4: New edges in the reduced CFG for method `selfRecursive`

edges added due to recursion. The dependence information flows along these new edges similarly to the processing of regular edges in the reduced CFG.

3. The SCC contains multiple methods, and some of them may be self-recursive. Processing the dependence information is done similarly to the previous cases, except in multiple iterations. After one iteration the dependence sets of the callees may not be complete, so a fixed-point computation is necessary. The dependence information is computed repeatedly for each method in the SCC until there is no change in the dependence information of any method.

At the very end of the SCC-DAG traversal, we have a complete set I_M for each method M, encoding the transitive dependencies from the formal parameters of M to the return value of M.

Our implementation of this whole-program analysis is based on the Soot analysis framework [81]. We use the call graph computed with 0-CFA by the `spark` module in Soot. Our implementation considers only call sites representing explicit calls, and ignores implicit calls to `finalize` and `clinit` methods in the call graph.[8] We also ignore all calls to native methods, as well as all calls through reflection — that is, any calls to methods with the following Soot signatures:

- `<java.lang.Class: java.lang.Object newInstance()>`
- `<java.lang.reflect.Constructor: java.lang.Object newInstance(java.lang.Object[])>`
- `<java.lang.reflect.Method: java.lang.Object invoke(java.lang.Object,java.lang.Object[])>`
- `<java.lang.reflect.Proxy: java.lang.Object newProxyInstance(java.lang.ClassLoader, java.lang.Class[],java.lang.reflect.InvocationHandler)>`

4.3 Summary Generation Analysis

The summary-generation analysis takes as input the set of library classes and produces a partial representation of their dependence information. The summary information is saved on disk, and is used in the subsequent summary-based interprocedural dependence analysis.

4.3.1 Summary Information

The summary information for a method is similar in structure to the output of phase 2 from the whole-program analysis (Section 4.2.2). Since this information is relevant only from the point of view of future callers of the method, only dependencies that may ultimately affect the return value of the method are considered. These are

[8]The analysis considers only call graph edges for which method `isExplicit` from class `soot.jimple.toolkits.callgraph.Edge` returns true.

the dependencies related to the return statement of the method and the pending call sites. Recall that for a statement `return l`, the MOP solution computed in phase 2 is $\lambda env.\Omega[l \mapsto \{s_1, s_2, \ldots\}]$ where s_i are identity statements or pending calls. Similarly, for any pending call with actual parameters `r1`, `r2`, ..., the MOP solution is $\lambda env.\Omega[r_1 \mapsto \{s_1^1, s_2^1, \ldots\}][r_2 \mapsto \{s_1^2, s_2^2, \ldots\}] \ldots$ where s_i^j are identity statements or pending calls.

As described below, in addition to the purely intraprocedural propagation which produces these MOP solutions, we also perform a restricted form of interprocedural propagation which "resolves" some of the pending calls. Still, in the general case, not all pending calls can be resolved, and they exists in the summary as sources and targets of dependencies. Conceptually, in a method's summary information, dependence information associated with a statement s (return statement or pending call) is represented as a set of pairs (t, s), where t is an identity statement or a pending call site from which dependencies flow directly or transitively to s. In cases when s is a pending call, the pair is annotated with the actual parameter of s which depends on t. The representation of call sites depends on the way the summary I/O is implemented (it can be as simple as an integer id), but it must be enough to uniquely identify the call site in the method body, and to query the call graph for targets in a summary-based dependence analysis of the user code.

4.3.2 Summary Generation

The algorithm for summary generation computes the intraprocedural dependencies, performs a limited form of interprocedural dependence propagation, and stores

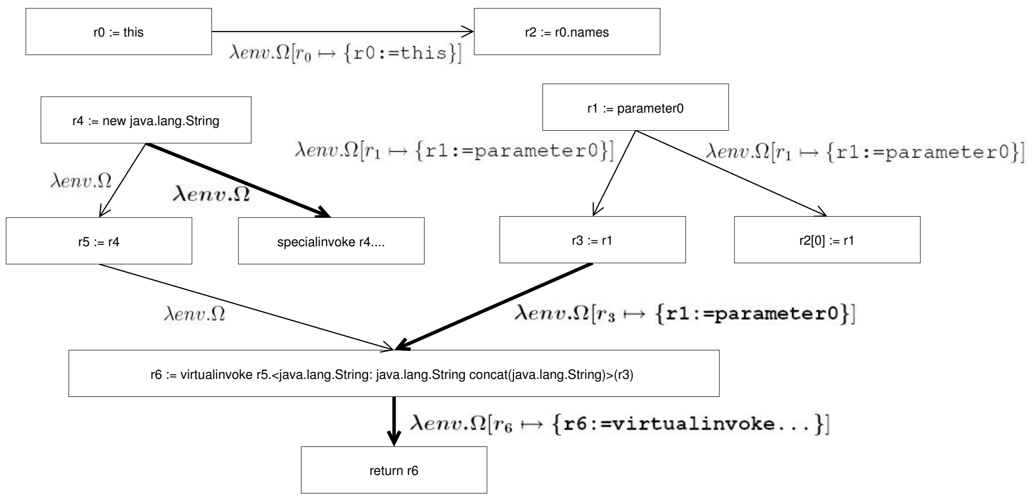

r0 := this
r2 := r0.names
$\lambda env.\Omega[r_0 \mapsto \{$r0:=this$\}]$
r4 := new java.lang.String
r1 := parameter0
$\lambda env.\Omega[r_1 \mapsto \{$r1:=parameter0$\}]$
$\lambda env.\Omega[r_1 \mapsto \{$r1:=parameter0$\}]$
$\lambda env.\Omega$
$\lambda env.\Omega$
r5 := r4
specialinvoke r4....
r3 := r1
r2[0] := r1
$\lambda env.\Omega$
$\lambda env.\Omega[r_3 \mapsto \{$r1:=parameter0$\}]$
r6 := virtualinvoke r5.<java.lang.String: java.lang.String concat(java.lang.String)>(r3)
$\lambda env.\Omega[r_6 \mapsto \{$r6:=virtualinvoke...$\}]$
return r6

Figure 4.5: Transitive intraprocedural dependencies for method `replaceName`

the result in a summary file. The algorithm has three phases that are similar to the phases of the whole-program dependence analysis.

In the first phase, the summary generation algorithm computes the set of reaching definitions for the statements of a method, and the resulting reduced CFG. In the second phase, the transitive closure of intraprocedural dependencies is computed for each node of the reduced CFG, exactly as done in phase 2 of the whole-program analysis (Section 4.2.2). The restrictions described in Section 4.2.4 (e.g., ignoring reflective calls and calls to native methods) are also used in the summary generation analysis. Figure 4.5 shows the transitive intraprocedural dependencies computed by phase 2 for the sample method `replaceName` presented in Figure 3.7. The thick edges highlight the dependencies related to the return statement of the method and the pending call sites. They are saved in the summary because they are relevant from the point of view of callers of the method. Figure 4.6 shows these dependencies and how the statements are affected. Note that the call to the constructor of the string

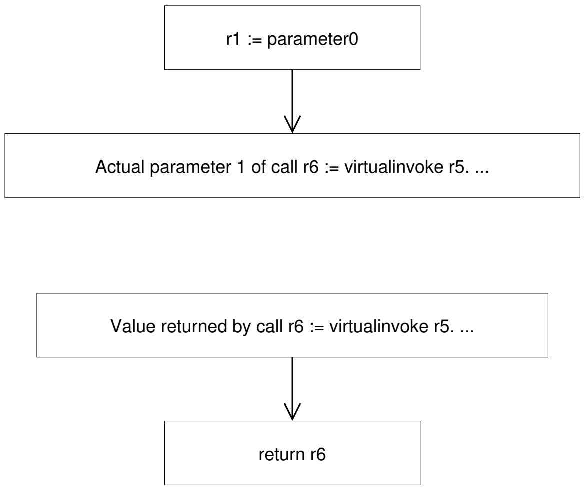

r1 := parameter0
Actual parameter 1 of call r6 := virtualinvoke r5. ...
Value returned by call r6 := virtualinvoke r5. ...
return r6

Figure 4.6: Transitive intraprocedural dependencies for method `replaceName` that are relevant for the summary

does not introduce any dependency, while the call to method `concat` may introduce interprocedural dependencies, although information about them is not available at this stage.

Complete interprocedural propagation is impossible because determining the run-time targets of library calls usually depends on information that is not available when the summary is built. Furthermore, some targets may be callback methods that are defined in future client code. However, there are calls for which the summary generation analysis can determine *precisely* the run-time target methods, and in some cases we can "inline" the dependence information of the callee into the summary of the caller, as described below.

4.3.3 Inlining Information about Fixed Methods

In order to describe the technique for inlining dependence information, we need to define the notion of a fixed method. Informally, for a *fixed method* we can determine exactly which methods it calls, directly or transitively, independently of any client code. The dependence information for the fixed methods can be computed completely and precisely at the moment the summary is built, and can be inlined into callers' dependence information.

Exit Calls

A call site is an *exit call* if it can invoke some method that "exits" the scope of the analysis and therefore the effects of the call cannot be modeled. An exit call is a virtual call `x.m()` for which all of the following hold: (1) the declared type of `x` has possible unknown subtypes, (2) the type which declares the compile-time target method of the call has possible unknown subtypes, and (3) the compile-time target method of the call can be overridden by unknown methods. Reflective calls (see end of Section 4.2.4) and calls for which the compile-time target method is a native method also exit the scope of the analysis, but we ignore them in our discussion.

The definition of exit call considers the possibility of callbacks to user methods. A library type T (class or interface type) is considered to have potential unknown subtypes in client code when T or some known subtype of T is public and not final. Note that this definition assumes that library packages are sealed — that is, client code cannot add new classes to existing library packages (and therefore non-public types cannot be accessed directly by client code). The compile-time target method m of the call site can have unknown overriding methods if (1) m is not private and is not final, and (2) at least one of m's known overriding methods (or m itself) is non-final and is also visible to the client code (i.e., it is public or protected).

Fixed Calls

A fixed call is a call for which there is exactly one possible target, no matter what the client code may be. The target is a library method and can be determined at compile time. Since it cannot invoke a method in the user code, a fixed call is obviously not an exit call.

A call that is not an exit call is fixed if it meets any of the following conditions:

- The call is a static invocation or special invocation. In this case the run-time target of the call is the same with the static target.

- The call is a virtual invocation with the compile-time type of the receiver being an array type. Array types are final, and there is no possibility for the static target method to be overridden.

- The call is a virtual invocation, the compile-time type of the receiver is a class/interface type, and at run time the receiver can point to objects of exactly one type.

  We consider receivers for which the set of possible types contains only one type. Such variables can be identified by looking at the types of objects that flow into them. The type set is determined using intraprocedural type analysis, as described in the previous chapter. We also make sure that nothing else affects the type of the receiver, that is, the receiver is not a variable that is assigned the value of another local variable, or the return of a call. If this is the case, the type set computed by intraprocedural type analysis contains all the possible types of the receiver. If the type set of the receiver contains exactly one type, the unique target method of the call can be determined based on this type.

- The call is a virtual invocation, the compile-time type of the receiver is a class/interface type, and it has exactly one possible run-time target in the library, as determined by the type hierarchy for the entire library.

For the sample method `replaceName` presented in Figure 3.7, both calls that appear in its body are fixed calls. The first call is a static invocation, and the second call is a virtual invocation for which the type of the receiver `java.lang.String` is final.

Fixed Methods

A fixed method is recursively defined as a method that either does not make any calls, or it has only fixed calls to fixed methods, possibly to itself. We present a fixed-point algorithm that marks the methods in the library as fixed or not fixed.

1. The library methods are iterated. A method is marked non-fixed if it contains a call site that is not fixed (not in the cases presented above), otherwise it is marked fixed.

2. The second step is a fixed-point iteration over all the methods that have been marked fixed. If such a method contains a call to a non-fixed method, it is marked as non-fixed. The corresponding call site is a fixed call site; otherwise the caller would have been marked as non-fixed in Step 1. The target of the call is determined using the cases presented above. This step is repeated until no new methods are marked as non-fixed.

Interprocedural Propagation

Dependency information related to fixed methods can be completely determined at summary generation time, independently of any client code. Therefore it can be inlined into the callers of fixed methods, in order to reduce the cost of subsequent summary-based analysis.

In the case of fixed methods that have calls, the dependency information depends on the targets of those calls. The information of a method cannot be inlined into its own fixed callers, unless the dependencies of all its callees have been already inlined into its own information, making it complete. Thus, the dependency information for fixed methods is propagated in a bottom-up traversal of the call graph, in a manner similar to the one described in Section 4.2.4. The call graph used by the propagation algorithm is a *fixed call graph*. The nodes are fixed methods, and the edges represent the effects of fixed calls. For each call site inside a fixed method, there is exactly one edge in the fixed call graph that connects it to the fixed method representing the target.

Similarly to the whole-program dependency analysis Section 4.2.4, the inlining algorithm traverses the fixed call graph in the reverse topological order of its SCCs. Each SCC is one of the following cases:

1. SCC contains only one method M, and M is not self-recursive. The called methods, if any, are all fixed methods, because M is itself fixed, and their dependence information is complete because it has been computed previously for their respective SCCs. This means that for each target t, the set I_t of parameters that affect the return value is known. The dependence information

is inlined in a very similar manner to Case 1 of the third phase of the whole-program dependence analysis algorithm presented in Section 4.2.4: the I_t set is used to jump over the call to t, and to introduce the transitive dependence effect of the call.

Adding the callees' information makes the dependency information of M complete. At this point the transitive dependencies due to the calls are computed. Since we are interested only in the (possibly transitive) dependencies that may affect a caller of M, the intraprocedural dependencies in M's information are redundant and can be removed. The set I_M of parameters that affect the return value is the relevant dependency information that is saved in the summary.

2. The SCC is made of one method M and M is self-recursive. The treatment of calls is similar to the previous case when the targets are different from the current method. When the target of the call is the same as the calling method, the dependence information is updated similarly as in Case 2 of the third phase of the whole-program dependence analysis algorithm from Section 4.2.4. New edges are added to the reduced CFG of the method to represent the dependencies introduced by recursive calls. Transitive dependencies are computed, and redundant intraprocedural dependencies are removed, similarly to Case 1.

3. The SCC is made of multiple methods, some of which are self-recursive. Similarly to Case 3 of the whole-program dependence analysis algorithm from Section 4.2.4, processing the dependence information is done in multiple iterations. A fixed-point computation is used, to determine the dependencies of the methods. A fixed point is reached when no more dependencies are added.

At the end of the SCC-DAG traversal, we have a complete set I_M for each fixed method M, encoding the transitive dependencies from the formal parameters of M to the return value of M.

The following example illustrates the technique of inlining dependencies. We assume that the sample method `replaceName` presented in Figure 3.7 has only fixed methods as targets of its fixed calls, thus itself is also a fixed method. We also assume that `replaceName` is the only method in its SCC. The algorithm outlined at Case 1 applies. The sets of parameters that affect the return value for the targets of the calls are known, because have been computed for a previous SCC. The set for the constructor of the string object $I_{constructor} = \emptyset$, because constructors do not have a return statement. We assume that the set for the `concat` method $I_{concat} = \{this, param0\}$. Computing the transitive dependencies yields a transitive dependence between the identity statement `r1 := parameter0` and the return statement of the method. The set of formals affecting the return value for method `replaceName` is $I_{replaceName} = \{param0\}$.

As a further improvement of the summary, the dependence information of fixed methods can be inlined into the dependence information of their non-fixed callers at fixed call sites. This procedure is similar to Case 1 of the algorithm, except that when removing redundant intraprocedural dependencies from the caller's information, the (non-fixed) call sites that were not inlined must be preserved.

Entry Methods

The summary can be further improved if we consider the methods for which the callers are in the library, and are fully known regardless of any future user code. For example, consider a method for which we know all the callers (which are all in the library), and each call site invoking the method is a fixed call. After inlining the dependency information of the method into its callers, we can remove it from the summary altogether, since it will never be used by other callers.

In order to find the methods with known callers, we need to define the notion of an *entry method*. Informally, an entry method can be directly called from outside of the analyzed library code. For an entry method the set of callers can not be completely determined at the time the summary is built.

A method is an entry method if it is any of the following:

- A method called at the startup of the Java virtual machine. There is a set of methods that are called directly by the virtual machine independently from the user code. This can affect their dependency information in ways that cannot be determined by a static analysis, therefore we ignore them in our discussion.

- A static initializer or a finalizer

- A method that overrides one of the following methods:
  - `void run()` in `java.lang.Runnable`
  - `java.lang.Object` `invoke` `(java.lang.Object, java.lang.reflect.Method, java.lang.Object[])` in `java.lang.reflect.InvocationHandler`
  - `void writeObject(java.io.ObjectOutputStream)` or `void readObject(java.io.ObjectInputStream)` or `java.lang.Object readResolve()` or `java.lang.Object writeReplace()`, and implements interface `java.io.Serializable`
  - `void writeExternal(java.io.ObjectOutput)` or `void readExternal(java.io.ObjectInput)`, and implements interface `java.io.Externalizable`

These methods are involved in Java's own special mechanisms for running threads, using reflection and defining serialization, and may be called directly by the virtual machine.

- A method that is public or protected, and its declaring class is public.
- A method that is declared in a non-public class, and this class has a public supertype that declares a public method with the same signature.

Methods with *imprecise caller information* are the methods that can have unknown callers outside the library, or for which the known callers have more than one target. Entry methods, as described above, have imprecise caller information. In this category are also the targets of virtual calls that are either exit calls, or that can have more than one possible target (are not fixed calls). As described in Section 4.3.3, the possible targets of a call can be determined by looking at the types of objects that flow into the receiver, when the value of the receiver is not affected by another variable or the return of a call. Otherwise analysis of the type hierarchy for the compile-type of the receiver can be used to determine the possible targets conservatively.

The library methods that do not fall in any of the cases presented above have precise caller information. This means that all the callers of such a method can be determined when the summary is built, and that the method is the only possible target for the respective call sites. For a method with precise caller information, we can inline the dependence information into the information of its callers. Once inlined, the information about the method becomes redundant and can be removed from the summary, thus reducing it size.

Inlining the information of a method that contains calls adds complications to the summary, because it introduces into the dependence information of the caller information about calls that are not part of the caller's body. Therefore, in our implementation we inline only the information corresponding to methods with precise caller information that are also fixed methods. The dependencies related to calls are resolved for fixed methods, so no calls are added into the caller's information when inlining such a method.

4.4 Experimental Study

The goal of the experimental study is to asses the effectiveness of using a summary-based analysis versus a whole-program analysis. We consider the following issues: (1) the size of the summary on disk, (2) what is the cost of creating the summary, (3) what are the effects of inlining fixed methods, and (4) the improvement in terms of running time and memory usage of the summary-based version compared to the whole-program version, and to a baseline implementation.

We implemented the algorithms presented in this chapter using the Soot framework [81]. The experiments were performed using a configuration identical to the one used in the experimental study for summary-based type analysis. The details are presented in Section 3.5.

4.4.1 Generating the summary

The summary information related to dependence analysis contains a data structure that represents pairs of statements that belong to the same method and between which there is a dependency. In a pair (s_1, s_2), s_1 can be an identity statement or a call, and s_2 can be the return statement of the method, or a call, in which case the position of the actual parameter affected by the dependency is also saved. If multiple actuals are affected, then multiple dependence pairs are saved in the summary for the same pair of statements.

The input of the summary generation were the classes from Java standard libraries from J2SE 1.4.2, which contain all classes in packages java., javax, com., COM., org., and sun. The output of the summary generation was a file that stores the dependence information for this methods methods, along with the summary information related to type analysis, presented in Section 3.5. Generating the summary took almost 40 minutes, and resulted in a file of size of 14.4 MB. In this file, the dependence information takes only 2.2 MB. The rest of 12.2 MB are taken by the summary for type analysis (see Section 3.5).

The library classes that served as input to the summary generation contain a total of 10238 classes, 77190 methods and 1496003 statements. Out of these statements, 307763 (20.6%) are calls. Out of all library methods, roughly one third, or 25490, are fixed. However only 11517 calls, which represents 3.7% of all calls, are in these fixed methods, which shows that the fixed methods have generally simple bodies with few or no calls. The intraprocedural dependence analysis computes 24470 dependence pairs in the fixed methods, which represents 6.1% of the total number of pairs for all methods (401070). The percentages of calls and dependence pairs in fixed methods are relatively small, and therefore limited savings will come out from interprocedural analysis of these methods.

Although it does not have a big impact on the summary, optimizing the representation of fixed methods through call inlining is straightforward. The callees of fixed methods are inlined using the technique presented in Section 4.3.3. Fixed methods are inlined into their callers at fixed calls. Out of all library methods, 19880 have precise caller information, which represents 25.75% of all the methods, and among these 7195 are also fixed. Due to inlining, the dependence information of the 7195 methods that are both fixed and have known callers does not need to be saved in the summary.

To summarize, there are two optimizations applied to the summary: (1) inlining all fixed methods (into callers that can be both fixed and non-fixed), and subsequently (2) completely removing from the summary all fixed methods that have known callers. After the first optimization is applied to the summary information, the number of calls decreases to 197267 (all of them left in non-fixed methods), which represents a reduction of 36.9% in the number of calls in the library. The number of dependence pairs left after inlining is 336249, which represents a reduction of 16.2% in the number of pairs resulted from the intraprocedural propagation. After the second optimization is applied, the number of dependence pairs further decreases to 333912 (a reduction of 16.8% in the number of pairs). These pairs are saved in the final summary. The second optimization does not affect the number of calls, since there are no calls in fixed methods.

In the final summary, there are only 5518 dependence pairs for fixed methods (1.7% of the total number of pairs in the summary), which shows that dependencies of non-fixed methods take up most of the summary. This suggests that improvements in the representation of non-fixed methods are of importance for future work in creating a compact summary.

Application	Regular	Summary	Reduced(%)	Baseline	Reduced(%)
`Rabbit2`	17.03	4.24	75.13	2.47	85.53
`compress`	15.41	3.30	78.56	2.12	86.25
`db`	15.66	2.97	81.03	2.22	85.84
`fractal`	20.05	4.43	77.91	3.21	83.98
`jack`	19.86	6.43	67.60	5.64	71.58
`javac`	19.34	6.11	68.41	5.47	71.74
`javacup-0.10j`	19.01	6.32	66.74	4.79	74.79
`jb-6.1`	16.34	3.42	79.05	3.03	81.42
`jess`	19.20	6.52	66.06	5.79	69.87
`jflex-1.4.1`	43.80	17.26	60.59	16.02	63.43
`jlex-1.2.6`	20.53	8.96	56.37	7.64	62.79
`jtar-1.21`	21.96	3.99	81.84	3.87	82.36
`mindterm-1.1.5`	42.74	28.62	33.04	26.54	37.91
`mpegaudio`	53.56	50.70	5.34	46.16	13.82
`muffin-0.9.3a`	27.56	10.73	61.06	9.93	63.97
`raytrace`	17.84	4.23	76.27	2.86	83.99
`sablecc-2.18.2`	25.96	11.24	56.71	9.05	65.13
`socksecho`	22.21	4.79	78.43	3.61	83.76
`socksproxy`	15.59	4.08	73.80	2.81	81.98
`violet`	31.05	6.28	79.78	4.28	86.22

Table 4.1: Comparison of the whole-program analysis, the summary-based analysis, and a baseline implementation in terms of running times (in seconds)

4.4.2 Summary-based analysis

This section describes the experiments we performed in order to evaluate the performance of the summary-based technique. We implemented dependence analysis (both whole-program and summary-based) as a separate module within the Soot 2.2.2 framework. The analysis uses as input the call graph resulted from the corresponding type analysis (whole-program or summary-based) described in the previous chapter. Similarly to the case of type analysis, whole-program dependence analysis uses the Jimple representation for all classes, while summary-based dependence analysis uses Jimple only for user classes. For library classes, the latter retrieves the already-computed representation of dependencies from the summary.

Application	Regular	Summary	Reduced(%)	Baseline	Reduced(%)
`Rabbit2`	34.05	1.38	95.94	1.24	96.35
`compress`	27.06	1.71	93.69	1.24	95.41
`db`	27.01	1.94	92.81	1.14	95.79
`fractal`	39.48	1.48	96.25	1.30	96.70
`jack`	30.18	1.74	94.23	1.41	95.34
`javac`	36.11	2.05	94.33	1.73	95.22
`javacup-0.10j`	29.72	1.53	94.86	1.34	95.48
`jb-6.1`	27.30	1.57	94.23	0.77	97.18
`jess`	30.02	1.83	93.91	1.36	95.48
`jflex-1.4.1`	43.78	1.61	96.31	1.21	97.24
`jlex-1.2.6`	28.98	1.81	93.77	1.74	93.98
`jtar-1.21`	42.15	1.79	95.76	0.99	97.65
`mindterm-1.1.5`	46.55	1.61	96.54	1.19	97.44
`mpegaudio`	30.51	1.43	95.30	1.19	96.11
`muffin-0.9.3a`	51.30	2.26	95.60	1.75	96.59
`raytrace`	28.03	1.56	94.43	1.28	95.42
`sablecc-2.18.2`	35.25	1.75	95.05	1.64	95.34
`socksecho`	41.05	1.63	96.02	0.97	97.65
`socksproxy`	28.38	1.54	94.59	1.24	95.65
`violet`	57.32	1.76	96.93	1.19	97.92

Table 4.2: Comparison of the regular analysis, the summary-based analysis, and a baseline implementation in terms of memory usage (in MB)

We ran our experiments on a collection of 20 Java programs, the same benchmarks that we used for the experiments on type analysis. The programs are presented in Table 3.1 and described in detail in Section 3.5.2.

For each benchmark, we ran the whole-program dependence analysis and the summary-based dependence analysis, and we compared the running time and memory usage for the two analyses. To asses the improvement brought by the summary-based analysis, we created a artificial summary that records empty dependence information for each library method. The analysis based on the artificial summary computes the dependencies for the user methods similarly to the whole-program analysis, while skipping any computation for the library methods, except for the disk access to the summary file. Although the solution computed this way is irrelevant, this version of the summary represents a baseline optimization: the time and memory measurements reported by the analysis with artificial summary represent a limit of how far a summary-based analysis can be improved.

Table 4.1 shows time measurements in seconds, and Table 4.2 shows memory usage in MB for each benchmark. In Table 4.1, Column "Regular" shows the running times for the whole-program dependence analysis, and Column "Summary" displays the times for the summary-based analysis. Column "Reduced%" displays the reduction of running time in the summary-based analysis compared to the whole-program analysis. The times obtained with the baseline version are displayed in Column "Baseline", and the last column "Reduced%" shows the reduction obtained by the baseline implementation compared to the whole program analysis. Table 4.2 displays similar measurements and comparisons for memory usage. The average reduction in time is 79.78% in the summary-based version, while the memory usage is reduced on average by 96.93%. The results show that the summary-based technique brings a significant improvement over the whole-program analysis. Comparison with the measurements for the baseline analysis indicate that the resource usage is reduced close to the limit of what can be achieved.

The results show clearly that using pre-computed dependence information for library methods takes much less in terms of time and space than creating it on the spot using the Jimple representation. The improvement is more dramatic compared to summary-based type analysis, due to the fact that dependence analysis is flow- and context-sensitive, and thus the propagation work it involves is more computation intensive than for type analysis. In the case of dependence analysis, the experimental study shows that our summary-based technique effectively addresses the problem of reducing analysis cost for applications built with large libraries.

Conclusions. (1) Summary-based dependence analysis can achieve significant savings of running time and memory usage, compared to its whole-program counterpart; for example, for all experimental subjects, the average running time reduction was roughly 80%. (2) Most savings come from avoiding the cost of analyzing the library code. (3) Interprocedural techniques for summary generation should be investigated in future work, in order to improve the representation of dependence information for non-fixed methods in the summary.

CHAPTER 5

RELATED WORK

5.1 Points-to Analysis for RMI Java Software

Points-to analysis for Java. Points-to analysis has been an active research field in the last fifteen years. Section 1.1 presents the dimensions that affect the cost/precision trade-offs of points-to analyses. Several groups have adapted points-to analyses defined for C to be used for analysis of Java (e.g., [3,9,37,39,51,60,76]). They have studied how analysis cost and precision are affected by the various dimensions of the algorithm design space. The effect of considering context sensitivity on the precision of the analysis in Java has been investigated by several researchers (e.g., [40,46,47]). A specific version of a subset-based analysis [4] suitable for implementing with binary decision diagrams (BDD) has been shown to exhibit good space and time behavior when used to analyze a range of Java programs. In [74] Sridharan et al. developed an efficient demand-driven points-to analysis algorithm for clients that only require points-to information for a subset of program variables. Other examples of points-to and similar analyses for Java include, among others, [23, 24, 65, 84].

One issue in analyzing Java programs is handling of dynamic class loading and reflection, which is either treated very conservatively (as in our work) or completely ignored. Livshits et al. [41] propose a static analysis algorithm that uses points-to information to approximate the targets of reflective calls as part of call graph construction.

Lhoták and Hendren [37] present the SPARK framework, which allows experimentation with many variations of points-to analyses for Java. The points-to analysis proposed in [37] is the closest related work to ours, and serves as the starting point for the PAG-based algorithm in our approach. We introduce various modifications of the techniques from [37]. For example, new kinds of PAG edges and propagation rules associated with them are required for analysis of RMI-based programs. The handling of calls is generalized to simulate the semantics of remote invocations, including the effects of serialization of object graphs. We also introduce a technique for efficient handling of the standard Java libraries across multiple distributed components.

Analysis of distributed applications. There has been very little work on generalizing points-to analyses to RMI-based Java software. The closest related work is [82], where a compile-time points-to analysis is used to optimize the serialization at remote calls in several ways, including the two optimization techniques described earlier. The analysis is described with very little detail, but it appears to be a flow-sensitive and context-insensitive variation of subset-based analysis. There is no theoretical definition of the analysis semantics, and no details are given about the algorithms and data structures used to implement this semantics. For example, it is unclear whether the approach uses two different points-to sets (remote and local) per variable, whether component-specific copies v^i of a variable v are used, whether the set of reachable methods is constructed during the analysis or is assumed to be part of the analysis input, and whether the underlying standard libraries are being analyzed. Our work provides a precise theoretical definition as well as specific algorithms and data structures. The experimental results in [82] focus on the effects of the optimizations on performance, while we are primarily interested in uses of the points-to information in tools for software understanding, testing, and verification.

5.2 Static Analysis in the Presence of Large Libraries

Various techniques have been used to achieve a certain degree of modularity in dataflow analysis. Below we describe some of the most relevant approaches. A more complete discussion is available in a summary paper by Cousot and Cousot [13], presented from an abstract-interpretation point of view.

The construction of summary dataflow functions for use in interprocedural analysis dates back to the "functional approach" to whole-program analysis defined by Sharir and Pnueli in [69]. Sagiv et al. propose efficient function representations with the formulation of IFDS problems [52] and the more general IDE problems [68]. Our work is most closely related to their formulation, since we define type analysis and dependence analysis as IDE problems. In [69], an analysis algorithm computes a dataflow function for a procedure, which is then used by the callers of that procedure. However, this work (as well as [52,68]) assumes a procedural language with no virtual calls; furthermore, there is no separation between client code and library code.

The summary-based analyses proposed in our work can be viewed as practical techniques based on the general theoretical framework for dataflow analysis for programs built with large extensible library components, a framework presented in [58].

The framework considers the model of interprocedural component-level dataflow analysis, defined in [56], in which the source code a program component is analyzed given some information on the environment of this component.

Summary information dependent on the rest of the program. The majority of dataflow analyses that use summaries perform bottom-up traversal of the call graph, and compute summary functions for procedures using the functions computed for the already visited callees. In [9], Chatterjee et al. present a bottom-up technique for a whole-program flow- and context-sensitive points-to analysis. The technique, called Relevant Context Inference (RCI), can be used to analyzed both whole programs and incomplete programs, such as libraries, and applies to points-to analysis of a C++ subset. RCI uses summaries to optimize the analysis of large programs by using partial information about procedures when performing whole-program analysis, such that the entire program is not needed in memory. However, the summaries are not saved on disk to be used for subsequent analyses. Furthermore, even when libraries are preanalyzed as in [8], the computation of summary functions cannot be performed in the presence of callbacks from the library to the client code. A points-to analysis for C programs that uses summaries in a similar manner is presented in [10]. Although their approach uses bottom-up traversal of the call graph, the propagation of summary transfer functions sometimes alternates with top-down propagation along the partially resolved call graph, in order to handle function pointers.

Various other summary-based analyses the bottom-up traversal of the call graph. Choi et al. [11] study the use of summaries for an efficient escape analysis. Their proposed data structure, the connection graph, establishes reachability relations between objects and objects references. Ruf [65] also presents an escape analysis in which method summaries are used to encode the alias and synchronization effects of methods and their transitive callees. Whaley et al. [85] present a summary-based combined pointer and escape analysis for Java programs. Their analysis produces a single parameterized result for each method, that can be specialized for use at all of the call sites that may invoke the method. It is a partial program analysis (it analyzes each method in isolation, and the result becomes more precise as callees are analyzed). The analysis results for the library are used as summaries, and once computed, can be reloaded in the analysis of subsequent programs. Similarly, the analysis in [18] computes a function dependence graph that encodes the dataflow within a program. The graph contains the summary functions for all procedures. The functions for a callee must be created before the function for the caller. Other analyses of a similar nature include [6, 38, 80].

Another approach is to traverse the call graph in a top-down manner, and to compute the summary function for a procedure by taking using the already computed functions of the callers. In [28] may alias information is computed for a module (which is defined as a method or a group of methods that has one entry point) by introducing all possible contexts at the entry of the module. The traversal goes in top-down manner, and the information about a module is later reused as a summary.

Summary information independent of the rest of the program. Some techniques compute summary information for a software component independently of the callers and callees of that component. One particular technique is a modular approach which computes partial analysis results for each component, combines the results for all components in the program, and then performs the rest of the analysis. Such an approach is used in Flanagan and Felleisen's componential analysis for set-based analysis [17], where they adapt set-based analysis to large programs. The work in [12] presents a compositional analysis in which a program is seen as a collection of modules. Similarly to our summary-based approach, their technique improves analysis efficiency by using partial information about modules of the program under the analysis. The work in [14] presents a points-to analysis in C that improves Steensgaard's algorithm [75], while being more scalable than Andersen's [2]. In their analysis, each individual source file is first parsed, and the assignment statements therein are used to construct a flow graph, which is "serialized" and written to disk. In a subsequent phase, the summary files are read from disk, and the propagation step is reapplied to obtain global points-to information. Heintze [31] creates summary information that encodes the structure of the source code of the component, without performing any analysis work in advance. Rountev and Ryder [54, 63] define an approach for creating summary information for C libraries based on a similar idea: the summaries represent the structure of the source code of the library, and can be used to perform a variety of points-to and side-effect analyses of library clients. Summary optimizations are used to filter out parts of the summary that are irrelevant for the clients of the library.

User-defined summary information. There have been proposals for employing summary information provided by the analysis user (e.g., programmer, tester, etc.). Guyer [25, 26] proposes annotations for describing libraries in the domain of high-performance computing. The annotations encode high-level semantic information (e.g., points-to and side-effect properties) and are produced by a library expert. Rugina and Rinard [66] propose the use of design information in the context of optimizing compilers. They present summary information that describes how a called procedure affects points-to relationships and how it accesses regions of arrays; this information is used to perform automatic program parallelization. Dwyer [15] presents a modular dataflow analysis for verifying correctness properties of concurrent programs. Information about the surrounding environment of a module is represented using an environment automaton. This automaton describes the possible interactions between the module and the environment, and is provided by the user.

Using conservative assumptions about external code. Certain approaches analyze a software component when there is no available information about the surrounding environment, using conservative assumptions about the possible effects of unknown external code. Harrold and Rothermel [28] describe an approach for applying Landi-Ryder's whole-program pointer analysis for C [34] to a software module. All possible contexts are introduced at the entry of the module, and then information is propagated in top-down manner inside the module. Rountev et al. [64] define a theoretical approach for creating conservative assumptions, and apply the approach to flow- and context-sensitive pointer analysis of C components. The analysis by Chatterjee et al. [9] has been modified to perform analysis of library modules [8]. The library analysis processes the library and all of its callees in bottom-up manner, and then performs a top-down traversal that propagates conservative assumptions about the clients of the library. Sreedhar et al. [73] define extant analysis that determines whether a reference variables may point to instances of unknown classes. Tip et al. [77, 79] describe analyses and optimizations for the removal of unused functionality in Java modules, and use conservative assumptions to approximate the effects of code located outside of the optimized modules. Ghemawat et al. [19] present several analyses of the properties of fields in Java modules. Whole-program escape analyses [6, 11, 85] can be modified to analyze a component without having any information about the callers or callees of the component. For example, Vivien and Rinard [83] present an incrementalized escape analysis that is based on the whole-program analysis from [85]. The analysis dynamically grows the analyzed program region, and makes conservative assumptions about the rest of the program. Harrold and Rothermel [27] present a technique for class-level def-use analysis for the purposes of dataflow-based unit testing. The technique constructs a placeholder driver that represents all possible sequences of method invocations initiated by client code; the driver does not take into account the effects of aliasing, polymorphism, and dynamic binding. Work by Rountev et al. [54, 61, 62] on points-to analysis shows how to solve this problem in a general manner by creating a placeholder driver that simulates the possible flow of reference values due to unknown callers; this approach has also been generalized to side-effect analysis [55].

Incremental and parallel dataflow analysis. Finally, there is related work on incremental and parallel dataflow analysis, in which intensive analysis is performed first locally, and then followed by a quick traversal to recover the actual solutions. Reference [35] presents a technique of parallelizing dataflow algorithms, where the analysis is done on procedures in a data parallel way. Marlowe and Ryder [43] propose a hybrid incremental method that combines iterative and elimination methods. Their method decomposes the program into regions in which several local problems are solved, and representative (placeholder) values are used for actual data-flow information that is external to the region.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

The traditional model of whole-program data-flow analysis has several limitations that make it unsuitable for many real-world software systems. Whole-program analysis cannot be applied to distributed software, and to very large programs. The impact of data-flow analysis research can be broadened significantly if the limitations of whole-program analysis are addressed and resolved. As a step towards achieving this goal, in this thesis we proposed a theoretical model for points-to analysis of distributed Java applications, and an analysis approach which employs precomputed library summary information.

6.1 Static Analysis for RMI Java Software

The work presented in Chapter 2 is a work that generalizes the existing formalisms for points-to analysis to handle RMI features (Section 2.3) such as remote references, remote calls, and parameter passing through serialization. The key to this generalization is maintaining two separate points-to sets per variable: one for ordinary references and one for remote references. Second, the PAG-based propagation algorithm from [37] is generalized with the help of remote PAG edges (Section 2.3). The specialized propagation rules for such edges allow the algorithm to model 1) the propagation of remote references, which is relatively straightforward, and 2) the propagation of references to deserialized copy objects, which requires several extensions of the algorithm. A similar approach can potentially be useful for other existing points-to analysis algorithms for non-RMI Java programs.

Our work on points-to analysis for RMI applications is a first step in establishing a body of research on static analysis for distributed Java applications. Obvious targets for future work are various flow- or context-sensitive points-to analyses. Such analyses can be defined as extensions of the approach from Chapter 2, and their scalability would have to be investigated carefully. Furthermore, the theoretical complexity of the proposed analysis should be considered: does the subset-based flow- and context-insensitive points-to analysis of RMI applications still have the cubic complexity of the corresponding analysis for non-distributed applications? Another interesting problem would be to define the RMI-specific generalizations of other categories of analyses such as side-effect analysis, def-use analysis, and escape analysis. These analyses could be evaluated experimentally in the context of program understanding tools (e.g., for change impact analysis) and test coverage tools (e.g., for round-trip-scenario coverage [5]). Finally, these static analyses could be generalized and evaluated for more powerful RMI-based middleware platforms such as Enterprise JavaBeans.

6.2 Summary-Based Static Analyses for Large Java Software

The use of library summaries is essential for interprocedural dataflow analysis of modern systems that are built with large library components. We investigated the impact of summary usage on two important dataflow analyses, type analysis and dependence analysis. Our approach uses precomputed summary information that is created with a summary-generation analysis independent of any client code. The summary is loaded from disk and used in the analysis of software built with reusable library components. Our experimental studies indicate that the cost of whole-program type analysis and dependence analysis can be reduced dramatically by this approach.

Future work should apply the summary-based approach to other dataflow analyses, implement the summary-based algorithms, and evaluate their performance. This will lead to deeper understanding of the benefits of abstracting library information and reusing it for subsequent analyses. Such work should consider not only Java, but also programming languages with more challenging features (e.g., the more complex aliasing relationships in C programs). Future investigations also need to focus on systems built with multiple library components, with various inter-component dependencies, and need to evaluate the analysis algorithms on large-scale programs that are built with multiple libraries.

APPENDIX A

WORKLIST ALGORITHM FOR POINTS-TO ANALYSIS

This appendix contains a more detailed description of the worklist algorithm outlined in Section 2.4.4. Given the source code for all classes in sets $\mathit{cls}(C_i)$ for all program components $C_i \in \mathcal{C}$, the algorithm produces:

- A pointer assignment graph (PAG)
- Local points-to set Pt_L and remote points-to set Pt_R for PAG nodes $\mathit{node}(v^i)$ and $\mathit{node}(o.\mathit{fld})$
- A set of reachable methods $\mathit{Reach} = \bigcup_i \mathit{Reach}_i$, where $m^i \in \mathit{Reach}_i$ represents the copy of method m in component C_i
- A call graph with nodes $m^i \in \mathit{Reach}$ and edges $e \in \mathit{Reach} \times \mathit{Reach} \times \mathit{CallSites} \times \{L, R\}$. A remote call graph edge $(m^i, n^j, c)_R$ indicates that method m in component C_i contains a call site c at which one possible remote run-time target method is n in component C_j. A local call graph edge $(m^i, n^i, c)_L$ shows that method m in C_i contains a call site c at which one possible non-remote run-time target method is n in the same component.

```
input    program code
output   $PAG = (N, E)$ with $N \subseteq V \cup O \cup (V \times F) \cup (O \times F)$ and $E \subseteq N \times N$;
             initialized to $(\emptyset, \emptyset)$
         $Pt_L : (V \cup (O \times F)) \rightarrow 2^O$ and $Pt_R : (V \cup (O \times F)) \rightarrow 2^O$;
             for all $x$, $Pt_{L/R}(x)$ are initialized to $\emptyset$
         $\mathit{Reach}$ : set of reachable methods; initialized to $\emptyset$
         $\mathit{CallGraph}$ : call graph; the set of nodes is $\mathit{Reach}$; initialized to $(\emptyset, \emptyset)$
declare  $\mathit{NodeWorklist}$ : set of $PAG$ nodes; initialized to $\emptyset$
         $\mathit{MethodWorklist}$ : set of methods; initialized to $\emptyset$
[1]      $\mathit{Reach} := \bigcup_i \{\mathit{MainMethod}^i\} \cup \{\mathit{StaticInitializerOfStartClass}^i\} \cup \{\mathit{JVMStartupMethods}^i\}$
[2]      foreach $m^i \in \mathit{Reach}$ do $\mathit{ProcessBody}(m^i)$
[3]      $\mathit{PropagatePointsToSets}$
```

Figure A.1: Top level of the analysis algorithm

Top Level. The top-level functionality of the algorithm is shown in Figure A.1. Line 1 initializes *Reach* with (1) the main method of each component C_i, (2) the static initializer (if any) of the class containing that main method, and (3) the set of library methods that are executed in C_i at JVM startup, before the main method is invoked. We assume that all initialization code for static fields inside a class (i.e., initialization expressions in field declarations [21, Section 8.3.2.1] and static initialization blocks [21, Section 8.7]) is combined in an artificial static method *StaticInitializer* for that class. Since the class containing the main method for a component C_i could be initialized as a result of the invocation of that main method, the code in *StaticInitializer* should be considered executable and this artificial method should be added to *Reach*.

ProcessBody. For each method m^i in this initial set *Reach*, the analysis processes the method body using procedure *ProcessBody*. Later, during the rest of the analysis, this procedure is also executed (exactly once) on each newly discovered reachable

```
procedure $ProcessBody(m^i)$
[4]  foreach statement $st$ in $m^i$ do
[5]      if $st$ is $v = new\ X$ then $ProcessAllocation(node(s^i), node(v^i))$
[6]      if $st$ is $v_1 = v_2$ then add edge $node(v_2^i) \longrightarrow node(v_1^i)$ to $PAG$
[7]      if $st$ is $v_1 = v_2.fld$ then add edge $node(v_2^i.fld) \longrightarrow node(v_1^i)$ to $PAG$
[8]      if $st$ is $v_1.fld = v_2$ then add edge $node(v_2^i) \longrightarrow node(v_1^i.fld)$ to $PAG$
[9]      if $st$ is $v = X.fld$ then add edge $node(X.fld^i) \longrightarrow node(v^i)$ to $PAG$
[10]     if $st$ is $X.fld = v$ then add edge $node(v^i) \longrightarrow node(X.fld^i)$ to $PAG$
[11] foreach method $n^j \in Reach$ for which $ProcessBody$ has already been executed do
[12]     foreach initial remote references $(v^i, w^j) \in I_{i \to j}$ where $v^i \in Locals(m^i)$
         and $w^j \in Locals(n^j)$ do
[13]         add edge $node(v^i) \xrightarrow{remote} node(w^j)$ to $PAG$
[14]         add $node(v^i)$ to $NodeWorklist$
[15]     foreach initial remote references $(w^j, v_i) \in I_{j \to i}$ where $v^i \in Locals(m^i)$
         and $w^j \in Locals(n^j)$ do
[16]         add edge $node(w^j) \xrightarrow{remote} node(v^i)$ to $PAG$
[17]         add $node(w^j)$ to $NodeWorklist$
[18] foreach monomorphic call site $c$ in $m^i$, where the invoked method is $n^i$ do
[19]     $AddCallGraphEdge(m^i, n^i, c, L)$
```

Figure A.2: Processing the body of a newly discovered reachable method

method. Given a method body, PAG edges are created to represent the value-flow semantics of the statements inside that body (lines 4–10 in Figure A.2), excluding the effects of method calls. For an assignment $v = new\ X$, procedure *ProcessAllocation* (see Figure A.6) adds s^i to $Pt_L(v^i)$ and puts $node(v^i)$ on a worklist *NodeWorklist* of PAG nodes. Throughout the algorithm, the elements of this worklist are PAG nodes $node(v^i)$ whose points-to sets may need to be propagated to other PAG nodes—for example, to $node(w^i)$ due to a PAG edge $node(v^i) \longrightarrow node(w^i)$.

Next, *ProcessBody* considers all pairs of local variables that could be used to "bootstrap" initial remote references between m^i and reachable methods n^j (lines 11–17). The source nodes of the new PAG edges are added to *NodeWorklist*, because the points-to sets of these source nodes may have to be propagated to the corresponding

```
procedure $AddCallGraphEdge(m^i, n^j, c, x)$
[20]  if $(m^i, n^j, c)_x \in CallGraph$ then return
[21]  add $(m^i, n^j, c)_x$ to $CallGraph$
[22]  if $n^j \notin Reach$ then add $n^j$ to $Reach$ and to $MethodWorklist$
[23]  foreach actual $v^i_t$ of $c$ with corresponding formal $p^j_t$ of $n^j$ do
[24]      add edge $node(v^i_t) \xrightarrow{x} node(p^j_t)$ to $PAG$
[25]      add $node(v^i_t)$ to $NodeWorklist$
[26]  add edge $node(ret^j) \xrightarrow{x} node(w^i)$ to $PAG$, where the return value at $c$ is assigned to $w$
[27]  add $node(ret^j)$ to $NodeWorklist$
```

Figure A.3: Creating a new call graph edge and the corresponding PAG edges

target nodes. Finally, at lines 18–19, for all compile-time monomorphic call sites in m^i, the corresponding call graph edges are created using procedure *AddCallGraphEdge* (this procedure is discussed below). This is done for each call site c that represents a static call, a constructor call, or a non-polymorphic instance call (e.g., call to a final method). Note that remote calls are never processed in this step, because such calls are always performed through remote interfaces. Polymorphic calls (including remote calls) are processed later, when the points-to sets of PAG nodes can be used to infer that certain receiver objects are possible at these call sites.

AddCallGraphEdge. Procedure *AddCallGraphEdge*, shown in Figure A.3, is invoked when a potential target method is detected at a call site. The procedure updates the call graph, the set of reachable methods, and the PAG. The actual parameters are the calling method m^i, the target method n^j, the call site c, and a value $x \in \{L, R\}$ signifying a local or a remote call. If edge $(m^i, n^j, c)_x$ is new, it is added to the call graph. If the target method n^j is discovered for the first time, it is added to the set of reachable methods. The new method is also added to a worklist *MethodWorklist* of newly discovered reachable methods; as discussed later, each method on this worklist is eventually processed using *ProcessBody*. Each reachable method (except for the initial methods from line 1 in Figure A.1) is added to *MethodWorklist* exactly once.

Lines 23–27 in Figure A.3 create new PAG edges representing the flow of values due to actual-formal parameter bindings and due to return values. The newly created PAG edges are remote if the call graph edge is remote, and local otherwise. The source nodes of the new edges are added to *NodeWorklist*, because their points-to sets may contain objects that need to be propagated to the PAG nodes for p_t^j and w^i.

PropagatePointsToSets. Procedure *PropagatePointsToSets* in Figure A.4 contains the main loop of the algorithm. This loop propagates objects to points-to sets and performs on-the-fly call graph construction. The procedure completes when there are no more objects to be propagated to points-to sets. The completion is signaled by two conditions (line 43):

1. Worklist *NodeWorklist* is empty, meaning that there are no more variables with points-to sets that need to be propagated.

2. Boolean variable *SerializableToPropagate* is *false*, meaning that there are no more object fields *o.fld* whose points-to sets need to be propagated. As described below, a value *true* for this flag indicates that points-to sets may have changed for object fields that are subject to serialization at remote calls, and therefore further propagation may be necessary.

Each iteration of the inner loop processes a PAG node $node(v_i)$. This node was earlier put on *NodeWorklist* because the (local or remote) points-to set of v^i changed, and this change required further propagation. First, the analysis considers

```
procedure $PropagatePointsToSets$
declare   $RemoteFieldEdges$ : set of PAG edges
          $SerializableToPropagate$ : boolean
[28]  repeat
[29]    repeat
[30]      remove $node(v^i)$ from $NodeWorklist$
[31]      $BuildCallGraphOnTheFly(v^i)$
[32]      foreach $node(v^i) \longrightarrow node(w^i)$ in $PAG$ do
            $processSimpleEdge(node(v^i), node(w^i))$
[33]      foreach $node(v^i) \longrightarrow node(w^i.fld)$ in $PAG$ do
            $processStoreEdge(node(v^i), node(w^i.fld))$
[34]      foreach $node(w^i) \longrightarrow node(v^i.fld)$ in $PAG$ do
            $processStoreEdge(node(w^i), node(v^i.fld))$
[35]      foreach $node(v^i.fld) \longrightarrow node(w^i)$ in $PAG$ do
            $processLoadEdge(node(v^i.fld), node(w^i))$
[36]      foreach $node(v^i) \xrightarrow{remote} node(w^j)$ in $PAG$ do
            $processRemoteEdge(node(v^i), node(w^j))$
[37]    until $NodeWorklist = \emptyset$
[38]    foreach $node(v^i) \longrightarrow node(w^i.fld)$ in $PAG$ do
            $processStoreEdge(node(v^i), node(w^i.fld))$
[39]    foreach $node(v^i.fld) \longrightarrow node(w^i)$ in $PAG$ do
            $processLoadEdge(node(v^i.fld), node(w^i))$
[40]    $RemoteFieldEdges := \{ e \in PAG \mid e$ is an edge $node(o_1.fld) \xrightarrow{remote} node(o_2.fld) \}$
[41]    $SerializableToPropagate := false$
[42]    foreach $e \in RemoteFieldEdges$ do $ProcessRemoteFieldEdge(source(e), target(e))$
[43]  until $NodeWorklist = \emptyset$ and $SerializableToPropagate = false$
```

Figure A.4: Propagation of objects to points-to sets

the current values of $Pt_{L/R}(v^i)$ and attempts to grow the call graph using function *BuildCallGraphOnTheFly* (line 31). Inside this function, all call graph edges related to v^i are created (if necessary) and the the points-to sets of the corresponding formals *this* are updated. Furthermore, if there are reachable methods whose bodies have not been processed yet (e.g., because they were just discovered by the call at line 47), they are removed from *MethodWorklist* and their bodies are examined. Note that the

```
procedure $BuildCallGraphOnTheFly(v^i)$
[44]    foreach polymorphic call site $c$ of the form $v.m(\ldots)$ in method $m^i$ do
[45]        foreach object $o^j \in Pt_x(v^i)$ do
[46]            $n^j$ := target method for receiver $o^j$ at call site $c$
[47]            $AddCallGraphEdge(m^i, n^j, c, x)$
[48]            $Pt_L(this_{n^j}) := Pt_L(this_{n^j}) \cup \{o^j\}$
[49]            if $Pt_L(this_{n^j})$ changed then add $node(this_{n^j})$ to $NodeWorklist$
[50]    while $MethodWorklist \neq \emptyset$ do
[51]        remove $m^i$ from $MethodWorklist$
[52]        $ProcessBody(m^i)$
```

Figure A.5: On-the-fly call graph construction

call to *ProcessBody* at line 52 could add methods to *MethodWorklist* because of the call at line 19.

After all necessary changes to the call graph, the PAG edges related to v^i are examined and the corresponding points-to sets are updated (lines 32–36). The processing of these edges is based on the points-to analysis algorithm from [36, 37] with the appropriate addition (at line 36) to handle PAG edges labeled as remote. The outer loop considers all load and store edges in the PAG (lines 38–39), not just the ones that are related to v^i. This is necessary to propagate points-to relationships that may be missed by the inner loop due to aliasing [36].

Lines 40–42 are needed to propagate the points-to sets of fields *o.fld* along remote edges corresponding to serialization at parameter passing for remote calls. Each such edge is of the form $node(o.fld) \xrightarrow{remote} node(\mu_j(o).fld)$ and represents the flow of values from field *o.fld* to the deserialized copy of that field in component C_j. Using the points-to sets of *o.fld*, the analysis updates the points-to sets of $\mu_j(o).fld$ (lines 73–76 in Figure A.7). This process may also add new remote edges between object fields (lines 77–78). If the points-to sets of $\mu_j(o).fld$ change, flag *SerializableToPropagate*

```
procedure $ProcessAllocation(node(s^i), node(v^i))$
[53]    add edge $node(s^i) \longrightarrow node(v^i)$ to $PAG$
[54]    $Pt_L(v^i) := Pt_L(v^i) \cup \{s^i\}$
[55]    add $node(v^i)$ to $NodeWorklist$

procedure $ProcessSimpleEdge(node(v^i), node(w^i))$
[56]    $Pt_L(w^i) := Pt_L(w^i) \cup Pt_L(v^i)$
[57]    $Pt_R(w^i) := Pt_R(w^i) \cup Pt_R(v^i)$
[58]    if $Pt_L(w^i)$ changed or $Pt_R(w^i)$ changed then add $node(w^i)$ to $NodeWorklist$

procedure $ProcessStoreEdge(node(v^i), node(w^i.fld))$
[59]    foreach $o \in Pt_L(w^i)$ do
[60]        $Pt_L(o.fld) := Pt_L(o.fld) \cup Pt_L(v^i)$
[61]        $Pt_R(o.fld) := Pt_R(o.fld) \cup Pt_R(v^i)$

procedure $ProcessLoadEdge(node(v^i.fld), node(w^i))$
[62]    foreach $o \in Pt_L(v^i)$ do
[63]        $Pt_L(w^i) := Pt_L(w^i) \cup Pt_L(o.fld)$
[64]        $Pt_R(w^i) := Pt_R(w^i) \cup Pt_R(o.fld)$
[65]    if $Pt_L(w^i)$ changed or $Pt_R(w^i)$ changed then add $node(w^i)$ to $NodeWorklist$
```

Figure A.6: Propagation along non-remote PAG edges

is raised to ensure that at least one more iteration of the outer loop at lines 28–43 is executed, in order to propagate these changed sets. It is easy to show that when new remote PAG edges are created at line 78, line 79 is guaranteed to raise the flag, thus forcing future propagation along those new edges. When the flag remains *false* after line 42, neither the PAG nor the points-to sets have changed due to the effects of object serialization. Together with $NodeWorklist \neq \emptyset$, this guarantees that further propagation is not necessary.

```
procedure $\mathit{ProcessRemoteEdge}(node(v^i), node(w^j))$
[66]    $Pt_R(w^j) := Pt_R(w^j) \cup \{\, o \mid o \in Pt_L(v^i) \ \wedge\ o \textit{ is remote} \,\}$
[67]    $Pt_R(w^j) := Pt_R(w^j) \cup Pt_R(v^i)$
[68]    foreach non-remote serializable $o \in Pt_L(v^i)$ do
[69]        $Pt_L(w^j) := Pt_L(w^j) \cup \{\, \mu_j(o) \,\}$
[70]        foreach non-transient field $\mathit{fld}$ of $o$ do
[71]            add edge $node(o.\mathit{fld}) \xrightarrow{remote} node(\mu_j(o).\mathit{fld})$ to $PAG$,
                    if the edge does not exist already
[72]    if $Pt_L(w^j)$ changed or $Pt_R(w^j)$ changed then add $node(w^j)$ to $\mathit{NodeWorklist}$

procedure $\mathit{ProcessRemoteFieldEdge}(node(o.\mathit{fld}), node(\mu_j(o).\mathit{fld}))$
[73]    $Pt_R(\mu_j(o).\mathit{fld}) := Pt_R(\mu_j(o).\mathit{fld}) \cup \{\, o' \mid o' \in Pt_L(o.\mathit{fld}) \ \wedge\ o' \textit{ is remote} \,\}$
[74]    $Pt_R(\mu_j(o).\mathit{fld}) := Pt_R(\mu_j(o).\mathit{fld}) \cup Pt_R(o.\mathit{fld})$
[75]    foreach non-remote serializable $o' \in Pt_L(o.\mathit{fld})$ do
[76]        $Pt_L(\mu_j(o).\mathit{fld}) := Pt_L(\mu_j(o).\mathit{fld}) \cup \{\, \mu_j(o') \,\}$
[77]        foreach non-transient field $\mathit{fld2}$ of $o'$ do
[78]            add edge $node(o'.\mathit{fld2}) \xrightarrow{remote} node(\mu_j(o').\mathit{fld2})$ to $PAG$,
                    if the edge does not exist already
[79]    if $Pt_L(\mu_j(o).\mathit{fld})$ changed or $Pt_R(\mu_j(o).\mathit{fld})$ changed then
            $\mathit{SerializableToPropagate} := \mathit{true}$
```

Figure A.7: Propagation along remote PAG edges

BIBLIOGRAPHY

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

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

[3] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDD's. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 103–114, 2003.

[4] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 103–114, 2003.

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

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

[7] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 47–56, 1988.

[8] R. Chatterjee and B. G. Ryder. Data-flow-based testing of object-oriented libraries. Technical Report DCS-TR-433, Rutgers University, 2001.

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

[10] B. Cheng and W. Hwu. Modular interprocedural pointer analysis using access paths. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–69, 2000.

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

[12] M. Codish, S. Debray, and R. Giacobazzi. Compositional analysis of modular logic programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 451–464, 1993.

[13] P. Cousot and R. Cousot. Modular static program analysis. In *International Conference on Compiler Construction*, LNCS 2304, pages 159–178, 2002.

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

[15] M. Dwyer. Modular flow analysis of concurrent software. In *International Conference on Automated Software Engineering*, pages 264–273, 1997.

[16] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[17] C. Flanagan and M. Felleisen. Componential set-based analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 235–248, 1997.

[18] 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.

[19] S. Ghemawat, K. Randall, and D. Scales. Field analysis: Getting useful and low-cost interprocedural information. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 334–344, 2000.

[20] S. Ghosh, N. Bawa, S. Goel, and Y. R. Reddy. Validating run-time interactions in distributed Java applications. In *IEEE International Conference on Engineering of Complex Computer Systems*, pages 7–16, 2002.

[21] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Addison-Wesley, 3 edition, 2005.

[22] W. Grosso. *Java RMI.* O'Reilly, 2002.

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

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

[25] S. Guyer. *Incorporating Domain-Specific Information into the Compilation Process.* PhD thesis, University of Texas, Austin, 2003.

[26] S. Guyer and C. Lin. Optimizing the use of high performance software libraries. In *International Workshop on Languages and Compilers for Parallel Computing*, LNCS 2017, pages 227–243, 2000.

[27] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 154–163, 1994.

[28] M. J. Harrold and G. Rothermel. Separate computation of alias information for reuse. *IEEE Transactions on Software Engineering*, 22(7):442–460, July 1996.

[29] B. Haumacher and M. Philippsen. Exploiting object locality in JavaParty, a distributed computing environment for workstation clusters. In *Workshop on Compilers for Parallel Computers*, pages 83–94, June 2001.

[30] N. Heintze. *Set Based Program Analysis.* PhD thesis, Carnegie Mellon University, 1992.

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

[32] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.

[33] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, Jan. 1990.

[34] W. Landi and B. G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 235–248, 1992.

[35] Y.-F. Lee and B. G. Ryder. A comprehensive approach to parallel data flow analysis. In *ACM International Conference on Supercomputing*, pages 236–247, 1992.

[36] O. Lhoták. Spark: A scalable points-to analysis framework for Java. Master's thesis, McGill University, Dec. 2002.

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

[38] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *Static Analysis Symposium*, LNCS 2126, pages 279–298, 2001.

[39] 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, June 2001.

[40] D. Liang, M. Pennings, and M. J. Harrold. Evaluating the impact of context-sensitivity on Andersen's algorithm for Java programs. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2005.

[41] B. Livshits, J. Whaley, and M. Lam. Reflection analysis for Java. In *Asian Symposium on Programming Languages and Systems*, LNCS 3780, pages 139–160, 2005.

[42] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems*, 23(6):747–775, Nov. 2001.

[43] T. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184–196, 1990.

[44] Sun Microsystems. *RMI Specification.* Sun Microsystems, 2002.

[45] Sun Microsystems. *Serialization Specification.* Sun Microsystems, 2003.

[46] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–11, 2002.

[47] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, Jan. 2005.

[48] M. Philippsen and B. Haumacher. Locality optimization in JavaParty by means of static type analysis. *Concurrency: Practice and Experience*, 12(8):613–628, July 2000.

[49] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, May 2000.

[50] B. Quig, J. Rosenberg, and M. Kölling. Supporting interactive invocation of remote services. In *International Conference on Principles and Practice of Programming in Java*, pages 195–200, 2003.

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

[52] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[53] T. Reps, M. Sagiv, and S. Horwitz. Interprocedural dataflow analysis via graph reachability. Technical Report DIKU-TR94-14, University of Copenhagen, Apr. 1994.

[54] A. Rountev. *Dataflow Analysis of Software Fragments.* PhD thesis, Rutgers University, Aug. 2002. Available as Techical Report DCS-TR-501.

[55] A. Rountev. Precise identification of side-effect-free methods in Java. In *IEEE International Conference on Software Maintenance*, pages 82–91, 2004.

[56] A. Rountev. Component-level dataflow analysis. In *International SIGSOFT Symposium on Component-Based Software Engineering*, LNCS 3489, pages 82–89, 2005.

[57] A. Rountev and B. H. Connell. Object naming analysis for reverse-engineered sequence diagrams. In *International Conference on Software Engineering*, pages 254–263, 2005.

[58] A. Rountev, S. Kagan, and T. Marlowe. Interprocedural dataflow analysis in the presence of large libraries. In *International Conference on Compiler Construction*, LNCS 3923, pages 2–16, 2006.

[59] A. Rountev, S. Kagan, and T. Marlowe. Interprocedural dataflow analysis in the presence of large libraries. Technical Report OSU-CISRC-1/06-TR01, Ohio State University, Jan. 2006.

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

[61] 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, 2003.

[62] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in Java software. *IEEE Transactions on Software Engineering*, 30(6):372–387, June 2004.

[63] A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *International Conference on Compiler Construction*, LNCS 2027, pages 20–36, 2001.

[64] A. Rountev, B. G. Ryder, and W. Landi. Data-flow analysis of program fragments. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, LNCS 1687, pages 235–252, 1999.

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

[66] R. Rugina and M. Rinard. Design-driven compilation. In *International Conference on Compiler Construction*, LNCS 2027, pages 150–164, 2001.

[67] 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.

[68] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167:131–170, 1996.

[69] 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.

[70] M. Sharp and A. Rountev. Static analysis of object references in RMI-based Java software. In *IEEE International Conference on Software Maintenance*, pages 101–110, 2005.

[71] M. Sharp and A. Rountev. Static analysis of object references in RMI-based Java software. *IEEE Transactions on Software Engineering*, 32(9):664–681, Sept. 2006.

[72] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering*, 26(9):849–871, Sept. 2000.

[73] V. Sreedhar, M. Burke, and J. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–207, 2000.

[74] M. Sridharan, D. Gopan, L. Shan, and R. Bodik. Demand-driven points-to analysis for Java. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 59–76, 2005.

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

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

[77] P. Sweeney and F. Tip. Extracting library-based object-oriented applications. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 98–107, 2000.

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

[79] F. Tip, P. Sweeney, C. Laffra, A. Eisma, and D. Streeter. Practical extraction techniques for Java. *ACM Transactions on Programming Languages and Systems*, 24(6):625–666, 2002.

[80] S. Triantafyllis, M. Bridges, E. Raman, G. Ottoni, and D. August. A framework for unrestricted whole-program optimization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 61–71, 2006.

[81] 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.

[82] R. Veldema and M. Philippsen. Compiler optimized remote method invocation. In *IEEE International Conference on Cluster Computing*, pages 127–137, 2003.

[83] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, 2001.

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

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