The Separate Compilation Assumption

by

Karim Ali

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2014

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

I would like to acknowledge the names of my co-authors who contributed to the research described in this dissertation. These include:

- Dr. Ondřej Lhoták
- Dr. Frank Tip
- Dr. Julian Dolby
- Marianna Rapoport

Abstract

Call graphs are an essential requirement for almost all inter-procedural analyses. This motivated the development of many tools and frameworks to generate the call graph of a given program. However, the majority of these tools focus on generating the call graph of the whole program (i.e., both the application and the libraries that the application depends on). A popular compromise to the excessive cost of building a call graph for the whole program is to build an application-only call graph. To achieve this, all the effects of the library code and any calls that the library makes back into the application are usually ignored. This results in potential unsoundness in the generated call graph and therefore in analyses that use it. Additionally, the scope of the application classes to be analyzed by such an algorithm has been often arbitrarily defined.

In this thesis, we define the *separate compilation assumption*, which clearly defines the division between the application and the library based on the fact that the code of the library has to be compiled without access to the code of the application. We then use this assumption to define more specific constraints on how the library code can interact with the application code. These constraints make it possible to generate sound and reasonably precise call graphs without analyzing libraries.

We investigate whether the separate compilation assumption can be encoded universally in Java bytecode, such that all existing whole-program analysis frameworks can easily take advantage of it. We present and evaluate Averroes, a tool that generates a placeholder library that over-approximates the possible behaviour of an original library. The placeholder library can be constructed quickly without analyzing the whole program, and is typically in the order of 80 kB of class files (comparatively, the Java standard library is 25 MB). Any existing whole-program call graph construction framework can use the placeholder library as a replacement for the actual libraries to efficiently construct a sound and precise application call graph.

Averroes improves the analysis time of whole-program call graph construction by a factor of 3.5x to 8x, and reduces memory requirements by a factor of 8.4x to 12x. In addition, Averroes makes it easier for whole-program frameworks to handle reflection soundly in two ways: it is based on conservative assumptions about all behaviour within the library, including reflection, and it provides analyses and tools to model reflection in the application. We also evaluate the precision of the call graphs built with Averroes in existing whole-program frameworks. Finally, we provide a correctness proof for Averroes based on Featherweight Java.

Acknowledgements

First and foremost, all thanks are due to God, The most merciful, The most compassionate. I would not have made it this far without His continuous guidance and support.

Sarah, my wife and the love of my life, without whom my life is a complete mess. I cannot thank you enough for being very supportive at times when I had loads of work to do, giving me hope when I thought I cannot do this any more. I think you are entitled to claim no less than half of this PhD. I cannot wait to start a new chapter in our life together in Germany. Like you said, now we are "a couple of doctors" :-)

My dear parents, Assaad and Nabila, you guys are the ones who put me on this path by giving me the best education I could get. Then you bared with me going away while being your only son, which I know was really hard for you. I hope that all those years of anticipation have come to a good ending.

My in-laws, whom I consider as the natural extension to my family. I am eternally grateful to you guys for your continuous support during all the hard times Sarah and I have been through. Amr, who is my brother from another mother. Ghada, Aly, Hussein, and the new addition to their family Yousif. I love you guys!

Ondřej Lhoták, the best supervisor one could ever wish for. You are one of the kindest and smartest people I have seen in my life. You taught me what to do, and not to do, in order to be a good supervisor. I am not sure though if I will ever match that with my future students. I was very fortunate to be advised by you during my PhD, and I am sure all of your previous/current students feel the same. I wish you best of luck with your future endeavours, and hope that our paths cross again. You will not get rid of me that easily!

Frank Tip, I pretty much consider you my co-supervisor. Even Ondřej once thought you really are! You have been a great mentor during the two years that we have worked together. You have taught me how to be organized in my work, and how to always think about the big picture of a project. I am sure we will continue working together on many more projects.

I would like to also thank Werner Dietl, Reid Holmes, and my external examiner Jan Vitek for serving on my committee. Your comments and remarks have served me well in making this dissertation better.

I cannot forget the support I got from my colleagues in the PLG lab, especially my friend Marianna who brings a lot of energy to the lab. I know she will not agree with this, as usual, but I am sure she will make one hell of a researcher during her PhD and afterwards. I wish you and Abel best of luck.

The gang, my closest and dearest friends: Shafei, Sherif. You guys have always been there, especially Sherif visiting us from almost everywhere in the North America (fifi poutine). And Shafei whose parents duaa is probably one of the reasons I have made it this far.

Thank you Ramy, Khairy, Hadidy, Hamouda, Abdelrazik, Bayoumy, Khalifa for all the wonderful memories we have made together in Waterloo. I will always cherish them.

The Menoufys, Mona and Adham! You were more than just friends to Sarah and myself. You were neighbours, friends, partners in many crimes, adventures, and so many things that brought joy and action to our lives. We will truly miss you guys when we move to Germany, especially the little ones Maya and Laila. I am sure we will get to meet each other again. Who knows, we might go back to Canada after all!

Karim Ali
University of Waterloo
September 2014

To the brave people who strive to rid this world of tyranny, injustice, and occupation ...

Table of Contents

List of Tables

List of Figures

"O my Lord! advance me in knowledge."

[The Holy Qur'an, Ta-Ha, 20:114]

Chapter 1

Introduction

Starting four decades ago, call graphs were first used for inter-procedural analysis and optimization [5]. To this day, call graphs are used by various software tools to compute the calls between methods in a given program. Such information is necessary for compilers to determine whether specific optimizations can be applied. For example, a compiler can use a call graph to eliminate unreachable code (e.g., debugging code and unused parts of libraries) [24], propagate constants across a program [27], or to inline statically-bound calls [24]. Additionally, numerous software engineering tools use call graph information to help software engineers increase their understanding of a program. For example, an integrated development environment (IDE), like Eclipse, uses call graph information to provide features like code navigation and completion [34], "Jump to Declaration" [21], and automated code refactoring [49].

The call graph of a program is the relation describing exactly the calls made from one method to another in all possible executions of that program. However, computing this relation is undecidable, as the halting problem can be reduced to the precise call graph construction problem [39]. Therefore, we compute over-approximations of the precise call graph using static analysis. Such over-approximations are typically referred to as static call graphs. For every method invocation instruction in the program (a *call site*), the static call graph contains an *edge* to every *target* method that might be invoked by that instruction.

In Java, as well as other object-oriented languages, the targets of method calls are selected using the runtime type of the receiver object. Therefore, static call graph construction requires a combination of two inter-related static analyses: (1) calculating the sets of possible receiver types, and (2) determining the targets of method calls. Determining the targets of calls requires computing two sets: *reachable methods* and *call edges*. On the other hand, a points-to analysis is usually required to precisely determine the sets of possible receiver types. A typical points-to analysis is defined by two parameters: *points-to sets* and *points-to constraints*. Figure 1.1 depicts the four parameters of static call graph construction and their inter-dependencies.

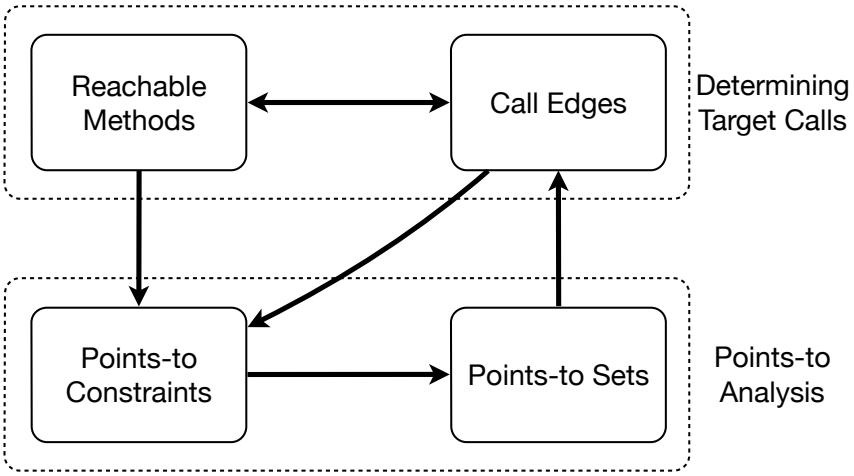


Figure 1.1: Inter-dependent parameters of a static call graph construction algorithm.

1.1 Static Call Graph Construction

The main purpose of a static call graph construction algorithm is to compute the set of call edges. A call edge connects a call site to a target method that may be invoked from that call site. Figure 1.1 shows that computing the set of call edges depends on two parameters: reachable methods and points-to sets. First, we are only interested in call sites that do not occur in unreachable code. Therefore, a precise static call graph construction algorithm keeps track of the set of methods that are transitively reachable from the entry points of the program (e.g., its `main()` method). Second, the target of a given call depends on the runtime type of the receiver of the call. A precise static call graph construction algorithm therefore computes the points-to sets abstracting the objects that each variable could point to. There are two common methods of abstraction to represent objects: either by their allocation site (from which their runtime type can be deduced) or by their runtime type. Thus, the points-to set of a variable at a call site over-approximates the runtime types of the receiver of that call.

Points-to sets are computed by finding the least fixed-point solution of a system of subset constraints [7] that model all possible assignments between variables in the program. Thus, an abstract object "flows" from its allocation site into the points-to sets of all variables to which it could be assigned. Eventually, the abstract object reaches all of the call sites at which its methods could be called. The dependency of the points-to set on the set of call edges is illustrated in Figure 1.1. The calculation of the points-to sets is expressed in terms of the points-to constraints. The points-to constraints model intra-procedural assignments between variables due to explicit instructions within methods. They also model inter-procedural assignments due to parameter passing and returns from methods. Since only intra-procedural assignments in reachable methods

are considered, the set of points-to constraints depends on the set of reachable methods. The set of call edges is another dependency because it determines the inter-procedural assignments.

Finally, the set of reachable methods depends, of course, on the set of call edges. A method n is reachable if there exists a call edge $(m \rightarrow n)$ that leads to it. A precise static call graph construction algorithm computes these four inter-dependent relations concurrently until it reaches a mutual least fixed-point. This is commonly referred to as *on-the-fly* call graph construction.

1.2 Whole-Program Call Graph Construction

A static call graph is a necessary prerequisite for most inter-procedural analyses used in compilers, verification tools, and program understanding tools [35]. However, constructing sound and precise static call graphs for object-oriented programs is often difficult and expensive.[1] The key reason is dynamic dispatch: the target of a method call depends on the runtime type of the receiver of the call. One approach, Class Hierarchy Analysis (CHA) [16], conservatively assumes that the receiver could be any object admitted by the statically declared type of the receiver of the call. Because call graphs constructed with this assumption are imprecise, most call graph construction algorithms attempt to track the flow of potential receivers through the program [1, 12, 29, 36, 56]. Since the receiver of the call could have been created anywhere in the program, these algorithms generally analyze the whole program. However, common libraries (e.g., the Java standard library) are usually large in size, making it very expensive to construct a call graph, even for a small object-oriented program. For example, constructing the call graph of a Java "Hello, World!" program using SPARK [37], with JDK 1.4 as the Java standard library, takes up to 30 seconds, and produces a call graph with more than 5,000 reachable methods and more than 23,000 call edges.

In general, even if the call graph construction algorithm itself is cheap, just reading all of the library dependencies of a program takes a long time.[2] Additionally, in many cases, the whole program may not even be available for analysis. This generated interest in the development of algorithms that analyze only parts of a program.

1.3 Partial-Program Call Graph Construction

A partial call graph is a call graph that explicitly represents the call relations between the analyzed parts of the program, usually the application, and summarizes the unanalyzed parts of the program, usually the library, by a single node in the graph. Partial call graph construction is an

[1] Hereafter, we will use the terms "call graph" and "static call graph" interchangeably.

[2] Hereafter, we will use the singular "library" to refer to all of the libraries that a program depends on.

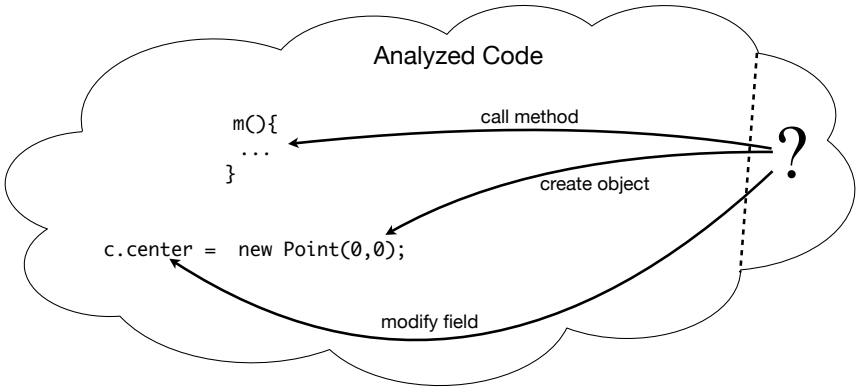


Figure 1.2: Conservative assumptions that a sound partial-program analysis must make.

often-requested feature in static analysis frameworks for Java. On the mailing list of Soot [56], which analyzes the whole program to construct its call graph, dozens of users have requested support for partial call graph construction [10].

A common approach for constructing partial call graphs, used for example in Wala [29], is to define an analysis scope of the classes to be analyzed. The analysis scope then represents the application part of the program. The effects of code outside this scope (i.e., the library) are ignored. As a consequence, the generated call graph lacks edges representing callbacks from the library to the application. Methods that should be reachable due to those callback edges are ignored as well. Since this approach ignores the effects of library code, any store or load operation in the library that involves an object created in the application is ignored. Therefore, the points-to sets of the application objects will be incomplete, potentially causing even more call graph edges to be missing.

A whole-program call graph is sound if the set of call edges it computes is a superset of the possible runtime calls in the whole program. On the other hand, a partial-graph is sound if the set of call edges within the application part of the program is a superset of the possible runtime calls that could take place within the same application part. In general, if a sound call graph is to be constructed without analyzing the whole program, conservative assumptions must be made for all four of the inter-dependent relations in Figure 1.1. A sound analysis must assume that any unanalyzed code could do "anything". In particular, the unanalyzed code could call any method, assign any value to any field, and create new objects of any type, as summarized in Figure 1.2. Due to the dependencies between the four relations, imprecision in any one relation can quickly pollute the others. Therefore, without any assumptions about the unanalyzed part of the code, a sound algorithm generates a call graph that is so imprecise that it is useless. However, it is frequently the case that the unanalyzed code is a library that is developed separately and can be compiled without access to the rest of the program.

1.4 Motivation

Our goal is to construct a partial call graph that soundly over-approximates the set of targets of every call site, and the set of reachable methods in the analysis scope. Such call graph uses a single summary node to represent all methods in the library. However, the call graph should be accurate for the application code. To achieve this, we have made less conservative assumptions about the library code, which is not analyzed, while still generating a precise and sound call graph for the application. The essential observation behind our approach is that the division between an application and its library is not arbitrary. If the analysis scope could be any set of classes, then the call graph would necessarily be very imprecise. In particular, a sound analysis would have to assume that the unanalyzed code could call any non-private method and modify any non-private field in the analysis scope.[3]

A realistic, yet useful assumption, is that the code of the library has been compiled without access to the code of the application. We refer to this as the *separate compilation assumption*. From this assumption, we can deduce more specific restrictions on how the library can interact with the application. In particular, the library cannot call a method, access a field, or instantiate a class of the application if the library author does not know the name of that method, field, or class. It is theoretically possible to discover this information using reflection, and some special-purpose "libraries" such as JUnit [41] actually do so. We assume that such reflective poking into the internals of an application is rare in most general-purpose libraries. Section 4.3.3 provides more details about handling reflection in the context of the separate compilation assumption.

1.5 Thesis Research Theme

The *separate compilation assumption* enables the construction of a precise and sound call graph for the part of the program that is analyzed (the *application*) without analyzing its libraries. The call graph soundly over-approximates the set of targets of every call site in the application part of a program (the analysis scope), while using a single summary node to represent all methods in the library. Calls from application methods to library methods and vice versa are represented as call edges to and from the library summary node, respectively. A call edge is created for each possible call between application methods, but no edges are created to represent calls within the library. It is implicitly assumed that any library method could call any other library method. Similarly, a single summary points-to set is used to represent the points-to sets of all variables within the library. Intra-library pointer flow, however, is not tracked precisely.

More formally, a partial call graph for a program P consisting of an application A and a library L is sound if for every call $m \mapsto m'$ that can occur at runtime:

[3] Some field modifications could theoretically be ruled out if an escape analysis determined that some objects are not reachable through the heap from any objects available to the unanalyzed code.

```
1 public class Main {
2   public static void main(String[] args) {
3     MyHashMap<String,String> myHashMap = new MyHashMap<String,String>();
4     System.out.println(myHashMap);
5   }
6 }
7
8 public class MyHashMap<K,V> extends HashMap<K,V> {
9   public void clear() { }
10  public int size() { return 0; }
11  public String toString() { return "MyHashMap"; }
12 }
```

Figure 1.3: A sample Java program that will be used for demonstration.

1. if both m and m' are in A, then there exists an edge from m to m' in the partial call graph,
2. if m is in A and m' is in L, then there exists an edge from m to the library summary node,
3. if m is in L and m' is in A, then there exists an edge from the library summary node back to m', and
4. if both m and m' are in L, then the call is ignored in the partial call graph.

Figure 1.3 shows a sample Java program that we will use to demonstrate our approach. Figure 1.4 compares two branches from the call graphs generated for this sample program by (a) whole-program analysis and (b) partial-program analysis (e.g., using the separate compilation assumption). Both branches are computed by following the paths from the entry-point method of the call graph, `Main.main()`. In Figure 1.4a, we can see that the first branch shows all of the call edges from the method `MyHashMap.<init>()` all the way up to `java.lang.Object.<init>()`. The second branch in the call graph shows the call edges between library methods (e.g., the call edge from the method `java.io.PrintStream.println(Object)` to the method `java.lang.String.valueOf(Object)`). The target of that edge calls back to the application method `MyHashMap.toString()`.

On the other hand, Figure 1.4b shows how a partial call graph represents the same branches. All of the edges beyond the predefined point of interest of the user (i.e., the application classes `MyHashMap` and `Main`) are considered as part of the library and are not explicitly represented in the graph. Therefore, all library methods are reduced to one library node that can have edges to and from application methods. The figure also shows that even for such a small sample program, the graph generated using the separate compilation assumption is easier to visualize and inspect. This will give the users a more focused view of the classes they are interested in, similar to what

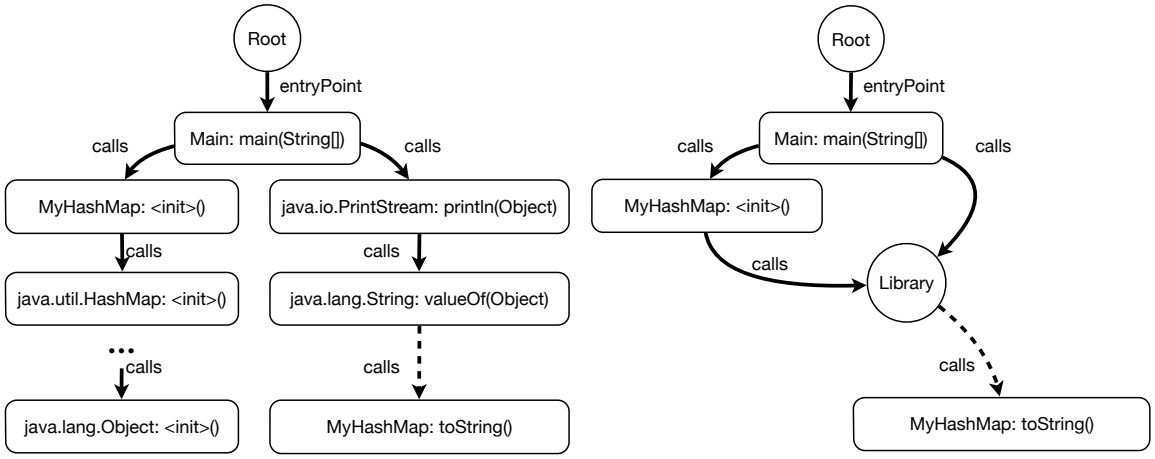


Figure 1.4: Two branches from the call graph for the sample program in Figure 1.3 as computed by (a) a whole-program call graph analysis and (b) a partial-program call graph analysis using the separate compilation assumption. The dashed line represents a call from a library method to an application method (i.e., a library callback).

they would do during manual code inspection [28]. Having a more focused and precise view of the call graph should not ignore any of the potential call edges. Ignoring the library callback edge in Figure 1.4a, for example, will render the generated call graph unsound. Thus, it is crucial to precisely define, based on the separate compilation assumption, how the library summary node interacts with the application methods in the call graph.

We briefly introduce and evaluate CGC, a proof-of-concept extension to DOOP [12] that constructs call graphs using a pointer analysis based on the separate compilation assumption. We next investigate whether the separate compilation assumption, and the constraints that follow from it, can be encoded in a form that is universal to all Java program analysis frameworks, the Java bytecode. Our goal is to enable any existing whole-program analysis framework to take advantage of the separate compilation assumption without modifications to the framework.

To accomplish this, we present AVERROES, a Java bytecode generator that, for a given program, generates a replacement for the program's library that embodies the constraints that follow from the separate compilation assumption. An existing, unmodified whole-program analysis framework needs only to read the replacement library instead of the original library to automatically gain the benefits of the separate compilation assumption. For example, instead of going through all of the work that was necessary to implement CGC, one can now achieve the same effect automatically by running AVERROES followed by DOOP. The same adaptation can be applied not only to DOOP, but to any other whole-program call graph construction framework.

1.6 Results

In this thesis, we evaluate the hypothesis that the separate compilation assumption is sufficient to construct precise call graphs. The evaluation process is twofold. First, we evaluate soundness by comparing call graphs produced by AVERROES-based tools against the dynamic call graphs observed at run time by *J [20]. Second, we evaluate precision by comparing call graphs produced by AVERROES-based tools against the call graphs constructed by whole-program analysis (SPARK [37] and DOOP [12]).

We also evaluate the performance improvements that AVERROES offers for the whole-program analysis frameworks SPARK and DOOP. Our experiments have shown that the improvements are very significant because the replacement library is much smaller than the original library. For example, version 1.4 of the Java standard library contains 25 MB of class files, whereas the AVERROES replacement library contains in the order of only 80 kB of class files. Depending on the size of the analyzed client program, AVERROES improves the running time of SPARK and DOOP by a factor of 3.5x and 2.8x, respectively, and reduces memory requirements by a factor of 12x and 8.4x, respectively.

AVERROES also enables other benefits in addition to performance. One such benefit is generality. For example, many whole-program analysis frameworks are designed to soundly model some specific version of the Java standard library. However, the replacement library constructed by AVERROES soundly over-approximates all possible implementations of the library that have the interface used by the client application. Therefore, AVERROES makes any existing whole-program analysis framework independent of the Java standard library version.

A related benefit is the handling of difficult features such as reflection and native methods. A whole-program analysis must correctly model in detail all such unanalyzable behaviour within the library in order to maintain soundness. On the other hand, AVERROES is automatically sound for such behaviour because it already assumes that the library could "do anything". That said, the generated library must still model reflective effects of the library on the client application (e.g., reflective instantiation of classes of the application). However, this issue is also made easier by AVERROES. Any tools or analyses that provide information about such reflective effects (e.g., analysis of strings passed to reflection methods or dynamic traces summarizing actual reflective behaviour) can be implemented in AVERROES. Whole-program analysis frameworks can then take advantage of these effects without modification.

1.7 Contributions

In this thesis, we make the following contributions.

- We identify the *separate compilation assumption* as key to partial call graph construction, and specify the assumptions about the effects of library code that can be derived from it.
- We provide a correctness proof for the separate compilation assumption based on Featherweight Java [30], a minimal core calculus for Java.
- We briefly introduce CGC, a proof-of-concept extension for DOOP, written in Datalog[4], that implements partial-program call graph construction.
- We present AVERROES, a Java bytecode generator that, for a given program, generates a replacement library that embodies the constraints that follow from the separate compilation assumption.[5]
- We empirically show that the separate compilation assumption is sufficient for constructing precise and sound application-only call graphs.
- We identify the performance gains of using an AVERROES replacement library as opposed to the original library of a Java program.

1.8 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 discusses related work. Chapter 3 defines the *separate compilation assumption* and the constraints that follow from it. In Chapter 4, we explain how AVERROES encodes the constraints that follow from the separate compilation assumption as Java bytecode instructions. Chapter 5 provides an extensive evaluation with respect to soundness, precision and performance improvements of using the replacement library generated by AVERROES instead of the original library code. Chapter 6 provides a correctness proof for the separate compilation assumption based on Featherweight Java. Chapter 7 summarizes our conclusions for the work presented in this thesis.

[4] Datalog is a logic-based language for (recursively) defining relations.

[5] AVERROES implements the constraints of the separate compilation assumption for versions of Java up to 1.7. Java 8 introduces additional features such as `invokedynamic` that would require additional support in AVERROES.

Chapter 2

Related Work

Call graphs are the cornerstone of most inter-procedural analyses used in program analysis tools to provide a wide range of tasks such as refactoring, bug-finding, code optimization, and code verification. Therefore, the problem of call graph construction, as well as pointer analysis, has received significant attention. In this chapter, we present a survey of the work most related to the area of call graph construction and other topics discussed in this dissertation.

Chapter Organization. In Section 2.1, we present early work on call graph construction and pointer analysis. Section 2.2 then gives a brief overview of the most popular pointer analysis frameworks commonly used to generate call graphs. In Section 2.3, we describe a line of work which enables demand-driven and incremental pointer analysis. Section 2.4 presents a category of algorithms that are capable of analyzing fragments of a program, in the absence of the rest of the program. In Section 2.5, we discuss previous research which determines objects that may escape from a predefined analysis scope.

2.1 Call Graph Construction

A differentiating feature of whole-program call graph construction algorithms is how they approximate the targets of dynamically dispatched method calls. This affects how they approximate the runtime types of the receivers of those calls. A precise call graph construction algorithm requires knowledge of the runtime types of the receivers of method calls, through pointer analysis, to compute the target methods of those calls. A pointer analysis determines when two pointer expressions refer to the same heap abstraction. A points-to analysis determines what heap abstractions a pointer can point to. We will use both terms interchangeably in this thesis to mean the points-to analysis.

Early work on call graph construction uses simple approximations of runtime types to model dynamic dispatch. Dean et al. [16] formulate *class hierarchy analysis* (CHA), which uses the assumption that the runtime type of a receiver could be any subtype of its statically declared type at the call site. Thus, CHA uses only static type information, and does not maintain any points-to sets of the possible runtime types of objects. Bacon et al. [8] define *rapid type analysis* (RTA), a refinement of CHA that restricts the possible runtime types to classes that are instantiated in the reachable part of the program. Diwan et al. [19] present more precise call graph construction algorithms that remain simple and fast. Rather than maintaining a single set of possible runtime types in the whole program, as in RTA, the authors compute separate sets of runtime types for individual local variables. Sundaresan et al. [53] introduce *variable type analysis* (VTA). VTA generates subset constraints to model the possible assignments between variables within the program. It then propagates points-to sets of the specific runtime types of each variable along these constraints. Unlike the analyses of Diwan et al. [19], VTA computes these points-to sets for heap-allocated objects in addition to local variables.

Tip et al. [54] study a wide range of call graph construction algorithms. The authors define several analyses whose precision vary between RTA [8] and 0-CFA [24, 50]. Separate object sets for methods and fields are used to approximate the runtime values of expressions. The authors have experimented with using distinct sets for classes (CTA), distinct sets for classes and fields (MTA), distinct sets for classes and methods (FTA), and distinct sets for classes, fields and methods (XTA). The XTA analysis resulted in the best trade-off between performance and precision for call graph construction.

The previous algorithm was later used by Tip et al. [55] to implement Jax, an application extractor for Java. Jax uses type-based algorithms to model the library dependencies of a given Java program on-the-fly. In contrast, the separate compilation assumption flow-based algorithms to create a model for the behaviour of the unanalyzed library. Additionally, the separate compilation assumption formally defines the constraints that model the side effects of the unanalyzed library code.

In general, pointer analyses differ in how they handle four aspects of the analyzed program:

1. execution flow: a *flow-sensitive* analysis takes into consideration the order of the execution of statements in the program.

2. calling context: an analysis is *context-sensitive* if it distinguishes between different calling contexts of a method.

3. field sensitivity: a *field-sensitive* analysis distinctly represents fields of an object.

4. object representation: an analysis can use one abstract object to represent all instantiations of a class, or use different abstract object to represent different allocation sites of a class.

Ryder [47] provides a comprehensive taxonomy of many dimensions that affect the precision and cost of a given pointer analysis.

2.2 Static Analysis Frameworks

Several static analysis frameworks for Java now include call graph construction implementations with a range of algorithms that can be configured for the desired trade-off between the precision and the cost of the underlying pointer analysis. Lhoták et al. [37] introduce SPARK, a flexible framework for experimenting with points-to analyses for Java programs. SPARK provides a SOOT [56] transformation that constructs the call graph of the input program on-the-fly while calculating the points-to sets. Although SOOT requires the entire program to generate a whole-program call graph, it is possible to configure the SOOT classes so that some of the input classes are ignored. This is usually achieved through setting the *allow_phantom_refs* option to *true*, which means that the ignored class will be completely discarded. However, crucial information about the signatures of the classes, methods, and fields is lost. Nevertheless, SOOT will continue the analysis on a best-effort basis while warning the user that the results are probably unsound [31]. However, SPARK, which is a call graph analysis tool within SOOT, throws an exception if it encounters a call to one of those phantom classes.

WALA [29] is a Java-based static analysis library from IBM Research designed to support various pointer analysis configurations. WALA is capable of building a call graph for a program by performing pointer analysis with on-the-fly call graph construction to resolve the targets of dynamically dispatched calls. WALA provides the option of excluding individual classes or entire packages while constructing the call graph. In fact, WALA excludes all the user-interface related packages (e.g., `java.awt`, `javax.swing`) from the Java runtime library by default when constructing a call graph. When the *exclude* option is set, WALA limits the scope of its pointer analysis to the set of included classes. This ignores any effects the excluded classes might have on the calculation of the points-to sets. Therefore, the generated call graphs may be unsound and/or imprecise. Moreover, it is not possible to exclude crucial classes (e.g., `java.lang.Object`) from the analysis as this will cause WALA to throw an exception.

PADDLE [36] is a BDD-based framework that offers various call graph analyses for Java, as well as client analyses that uses their results. PADDLE offers several variations of context-sensitive and object-sensitive analyses. Since PADDLE was designed to scale well to benchmarks using large libraries, it has no support for analyzing incomplete/partial programs.

Bravenboer et al. [12] present DOOP, a novel framework for points-to analysis of Java programs. DOOP offers various pointer analysis algorithms, all defined declaratively in Datalog. The modular and declarative implementation of the framework achieves substantial speedups, up to 15x, when compared to the BDD-based framework PADDLE [36] for various analyses (e.g.,

1-call-site sensitive analysis). DOOP is also capable of constructing the call graph on-the-fly while computing the points-to sets. There is no way to exclude some classes (e.g., the Java standard library classes) from the analysis as DOOP analyzes the entire input program. However, we were able to extend DOOP to support partial program analysis using our extension, CGC [2].

2.3 Demand-Driven Pointer Analysis

One possible approach to reduce the excessive cost of performing a whole-program points-to analysis is using a demand-driven or an incremental pointer analysis. A demand-driven pointer analysis computes the points-to sets of a given list of pointers using the least possible relevant program information. This can enable pointer analyses in applications where users expect fast responses (e.g., integrated development environments, IDEs). However, Hind [26] explains that identifying useful demand-driven/incremental analyses and incomplete program analyses are some of the remaining problems that face pointer analysis research.

Yur et al. [60] provide an incremental flow-sensitive and context-sensitive pointer analysis that updates data-flow information after a program change rather than recomputing it from scratch, under the assumption that the change impact will be limited. The analysis achieved an average of six-fold speedup while retaining the same precision as exhaustive analyses. However, the experiments carried out in this work involved using artificial program changes crafted by the authors that do not necessarily reflect modifications likely to be made by programmers.

Heintze et al. [25] introduce a demand-driven approach for pointer analysis that performs just enough computation to determine the points-to sets for a given list of pointer variables. The analysis requires access to the full code of the defined scope. Therefore, it performs badly in cases when computing the points-to sets of all the variables in the program is required. The analysis is also memory-hungry, and slower than whole-program analyses. Vivien et al. [59] present a similar algorithm that analyzes the small regions of the program that surround each allocation site. It then uses this information to deliver useful pointer analysis results. The algorithm delivers almost all of the benefit of a whole-program analysis. However, it does not soundly deal with situations where it is unable to locate all of the potential callers of a given method.

In an effort to compute more sound pointer analysis information using a demand-driven approach, Rountev et al. [45] propose a novel whole-program call graph construction analysis for C. Although the analysis requires the whole program, it analyzes each module of the program separately. A given C program can take the address of a function, and later invoke it by dereferencing the resulting function pointer. Therefore, the function that is invoked depends on the target of the function pointer. The analysis proceeds in two steps. First, in contrast to AVERROES, conservative assumptions are made about all possible applications that could use a given library. The analysis then builds up a set of constraints that model the precise behaviour of the library.

Second, these constraints are used to model the library in an analysis of a specific application. The main target of this work is software built with large extensible reusable library components. The authors have shown they can do the whole-program analysis of the software by combining the summary information of the library to the analysis of the main component without any loss of precision. However, the whole program is still analyzed, just each module separately (i.e., not at the same time).

Rountev et al. [43] apply a similar approach to summarize the precise effects of Java libraries for the purpose of the inter-procedural finite distributive subset (IFDS) and inter-procedural distributive environment (IDE) algorithms [48]. Although these algorithms already inherently construct summaries of callees to use in analyzing callers, they had to be extended in order to deal with the library calling back into application code. This is done by splitting methods into the part *before* and *after* an unknown call. Summaries are then generated for each part rather than the whole method. When the target of the unknown call later becomes available, the partial summaries are composed. Rountev et al. [46] evaluate the approach on two instances of IDE, a points-to analysis and a system dependence graph construction analysis.

Sridharan et al. [52] suggest a novel approach to improve the scalability and precision of demand-driven pointer analyses. The authors present a demand-driven, context-sensitive points-to analysis that filters out unrealizable paths to generate context-sensitive call graphs. Although the analysis can be used to precisely analyze large programs, it is not capable of handling incomplete programs or program fragments where some parts of the source code are unavailable.

2.4 Program Fragments Analysis

Although demand-driven approaches help reduce the cost of performing an exhaustive whole-program analysis at once, they still require access to the source code of the parts of the program they analyze. This approach cannot be used in situation when parts of the source code are unavailable (e.g., developing the library components of a program separately from the application components).

Diwan et al. [19] present simple and fast algorithms for type hierarchy analysis, aggregate analysis, intra-procedural analysis, and inter-procedural analysis. The analyses assume that the entire program, except for the library code, is available. However, the analyses assume that the library code does not create subtypes of any types declared outside the library. In other words, the library cannot create an object of an application class. This limits the ability of the analysis to handle current programming trends (e.g., reflection).

DeFouw et al. [17] describe a general analysis framework that integrates propagation-based and unification-based analyses to produce a fast inter-procedural class analysis. The analysis assumes that it has access to the entire program but could be extended to support analyzing

program fragments. However, the summary data-flow graphs for the missing components (i.e., the missing source code) should be available.

Tip et al. [54] explore many call graph construction algorithms in which the scope of the points-to sets is varied between a single set for the whole program (like RTA) and a separate set for each variable (like VTA). The authors present a scalable propagation-based call graph construction algorithm (XTA), where separate object sets for methods and fields are used to approximate the run-time values of expressions. The algorithm is capable of analyzing incomplete applications by associating a single set of objects, S_E, with the outside world (i.e., the library). The algorithm conservatively assumes that the library calls back any application method that overrides a library method. The set S_E is then used to determine the set of methods that the external code can invoke by the dynamic dispatch mechanism. A separate set of objects S_C is associated with an external (i.e., library) class if the objects passed to the methods in class C interact with other external classes in limited ways. An example of this case is the class `java.lang.Vector`. This step requires the analysis of the external classes to model the separate propagation sets. In addition, this technique varies based on the library dependencies of the input program.

Expanding on Tip et al.'s initial idea of analyzing incomplete applications, we formulated the separate compilation assumption, and worked out the specific assumptions that follow from it. We have also derived a set of constraints from those assumptions. Additionally, we have empirically analyzed the precision and soundness of the partial call graphs generated compared to call graphs generated by analyzing the whole program.

Rountev et al. [44] adapt the C-based analysis from [45] to support Java programs. The authors present a general approach for adapting whole program class analyses to operate on program fragments by creating placeholders to serve as representatives for and simulate potential effects of unknown code. Like Averroes, the fragment class analysis encodes the constraints collected from the library in executable placeholder code. The placeholder code can then be added to the input classes and the result is treated as a complete program which can be analyzed using whole program class analyses. Unlike Averroes, this placeholder code is a precise and detailed summary of the exact effects of the library, and its construction requires the entire library to be analyzed. Moreover, some of the constraints require changes to the application code in addition to the placeholder library. In contrast, the purpose of Averroes is to generate a minimal library stub that enables a sound analysis of the original application code.

Our work is also related to the work of Zhang et al. [61]. They provide a fine-tuned data reachability algorithm to resolve library call-backs, $V^a - DataReach^{ft}$. Similar to the separate compilation assumption, their algorithm distinguishes library code from application code in the formulation of the constraints. The purpose of their algorithm is to compute more precise library callback information than can be expressed by a call graph. For each call from the application to the library, the algorithm computes a specialized set of library callbacks. In contrast, our aim is not to analyze the library at all, and generate a possibly less precise but sound call graph.

2.5 Confined Types

The main challenge encountered when analyzing the application part of a program while ignoring the library is determining the objects that may escape from the predefined application scope to the library, and vice versa. This directly affects the points-to sets of the local variables in the application and the library (or the summarized library points-to set in the case of partial-program analyses).

Grothoff et al. [23] present Kacheck/J, a tool that is capable of identifying accidental leaks of heap object abstractions. Kacheck/J achieves that by inferring the *confinement* property for Java classes [57, 58, 62]. A Java class is considered *confined* when objects of its type are encapsulated in its defining package. The analysis needs only to analyze the defining package of the given Java class to infer its confinement property. As part of the confinement analysis, Kacheck/J identifies *anonymous* methods which are guaranteed not to leak a reference to their receiver.

This approach can be used to develop a partial-program analysis where only the application classes are analyzed to construct the call graph of a program without the need to analyze any library code. To achieve this, the set of application classes can be thought of as one defining package. The confinement analysis can then determine which application class may leak objects of its type to the outside world (i.e., the library). However this is not enough to construct the call graph. Constructing the call graph would still require the points-to set information augmented with the confinement property information for the application classes.

In addition to the analyses that infer the confinement property, there is a large body of work on type systems that enforce encapsulation by restricting reference aliasing. Clarke et al. [14] and Noble et al. [40] introduce *ownership types* by forming a static type system that indicates object ownership. This provides a flexible mechanism to limit the visibility of object references and restrict access paths to objects. Dietl et al. [18] apply the notion of ownership types to structure the object store and to restrict how references can be passed in the Java Modelling Language, JML [33].

Genius et al. [22] present *sandwich types* that allow a coarser view on heap objects. *Sandwich types* are a generalization of *balloon types* [6]. All objects in a *balloon* can be accessed only via a distinguished balloon object. Both type systems were originally introduced to help improve the locality of object-oriented programs. Sandwich types could be used to compute the types of objects that local variables in the library can point to. However, this requires access to the source code of the library components which is something the separate compilation assumption tries to avoid.

Chapter 3

The Separate Compilation Assumption

Any given Java program can be thought of as two sets of classes. One set represents the libraries that the program uses to accomplish some tasks. The other set is the client/application code that uses these libraries to perform those tasks. In the context of call graph construction, the input to the algorithm can then be thought of as a set of classes designated as the *application classes*. The application classes may have dependencies on classes outside this set. We designate any class outside the set as a *library class*. We use the terms *application method* and *library method* to refer to the methods declared in application and library classes, respectively. Both the application classes and the library classes make up the whole program that is analyzed by the call graph construction algorithm. If a call graph construction algorithm does not analyze the library part of a program, it must make very conservative assumptions about the effects of the unanalyzed library code. In particular, a sound algorithm would have to assume the following:

1. a library class or interface may extend or implement any class or interface,
2. a library method may instantiate an object of any type and call its constructor,
3. a local variable in a library method may point to any object of any type consistent with its declared type,
4. a call site in a library class may call any accessible method of any class,
5. a library method may read or modify any accessible field of any object,
6. a library method may read or modify any element of any array,

7. a library method may cause the loading and static initialization (i.e., execution of the `<clinit>()` method) of any class, and

8. a library method may throw any exception of any subtype of `java.lang.Throwable`.

3.1 Definition

Our goal is to avoid the analysis of library code and instead make conservative assumptions about library behavior in a way that lets us construct a precise and sound call graph for the application. A realistic, yet useful, assumption is the **Separate Compilation Assumption** which states that:

> *All of the library classes can be compiled in the absence of the application classes.*

3.2 Constraints

Following from the separate compilation assumption, we identify a set of constraints that enable precise call graph construction for the application part of a Java program without analyzing its library dependencies.

Constraint 1 [*class hierarchy*]

> *A library class cannot extend or implement an application class or interface.*

If such a class existed, the library could not have been compiled in the absence of the application classes. This directly contradicts with the separate compilation assumption.

Constraint 2 [*class instantiation*]

> *An allocation site in a library method can instantiate an object whose runtime type is:*
>
> - *a library class, or*
> - *an application class whose name is known to the library (i.e., through reflection).*

An allocation site in a library method cannot instantiate an object whose runtime type is an application class. The runtime type of the object is specified in the allocation site, so compilation

of the allocation site would require the presence of the application class. The only exception to this rule is reflective allocation sites in a library class (i.e., using `Class.forName()` and `Class.newInstance()`) that could possibly create an object of an application class. Since Java semantics do not prevent the library from doing this, our analysis should handle these reflective allocations without analyzing the library code. The library can reflectively instantiate objects of an application class if the library knows the name of this particular application class. In other words, if a string corresponding to the name of an application class flows to the library (possibly as an argument to a call to `Class.forName()`), then the library can instantiate objects of that class.

Constraint 3 [*local variables*]

Local variables in the library can point to objects:

- *instantiated by the library,*
- *instantiated by the application and then passed in to the library due to inter-procedural assignments,*
- *stored in fields accessible by the library code, or*
- *whose runtime type is a subtype of* `java.lang.Throwable`.

Our approach computes a sound but non-trivial over-approximation of the abstract objects that local variables of library methods could point to. The library could create an object whose type is any library class. An object whose type is an application class can be instantiated only in an application method (except by reflection). In order for an object created in an application method to be pointed to by a local variable of a library method, an application class must pass the object to a library class in one of the following ways. First, an application method may pass the object as an argument to a call of a library method. This also applies to the receiver, which is passed as the `this` parameter. Second, an application method called back from a library method may return the object. Third, the application code may store the object in a field that can be read by the library code. Fourth, if the type of the object is a subtype of `java.lang.Throwable`, an application method may throw the object and a library method may catch it.

Thus, our approach computes a set, `LibraryPointsTo`, of the abstract objects allocated in the application that a local variable of a library method can point to. Implicitly, the library can point to objects whose type is any library class since these can be created in the library. Only the subset of application class objects that are passed into the library is included in `LibraryPointsTo`.

Constraint 4 [*method calls*]

A call site in the library can invoke:

- *any method in any library class visible at this call site, or*
- *a method m in an application class C, but only if:*
  1. *m is non-static and overrides a (possibly abstract) method of a library class, and*
  2. *a local variable in the library points to an object of type C or a subclass of C.*

Two conditions are necessary in order for a call site in a library class to call back a method m in an application class C. First, the method m must be non-static and override a (possibly abstract) method of some library class. Each call site in Java bytecode specifies the class and method signature of the method to be called. Since the separate compilation assumption states that the library has no knowledge about the application, the specified class and method must be in the library, not the application. The Java resolution rules [38, Section 5.4.3] could change the specified class to one if its superclasses, but this must also be a library class due to Constraint 1. Therefore, the only way in which an application method could be invoked is if it is selected by dynamic dispatch. This requires the application method to be non-static and to override the method specified at the call site. Second, the receiver variable of the call site must point to an object of class C or a subclass of C such that calling m on that subtype resolves to the implementation in C. Therefore, an object of class C or of the appropriate subclass must be in the `LibraryPointsTo` set.

Constraint 5 [*field access*]

A statement in the library can access (i.e., read or modify):

- *any field in any library class visible at this statement, or*
- *a field f of an object O of class C created in the application code, if:*
  1. *f is originally declared in a library class, and*
  2. *a local variable in the library points to the object O.*

Similar to method calls, two conditions are necessary in order for a library method to read or modify a field f of an object O of class C created in the application code. First, the field f must originally be declared in a library class, though C can be a subclass of that class, and

could therefore be an application class. Each field access in Java bytecode specifies the class and the name of the field. This class must be a library class due to the separate compilation assumption. The Java resolution rules could change the specified class, but again only to one of its superclasses, which must also be a library class. Second, it must be possible for the local variable whose field is accessed to point to the object O. In other words, the `LibraryPointsTo` set must contain the abstract object representing O. In the case of a field write, the object being stored into the field must also be pointed to by a local variable in the library. Therefore, its abstraction must be in the `LibraryPointsTo` set. Additionally, the library can access any static field of a library class, and any field of an object that was instantiated in the library.

Constraint 6 [*array access*]

The library can only access array objects pointed to by its local variables.

If the library has access to an array, it can access any of its elements through its index. This is unlike an instance field, which is only accessible if its name is known to the library. However, the library is limited to accessing only the elements of arrays that it has a reference to (i.e., ones that are in the `LibraryPointsTo` set). Similar to field writes, objects written into an array element must be pointed to by a local variable in the library (i.e., such objects must also be in the `LibraryPointsTo` set).

Constraint 7 [*static initialization*]

The library causes the loading and static initialization (i.e., execution of the `<clinit>()` *method) of classes that it instantiates.*

Due to the separate compilation assumption, the library does not contain any direct references to application classes. Thus, the library cannot cause a static initializer of an application class to be executed except by using reflection. When determining which static initializers will execute, our analysis includes those classes that are referenced from application methods reachable through the call graph, as well as classes that may be instantiated using reflection as discussed above in Constraint 2.

Constraint 8 [*exception handling*]

The library can throw an exception object e if:

- *e is instantiated by the library, or*
- *e is instantiated by the application and passed to the library.*

The library can throw an exception either if it creates the exception object (in which case its type must be a library class) or if the exception object is created in an application class and passed into the library per Constraint 3 (in which case its abstraction appears in the `LibraryPointsTo` set). We conservatively assume that the library can catch any thrown exception. Consequently, we add the abstractions of all thrown exception objects to the `LibraryPointsTo` set.

Additional Constraints

In addition to the constraints that follow from the separate compilation assumption, we strictly enforce the restrictions imposed by declared types.

- When the library calls an application method, the arguments passed in the call must be in the `LibraryPointsTo` set, and must also be compatible with the declared types of the corresponding parameters.
- When an application method calls a library method, the returned object must be in the `LibraryPointsTo` set and compatible with the declared return type of the library method.
- When the library modifies a field, the object it may write into the field must be compatible with the declared type of the field.
- When an application method catches an exception, only exception objects whose type is compatible with the declared type of the exception handler are propagated.

Chapter 4

AVERROES

To evaluate the separate compilation assumption, we have implemented two systems, CGC and AVERROES. CGC is a proof-of-concept extension for DOOP that adds support for doing partial-program call graph construction using the separate compilation assumption.[1]

To generalize the use of the separate compilation assumption in all Java whole-program analysis frameworks, we have implemented AVERROES. Given any Java program, AVERROES generates an alternative placeholder library that models the constraints that follow from the separate compilation assumption. The placeholder library can then be used instead of the original library classes as input to a whole-program analysis.[2]

Chapter Organization. We first present the proof-of-concept tool CGC in Section 4.1. We then discuss how the AVERROES system generates a placeholder replacement library for a given Java program in Section 4.2. Finally, Section 4.3 explains in more details the contents of the placeholder library in terms of classes and method bodies.

4.1 Proof-Of-Concept

The usual context of a whole-program analysis tool is depicted in Figure 4.1. The tool expects to analyze all of the classes of the program, including any libraries that it uses. The tool does not necessarily distinguish between application and library classes. Optionally, the tool may also make use of additional information about the uses of reflection in the program. This information could be provided by the user or collected during execution of the program being analyzed with a tool such as TAMIFLEX [11].

[1] CGC is available for download at http://plg.uwaterloo.ca/~karim/projects/cgc/

[2] We have made AVERROES available at http://plg.uwaterloo.ca/~karim/projects/averroes/

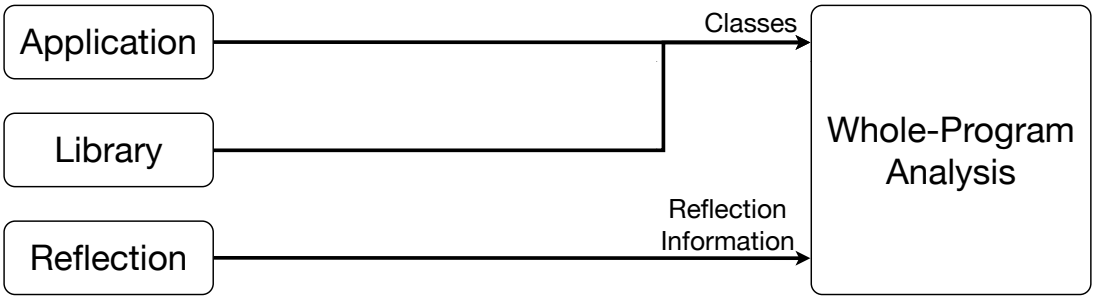


Figure 4.1: The usual context of a whole-program analysis.

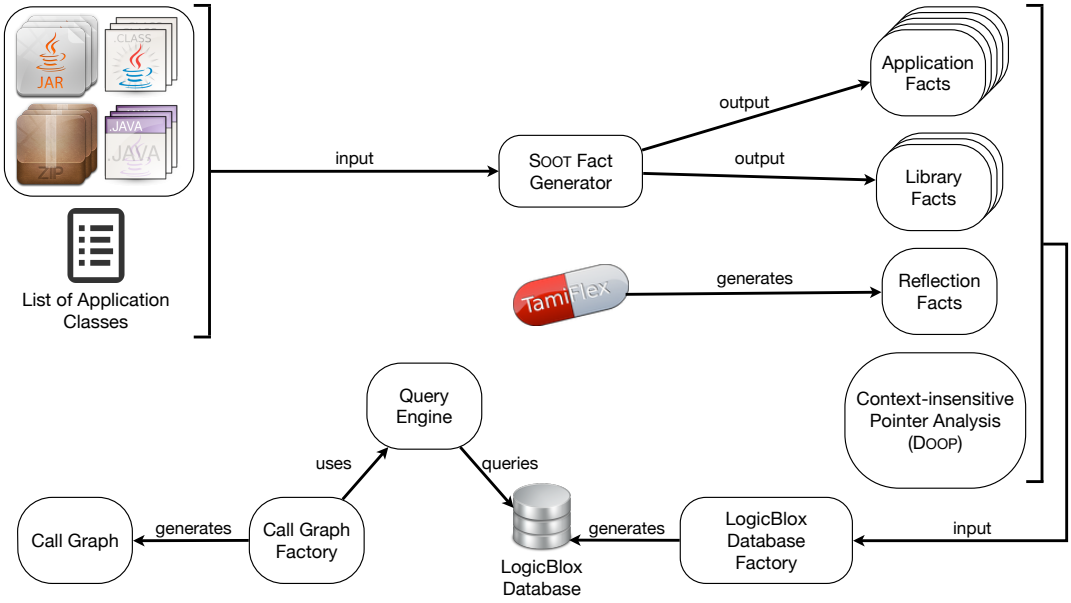


Figure 4.2: An overview of the workflow of Cgc.

A whole-program analysis tool could support analyzing partial programs if it implements the separate compilation assumption. CGC is an example of such a tool. Figure 4.2 shows an overview of the workflow of CGC. Like DOOP, CGC uses a fact generator based on SOOT to preprocess the input code and generate the input facts for the Datalog program. The fact generator receives a collection of input files and a specification of the set of application classes. The rest of the classes are considered library classes. The fact generator then generates two sets of facts. The first set is for the application classes and contains all details about those classes: signatures for classes, methods, fields as well as facts about method bodies. The second set is dedicated to the library and contains the following: signatures for classes, methods, and fields in the library classes that are referenced in the application and their (transitive) superclasses and superinterfaces.

We generate a third set of facts which holds information about reflection code in the application. This set is generated using TAMIFLEX [11], a tool suite that records actual uses of reflection during a run of a program, and summarizes them in a format suitable as input to a static analysis. CGC uses the output of TAMIFLEX to model calls to `java.lang.reflect.Method.invoke()`, and application class name string constants to model reflective class loading.

The three sets of facts along with the Datalog rules that define the pointer analysis are then used to initialize a LOGICBLOX [42] database. Once the database is created and the analysis completes, CGC queries it for information about the call graph entry points and the various types of call graph edges: application-to-application edges, application-to-library edges, and library-to-application call back edges. Finally, CGC uses those derived facts to generate the call graph for the given input program files and to save it as a GXL document [35][3] or a directed DOT [32] graph file. The DOT graph can be visualized using Graphviz [51] or converted by CGC to a PNG or a PS file that can be visualized using any document previewer.

Since CGC is a specific implementation of the separate compilation assumption for DOOP, other whole-program analysis frameworks have to provide their own implementations. This will lead to unnecessary redundant implementations for the separate compilation assumption, each supporting one, or more, analysis frameworks. This motivates the need to have an implementation of the separate compilation assumption that is universal to all Java analysis frameworks.

4.2 Workflow

We have implemented AVERROES, a tool that intends to provide the same input environment to the whole-program analysis, but without analyzing any actual code of the original library classes.[4]

[3]The DTD schema can be found at http://plg.uwaterloo.ca/~karim/projects/cgc/schemas/callgraph.xml

[4]We have made AVERROES available at http://plg.uwaterloo.ca/~karim/projects/averroes/

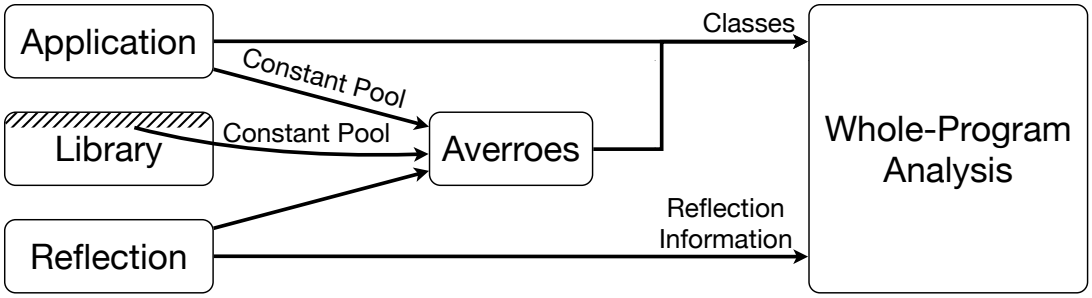


Figure 4.3: The context of whole-program analysis using AVERROES.

Figure 4.3 depicts the context in which AVERROES is used. Given any Java program, AVERROES generates an alternative placeholder library that models the constraints that follow from the separate compilation assumption. To achieve that, AVERROES uses SOOT [56] to consult the classes of the input program. Unlike a whole-program analysis, AVERROES does not inspect all classes, and does not analyze any Java bytecode instructions. For each application class, AVERROES examines only the constant pool to find all references to library classes, methods, and fields. Among library classes, AVERROES consults only the classes that are directly referenced by the application and their superclasses and superinterfaces. Within this restricted set of classes, AVERROES examines only the constant pool. AVERROES uses this information in order to build a model of the class hierarchy and the overriding relationships between methods in the program.

Since AVERROES examines only a small fraction of classes and only a small fraction of each class file, the execution of AVERROES can be much faster than a whole-program analysis that reads and analyzes the code of the whole program. In addition, if the library code itself calls other dependent libraries, AVERROES can process the library even if those dependencies are not available for analysis, assuming they are not directly referenced from the application code. AVERROES also, optionally, reads in the reflection facts generated by TAMIFLEX for the input program.

4.3 Placeholder Library

The output of AVERROES is a placeholder library. AVERROES generates this library using SOOT. Additionally, AVERROES uses the Java bytecode verification tools offered by BCEL [13] to verify that the generated library satisfies the specifications of valid Java bytecode. The placeholder library contains stubs of all of the library classes, methods, and fields referenced by the application, so that the application could be compiled with the placeholder library instead of the original library. As a consequence, the application classes together with the generated placeholder library make up a self-contained whole program that can be given as input to any whole-program

analysis. The placeholder library is designed to be as small as possible, while still being self-contained, such that the whole-program analysis can analyze it much more efficiently than the original library. In addition, the placeholder library over-approximates all possible behaviours of the original library, so that the call graph analysis produces a sound call graph when analyzing the placeholder library instead of the original library. The rest of this section defines in detail the contents of the generated placeholder library.

4.3.1 Library Classes

The AVERROES placeholder library contains three kinds of classes: referenced library classes, concrete implementation classes, and the AVERROES library class. We define the structure of these classes first, and we define the contents of their methods in Section 4.3.2.

Referenced Library Classes

The AVERROES placeholder library contains every library class directly referenced by the application and their superclasses and superinterfaces. In addition, it contains a small fixed set of basic classes that are mentioned explicitly in the Java Language Specification [38] and expected by whole-program analyses (e.g., `java.lang.Object` and `java.lang.Throwable`).

Each such referenced class contains placeholders for all of the methods and fields that are referenced by the application. A method m is considered to be referenced by the application if:

- a reference to m appears in the constant pool of an application class,
- m is a constructor or a static initializer in the original library class, or
- a call to some method m' referenced by the application may resolve to m.

A field f of type t is considered to be referenced by the application if a reference to f appears in the constant pool of an application class. If an included library method is native in the original library, its placeholder is made non-native. This is because the generated placeholder library should not depend on other code, including native code. Furthermore, the *throws* clause of a generated placeholder library method can only contain exception classes that are referenced by the application. To ensure that every library class has at least one accessible constructor, AVERROES adds a default constructor (i.e., a public constructor that takes no arguments) to every included library class.

```
13 class Main {
14   void foo() {
15     Vector v = new Vector();
16     ...
17     Enumeration e = v.elements();
18     while(e.hasMoreElements()) {
19     ...
20     }
21   }
22 }
```

(a)

```
23 class EnumerationConcrete
24     implements Enumeration {
25   boolean hasMoreElements() {
26     doItAll();
27     return true;
28   }
29
30   Object nextElement() {
31     doItAll();
32     return (Object)libraryPointsTo;
33   }
34 }
```

(b)

Figure 4.4: An example illustrating the concept of concrete implementation classes in AVERROES: (a) sample application Java code that uses the class `java.util.Enumeration`, (b) the concrete implementation class that AVERROES creates for `java.util.Enumeration` in the placeholder library.

Concrete Implementation Classes

Constraint 2 of the separate compilation assumption states that the library code can create an object of any library type. This includes types that are not referenced by the application. The object of an unreferenced type could still be accessed by the application through one of its supertypes. For the purpose of constructing an application-only call graph, the exact run-time type of the object is not important, since any calls on the object will just resolve to the library summary node. However, the call graph construction algorithm must be aware that an object of such an unknown type could be the receiver of a call.

Figure 4.4a shows a sample Java program where method `foo()` calls the method `java.util.Vector.elements()`. The return type of the method is `java.util.Enumeration`, which is an interface. If the application then calls a method such as `hasMoreElements()` or `nextElement()` on the value returned from `java.util.Vector.elements()`, the call should resolve to the library. Therefore, the call graph construction analysis must be aware that the receiver of the call could be some object that implements the `java.util.Enumeration` interface. However, if the application does not implement the interface itself, and if it does not reference any library class that implements it, then the whole-program analysis would not know about the existence of any concrete class that implements the interface. In this case, AVERROES adds to the placeholder library a concrete class that implements the interface, so that the call graph construction algorithm can resolve the call on this class. Figure 4.4b illustrates the contents of the concrete

implementation class that AVERROES generates for `java.util.Enumeration` in the placeholder library.

Specifically, AVERROES creates a concrete implementation class for each interface and abstract class in the library that is referenced by the application, but is not implemented by any concrete class already in the placeholder library. If the original library contains a concrete class implementing the given interface or abstract class, that concrete class would already be in the placeholder library only if the application references that concrete class. Each concrete implementation class contains implementations of all abstract methods in the interface or abstract class that caused the concrete implementation class to be created, including abstract methods inherited from superclasses and superinterfaces.

AVERROES Library Class

All of the conservative approximations of the possible behaviours of the library defined by the constraints listed in Chapter 3 are implemented in one class in the placeholder library, `AverroesLibraryClass`. In particular, this class models the following library behaviours: object instantiation, callbacks to application methods, array writes, and exception handling. The `AverroesLibraryClass` has two members:

1. The field `libraryPointsTo` is a public, static field of type `java.lang.Object`. It represents all local variables in the original library code. Every object that could be assigned to a local variable in the original library is assigned to this field. The points-to set of the `libraryPointsTo` field corresponds to the `LibraryPointsTo` set that the separate compilation assumption computes (Constraint 3).

2. The method `doItAll()` is a public, static method. It is the main AVERROES method that models all of the potential side effects that the original library code could have.

4.3.2 Library Methods

Referenced Library Method Bodies

Each placeholder method in the referenced library classes and in the concrete implementation classes is an entry point from the application into the library, and should conservatively implement the behaviours specified in Chapter 3. Most of these behaviours are implemented just by calling the `doItAll()` method of the `AverroesLibraryClass`. In addition, each placeholder method stores all of its parameters to the `libraryPointsTo` field. The return value of the method is also taken from `libraryPointsTo`.

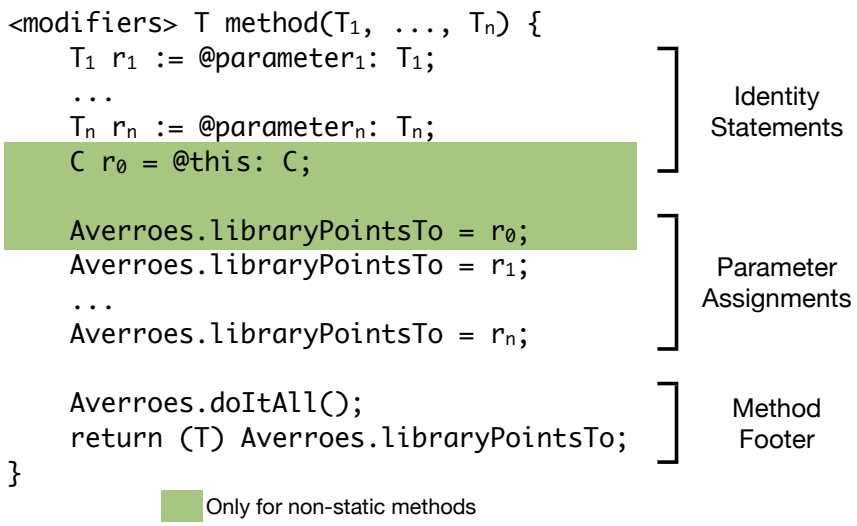


Figure 4.5: The Jimple template AVERROES uses to generate bodies for referenced library methods.

More specifically, the body of each placeholder method is constructed according to the template shown in Figure 4.5. The template is shown in the Jimple intermediate language of the SOOT framework [56], which is used by AVERROES to generate the placeholder library. The template has three code regions:

1. Identity statements define the variables that will hold the method parameters. Non-static methods have an additional identity statement for the implicit `this` parameter.
2. Parameter assignment statements assign the parameters to the `libraryPointsTo` field in order to model the inter-procedural flow of objects from the application through parameters into the library (Constraint 3).
3. The method footer contains two statements. The first statement is a call to the `doItAll()` method in the `AverroesLibraryClass` to model the side effects of the library. The second statement is the `return` statement of the method. The method can return any object from the library whose type is compatible with the return type of the method. This is modelled by reading the `libraryPointsTo` field and casting its value to the method return type. This completes the implementation of Constraint 3. If the return type of the method is primitive, the constant value 1 is returned. Methods with return type `void` will just have an empty `return` statement.

The bodies of constructors of placeholder library classes are generated using the same Jimple template. However, a call to the default constructor of the direct superclass is generated before

accessing the `this` parameter in the constructor body. Moreover, AVERROES generates statements that initialize the instance fields of the declaring class. Each instance field is initialized by assigning it the value of the `libraryPointsTo` field after casting it to the appropriate type (Constraint 5).

The bodies of library static initializers are simpler. Since static initializers have no parameters or return value, no identity statements or parameter assignment statements are generated for them. In addition, they have an empty `return` statement (i.e., one that does not return any value). Moreover, for each static initializer, AVERROES initializes the static fields of its declaring class with the value from the `libraryPointsTo` field through the appropriate cast (Constraint 5).

AVERROES `doItAll()` Method Body

The `doItAll()` method implements most of the conservative approximation of the behaviour of the library. It is a static method with no parameters, and therefore does not have any identity statements. The body of the `doItAll()` method implements the following behaviours:

1. Class instantiation (Constraint 2 and Constraint 7): According to Constraint 2, the library can create an object of any concrete class in the library or any application class that is instantiated by reflection. For each such class C, two statements are generated: a `new` instruction to allocate the object, and a special invocation instruction (corresponding to the `invokespecial` bytecode) to an accessible constructor of the class. Finally, if the class C declares a static initializer, AVERROES generates a call to it.

2. Library callbacks (Constraint 3 and Constraint 4): Following Constraint 4, the `doItAll()` method contains calls to all methods of the library that are overridden by some method of the application, since at run time, any such call could dispatch to the application method. In addition, the `doItAll()` method calls all application methods known to be invoked by reflection. The receiver of all of these calls is taken from the `libraryPointsTo` field, as are all arguments to the method. The values from the `libraryPointsTo` field are cast to the appropriate types as required by the method signature. Additionally, Constraint 3 states that objects may flow from the application to the library due to inter-procedural assignments. Therefore, in AVERROES, if the target method of a library call back has a non-primitive return type, its return value is assigned to the field `libraryPointsTo`.

3. Array element writes (Constraint 6): The library could store any object reference that it has into any element of any array to which it has a reference. Two statements are generated to simulate this. The first statement casts the value of the `libraryPointsTo` field to an array of `java.lang.Object`, which is a supertype of all arrays of non-primitive types. The second statement assigns the value of the `libraryPointsTo` field to element 0 of the array.

4. Exception handling (Constraint 8): The library code could throw any exception object to which it has a reference. To model this, AVERROES generates code that casts the value of the `libraryPointsTo` field to the type `java.lang.Throwable`, and throws the resulting value using the Jimple `throw` statement (which corresponds to the `athrow` bytecode instruction).

In the current implementation of AVERROES, the `doItAll()` method is a single straight-line piece of code with no control flow. If AVERROES were to be used with a flow-sensitive analysis, control flow instructions should be added to all library methods, including the `doItAll()` method. This allows the instructions to be executed in an arbitrary order for an arbitrary number of times. This enables a sound over-approximation for all possible control flow in the original library. Although this would be easy to implement, we have not done it because all of the call graph construction frameworks for Java that we are aware of mainly do flow-insensitive analysis.

Similarly, the `doItAll()` method writes only to element 0 of every array. If a framework attempts to distinguish different array elements, this should be changed to a loop that writes to all array elements. Again, we are not aware of any call graph construction frameworks for Java that distinguish different array elements.

4.3.3 Modelling Reflection

AVERROES models reflective behaviour in the library in two ways. First, whenever a call site in the application calls a library method, AVERROES assumes that any argument of the call that is a string constant could be the name of an application class that the library instantiates by reflection. For every such string constant that is the name of an application class, AVERROES generates a `new` instruction and a call to the default constructor of the class in the `doItAll()` method.

Second, AVERROES reads information about uses of reflection in the format of TAMIFLEX [11]. TAMIFLEX is a dynamic tool that observes the execution of a program and records the actual uses of reflection that occur. AVERROES then generates the corresponding behaviour in the `doItAll()` method. Alternatively, a programmer who knows how reflection is used in the program could write a sound reflection specification by hand in the TAMIFLEX format. AVERROES generates the following code in the `doItAll()` method to model the reflective behaviour recorded in the TAMIFLEX format:

1. For every class that the TAMIFLEX file specifies as instantiated by `java.lang.Class.newInstance()`, or even just loaded by `java.lang.Class.forName()`, the `doItAll()` method allocates an instance of the class using a `new` instruction, and calls its default constructor.

2. For every unique appearance of `java.lang.reflect.Constructor.newInstance()` in the TamiFlex file, the `doItAll()` method allocates an instance of the specified class and calls the specified constructor on it.

3. For every unique appearance of `java.lang.reflect.Array.newInstance()` in the TamiFlex file, the `doItAll()` method allocates an array of the specified type.

4. For every unique appearance of `java.lang.reflect.Method.invoke()` in the TamiFlex file, the `doItAll()` method contains an explicit invocation of the appropriate method.

Even though these behaviours are triggered by reflection in the original library, the Averroes placeholder library implements all of them explicitly (non-reflectively) using standard Java bytecode instructions. Therefore, even if the whole-program analysis that follows Averroes does not itself handle reflection, it will automatically soundly handle the reflective behaviour that Averroes knows about. This is because Averroes encodes the behaviour explicitly in the placeholder library using standard bytecode instructions known to every analysis framework.

In addition, the placeholder library still contains the methods that implement reflection in the Java standard library. The `doItAll()` method also contains calls to `java.lang.Class.forName()` and `java.lang.Class.newInstance()` on the value of `libraryPointsTo` cast to `java.lang.String`. Therefore, if the whole-program framework knows about the special semantics of these reflection methods, or if it knows about some reflective behaviour that is unknown to Averroes, the whole-program framework can still model the additional reflective behaviour in the same way as if it were processing the original library instead of the Averroes placeholder library.

4.3.4 Code Verification

The placeholder library that is generated by Averroes is intended to be standard, verifiable Java bytecode that can be processed by any Java bytecode analysis tool. To guarantee this, Averroes verifies the generated placeholder library classes using the BCEL [13] verifier. BCEL closely follows the class file verification process defined in the Java Virtual Machine Specification [38, Section 4.9]. BCEL ensures the validity of the internal structure of the generated Java bytecode, the structure of each individual class, and the relationships between classes (e.g., the subclass hierarchy).

Chapter 5

Evaluation

We have evaluated the soundness of the proof-of-concept CGC compared to the dynamic call graph generated by *J, and its precision compared to whole-program analysis tools SPARK and DOOP. Our results have shown that not analyzing the library code does not affect the soundness of the resulting call graph. In fact, in many cases CGC was found to be more sound than SPARK and DOOP. Additionally, the call graphs generated by CGC are almost as precise as call graphs generated by DOOP, and sometimes more precise than SPARK. More detailed results can be found at [2].

In this chapter, we evaluate how well AVERROES achieves the goal of enabling whole-program analysis tools to construct sound and precise call graphs without analyzing the whole library.

Chapter Organization. Section 5.1 lays out our experimental setup. Section 5.2 provides a comparison of the call graphs computed by SPARK and DOOP when using AVERROES with dynamically observed call graphs to provide partial evidence that the static call graphs are sound. A correctness proof for AVERROES based on Featherweight Java is provided in Chapter 6. In Section 5.3, we compare the precision of call graphs constructed using AVERROES to those constructed by analyzing the original library code for both SPARK and DOOP. Section 5.4 compares the sizes of the call graphs computed by AVERROES-based tools to those generated by whole-program tools. In Section 5.5, we quantify the improvements in performance when AVERROES is used with both SPARK and DOOP in terms of call graph size, analysis time, and memory requirements. Section 5.6 summarizes our findings.

5.1 Experimental Setup

We have conducted our experiments on two benchmark suites: the DaCapo benchmark programs version 2006-10-MR2 [9], and the SPEC JVM98 benchmark programs [15]. All of these programs have been analyzed with the Java standard library from JDK 1.4 (jre1.4.2_11). We ran all of the experiments on a machine with four dual-core AMD Opteron 2.6 GHz CPUs (running in 64-bit mode) and 16 GB of RAM.

We evaluate AVERROES using the implementation of the same call graph analysis offered by SPARK and DOOP. The analysis uses a propagation-based pointer analysis that constructs the call graph on-the-fly. Additionally, the call graph analysis is field-sensitive, array-insensitive, flow-insensitive, and context-insensitive. We have used the default settings for both SPARK and DOOP. For the AVERROES-based SPARK (denoted by $\text{SPARK}_{\text{AVE}}$), we disable the option *simulate-natives*, and we disable the support for reflection for the AVERROES-based DOOP (denoted by DOOP_{AVE}). This is because we want to evaluate how AVERROES enables call graph analyses to easily handle features like native code and reflection by just using the placeholder library instead of analyzing the original library code.

We have created an artifact for the experiments that we conducted to evaluate AVERROES. The artifact includes a tutorial with detailed instructions on how to use AVERROES to generate the placeholder libraries for each program in our benchmark suites. It then shows how to reproduce all of the statistics we discuss in this chapter.[1] The artifact has been successfully evaluated by the ECOOP'13 Artifact Evaluation Committee and found to meet expectations [3].

5.2 Call Graph Soundness

Static call graph construction is made difficult in Java by dynamic features such as reflection and features that are difficult to analyze such as native methods. AVERROES makes it much easier to construct a sound call graph in the presence of these features in two ways. First, whereas whole-program analysis frameworks try to model all behaviour of the whole program precisely, AVERROES uses the conservative assumption that the library could have any behaviour consistent with the separate compilation assumption. Therefore, a whole-program analysis must model every detail of dynamic behaviour or risk becoming unsound. On the other hand, an analysis using AVERROES remains sound without having to precisely reason about dynamic behaviour within the library. Second, AVERROES contains analyses that model how the library affects the application using reflection. These analyses make use of information about strings passed into the library, as well as information about reflection generated by TAMIFLEX [11]. A whole-program analysis

[1] We have made the artifact available at http://plg.uwaterloo.ca/~karim/projects/averroes/tutorial.php

Table 5.1: Comparing the soundness of Spark and Doop when analyzing the whole program to using Averroes with respect to the dynamically observed call edges. The quantity Dyn \ Spark represents the number of edges in the static call graph computed by Spark that are missing in the dynamic call graph. The quantities Dyn \ Spark$_{\text{Ave}}$, Dyn \ Doop, and Dyn \ Doop$_{\text{Ave}}$ are defined similarly.

	Dyn	Dyn \ Spark	Dyn \ Spark$_{\text{Ave}}$	Dyn \ Doop	Dyn \ Doop$_{\text{Ave}}$
antlr	3,449	0	0	0	0
bloat	4,257	0	0	0	0
chart	657	0	0	0	0
hsqldb	1,627	61	0	331	0
luindex	726	4	0	303	0
lusearch	539	185	1	241	1
pmd	2,087	3	0	225	0
xalan	2,953	96	1	349	0
compress	43	0	0	0	0
db	54	0	0	0	0
jack	596	0	0	0	0
javac	2,538	0	0	0	0
jess	13	0	0	0	0
raytrace	330	0	0	0	0

that uses AVERROES can automatically benefit from the results of these analyses without having to implement the analyses themselves.

We have evaluated the soundness of static call graphs by comparing them against dynamic call graphs collected by *J [20]. Since a dynamic call graph results from only a single execution, it may miss edges that could execute in other executions. Therefore, such a comparison does not guarantee that the static call graph is sound for all executions. Nevertheless, the comparison can detect soundness violations, and the lack of detected violations provides at least partial assurance of soundness. The results of this comparison are shown in Table 5.1. The DYN column shows the number of call edges in the application portion of the dynamic call graph. The remaining columns show how many of these edges are missing in the static call graphs generated by SPARK and DOOP with and without using AVERROES. When using AVERROES, only two edges are missing from all of the call graphs. In LUSEARCH, a `NullPointerException` is thrown and the dynamic call graph records a call edge from the virtual machine to the constructor of this exception class. This behaviour is not modeled by either SPARK or DOOP. In XALAN, a call edge to `java.lang.ref.Finalizer.register()` from the application is missing from the call graph generated by SPARK using AVERROES since SPARK does not handle calls to this library method. On the other hand, the call graphs generated by SPARK and DOOP without using AVERROES are missing a significant number of dynamically observed edges in benchmarks that make heavy use of reflection. This is despite the immense effort that has been expended to make these analysis frameworks handle reflection soundly.

Finding 1: AVERROES reduces the difficulty of constructing sound static call graphs in the presence of reflection.

5.3 Call Graph Precision

In order for a call graph to be useful, it must also be precise in addition to being sound. We now compare the precision of call graphs generated by SPARK and DOOP using the AVERROES placeholder library to analyzing the original library code. We would expect AVERROES to be strictly less precise than SPARK and DOOP, since it makes conservative assumptions about the library code instead of precisely analyzing it. Since we found some dynamic call edges that were missing from the call graphs generated by both SPARK and DOOP, we first correct this unsoundness by adding the missing dynamic call edges to the static call graphs. This enables us to compare the precision of the static call graphs by counting only spurious call edges, and to avoid confounding due to differences in soundness. In Tables 5.2, 5.3 and 5.4, the quantity $\text{SPARK}_{\text{AVE}} \setminus \text{SPARK}$ represents the number of edges in the call graph generated by analyzing the AVERROES library that are missing in the call graph generated by SPARK, and are also not

Table 5.2: Comparing the precision of using AVERROES to analyzing the whole program in both SPARK and DOOP with respect to application call edges. The quantity $\text{SPARK}_{\text{AVE}} \setminus \text{SPARK}$ represents the number of edges in the call graph computed by $\text{SPARK}_{\text{AVE}}$ that are missing in the call graph generated by SPARK. The quantity $\text{DOOP}_{\text{AVE}} \setminus \text{DOOP}$ is defined similarly.

	SPARK	$\text{SPARK}_{\text{AVE}} \setminus \text{SPARK}$	DOOP	$\text{DOOP}_{\text{AVE}} \setminus \text{DOOP}$
ANTLR	6,299	14	6,293	17
BLOAT	13,419	2,301	12,433	2,471
CHART	6,732	32	1,811	110
HSQLDB	7,851	1,328	6,570	2,383
LUINDEX	1,412	5	693	231
LUSEARCH	2,879	253	1,700	384
PMD	6,893	270	4,109	585
XALAN	13,079	1,536	9,161	2,428
COMPRESS	40	0	40	0
DB	47	1	47	1
JACK	646	7	646	7
JAVAC	8,519	45	8,188	51
JESS	6	0	6	0
RAYTRACE	400	0	400	0

present in the dynamic call graph. The quantity $\text{DOOP}_{\text{AVE}} \setminus \text{DOOP}$ is defined similarly. We say that an AVERROES-based tool is *precise* when the call graph that it generates is identical to that generated through whole program analysis by SPARK or DOOP.

Application Call Edges

An *application call edge* is a call graph edge between two application methods. Table 5.2 shows that AVERROES-based tools generate precise call graphs with respect to application call edges when compared to both SPARK and DOOP for COMPRESS, JESS, and RAYTRACE. Across all benchmark programs, $\text{SPARK}_{\text{AVE}}$ generates a geometric mean of 2.18% extra application call edges when compared to SPARK and DOOP_{AVE} generates a geometric mean of 6.16% extra application call edges when compared to DOOP.

Table 5.3: Comparing the precision of using AVERROES to analyzing the whole program in both SPARK and DOOP with respect to library call edges. The quantity $\text{SPARK}_{\text{AVE}} \setminus \text{SPARK}$ represents the number of edges in the call graph computed by $\text{SPARK}_{\text{AVE}}$ that are missing in the call graph generated by SPARK. The quantity $\text{DOOP}_{\text{AVE}} \setminus \text{DOOP}$ is defined similarly.

	SPARK	$\text{SPARK}_{\text{AVE}} \setminus \text{SPARK}$	DOOP	$\text{DOOP}_{\text{AVE}} \setminus \text{DOOP}$
ANTLR	661	0	649	1
BLOAT	885	12	841	37
CHART	1,060	14	529	75
HSQLDB	869	115	792	171
LUINDEX	336	0	205	41
LUSEARCH	392	29	296	53
PMD	797	3	391	43
XALAN	1,055	116	860	261
COMPRESS	13	0	13	0
DB	23	1	24	0
JACK	97	10	98	9
JAVAC	317	2	313	0
JESS	6	1	6	1
RAYTRACE	34	0	34	0

Library Call Edges

A *library call edge* is a call graph edge from a method in the application part of the program to some method in the library. Table 5.3 shows that $\text{SPARK}_{\text{AVE}}$ generates precise call graphs with respect to library call graph when compared to SPARK for ANTLR, LUINDEX, COMPRESS, and RAYTRACE. On the other hand, DOOP_{AVE} generates precise call graphs when compared to DOOP for COMPRESS, DB, JAVAC, and RAYTRACE. Across all benchmark programs, $\text{SPARK}_{\text{AVE}}$ generates a geometric mean of 3.58% extra library call edges when compared to SPARK as opposed to DOOP_{AVE} that generates a geometric mean of 9.08% extra library call edges when compared to DOOP.

Table 5.4: Comparing the precision of using AVERROES to analyzing the whole program in both SPARK and DOOP with respect to library callback edges. The quantity $\text{SPARK}_{\text{AVE}} \setminus \text{SPARK}$ represents the number of edges in the call graph computed by $\text{SPARK}_{\text{AVE}}$ that are missing in the call graph generated by SPARK. The quantity $\text{DOOP}_{\text{AVE}} \setminus \text{DOOP}$ is defined similarly.

	SPARK	$\text{SPARK}_{\text{AVE}} \setminus \text{SPARK}$	DOOP	$\text{DOOP}_{\text{AVE}} \setminus \text{DOOP}$
ANTLR	73	3	42	12
BLOAT	223	26	84	119
CHART	490	9	55	94
HSQLDB	69	605	25	630
LUINDEX	132	2	21	16
LUSEARCH	146	6	29	36
PMD	135	23	66	52
XALAN	464	272	156	535
COMPRESS	10	0	0	0
DB	12	1	2	1
JACK	10	10	0	10
JAVAC	41	2	23	6
JESS	64	1	8	4
RAYTRACE	10	1	0	1

Library Callback Edges

A *library callback edge* is a call graph edge from some method in the library to a specific method in the application part of the program. Table 5.4 shows that AVERROES-based tools generate precise call graphs with respect to library callback edges when compared to SPARK and DOOP for COMPRESS. Across all benchmark programs, $\text{SPARK}_{\text{AVE}}$ generates a geometric mean of 10.9% extra library callback edges when compared to SPARK and DOOP_{AVE} generates a geometric mean of 110.18% extra library callback edges when compared to DOOP.

The conservative over-approximation of the side effects of the original unanalyzed library code in AVERROES causes imprecision in the computation of library call backs. These extra library callback edges then cause the imprecision that we observed in the application call edges and library call edges.

Table 5.5: Frequencies of extra library callback edges computed by SPARK$_{\text{AVE}}$ compared to SPARK. *Other* methods include all methods that are encountered only in one benchmark.

	ANTLR	BLOAT	CHART	HSQLDB	LUINDEX	LUSEARCH	PMD	XALAN	COMPRESS	DB	JACK	JAVAC	JESS	RAYTRACE	**Total**
`<init>`	1	1	1	5	1	1	15	16							**41**
`remove`		7					3								**10**
`finalize`				3	1	4		1							**9**
`run`				5		1		2						1	**9**
`close`				5				3							**8**
`write`				2				6							**8**
`hasMoreElements`	1							1				1			**3**
`nextElement`	1							1				1			**3**
`getType`				1				2							**3**
`next`		2		1											**3**
`clearParameters`				1				2							**3**
`previous`		1		1											**2**
Other	*0*	*15*	*8*	*581*	*0*	*0*	*5*	*238*	*0*	*1*	*10*	*0*	*1*	*0*	*859*
Total	**3**	**26**	**9**	**605**	**2**	**6**	**23**	**272**	**0**	**1**	**10**	**2**	**1**	**1**	**961**

We further investigate the specific causes of the extra library callback edges in the AVERROES-based call graphs. In Tables 5.5 and 5.6, we categorize these edges by the name of the application method that is being called from the library. In particular, we are interested to know whether the library calls a wide variety of application methods, or whether the imprecision is limited to a small number of well-known methods, which could perhaps be handled more precisely on an individual basis.

Table 5.5 shows that the most frequent extra library callback edges in SPARK$_{\text{AVE}}$ compared to SPARK target the methods: `<init>`, `remove`, `finalize`, `run`, `close`, and `write`. Table 5.6 shows that the most frequent extra library callback edges in DOOP$_{\text{AVE}}$ when compared to DOOP target the commonly overridden methods (in descending order): `clone`, `toString`, `equals`, `<init>`, `hashCode`, and `remove`. The number of extra library callback edges generated by SPARK$_{\text{AVE}}$ is

Table 5.6: Frequencies of extra library callback edges computed by DOOP$_{\text{AVE}}$ compared to DOOP. *Other* methods include all methods that are encountered only in one benchmark.

	ANTLR	BLOAT	CHART	HSQLDB	LUINDEX	LUSEARCH	PMD	XALAN	COMPRESS	DB	JACK	JAVAC	JESS	RAYTRACE	**Total**
`clone`	5	52	34		5	7		17				1			**121**
`toString`	3	6	14	10	7	7	6	2				2			**57**
`equals`	1		27	6	1	8	2	3							**48**
`<init>`	1	1	1	6	1	1	15	3							**29**
`hashCode`			10	2	1	8	2	1							**24**
`remove`		14					3								**17**
`write`				2				9							**11**
`run`				5		1		2						1	**9**
`close`				5				3							**8**
`next`		6		1			1								**8**
`printStackTrace`							5	2							**7**
`hasNext`		6					1								**7**
`getType`				1				3							**4**
`getAttributes`				1				3							**4**
`hasMoreElements`	1							1				1			**3**
`nextElement`	1							1				1			**3**
`clearParameters`				1				2							**3**
`read`				2								1			**3**
`accept`					1	1									**2**
`previous`		1		1											**2**
Other	*0*	*33*	*8*	*587*	*0*	*3*	*17*	*483*	*0*	*1*	*10*	*0*	*4*	*0*	*1,146*
Total	**12**	**119**	**94**	**630**	**16**	**36**	**52**	**535**	**0**	**1**	**10**	**6**	**4**	**1**	**1,516**

smaller than that compared to DOOP$_{\text{AVE}}$.

The constructor `<init>` ranks highly in HSQLDB, PMD, and XALAN. This is because these benchmarks use class constants. AVERROES conservatively assumes that if a class constant is created (using `Class.forName()`), an object of that type might also be instantiated by the library and its constructor called.

The benchmark programs HSQLDB and XALAN have the highest frequency of imprecise library callback edges. In the case of HSQLDB, most of those imprecise call backs are to methods of classes in the package `org.hsqldb.jdbc` (SPARK = 558, DOOP = 564). In XALAN, most of the imprecise call backs are to methods of classes in the packages `org.apache.xalan.*` (SPARK = 154, DOOP = 250) and `org.apache.xml.*` (SPARK = 110, DOOP = 266). In both of these benchmarks, the high imprecision is due to the fact that each benchmark contains its own implementation of a large subsystem (JDBC and XML) whose interface is defined in the library. Averroes conservatively assumes that the library could call all interface methods of the subsystem. On the other hand, a whole-program analysis computes a more restricted set of only those interface methods of the subsystem that the library actually calls.

Finding 2: Most of the imprecise call graph edges computed when using AVERROES are due to spurious library call back edges.

5.4 Call Graph Size

As we mentioned earlier, call graphs are a key prerequisite to all inter-procedural analyses. Therefore, any change in the size of the call graph will affect the performance of the analyses that use it as input. Since an AVERROES-based tool over-approximates the generated call graph, we evaluate the size of the generated call graph (in terms of the total number of edges) for AVERROES-based tools (SPARK$_{\text{AVE}}$ and DOOP$_{\text{AVE}}$) compared to whole-program tools (SPARK and DOOP). Table 5.7 shows that the AVERROES-based tools generate call graphs of equal or smaller size than SPARK and DOOP for the benchmark programs ANTLR and COMPRESS. Additionally, SPARK$_{\text{AVE}}$ generates call graphs of smaller size than SPARK for the benchmark programs CHART, LUINDEX, LUSEARCH, PMD, XALAN, DB, JAVAC, JESS, and RAYTRACE.

It is counterintuitive that the call graphs generated by the SPARK$_{\text{AVE}}$ are smaller than those generated by SPARK, which analyzes the whole program precisely. This result is primarily due to imprecisions in SPARK. To model objects created by the Java VM or by the Java standard library using reflection, SPARK uses special abstract objects whose type is not known (i.e., any subtype of `java.lang.Object`). SPARK does not filter these objects when enforcing declared types, so these objects pass freely through casts and pollute many points-to sets in the program. This affects the precision of the points-to sets of the method call receivers and leads to many

Table 5.7: Comparing the size of the call graph generated by the Averroes-based tools to Spark and Doop in terms of call graph edges.

	Spark	$\text{Spark}_{\text{Ave}}$	Doop	Doop_{Ave}
antlr	7,033	6,990	6,984	6,981
bloat	14,527	16,240	13,358	15,985
chart	8,282	2,674	2,395	2,674
hsqldb	8,789	10,585	7,387	10,571
luindex	1,880	1,211	919	1,207
lusearch	3,417	2,518	2,025	2,497
pmd	7,825	5,253	4,566	5,245
xalan	14,598	12,993	10,177	12,482
compress	63	53	53	53
db	82	75	73	75
jack	753	770	744	770
javac	8,877	8,693	8,524	8,581
jess	76	25	20	25
raytrace	444	435	434	435

imprecise call graph edges in SPARK. The precision of SPARK can be improved by redesigning the mechanism that it uses to model these objects.

> **Finding 3:** Call graphs generated by $\text{SPARK}_{\text{AVE}}$ are smaller in size compared to those generated by SPARK, while those generated by DOOP_{AVE} are similar in size compared to those generated by DOOP.

5.5 Performance

To evaluate how much work a whole-program analysis saves by using AVERROES, we first compare the size of the generated placeholder library with the size of the original Java standard library. We then measure the reductions in execution time and memory requirements of both SPARK and DOOP when using AVERROES.

5.5.1 AVERROES Placeholder Library Size

Over all of the benchmark programs that we have experimented with, the average size of the input library is 25 MB (min: 25 MB, max: 30 MB, geometric mean: 25 MB), while the average size of the generated AVERROES library is only 80 kB (min: 20 kB, max: 370 kB, geometric mean: 80 kB). Additionally, the average number of methods in the original input library is 36,000 (min: 19,462, max: 48,610, geometric mean: 35,615), while the average number of methods in the generated AVERROES library is only 600 (min: 137, max: 3,327, geometric mean: 570). That means that the number of methods in the placeholder library is smaller by a factor of 62x (min: 13x, max: 286x, geometric mean: 62x). As we will see, this reduction in the library size significantly reduces the time and memory required to do whole-program analysis.

> **Finding 4:** The placeholder library generated by AVERROES is very small compared to the original Java standard library.

5.5.2 Execution Time

We have compared the execution times of both SPARK and DOOP when analyzing each benchmark with the AVERROES placeholder library and the original Java standard library. We break the total time required to construct a call graph into three components. First, the AVERROES *library generation time* is the time required for AVERROES to inspect the application for references to the library and to generate the placeholder library. Second, the *overhead time* is the time required

for SPARK or DOOP to prepare for call graph construction analysis. In the case of SPARK, this preparation includes reading the whole program from disk and constructing internal data structures. In the case of DOOP, this preparation additionally includes generating the constraints required for the analysis and encoding them in Datalog relations. Third, the *analysis time* is the time required for SPARK or DOOP to solve the constraints and generate a call graph.

Figure 5.1 compares the times required for call graph construction by SPARK and DOOP with the original Java library (denoted SPARK and DOOP) and with the AVERROES placeholder library (denoted $\text{SPARK}_{\text{AVE}}$ and DOOP_{AVE}). AVERROES reduces the analysis time of SPARK by a factor of 8x (min: 3.2x, max: 45.9x, geometric mean: 7.9x) and of DOOP by a factor of 3.5x (min: 0.7x, max: 5.7x, geometric mean: 3.45x). In general, whole-program analysis is expensive not only because of the analysis itself, but also due to the overhead of reading a large whole program from disk and pre-processing it. Replacing the large Java library with the much smaller AVERROES placeholder library reduces the time that SPARK executes (including overhead and analysis time) by a factor of 5.4x (min: 2.9x, max: 15.2x, geometric mean: 5.4x), and the time that DOOP executes by a factor of 3.1x (min: 1.6x, max: 4.6x, geometric mean: 3.1x). When the AVERROES library generation time is added to the time taken by SPARK or DOOP to finish, the total overall time to execute $\text{SPARK}_{\text{AVE}}$ is faster than SPARK by a factor of 3.5x (min: 1.9x, max: 7.1x, geometric mean: 3.46x), and the total overall time to execute DOOP_{AVE} is faster than DOOP by a factor of 2.8x (min: 1.4x, max: 4.2x, geometric mean: 2.8x).

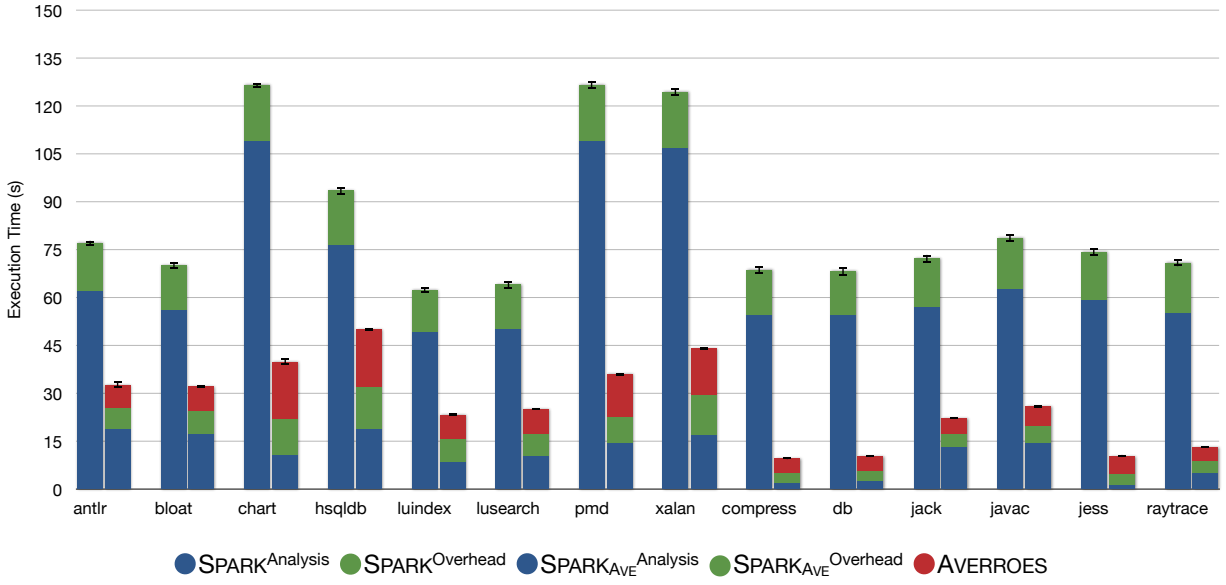
Finding 5: AVERROES enables whole-program tools to construct application call graphs faster.

5.5.3 Memory Requirements

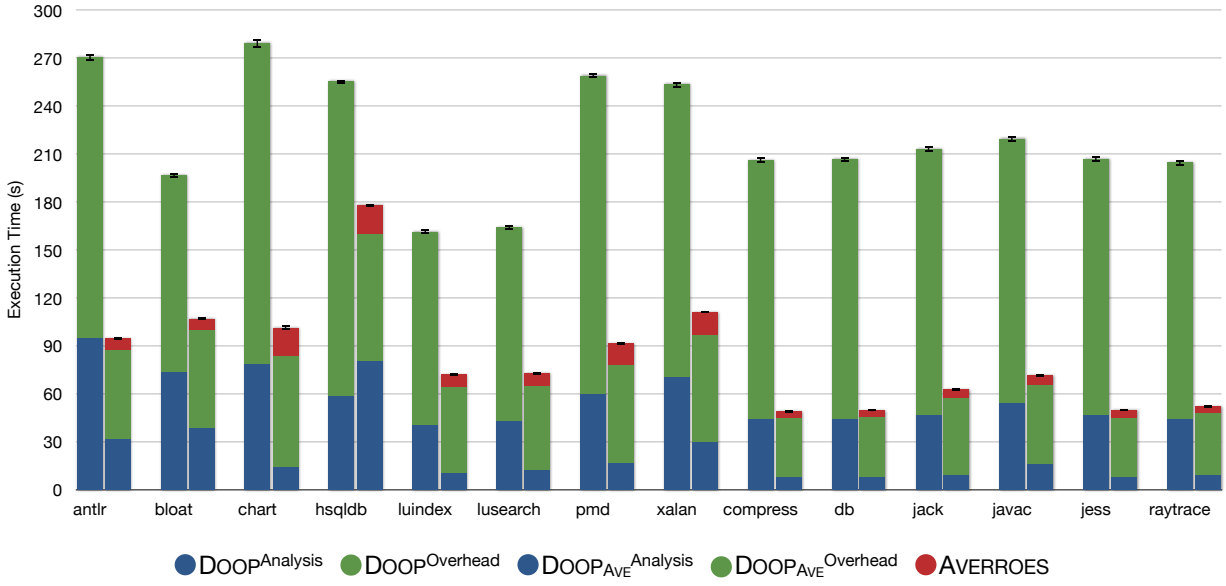
SPARK and DOOP store their intermediate results in different ways. SPARK does all the calculations in memory, while DOOP stores intermediate facts in a LogicBlox [42] database on disk. Therefore, we use different methods of calculating the memory requirements of each tool. We compare the maximum amount of heap space used during call graph construction by $\text{SPARK}_{\text{AVE}}$ and SPARK. On the other hand, we compare the on-disk size of the database of relations computed by DOOP_{AVE} and DOOP.

Figure 5.2 compares the memory usage of $\text{SPARK}_{\text{AVE}}$ against SPARK, and DOOP_{AVE} against DOOP. Overall, $\text{SPARK}_{\text{AVE}}$ requires 12x less heap space than SPARK (min: 4.9x, max: 26.6x, geometric mean: 11.9x), and DOOP_{AVE} uses 8.4x less disk space than DOOP (min: 2.6x, max: 23.7x, geometric mean: 8.4x).

Finding 6: Using AVERROES reduces the memory requirements of whole-program analysis tools.

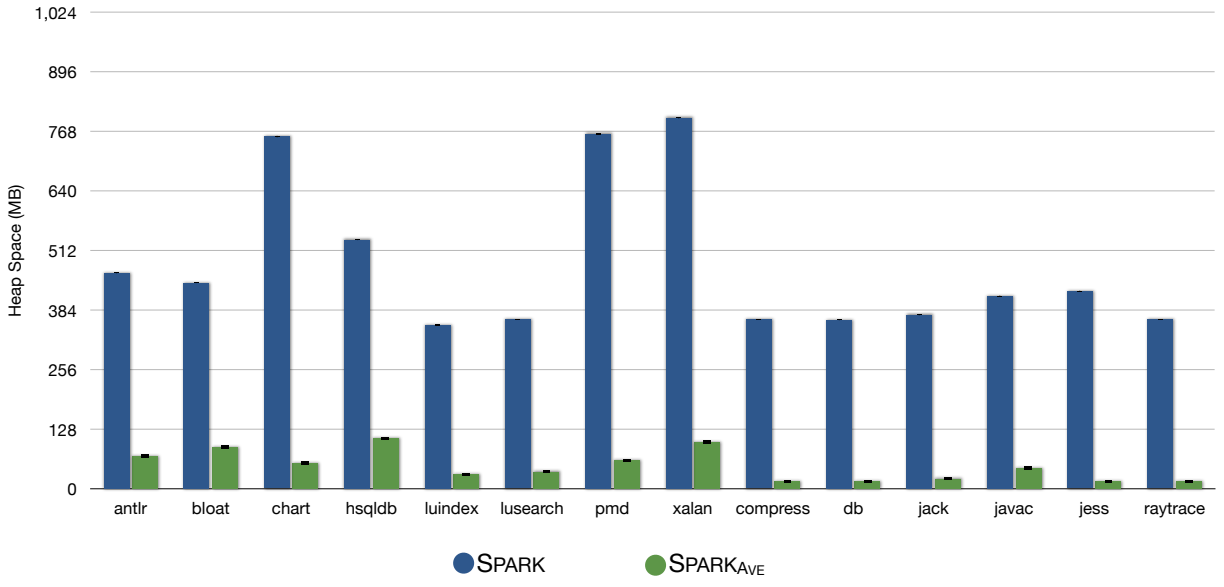


(a)

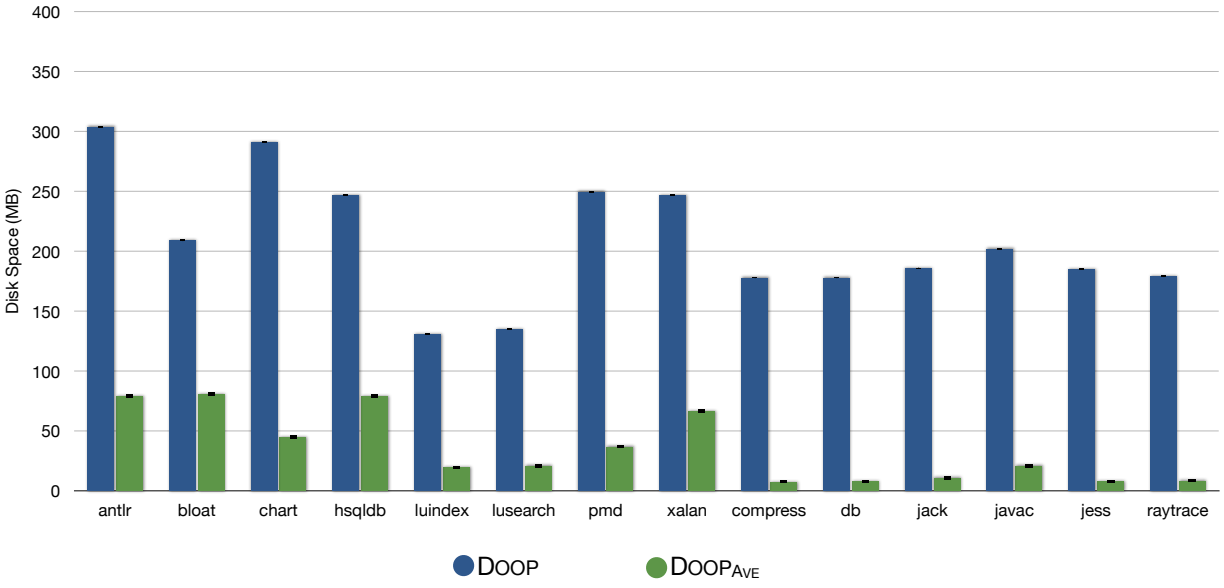


(b)

Figure 5.1: The execution time of whole-program tools (SPARK and DOOP) compared to AVERROES-based tools ($\text{SPARK}_{\text{AVE}}$ and DOOP_{AVE}).



(a)



(b)

Figure 5.2: The memory requirements of whole-program tools (SPARK and DOOP) compared to AVERROES-based tools ($\text{SPARK}_{\text{AVE}}$ and DOOP_{AVE}).

5.6 Summary

In this chapter, we have evaluated Averroes, a tool that generates a placeholder library that over-approximates the possible behaviour of an original library. This enables Averroes to reduce the difficulty of constructing sound static call graphs in the presence of reflection. Comparing the precision of the Averroes-based tools shows that the imprecise library call back edges computed by $\text{Spark}_{\text{Ave}}$ and Doop_{Ave} further caused some spurious edges to be computed between application methods and from application methods to the library. Nevertheless, the total number of edges computed by the Averroes-based tools is generally smaller than or equal to those computed by their whole-program counterparts.

Since Averroes does not analyze the whole program, it can construct the placeholder library quickly. The size of the placeholder library generated by Averroes is typically in the order of 80 kB of class files (comparatively, the Java standard library is 25 MB). Averroes improves the analysis time of whole-program call graph construction by a factor of 3.5x to 8x, while reducing the memory required to finish the analysis by a factor of 8.4x to 12x.

Chapter 6

Correctness Proof

This chapter provides a correctness proof for AVERROES, which implements the separate compilation assumption, based on Featherweight Java (FJ) [30]. The chapter is organized as follows. Section 6.1 covers necessary background information regarding Featherweight Java. Section 6.2 defines the additions that we made to FJ to be able to allow for a proof of correctness for AVERROES. We discuss how an FJ program is transformed to its corresponding AVERROES version, FJ_{AVE}, in Section 6.4. We define the library abstraction relation $\propto$ in Section 6.5, and additional reduction rules to FJ in Section 6.6. Section 6.3 provides the intuition of the proof. We go over the outline of the proof in Section 6.7. We then lay out the formal proof for all of the lemmas and theorems.

6.1 Featherweight Java

Featherweight Java (FJ) is a compact calculus that represents a lightweight version of Java [30]. FJ is typically used to do rigorous proofs for Java-based analyses as it comes with a proof of

L	::=	$\texttt{class}\ C\ \texttt{extends}\ C\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}$	(C-DECL)
K	::=	$C(\overline{C}\ \overline{f})\{\texttt{super}(\overline{f});\ \texttt{this}.\overline{f} = \overline{f};\}$	(K-DECL)
M	::=	$C\ m(\overline{C}\ \overline{x})\{\texttt{return}\ e;\}$	(M-DECL)
e	::=	$x \mid e.f \mid e.m(\overline{e}) \mid \texttt{new}\ C(\overline{e}) \mid (C)e$	(EXPR)

Figure 6.1: Featherweight Java syntax [30]

$$C <: C \qquad \text{(S-Ref)}$$

$$\frac{C <: D \qquad D <: E}{C <: E} \qquad \text{(S-Trans)}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\ldots\}}{C <: D} \qquad \text{(S-Sub)}$$

Figure 6.2: Featherweight Java subtyping rules [30]

type soundness. FJ allows the definition of classes, methods, class constructors, and expressions. For the sake of compactness, expressions are limited to five forms: variable, field access, method invocation, object creation, and casting. Figure 6.1 provides the full syntax of FJ. A class table CT is a mapping from class names C to class declarations L. An FJ program is defined as a pair (CT, e) of a class table and an expression. Figure 6.2 formally defines the subtyping relations between classes in FJ. Figure 6.3 shows auxiliary functions required for field, method type, and method body lookup in FJ. For all of those auxiliary functions, the original FJ rules leave the class table implicit. We have made the class table explicit, and we denote it by Σ, so that we can distinguish the class tables of the FJ program and the FJ_{Ave} program. The typing rules for expressions, method declarations, and class declarations are shown in Figure 6.4. More details about the type soundness of FJ can be found in [30]. Figure 6.5 shows all of the reduction rules for FJ. There are three main reduction rules for fields, method invocations, and casting (whose names start with R-). These rules can be applied to any sub-expression in an expression, which motivates the need for the congruence rules (whose names start with RC-).

6.2 Additions to Featherweight Java

We have made a few additions to FJ so that it can support proving the correctness of Averroes. First, we add a new form of expressions `LIB`, which is an empty expression that stands for any expression that the library could create and return to the application.

Second, we need to represent the Averroes field `libraryPointsTo` in the context of FJ. That field is a static field that represents the union of the points-to sets of all of the local variables in the original library code. However, FJ does not have mutable static fields. Therefore, we introduce `LPT`, which is a global points-to set for the library part of an FJ_{Ave} program. `LPT` represents all possible expressions that the library has access to according to the constraints of the separate compilation assumption discussed in Chapter 3. Since the evaluation strategy in FJ is call-by-name, the elements of `LPT` are expressions as opposed to values stored in the Averroes field `libraryPointsTo`. A configuration in the execution trace in FJ is just a partially reduced

$$\mathit{fields}(\texttt{Object}) = \bullet \qquad \text{(F-Object)}$$

$$\frac{\texttt{class } C \texttt{ extends } D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} \in \Sigma \qquad \Sigma \vdash \mathit{fields}(D) = \overline{D}\ \overline{g}}{\Sigma \vdash \mathit{fields}(C) = \overline{D}\ \overline{g}, \overline{C}\ \overline{f}} \qquad \text{(Fields)}$$

$$\frac{\begin{array}{c}\texttt{class } C \texttt{ extends } D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} \in \Sigma \\ B\ m(\overline{B}\ \overline{x})\{\texttt{return } e;\} \in \overline{M}\end{array}}{\Sigma \vdash \mathit{mtype}(m, C) = \overline{B} \to B} \qquad \text{(M-Type)}$$

$$\frac{\texttt{class } C \texttt{ extends } D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} \in \Sigma \qquad m \notin \overline{M}}{\Sigma \vdash \mathit{mtype}(m, C) = \Sigma \vdash \mathit{mtype}(m, D)} \qquad \text{(M-Type-Inh)}$$

$$\frac{\begin{array}{c}\texttt{class } C \texttt{ extends } D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} \in \Sigma \\ B\ m(\overline{B}\ \overline{x})\{\texttt{return } e;\} \in \overline{M}\end{array}}{\Sigma \vdash \mathit{mbody}(m, C) = \overline{x}.e} \qquad \text{(M-Body)}$$

$$\frac{\texttt{class } C \texttt{ extends } D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} \in \Sigma \qquad m \notin \overline{M}}{\Sigma \vdash \mathit{mbody}(m, C) = \Sigma \vdash \mathit{mbody}(m, D)} \qquad \text{(M-Body-Inh)}$$

Figure 6.3: Featherweight Java auxiliary functions [30]

$$\Gamma \vdash x \ : \ \Gamma(x) \qquad \text{(T-Var)}$$

$$\frac{\Gamma \vdash e_0 \ : \ C_0 \qquad \Sigma \vdash \mathit{fields}(C_0) = \overline{C}\ \overline{f}}{\Gamma \vdash e_0.f_i \ : \ C_i} \qquad \text{(T-Field)}$$

$$\frac{\Gamma \vdash e_0 \ : \ C_0 \qquad \Sigma \vdash \mathit{mtype}(m,\, C_0) = \overline{D} \to C \qquad \Gamma \vdash \overline{e} \ : \ \overline{C} \qquad \overline{C} <: \overline{D}}{\Gamma \vdash e_0.m(\overline{e}) \ : \ C} \qquad \text{(T-Invk)}$$

$$\frac{\Sigma \vdash \mathit{fields}(C) = \overline{D}\ \overline{f} \qquad \Gamma \vdash \overline{e} \ : \ \overline{C} \qquad \overline{C} <: \overline{D}}{\Gamma \vdash \texttt{new}\ C(\overline{e}) \ : \ C} \qquad \text{(T-New)}$$

$$\frac{\Gamma \vdash e_0 \ : \ D \qquad D <: C}{\Gamma \vdash (C)e_0 \ : \ C} \qquad \text{(T-UCast)}$$

$$\frac{\Gamma \vdash e_0 \ : \ D \qquad C <: D \qquad C \neq D}{\Gamma \vdash (C)e_0 \ : \ C} \qquad \text{(T-DCast)}$$

$$\frac{\Gamma \vdash e_0 \ : \ D \qquad C \not<: D \qquad D \not<: C \qquad \textit{stupid warning}}{\Gamma \vdash (C)e_0 \ : \ C} \qquad \text{(T-SCast)}$$

$$\frac{\begin{array}{c}\overline{x} : \overline{C},\ \texttt{this} : C \vdash e_0 \ : \ E_0 \qquad E_0 <: C_0 \\ \texttt{class}\ C\ \texttt{extends}\ D\ \{\ldots\} \\ \text{if } \Sigma \vdash \mathit{mtype}(m,\, D) = \overline{D} \to D_0, \text{ then } \overline{C} = \overline{D} \text{ and } C_0 = D_0\end{array}}{C_0\ m(\overline{C}\ \overline{x})\{\texttt{return}\ e_0;\}\ \texttt{OK IN}\ C} \qquad \text{(T-Method)}$$

$$\frac{K = C(\overline{D}\ \overline{g},\ \overline{C}\ \overline{f})\{\texttt{super}(\overline{g});\ \texttt{this}.\overline{f} = \overline{f};\} \qquad \Sigma \vdash \mathit{fields}(D) = \overline{D}\ \overline{g} \qquad \overline{M}\ \texttt{OK IN}\ C}{\texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}\ \texttt{OK}} \qquad \text{(T-Class)}$$

Figure 6.4: Featherweight Java typing rules [30]

$$\frac{\Sigma \vdash \mathit{fields}(C) = \overline{C}\ \overline{f}}{(\texttt{new}\ C(\overline{e})).f_i \xrightarrow{\Sigma} e_i} \quad \text{(R-Field)}$$

$$\frac{\Sigma \vdash \mathit{mbody}(m,\, C) = \overline{x}.e_0}{(\texttt{new}\ C(\overline{e})).m(\overline{d}) \xrightarrow{\Sigma} [\overline{d}/\overline{x},\ \texttt{new}\ C(\overline{e})/\texttt{this}]e_0} \quad \text{(R-Invk)}$$

$$\frac{C <: D}{(D)(\texttt{new}\ C(\overline{e})) \xrightarrow{\Sigma} \texttt{new}\ C(\overline{e})} \quad \text{(R-Cast)}$$

$$\frac{e_0 \xrightarrow{\Sigma} e_0{}'}{e_0.f \xrightarrow{\Sigma} e_0{}'.f} \quad \text{(RC-Field)}$$

$$\frac{e_0 \xrightarrow{\Sigma} e_0{}'}{e_0.m(\overline{e}) \xrightarrow{\Sigma} e_0{}'.m(\overline{e})} \quad \text{(RC-Invk-Recv)}$$

$$\frac{e_i \xrightarrow{\Sigma} e_i{}'}{e_0.m(\ldots, e_i, \ldots) \xrightarrow{\Sigma} e_0.m(\ldots, e_i{}', \ldots)} \quad \text{(RC-Invk-Arg)}$$

$$\frac{e_i \xrightarrow{\Sigma} e_i{}'}{\texttt{new}\ C(\ldots, e_i, \ldots) \xrightarrow{\Sigma} \texttt{new}\ C(\ldots, e_i{}', \ldots)} \quad \text{(RC-New-Arg)}$$

$$\frac{e_0 \xrightarrow{\Sigma} e_0{}'}{(C)e_0 \xrightarrow{\Sigma} (C)e_0{}'} \quad \text{(RC-Cast)}$$

Figure 6.5: Featherweight Java reduction rules [30]

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} \qquad B\ m(\overline{B}\ \overline{x})\{\texttt{return}\ e;\} \in \overline{M} \qquad C \in \Sigma}{\Sigma \vdash \mathit{mresolve}(m,\, C) = C} \tag{M-Res}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} \qquad B\ m(\overline{B}\ \overline{x})\{\texttt{return}\ e;\} \notin \overline{M} \qquad C \in \Sigma \qquad D \in \Sigma}{\Sigma \vdash \mathit{mresolve}(m,\, C) = \Sigma \vdash \mathit{mresolve}(m,\, D)} \tag{M-Res-Inh}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{E}\ \overline{f};\ K\ \overline{M}\} \in \Sigma}{\Sigma \vdash \mathit{declaringclass}(f) = C} \tag{D-Fields}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} \in \Sigma}{\Sigma \vdash \mathit{dmethods}(C) = \overline{M}} \tag{D-Methods}$$

Figure 6.6: Additional auxiliary functions used by FJ_{Ave}.

expression e. In the execution trace of the FJ_{Ave} program, a configuration is an expression e paired with the static mutable field LPT.

Additional auxiliary functions that we will use throughout the proof are given in Figure 6.6. The functions M-Res and M-Res-Inh are partial functions that resolve a method m in class C. If m is called on an object of concrete type C, $\Sigma \vdash \mathit{mresolve}(m,\, C)$ specifies the class that contains the body of the method that will be invoked. The function D-Fields finds the declaring class of a given field f. The function D-Methods computes the set of methods declared in a given class C.

6.3 Intuition

The main goal of the correctness proof is to show that if in the application code of the FJ program, method m calls m', then there is a trace for the transformed Averroes program in which m also calls m'. However, the trace of the Averroes program cannot contain code from the library methods of the FJ program because the Averroes program does not contain the bodies of those methods. Therefore, Averroes replaces the unanalyzed library code with the special expression LIB that simulates the side effects the original library code would have on the application code.

Figure 6.7 shows the expression tree of an FJ program, on the left-hand-side, and its Averroes counterpart, on the right-hand-side. In the FJ program, an A expression is an expression from the body of an application method, and an L expression represents an expression from the body of a library method. Notice that the expression tree on the left-hand-side contains calls

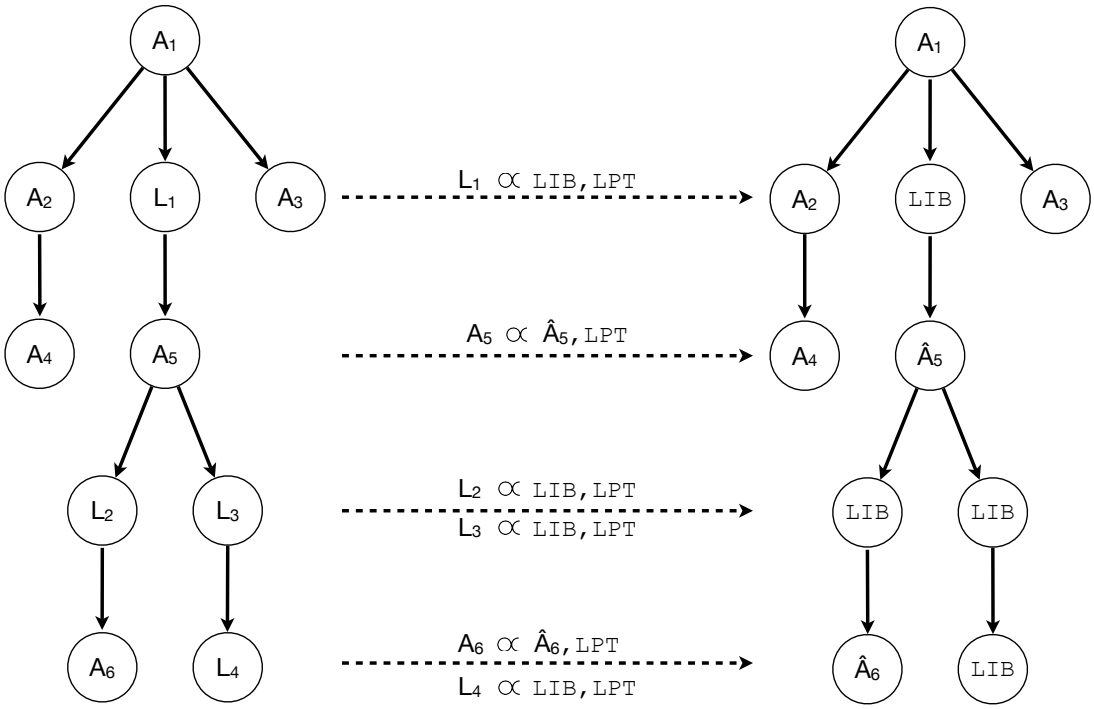


Figure 6.7: The intuition behind the correctness proof. The left-hand-side is the original FJ program, while the right hand side is the transformed AVERROES program (only the library code is transformed; application code stays the same). An "A" expression is an application expression, while an "L" expression is a library expression.

from the application to the library and from the library back to the application. Therefore, the expression tree contains sub-expressions from both application and library method bodies. The $\propto$ relation, which will be defined in detail in Section 6.5, relates sub-expressions in the FJ program with the sub-expressions that represent them in the FJ_{AVE} program.

6.4 AVERROES Transformation

Given an FJ program (CT, e_0), AVERROES transforms it to the FJ_{AVE} program (CT', e_0) according to the following rules.

Class Table [AVE-CT]

We split the class table CT of an FJ program into the disjoint union $CT_A \cup CT_L$, where CT_A represents the application classes and CT_L represents the library classes. The separate compilation assumption prohibits any library class (in CT_L) to be a subclass of any application class (in CT_A).

The class table CT' of the corresponding FJ_{AVE} program is then the disjoint union $CT_A \cup CT'_L$. The set CT_A remains the same as in the original FJ program. The set CT'_L is the set of transformed library classes found in the AVERROES placeholder library (Constraint 1). The classes in CT'_L are a subset of the classes in CT_L. Therefore, CT'_L retains the property that any of its element classes cannot subclass any application class. For each class in CT'_L, its superclass is also in CT'_L. Also, if a class is in CT_A and its superclass is a library class (i.e., in CT_L), then the superclass is also in CT'_L. In general, for every class in CT' (except `Object`), its superclass is also in CT'. The original bodies of all methods of classes in CT'_L are replaced with an empty body that just returns `LIB`:

$$M ::= C\ m(\overline{C}\ \overline{x})\{\texttt{return LIB;}\}$$

Library Methods [AVE-METHOD]

In all of the transformed library classes in CT'_L, AVERROES only keeps methods that are directly referenced, or could potentially be called from an application method. As we discussed earlier in Chapter 4, AVERROES implements this by looking at the method references in the constant pool of application classes (e.g., if it appears at a call site in the application). Such methods are retained in the transformed library class. Additionally, AVERROES analyzes the class hierarchy of a program and retains library methods that could be inherited by an application class.

Library Fields [AVE-FIELD]

Similar to library methods, AVERROES only keeps fields in classes in CT'_L that are directly referenced, or could potentially be accessed from an application method. AVERROES computes this set of fields by looking at the field references in the constant pool of application classes (e.g., if it appears at a field read or write site in the application). Such fields are retained in the transformed library class. Additionally, AVERROES analyzes the class hierarchy of a program and retains library fields that could be inherited by an application class.

6.5 Library Abstraction: The Relation $\propto$

The relation $\propto$ relates an expression in the trace of the FJ program with a corresponding expression in the trace of the corresponding FJ_{AVE} program. In particular, since some method bodies from the FJ program are missing from the FJ_{AVE} program, the relation $\propto$ must abstract some expressions (with `LIB`). Those expressions are mainly expressions that appear in the bodies of library methods that are in CT_L but not in CT'_L (according to AVE-METHOD). The special expression `LIB` corresponds to the method `doItAll()` in AVERROES. The FJ_{AVE} program should generally follow the same trace of execution as the FJ program, except that execution of library code in the FJ program is replaced by `LIB` in the FJ_{AVE} program.

In the FJ program in Figure 6.7, L expressions that are missing in the FJ_{AVE} program (i.e., L_1, L_2, L_3, and L_4) are replaced with `LIB`. Therefore, for each of their sub-expressions e, a corresponding sub-expression that is related to e by the $\propto$ relation is stored in `LPT`.

The full definition of the abstraction relation $\propto$ is given in Figure 6.8. We follow the same notation adopted by FJ. We write $\overline{e}$ as shorthand for a possibly empty sequence $e_1, \ldots, e_n$. We write $\overline{e} \propto \overline{\hat{e}}, \texttt{LPT}$ as shorthand for $\forall_i\, e_i \propto \hat{e}_i, \texttt{LPT}$. We also use $\overline{e} \propto \texttt{LIB}, \texttt{LPT}$ and $\overline{e} \propto \overline{\texttt{LIB}}, \texttt{LPT}$ interchangeably as shorthand for $\forall_i\, e_i \propto \texttt{LIB}, \texttt{LPT}$.

For each form of an FJ expression, there are generally two rules in the definition of the relation $\propto$: the REL-* rule relates the original expression to an expression of the same form in the FJ_{AVE} program, and the REL-LIB-* rule abstracts the original expression with `LIB`. The rules REL-FIELD and REL-LIB-FIELD relate field accesses. REL-NEW and REL-LIB-NEW relate object instantiations. REL-INVK and REL-LIB-INVK relate method invocations. REL-CAST and REL-LIB-CAST relate casting operations.

When the FJ program is executing within the library, it may be manipulating expressions that were passed into the library from the application, and are therefore contained in the `LPT` set in the FJ_{AVE} program. Therefore, any expression in the FJ program that is in `LPT` is also allowed to be abstracted by `LIB`. The rule REL-LPT formally defines this property.

$$\frac{e \propto \hat{e}, \texttt{LPT}}{e.f \propto \hat{e}.f, \texttt{LPT}} \quad \text{(Rel-Field)}$$

$$\frac{e \propto \texttt{LIB}, \texttt{LPT} \qquad CT \vdash \mathit{declaringclass}(f) \in CT_L}{e.f \propto \texttt{LIB}, \texttt{LPT}} \quad \text{(Rel-Lib-Field)}$$

$$\frac{C \in CT' \qquad \overline{e} \propto \overline{\hat{e}}, \texttt{LPT}}{\texttt{new}\ C(\overline{e}) \propto \texttt{new}\ C(\overline{\hat{e}}), \texttt{LPT}} \quad \text{(Rel-New)}$$

$$\frac{C \in CT_L \qquad \overline{e} \propto \texttt{LIB}, \texttt{LPT}}{\texttt{new}\ C(\overline{e}) \propto \texttt{LIB}, \texttt{LPT}} \quad \text{(Rel-Lib-New)}$$

$$\frac{e \propto \hat{e}, \texttt{LPT} \qquad \overline{e} \propto \overline{\hat{e}}, \texttt{LPT}}{e.m(\overline{e}) \propto \hat{e}.m(\overline{\hat{e}}), \texttt{LPT}} \quad \text{(Rel-Invk)}$$

$$\frac{e \propto \texttt{LIB}, \texttt{LPT} \qquad \overline{e} \propto \texttt{LIB}, \texttt{LPT} \qquad \exists C \in CT_L\,.\ m \in CT \vdash \mathit{dmethods}(C)}{e.m(\overline{e}) \propto \texttt{LIB}, \texttt{LPT}} \quad \text{(Rel-Lib-Invk)}$$

$$\frac{e \propto \hat{e}, \texttt{LPT}}{(C)e \propto (C)\hat{e}, \texttt{LPT}} \quad \text{(Rel-Cast)}$$

$$\frac{e \propto \texttt{LIB}, \texttt{LPT}}{(C)e \propto \texttt{LIB}, \texttt{LPT}} \quad \text{(Rel-Lib-Cast)}$$

$$\frac{e \propto \hat{e}, \texttt{LPT} \qquad \hat{e} \in \texttt{LPT}}{e \propto \texttt{LIB}, \texttt{LPT}} \quad \text{(Rel-Lpt)}$$

Figure 6.8: The definition of the abstraction relation $\propto$

6.6 AVERROES Reduction

Since the FJ reduction rules can only be used to reduce expressions, a new reduction relation is needed to reduce a pair $(e, \texttt{LPT})$ in the FJ_{AVE} program. Additional rules are also required to reduce the special expression `LIB`. Therefore, we define a new reduction relation $\twoheadrightarrow$ to satisfy both requirements.

Figure 6.9 provides the definition of a new reduction relation that executes expressions in the FJ_{AVE} program. The reduction relation is of the form $e, \texttt{LPT} \twoheadrightarrow \hat{e}, \texttt{LPT}'$, read: expression e reduces to expression $\hat{e}$ in one step, while adding some expressions to the set `LPT` to yield the set $\texttt{LPT}'$. We write $\twoheadrightarrow^*$ for the reflexive and transitive closure of $\twoheadrightarrow$.

The original FJ reduction rules are preserved by the rule RA-FJ which maps any reduction that could take place in FJ, with CT' as the class table, to a reduction that could take place in FJ_{AVE}. We then define four main reduction rules, each corresponding to one of the constraints of the separate compilation assumption as implemented in the AVERROES `doItAll()` method. Just as the `doItAll()` method manipulates values in the field `libraryPointsTo`, the corresponding constraints manipulate expressions in the set `LPT`.

- RA-FIELD models fields accessed by the library (Constraint 5).
- RA-NEW models classes that could be instantiated by the library (Constraint 2).
- RA-INVK models methods that could be called by the library (Constraint 4).
- RA-CAST models cast operations that the library is allowed to make.

All of these RA-* rules add to the set `LPT` expressions that the library could potentially create, satisfying Constraint 3. Other constraints of the separate compilation assumption do not have corresponding rules because of the limited forms an FJ expression can have:

- Constraint 6 does not apply because expressions in FJ cannot be array accesses.
- Constraint 7 does not apply because classes in FJ do not have static initializers.
- Constraint 8 does not apply because FJ omits exceptions from the full Java.

Three more reduction rules are defined to enable the following desired behaviour:

- RA-LIB-INVK models calls into the library that should reduce to the special expression `LIB`, which replaces the original library method body.

$$\frac{e \xrightarrow{CT'} e'}{e, \texttt{LPT} \twoheadrightarrow e', \texttt{LPT}} \quad \text{(RA-FJ)}$$

$$\frac{CT' \vdash \mathit{declaringclass}(f) \in CT'_L}{d, \texttt{LPT} \twoheadrightarrow d, \texttt{LPT} \cup \{\texttt{LIB}.f\}} \quad \text{(RA-Field)}$$

$$\frac{C \in CT'_L}{d, \texttt{LPT} \twoheadrightarrow d, \texttt{LPT} \cup \{\texttt{new } C(\overline{\texttt{LIB}})\}} \quad \text{(RA-New)}$$

$$\frac{m \in CT' \vdash \mathit{dmethods}(C) \qquad C \in CT'_L}{d, \texttt{LPT} \twoheadrightarrow d, \texttt{LPT} \cup \{\texttt{LIB}.m(\overline{\texttt{LIB}})\}} \quad \text{(RA-Invk)}$$

$$\frac{}{(C)\texttt{LIB}, \texttt{LPT} \twoheadrightarrow \texttt{LIB}, \texttt{LPT}} \quad \text{(RA-Cast)}$$

$$\frac{CT' \vdash \mathit{mresolve}(m,\, C) \in CT'_L}{(\texttt{new } C(\overline{e})).m(\overline{d}), \texttt{LPT} \twoheadrightarrow \texttt{LIB}, \texttt{LPT} \cup \{\texttt{new } C(\overline{e})\} \cup \overline{d}} \quad \text{(RA-Lib-Invk)}$$

$$\frac{e \in \texttt{LPT}}{\texttt{LIB}, \texttt{LPT} \twoheadrightarrow e, \texttt{LPT}} \quad \text{(RA-Return)}$$

$$\frac{e \in \texttt{LPT} \qquad e, \texttt{LPT} \twoheadrightarrow e', \texttt{LPT}'}{d, \texttt{LPT} \twoheadrightarrow d, \texttt{LPT} \cup \texttt{LPT}' \cup \{e'\}} \quad \text{(RA-Sub)}$$

$$\frac{e_0, \texttt{LPT} \twoheadrightarrow {e_0}', \texttt{LPT}'}{e_0.f, \texttt{LPT} \twoheadrightarrow {e_0}'.f, \texttt{LPT} \cup \texttt{LPT}'} \quad \text{(RAC-Field)}$$

$$\frac{e_0, \texttt{LPT} \twoheadrightarrow {e_0}', \texttt{LPT}'}{e_0.m(\overline{e}), \texttt{LPT} \twoheadrightarrow {e_0}'.m(\overline{e}), \texttt{LPT} \cup \texttt{LPT}'} \quad \text{(RAC-Invk-Recv)}$$

$$\frac{e_i, \texttt{LPT} \twoheadrightarrow {e_i}', \texttt{LPT}'}{e_0.m(\ldots, e_i, \ldots), \texttt{LPT} \twoheadrightarrow e_0.m(\ldots, {e_i}', \ldots), \texttt{LPT} \cup \texttt{LPT}'} \quad \text{(RAC-Invk-Arg)}$$

$$\frac{e_i, \texttt{LPT} \twoheadrightarrow {e_i}', \texttt{LPT}'}{\texttt{new } C(\ldots, e_i, \ldots), \texttt{LPT} \twoheadrightarrow \texttt{new } C(\ldots, {e_i}', \ldots), \texttt{LPT} \cup \texttt{LPT}'} \quad \text{(RAC-New-Arg)}$$

$$\frac{e_0, \texttt{LPT} \twoheadrightarrow {e_0}', \texttt{LPT}'}{(C)e_0, \texttt{LPT} \twoheadrightarrow (C){e_0}', \texttt{LPT} \cup \texttt{LPT}'} \quad \text{(RAC-Cast)}$$

Figure 6.9: The definition of the reduction relation $\twoheadrightarrow$

- RA-RETURN models returning from a method call to the library by reducing LIB to an expression in the set LPT.
- RA-SUB allows the set LPT to expand by including the reduction of any of its elements. This rule is needed because the FJ semantics is call-by-name, so any sub-expression can be reduced at any time, including expressions in the set LPT.

Similar to FJ, all of the previous RA-* reduction rules may be applied at any point in an expression. Therefore, we define the congruence rules (whose names start with RAC-). For example, if $e, \texttt{LPT} \twoheadrightarrow \hat{e}, \texttt{LPT}'$ then $e.f, \texttt{LPT} \twoheadrightarrow \hat{e}.f, \texttt{LPT}'$.

6.7 Proof Outline

In general, if a method $m \in CT \vdash dmethods(C)$ calls method $m' \in CT \vdash dmethods(C')$, then there are four types of call edges that AVERROES needs to handle:

- A-A: an application to application call edge, where $C \in CT_A$ and $C' \in CT_A$
- A-L: an application to library call edge, where $C \in CT_A$ and $C' \in CT_L$
- L-A: a library to application call back edge, where $C \in CT_L$ and $C' \in CT_A$
- L-L: a library to library call edge, where $C \in CT_L$ and $C' \in CT_L$

In this proof, we will inductively prove that for any trace of an FJ program, there is a corresponding trace in its FJ_{AVE} counterpart whose steps are related by the abstraction relation $\propto$. More importantly, if an A-A or L-A call expression is related to an FJ_{AVE} expression, then we will prove that the corresponding expression in the FJ_{AVE} trace is also a call with the same call target. If an A-L expression is related to an FJ_{AVE} expression, then the FJ_{AVE} expression is a call to the special expression LIB. As a result, for each call edge in the original trace, a call graph constructed for the FJ_{AVE} trace will have:

- the same A-A edges,
- an edge to LIB for each call into the library,
- an edge from LIB to the correct application method for each library callback edge, and
- no call edges between library methods (L-L).

This definition of the call graph constructed for the FJ_{AVE} trace is similar to the definition of the summary call graph defined for the separate compilation assumption in Chapter 1.

The proof is structured as follows. Lemmas 1-2 prove some properties of the partial function *mresolve*. Lemmas 3-6 prove some properties for the abstraction relation $\propto$. Lemmas 7-9 prove some properties for the reduction relation $\twoheadrightarrow$. Lemma 10 proves some properties for expressions reduced within the library points-to set. Lemma 11 is the main lemma in our proof, and proves that for any trace of an FJ program, there is a corresponding trace in its FJ_{AVE} counterpart whose steps are related by the abstraction relation $\propto$. Lemma 12 relates the initial expression of an FJ program to its FJ_{AVE} counterpart. Finally, Theorem 1 proves that any call graph that is sound for the FJ_{AVE} program soundly over-approximates the application part of the call graph of the original FJ program. This is because, for any call edge that can occur in an execution of the original FJ program, a corresponding call edge can occur in an execution of the FJ_{AVE} program.

6.8 Proof

Lemma 1 shows that if a method call resolves to an application method, then the dynamic type of the receiver is an application class. This is because a library class cannot extend an application class (Constraint 1).

Lemma 1. $\dfrac{\Sigma \vdash \mathit{mresolve}(m,\, C) \in CT_A}{C \in CT_A}$

Proof. By structural induction on the derivation of $\Sigma \vdash \mathit{mresolve}(m,\, C)$.

Case M-RES**:**

$\Sigma \vdash \mathit{mresolve}(m,\, C) = C$ ⟅M-RES⟆

$C \in CT_A$ ⟅$\Sigma \vdash \mathit{mresolve}(m,\, C) \in CT_A$⟆

Case M-RES-INH**:**

$\texttt{class}\ C\ \texttt{extends}\ D$
$\Sigma \vdash \mathit{mresolve}(m,\, C) = \Sigma \vdash \mathit{mresolve}(m,\, D)$ ⟅M-RES-INH⟆

$\Sigma \vdash \mathit{mresolve}(m,\, D) \in CT_A$ ⟅lemma precondition⟆

$D \in CT_A$ ⟅induction hypothesis⟆

$C \in CT_A$ ⟅Ave-CT⟆

□

Lemma 2 shows that if a method call resolves to method m in an application class C, then the dynamic type of the receiver is either the application class C or one of its superclasses in the transformed library CT'_L. This is because the method m can either be defined in C or inherited. If m is inherited, then it cannot be inherited from an application class, otherwise the call would have resolved to that application superclass. Therefore, it must be inherited from a library class which should be transformed by Averroes and should be present in CT'_L (Ave-Method).

Lemma 2. $$\frac{C \in CT_A \qquad CT \vdash \mathit{mresolve}(m,\, C)\ \mathit{is\ defined}}{CT \vdash \mathit{mresolve}(m,\, C) \in CT_A \quad \vee \quad CT' \vdash \mathit{mresolve}(m,\, C) \in CT'_L}$$

Proof. By structural induction on the derivation of $CT \vdash \mathit{mresolve}(m,\, C)$.

Case M-Res**:**

$CT \vdash \mathit{mresolve}(m,\, C) = C$ ⟅M-Res⟆

$CT \vdash \mathit{mresolve}(m,\, C) \in CT_A$ ⟅$C \in CT_A$⟆

Case M-Res-Inh**:**

`class` C `extends` D
$CT \vdash \mathit{mresolve}(m,\, C) = CT \vdash \mathit{mresolve}(m,\, D)$ ⟅M-Res-Inh⟆

$CT \vdash \mathit{mresolve}(m,\, D)$ is defined ⟅lemma precondition⟆

D can either be in CT_A or CT_L

If $D \in CT_A$**:**
$CT \vdash \mathit{mresolve}(m,\, D) \in CT_A \quad \vee \quad CT' \vdash \mathit{mresolve}(m,\, D) \in CT'_L$
⟅induction hypothesis⟆

$\therefore CT \vdash \mathit{mresolve}(m,\, C) \in CT_A \quad \vee \quad CT' \vdash \mathit{mresolve}(m,\, C) \in CT'_L$

If $D \in CT_L$**:**
$D \in CT'_L$ ⟅Ave-CT⟆

$CT' \vdash mresolve(m,\ D) \in CT'_L$ ⟅Ave-Method⟆

$\therefore CT' \vdash mresolve(m,\ C) \in CT'_L$

□

A method call in FJ is reduced to the body of the target method, with the receiver substituted for the `this` variable and the arguments of the call substituted for the method parameters. Lemma 3 shows that if the receiver and arguments of a method call in the original FJ program are replaced in the corresponding FJ_{Ave} program by expressions related to them by the abstraction relation $\propto$, then the resulting method bodies after the respective substitutions are also related by $\propto$.

Lemma 3. *Let e_0 be any FJ expression that type-checks in CT', let σ be the substitution $[\overline{d}/\overline{x},\ q/\texttt{this}]$, and let $\hat{\sigma}$ be the substitution $[\overline{\hat{d}}/\overline{x},\ \hat{q}/\texttt{this}]$. Then, the following statement holds:*

$$\frac{\overline{d} \propto \overline{\hat{d}}, \texttt{LPT} \qquad q \propto \hat{q}, \texttt{LPT}}{\sigma\, e_0 \propto \hat{\sigma}\, e_0, \texttt{LPT}}$$

Proof. By structural induction on the form of the expression e_0.

Case Var**:**

$e_0 = x_i$
$\therefore \sigma\, e_0 = \sigma\, x_i = d_i$ and
$\quad \hat{\sigma}\, e_0 = \hat{\sigma}\, x_i = \hat{d}_i$

$\sigma\, e_0 \propto \hat{\sigma}\, e_0, \texttt{LPT}$ ⟅$\overline{d} \propto \overline{\hat{d}}, \texttt{LPT}$⟆

Case This**:**

$e_0 = \texttt{this}$
$\therefore \sigma\, e_0 = \sigma\, \texttt{this} = q$ and
$\quad \hat{\sigma}\, e_0 = \hat{\sigma}\, \texttt{this} = \hat{q}$

$\sigma\, e_0 \propto \hat{\sigma}\, e_0, \texttt{LPT}$ ⟅$q \propto \hat{q}, \texttt{LPT}$⟆

Case FIELD**:**

$e_0 = e.f$

$\sigma\, e \propto \hat{\sigma}\, e, \texttt{LPT}$ ⟅induction hypothesis⟆

$(\sigma\, e).f \propto (\hat{\sigma}\, e).f, \texttt{LPT}$ ⟅REL-FIELD⟆

$\sigma\, (e.f) \propto \hat{\sigma}\, (e.f), \texttt{LPT}$

$\sigma\, e_0 \propto \hat{\sigma}\, e_0, \texttt{LPT}$

Case NEW**:**

$e_0 = \texttt{new}\ C(\overline{p})$

$\sigma\, \overline{p} \propto \hat{\sigma}\, \overline{p}, \texttt{LPT}$ ⟅induction hypothesis⟆

$C \in CT'$ ⟅e_0 type-checks in CT'⟆

$\texttt{new}\ C(\sigma\, \overline{p}) \propto \texttt{new}\ C(\hat{\sigma}\, \overline{p}), \texttt{LPT}$ ⟅REL-NEW⟆

$\sigma\, (\texttt{new}\ C(\overline{p})) \propto \hat{\sigma}\, (\texttt{new}\ C(\overline{p})), \texttt{LPT}$

$\sigma\, e_0 \propto \hat{\sigma}\, e_0, \texttt{LPT}$

Case INVK**:**

$e_0 = e.m(\overline{d})$

$\sigma\, e \propto \hat{\sigma}\, e, \texttt{LPT}$
$\sigma\, \overline{d} \propto \hat{\sigma}\, \overline{d}, \texttt{LPT}$ ⟅induction hypothesis⟆

$(\sigma\, e).m(\sigma\, \overline{d}) \propto (\hat{\sigma}\, e).m(\hat{\sigma}\, \overline{d}), \texttt{LPT}$ ⟅REL-INVK⟆

$\sigma\, (e.m(\overline{d})) \propto \hat{\sigma}\, (e.m(\overline{d})), \texttt{LPT}$

$\sigma\, e_0 \propto \hat{\sigma}\, e_0, \texttt{LPT}$

Case CAST**:**

$e_0 = (C)e$

$\sigma\, e \propto \hat{\sigma}\, e, \texttt{LPT}$ ⟅induction hypothesis⟆

$(C)(\sigma\, e) \propto (C)(\hat{\sigma}\, e), \texttt{LPT}$ ⟅REL-CAST⟆

$\sigma\, ((C)e) \propto \hat{\sigma}\, ((C)e), \texttt{LPT}$

$\sigma\, e_0 \propto \hat{\sigma}\, e_0, \texttt{LPT}$

□

Lemma 4 shows a similar property as shown in Lemma 3 but for expressions abstracted by the special expression LIB.

Lemma 4. *Let e_0 be any FJ expression that type-checks in CT_L, and let σ be the substitution $[\overline{d}/\overline{x},\ q/\texttt{this}]$. Then, the following statement holds:*

$$\frac{\overline{d} \propto \texttt{LIB}, \texttt{LPT} \qquad q \propto \texttt{LIB}, \texttt{LPT}}{\sigma\, e_0 \propto \texttt{LIB}, \texttt{LPT}}$$

Proof. By structural induction on the form of the expression e_0.

Case VAR**:**

$e_0 = x_i$
$\therefore \sigma\, e_0 = \sigma\, x_i = d_i$

$\sigma\, e_0 \propto \texttt{LIB}, \texttt{LPT}$ ⟅$\overline{d} \propto \texttt{LIB}, \texttt{LPT}$⟆

Case THIS**:**

$e_0 = \texttt{this}$
$\therefore \sigma\, e_0 = \sigma\, \texttt{this} = q$

$\sigma\, e_0 \propto \texttt{LIB}, \texttt{LPT}$ ⟅$q \propto \texttt{LIB}, \texttt{LPT}$⟆

Case FIELD**:**

$e_0 = e.f$

$\sigma\, e \propto \texttt{LIB}, \texttt{LPT}$ ⟅induction hypothesis⟆

$CT \vdash \mathit{declaringclass}(f) \in CT_L$ ⟅e_0 type-checks in CT_L⟆

$(\sigma\, e).f \propto \texttt{LIB}, \texttt{LPT}$ ⟅REL-LIB-FIELD⟆

$\sigma\,(e.f) \propto \texttt{LIB}, \texttt{LPT}$

$\sigma\, e_0 \propto \texttt{LIB}, \texttt{LPT}$

Case NEW**:**

$e_0 = \texttt{new}\ C(\overline{p})$

$\sigma\, \overline{p} \propto \texttt{LIB}, \texttt{LPT}$ ⟅induction hypothesis⟆

$C \in CT_L$ ⟅e_0 type-checks in CT_L⟆

$\texttt{new}\ C(\sigma\, \overline{p}) \propto \texttt{LIB}, \texttt{LPT}$ ⟅REL-LIB-NEW⟆

$\sigma\,(\texttt{new}\ C(\overline{p})) \propto \texttt{LIB}, \texttt{LPT}$

$\sigma\, e_0 \propto \texttt{LIB}, \texttt{LPT}$

Case INVK**:**

$e_0 = e.m(\overline{d})$

$\sigma\, e \propto \texttt{LIB}, \texttt{LPT}$
$\sigma\, \overline{d} \propto \texttt{LIB}, \texttt{LPT}$ ⟅induction hypothesis⟆

$\exists\, C \in CT_L\,.\ m \in CT \vdash \mathit{dmethods}(C)$ ⟅e_0 type-checks in CT_L⟆

$(\sigma\, e).m(\sigma\, \overline{d}) \propto \texttt{LIB}, \texttt{LPT}$ ⟅REL-LIB-INVK⟆

$\sigma\,(e.m(\overline{d})) \propto \texttt{LIB}, \texttt{LPT}$

$\sigma\, e_0 \propto \texttt{LIB}, \texttt{LPT}$

Case CAST**:**

$e_0 = (C)e$

$\sigma\, e \propto \texttt{LIB}, \texttt{LPT}$ ⟅induction hypothesis⟆

$(C)(\sigma\, e) \propto \texttt{LIB}, \texttt{LPT}$ ⟅REL-LIB-CAST⟆

$\sigma\, ((C)e) \propto \texttt{LIB}, \texttt{LPT}$

$\sigma\, e_0 \propto \texttt{LIB}, \texttt{LPT}$

□

Lemma 5 shows that expanding the set LPT, by adding more expressions to it, does not affect the abstraction relation $e \propto \hat{e}, \texttt{LPT}$.

Lemma 5. $\dfrac{e \propto \hat{e}, \texttt{LPT}}{e \propto \hat{e}, \texttt{LPT} \cup \overline{\hat{y}}}$

Proof. By structural induction on the derivation of $e \propto \hat{e}, \texttt{LPT}$.

Case REL-FIELD**:**

$e = q.f$
$\hat{e} = \hat{q}.f$ ⟅REL-FIELD⟆
$q \propto \hat{q}, \texttt{LPT}$

$q \propto \hat{q}, \texttt{LPT} \cup \overline{\hat{y}}$ ⟅induction hypothesis⟆

$q.f \propto \hat{q}.f, \texttt{LPT} \cup \overline{\hat{y}}$ ⟅REL-FIELD⟆

Case Rel-Lib-Field**:**

$e = q.f$
$\hat{e} = \texttt{LIB}$
$CT \vdash \mathit{declaringclass}(f) \in CT_L$ ⟅Rel-Lib-Field⟆
$q \propto \texttt{LIB}, \texttt{LPT}$

$q \propto \texttt{LIB}, \texttt{LPT} \cup \overline{\hat{y}}$ ⟅induction hypothesis⟆

$q.f \propto \texttt{LIB}, \texttt{LPT} \cup \overline{\hat{y}}$ ⟅Rel-Lib-Field⟆

Case Rel-New**:**

$e = \texttt{new}\ C(\overline{p})$
$\hat{e} = \texttt{new}\ C(\overline{\hat{p}})$
$C \in CT'$ ⟅Rel-New⟆
$\overline{p} \propto \overline{\hat{p}}, \texttt{LPT}$

$\overline{p} \propto \overline{\hat{p}}, \texttt{LPT} \cup \overline{\hat{y}}$ ⟅induction hypothesis⟆

$\texttt{new}\ C(\overline{p}) \propto \texttt{new}\ C(\overline{\hat{p}}), \texttt{LPT} \cup \overline{\hat{y}}$ ⟅Rel-New⟆

Case Rel-Lib-New**:**

$e = \texttt{new}\ C(\overline{p})$
$\hat{e} = \texttt{LIB}$
$C \in CT_L$ ⟅Rel-Lib-New⟆
$\overline{p} \propto \texttt{LIB}, \texttt{LPT}$

$\overline{p} \propto \texttt{LIB}, \texttt{LPT} \cup \overline{\hat{y}}$ ⟅induction hypothesis⟆

$\texttt{new}\ C(\overline{p}) \propto \texttt{LIB}, \texttt{LPT} \cup \overline{\hat{y}}$ ⟅Rel-Lib-New⟆

Case Rel-Invk**:**

$e = q.m(\overline{d})$
$\hat{e} = \hat{q}.m(\overline{\hat{d}})$
$q \propto \hat{q}, \texttt{LPT}$ ⟅Rel-Invk⟆
$\overline{d} \propto \overline{\hat{d}}, \texttt{LPT}$

$q \propto \hat{q}, \texttt{LPT} \cup \overline{\hat{y}}$
$\overline{d} \propto \overline{\hat{d}}, \texttt{LPT} \cup \overline{\hat{y}}$ ⟅induction hypothesis⟆

$q.m(\overline{d}) \propto \hat{q}.m(\overline{\hat{d}}), \texttt{LPT} \cup \overline{\hat{y}}$ ⟅Rel-Invk⟆

Case Rel-Lib-Invk**:**

$e = q.m(\overline{d})$
$\hat{e} = \texttt{LIB}$
$\exists C \in CT_L\,.\ m \in CT \vdash dmethods(C)$ ⟅Rel-Lib-Invk⟆
$q \propto \texttt{LIB}, \texttt{LPT}$
$\overline{d} \propto \texttt{LIB}, \texttt{LPT}$

$q \propto \texttt{LIB}, \texttt{LPT} \cup \overline{\hat{y}}$
$\overline{d} \propto \texttt{LIB}, \texttt{LPT} \cup \overline{\hat{y}}$ ⟅induction hypothesis⟆

$q.m(\overline{d}) \propto \texttt{LIB}, \texttt{LPT} \cup \overline{\hat{y}}$ ⟅Rel-Lib-Invk⟆

Case Rel-Cast**:**

$e = (C)q$
$\hat{e} = (C)\hat{q}$ ⟅Rel-Cast⟆
$q \propto \hat{q}, \texttt{LPT}$

$q \propto \hat{q}, \texttt{LPT} \cup \overline{\hat{y}}$ ⟅induction hypothesis⟆

$(C)q \propto (C)\hat{q}, \texttt{LPT} \cup \overline{\hat{y}}$ ⟅Rel-Cast⟆

Case Rel-Lib-Cast**:**

$e = (C)q$
$\hat{e} = \texttt{LIB}$ ⟅Rel-Lib-Cast⟆
$q \propto \texttt{LIB}, \texttt{LPT}$

$q \propto \texttt{LIB}, \texttt{LPT} \cup \overline{\hat{y}}$ ⟅induction hypothesis⟆

$(C)q \propto \texttt{LIB}, \texttt{LPT} \cup \overline{\hat{y}}$ ⟅Rel-Lib-Cast⟆

Case REL-LPT**:**

$\hat{e} = \texttt{LIB}$
$\exists\, \hat{q} \in \texttt{LPT}\,.\ e \propto \hat{q}, \texttt{LPT}$ ⟅REL-LPT⟆

$e \propto \hat{q}, \texttt{LPT} \cup \overline{\hat{y}}$ ⟅induction hypothesis⟆

$\hat{q} \in \texttt{LPT} \cup \overline{\hat{y}}$ ⟅$\hat{q} \in \texttt{LPT}$⟆

$e \propto \texttt{LIB}, \texttt{LPT} \cup \overline{\hat{y}}$ ⟅REL-LPT⟆

□

The result of applying the the reduction rule $\twoheadrightarrow$ along an FJ_{AVE} trace should only refer to fields, classes, and methods declared in classes in CT'. Definition 1 formalizes this notion.

Definition 1. *Let $\mathcal{P}(\hat{e}, \texttt{LPT})$ be true if all of the fields, classes, and methods referenced in $\hat{e}$ and in all of the expressions in* LPT *are declared in classes in CT'.*

Lemma 6 shows that if an expression e is abstracted by an expression $\hat{e}$ that flows into the library (i.e., $\hat{e} \in \texttt{LPT}$), then the library (i.e., the special expression LIB) can also abstract the expression e. This lemma factors out the proof of the additional subcase for the rule REL-LPT in all the cases handled by Lemma 11.

Lemma 6. *If $e \propto \hat{e}, \texttt{LPT}$ and $\mathcal{P}(\hat{e}, \texttt{LPT})$ holds, then there exists an expression $\hat{y}$ such that $\mathcal{P}(\hat{y}, \texttt{LPT})$ holds, $e \propto \hat{y}, \texttt{LPT}$ is derived by a rule other than* REL-LPT*, and $\hat{e}, \texttt{LPT} \twoheadrightarrow^* \hat{y}, \texttt{LPT}$.*

Proof. By structural induction on the derivation of $e \propto \hat{e}, \texttt{LPT}$.

Case $e \propto \hat{e}, \texttt{LPT}$ **is derived by** REL-LPT**:**

$\hat{e} = \texttt{LIB}$
$\exists\, \hat{q} \in \texttt{LPT}\,.\ e \propto \hat{q}, \texttt{LPT}$ ⟅REL-LPT⟆

$\mathcal{P}(\hat{q}, \texttt{LPT})$ holds ⟅$\hat{q} \in \texttt{LPT}$⟆

There exists an expression $\hat{d}$ such that
$\mathcal{P}(\hat{d}, \texttt{LPT})$ holds
$e \propto \hat{d}, \texttt{LPT}$ is derived by a rule other than REL-LPT and ⟅induction hypothesis⟆
$\hat{q}, \texttt{LPT} \twoheadrightarrow^* \hat{d}, \texttt{LPT}$

$\texttt{LIB}, \texttt{LPT} \twoheadrightarrow \hat{q}, \texttt{LPT}$ ⟅RA-Return⟆
$\twoheadrightarrow^* \hat{d}, \texttt{LPT}$

Letting $\hat{y} = \hat{d}$ satisfies the conclusions of the lemma.

Case $e \propto \hat{e}, \texttt{LPT}$ is derived by a rule other than Rel-Lpt:

Letting $\hat{y} = \hat{e}$ satisfies the conclusions of the lemma.

□

Lemma 7 shows that expanding the set LPT, by adding more expressions to it, also preserves the reduction relation $\hat{e}, \texttt{LPT} \twoheadrightarrow \hat{e}', \texttt{LPT}'$.

Lemma 7. $\dfrac{\hat{e}, \texttt{LPT} \twoheadrightarrow \hat{e}', \texttt{LPT}'}{\hat{e}, \texttt{LPT} \cup \overline{\hat{y}} \twoheadrightarrow \hat{e}', \texttt{LPT}' \cup \overline{\hat{y}}}$

Proof. By trivial structural induction on the derivation of $\hat{e}, \texttt{LPT} \twoheadrightarrow \hat{e}', \texttt{LPT}'$.

□

Lemma 8 shows that the reduction relation $\hat{e}, \texttt{LPT} \twoheadrightarrow \hat{e}', \texttt{LPT}'$ preserves the expressions in the set LPT, and possibly adds new expressions to it.

Lemma 8. $\dfrac{\hat{e}, \texttt{LPT} \twoheadrightarrow \hat{e}', \texttt{LPT}'}{\texttt{LPT} \subseteq \texttt{LPT}'}$

Proof. By trivial structural induction on the derivation of $\hat{e}, \texttt{LPT} \twoheadrightarrow \hat{e}', \texttt{LPT}'$.

□

An FJ expression e_0 could be illegal in CT' because it references fields, classes or methods that are not in the FJ_{Ave} transformation of the given FJ program. Such an expression e_0 is abstracted by the special expression LIB ($e_0 \propto \texttt{LIB}, \texttt{LPT}_0$). If a sub-expression e_n of the expression e_0 is reduced, then both e_0 and e_n should still be abstracted by the special expression LIB.

Lemma 9 shows that if an expression e_0 is reduced to an expression e_n outside the library points-to set ($e_0, \texttt{LPT}_0 \twoheadrightarrow^* e_n, \texttt{LPT}_n$), then if e_0 is in some $\texttt{LPT}'_0$, then it can also reduce to e_n inside the library points-to set ($\texttt{LIB}, \texttt{LPT}'_0 \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}'_n$ and $e_n \in \texttt{LPT}'_n$).

Lemma 9. *Let e_0 be an expression that is not* LIB*, such that $e_0 \in \texttt{LPT}_0$. Then, if*

$$e_0, \texttt{LPT}_0 \twoheadrightarrow e_1, \texttt{LPT}_1 \twoheadrightarrow \ldots \twoheadrightarrow e_n, \texttt{LPT}_n$$

then

$$\texttt{LIB}, \texttt{LPT}'_0 \twoheadrightarrow \texttt{LIB}, \texttt{LPT}'_1 \twoheadrightarrow \ldots \twoheadrightarrow \texttt{LIB}, \texttt{LPT}'_n$$

where

$$\texttt{LPT}'_i = \texttt{LPT}_i \cup \{e_k : k \leq i\}$$

Proof. By induction on the number of steps n taken to derive $e_0, \texttt{LPT}_0 \twoheadrightarrow^* e_n, \texttt{LPT}_n$.

Case $n = 0$:

$e_n = e_0$
$\texttt{LPT}_n = \texttt{LPT}_0$
$\texttt{LPT}'_n = \texttt{LPT}'_0$

Then it follows directly that
$\texttt{LIB}, \texttt{LPT}'_0 \twoheadrightarrow \texttt{LIB}, \texttt{LPT}'_1 \twoheadrightarrow \ldots \twoheadrightarrow \texttt{LIB}, \texttt{LPT}'_n$

Case $n > 0$:

$e_0 \in \texttt{LPT}_0$
$e_0, \texttt{LPT}_0 \twoheadrightarrow e_1, \texttt{LPT}_1 \twoheadrightarrow \ldots \twoheadrightarrow e_i, \texttt{LPT}_i \twoheadrightarrow e_{i+1}, \texttt{LPT}_{i+1}$ — ⟅lemma precondition⟆

$\texttt{LPT}_0 \subseteq \texttt{LPT}_1 \subseteq \ldots \subseteq \texttt{LPT}_i \subseteq \texttt{LPT}_{i+1}$ — ⟅Lemma 8⟆

$\forall_i\, e_i \in \texttt{LPT}'_i$
$\texttt{LIB}, \texttt{LPT}'_0 \twoheadrightarrow \texttt{LIB}, \texttt{LPT}'_1 \twoheadrightarrow \ldots \twoheadrightarrow \texttt{LIB}, \texttt{LPT}'_i$ — ⟅induction hypothesis⟆

$e_i, \texttt{LPT}_i \cup \texttt{LPT}'_i \twoheadrightarrow e_{i+1}, \texttt{LPT}_{i+1} \cup \texttt{LPT}'_i$ — ⟅Lemma 7⟆
$e_i, \texttt{LPT}'_i \twoheadrightarrow e_{i+1}, \texttt{LPT}_{i+1} \cup \texttt{LPT}'_i$ — ⟅$\texttt{LPT}_i \subseteq \texttt{LPT}'_i$⟆

$\texttt{LIB}, \texttt{LPT}'_i \twoheadrightarrow \texttt{LIB}, \texttt{LPT}'_i \cup \texttt{LPT}_{i+1} \cup \{e_{i+1}\}$ — ⟅RA-Sub⟆
$= \texttt{LIB}, \texttt{LPT}'_{i+1}$ — ⟅$\texttt{LPT}_i \subseteq \texttt{LPT}_{i+1}$ and by the definition of $\texttt{LPT}'_{i+1}$⟆

$\therefore \texttt{LIB}, \texttt{LPT}'_0 \twoheadrightarrow \texttt{LIB}, \texttt{LPT}'_1 \twoheadrightarrow \ldots \twoheadrightarrow \texttt{LIB}, \texttt{LPT}'_{i+1}$

□

Lemma 10 shows that if an expression e is returned from the library points-to set in a reduction sequence $(\texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* e, \texttt{LPT}')$, then there also exists a reduction sequence starting at $\texttt{LIB}, \texttt{LPT}$ in which e is kept within the library points-to set ($\texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}''$ and $e \in \texttt{LPT}''$).

Lemma 10. *If*

$$\begin{array}{ccc} e_0 & \propto & \texttt{LIB}, \texttt{LPT} \\ {\scriptstyle CT}\downarrow & & \downarrow{\scriptstyle *} \\ e_0' & \propto & \hat{e}_0', \texttt{LPT}' \end{array}$$

then there exists $\texttt{LPT}''$ *such that*

$$\begin{array}{ccc} e_0 & \propto & \texttt{LIB}, \texttt{LPT} \\ {\scriptstyle CT}\downarrow & & \downarrow{\scriptstyle *} \\ e_0' & \propto & \texttt{LIB}, \texttt{LPT}'' \end{array}$$

Proof. By case analysis on the form of $\hat{e}_0'$.

Case $\hat{e}_0' = \texttt{LIB}$**:**

Let $\texttt{LPT}'' = \texttt{LPT}'$. Then, it follows directly that
${e_0}' \propto \texttt{LIB}, \texttt{LPT}'$ and
$\texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}'$

Case $\hat{e}_0' \neq \texttt{LIB}$**:**

Since $\hat{e}_0' \neq \texttt{LIB}$, there must be at least one intermediate reduction step to reduce $\texttt{LIB}, \texttt{LPT}$ to $\hat{e}_0', \texttt{LPT}'$. Therefore, there exists a set $\texttt{LPT}_0$, such that $\texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* e, \texttt{LPT}_0 \twoheadrightarrow^* \hat{e}_0', \texttt{LPT}'$.

The only rule that reduces LIB to an expression e that is not LIB is RA-Return.

There exists a set $\texttt{LPT}'''$ such that
$\texttt{LIB}, \texttt{LPT}_0 \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}'''$
$\texttt{LPT}' \subseteq \texttt{LPT}'''$ ⟅Lemma 9⟆
$\hat{e}_0' \in \texttt{LPT}'''$

$e_0{}' \propto \hat{e}'_0, \texttt{LPT}'''$ ⟅lemma precondition and Lemma 5⟆

$e_0{}' \propto \texttt{LIB}, \texttt{LPT}'''$ ⟅Rel-Lpt⟆

Letting $\texttt{LPT}'' = \texttt{LPT}'''$ satisfies the conclusions of the lemma.

□

Lemma 11 shows that for any trace of an FJ program, there is a corresponding trace in its FJ_{Ave} counterpart whose reduction steps are related by the abstraction relation $\propto$.

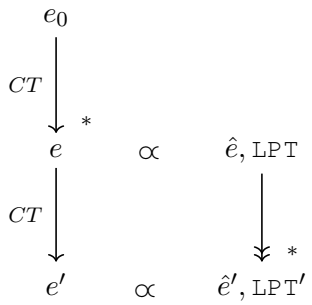
Lemma 11. *Let e_0 be an initial FJ program that type-checks in CT. If*

- $e_0 \xrightarrow{CT}{}^* e \xrightarrow{CT} e'$
- $e \propto \hat{e}, \texttt{LPT}$ *and*
- $\mathcal{P}(\hat{e}, \texttt{LPT})$ *holds*

then

- *e type-checks in CT (by the type preservation proof for FJ [30])*
- *there exists an expression $\hat{e}'$ and a set $\texttt{LPT}'$ such that $\hat{e}, \texttt{LPT} \twoheadrightarrow^* \hat{e}', \texttt{LPT}'$ and $e' \propto \hat{e}', \texttt{LPT}'$ and*
- *$\mathcal{P}(\hat{e}', \texttt{LPT}')$ holds*

This is summarized in the following diagram



Proof. By Lemma 6, there exists an expression $\hat{y}$ such that $e \propto \hat{y}, \texttt{LPT}$ is derived by a rule other than Rel-Lpt. We will show that $\hat{y}, \texttt{LPT} \twoheadrightarrow^* \hat{e}', \texttt{LPT}'$ which implies that $\hat{e}, \texttt{LPT} \twoheadrightarrow^* \hat{e}', \texttt{LPT}'$. We will prove this lemma by structural induction on the derivation of the reduction $e \xrightarrow{CT} e'$.

Case R-FIELD**:**

$e = (\texttt{new}\ C(\overline{p})).f_i$
$e' = p_i$ ⟅$e \xrightarrow{CT} e'$ by R-FIELD⟆
$\Sigma \vdash \mathit{fields}(C) = \overline{C}\ \overline{f}$

From the form of e, the fact $e \propto \hat{y}, \texttt{LPT}$ could be derived by either REL-FIELD or REL-LIB-FIELD.

Subcase REL-FIELD**:**

$\hat{y} = \hat{q}.f_i$
$\texttt{new}\ C(\overline{p}) \propto \hat{q}, \texttt{LPT}$ ⟅REL-FIELD⟆

C can either be in CT_A or CT_L ⟅e type-checks in CT⟆

If $C \in CT_A$**:**
$\exists\, \hat{z}\,.\ \hat{q}, \texttt{LPT} \twoheadrightarrow^* \hat{z}, \texttt{LPT}$ and
$\texttt{new}\ C(\overline{p}) \propto \hat{z}, \texttt{LPT}$ is derived by REL-NEW ⟅Lemma 6⟆

$\hat{z} = \texttt{new}\ C(\overline{\hat{p}})$
$\overline{p} \propto \overline{\hat{p}}, \texttt{LPT}$ ⟅REL-NEW⟆

$\hat{q}.f_i, \texttt{LPT} \twoheadrightarrow^* (\texttt{new}\ C(\overline{\hat{p}})).f_i, \texttt{LPT}$ ⟅RAC-FIELD⟆
$\twoheadrightarrow \hat{p}_i, \texttt{LPT}$ ⟅R-FIELD and RA-FJ⟆

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \hat{p}_i, \texttt{LPT}$

Letting $\hat{e}' = \hat{p}_i$ and $\texttt{LPT}' = \texttt{LPT}$ satisfies the conclusions of the lemma.

If $C \in CT_L$**:**
$\exists\, \hat{z}\,.\ \hat{q}, \texttt{LPT} \twoheadrightarrow^* \hat{z}, \texttt{LPT}$ and
$\texttt{new}\ C(\overline{p}) \propto \hat{z}, \texttt{LPT}$ is derived by REL-LIB-NEW ⟅Lemma 6⟆
$\mathcal{P}(\hat{y}, \texttt{LPT})$ holds

$\hat{z} = \texttt{LIB}$
$\overline{p} \propto \texttt{LIB}, \texttt{LPT}$ ⟅REL-LIB-NEW⟆

Let $\hat{C} = CT' \vdash \mathit{declaringclass}(f_i)$. Then,
$\hat{C} \in CT_L$ ⟅AVE-FIELD⟆
$\hat{C} \in CT'_L$ ⟅$\mathcal{P}(\hat{y}, \texttt{LPT})$ holds⟆

$\hat{q}.f_i, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}.f_i, \texttt{LPT}$ ⟅RAC-Field⟆
$\twoheadrightarrow \texttt{LIB}.f_i, \texttt{LPT} \cup \{\texttt{new}\ \hat{C}(\overline{\texttt{LIB}})\}$ ⟅RA-New⟆
$\twoheadrightarrow (\texttt{new}\ \hat{C}(\overline{\texttt{LIB}})).f_i, \texttt{LPT} \cup \{\texttt{new}\ \hat{C}(\overline{\texttt{LIB}})\}$ ⟅RA-Return and RAC-Field⟆
$\twoheadrightarrow \texttt{LIB}, \texttt{LPT} \cup \{\texttt{new}\ \hat{C}(\overline{\texttt{LIB}})\}$ ⟅R-Field and RA-FJ⟆

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT} \cup \{\texttt{new}\ \hat{C}(\overline{\texttt{LIB}})\}$

$\overline{p} \propto \texttt{LIB}, \texttt{LPT} \cup \{\texttt{new}\ \hat{C}(\overline{\texttt{LIB}})\}$ ⟅Lemma 5⟆

Letting $\hat{e}' = \texttt{LIB}$ and $\texttt{LPT}' = \texttt{LPT} \cup \{\texttt{new}\ \hat{C}(\overline{\texttt{LIB}})\}$ satisfies the conclusions of the lemma.

Subcase Rel-Lib-Field**:**

$\hat{y} = \texttt{LIB}$
$\texttt{new}\ C(\overline{p}) \propto \texttt{LIB}, \texttt{LPT}$ ⟅Rel-Lib-Field⟆
$CT \vdash \mathit{declaringclass}(f_i) \in CT_L$

C can either be in CT_A or CT_L ⟅e type-checks in CT⟆

If $C \in CT_A$**:**
$\exists\, \hat{z}\,.\ \texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* \hat{z}, \texttt{LPT}$ and
$\texttt{new}\ C(\overline{p}) \propto \hat{z}, \texttt{LPT}$ is derived by Rel-New ⟅Lemma 6⟆

$\hat{z} = \texttt{new}\ C(\overline{\hat{p}})$
$\overline{p} \propto \overline{\hat{p}}, \texttt{LPT}$ ⟅Rel-New⟆

$CT' \vdash \mathit{declaringclass}(f_i) \in CT'_L$ ⟅Ave-Field⟆

$\texttt{LIB}, \texttt{LPT} \twoheadrightarrow \texttt{LIB}, \texttt{LPT} \cup \{\texttt{LIB}.f_i\}$ ⟅RA-Field⟆
$\twoheadrightarrow \texttt{LIB}.f_i, \texttt{LPT} \cup \{\texttt{LIB}.f_i\}$ ⟅RA-Return⟆
$\twoheadrightarrow^* (\texttt{new}\ C(\overline{\hat{p}})).f_i, \texttt{LPT} \cup \{\texttt{LIB}.f_i\}$ ⟅RAC-Field⟆
$\twoheadrightarrow \hat{p}_i, \texttt{LPT} \cup \{\texttt{LIB}.f_i\}$ ⟅R-Field and RA-FJ⟆

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \hat{p}_i, \texttt{LPT} \cup \{\texttt{LIB}.f_i\}$

$\overline{p} \propto \overline{\hat{p}}, \texttt{LPT} \cup \{\texttt{LIB}.f_i\}$ ⟅Lemma 5⟆

Letting $\hat{e}' = \hat{p}_i$ and $\texttt{LPT}' = \texttt{LPT} \cup \{\texttt{LIB}.f_i\}$ satisfies the conclusions of the lemma.

If $C \in CT_L$**:**

$\exists\, \hat{z}\,.\ \texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* \hat{z}, \texttt{LPT}$ and
$\texttt{new}\ C(\overline{p}) \propto \hat{z}, \texttt{LPT}$ is derived by Rel-Lib-New — ⟅Lemma 6⟆

$\hat{z} = \texttt{LIB}$
$\overline{p} \propto \texttt{LIB}, \texttt{LPT}$ — ⟅Rel-Lib-New⟆

Letting $\hat{e}' = \texttt{LIB}$ and $\texttt{LPT}' = \texttt{LPT}$ satisfies the conclusions of the lemma.

Case R-Invk**:**

$e = (\texttt{new}\ C(\overline{p})).m(\overline{d})$ and $e' = \sigma\, e_0$
where $\sigma = [\overline{d}/\overline{x},\ \texttt{new}\ C(\overline{p})/\texttt{this}]$
$CT \vdash \mathit{mbody}(m, C) = \overline{x}.e_0$ — ⟅$e \xrightarrow{CT} e'$ by R-Invk⟆

From the form of e, the fact $e \propto \hat{y}, \texttt{LPT}$ could be derived by either Rel-Invk or Rel-Lib-Invk. Additionally, the function $CT \vdash \mathit{mresolve}(m, C)$ yields a class either in CT_A or CT_L.

Subcase Rel-Invk **and** $CT \vdash \mathit{mresolve}(m, C) \in CT_A$**:**

$\hat{y} = \hat{q}.m(\overline{\hat{d}})$
$\texttt{new}\ C(\overline{p}) \propto \hat{q}, \texttt{LPT}$
$\overline{d} \propto \overline{\hat{d}}, \texttt{LPT}$ — ⟅Rel-Invk⟆

$C \in CT_A$ — ⟅Lemma 1⟆

$\exists\, \hat{z}\,.\ \hat{q}, \texttt{LPT} \twoheadrightarrow^* \hat{z}, \texttt{LPT}$ and
$\texttt{new}\ C(\overline{p}) \propto \hat{z}, \texttt{LPT}$ is derived by Rel-New — ⟅Lemma 6⟆

$\hat{z} = \texttt{new}\ C(\overline{\hat{p}})$ — ⟅Rel-New⟆

$\hat{q}.m(\overline{\hat{d}}), \texttt{LPT} \twoheadrightarrow^* (\texttt{new}\ C(\overline{\hat{p}})).m(\overline{\hat{d}}), \texttt{LPT}$ — ⟅RAC-Invk-Recv⟆
$\twoheadrightarrow \hat{\sigma}\, e_0, \texttt{LPT}$ — ⟅R-Invk and RA-FJ⟆
where $\hat{\sigma} = [\overline{\hat{d}}/\overline{x},\ \texttt{new}\ C(\overline{\hat{p}})/\texttt{this}]$

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \hat{\sigma}\, e_0, \texttt{LPT}$

$\sigma\, e_0 \propto \hat{\sigma}\, e_0, \texttt{LPT}$ — ⟅Lemma 3⟆

Letting $\hat{e}' = \hat{\sigma}\, e_0$ and $\texttt{LPT}' = \texttt{LPT}$ satisfies the conclusions of the lemma.

Subcase Rel-Invk **and** $CT \vdash \mathit{mresolve}(m,\, C) \in CT_L$**:**

$\hat{y} = \hat{q}.m(\overline{\hat{d}})$
$\texttt{new}\ C(\overline{p}) \propto \hat{q}, \texttt{LPT}$ ⟅Rel-Invk⟆
$\overline{d} \propto \overline{\hat{d}}, \texttt{LPT}$

C can either be in CT_A or CT_L ⟅e type-checks in CT⟆

If $C \in CT_A$**:**
$\exists\, \hat{z}\,.\ \hat{q}, \texttt{LPT} \twoheadrightarrow^* \hat{z}, \texttt{LPT}$ and
$\texttt{new}\ C(\overline{p}) \propto \hat{z}, \texttt{LPT}$ is derived by Rel-New ⟅Lemma 6⟆
$\mathcal{P}(\hat{z}, \texttt{LPT})$ holds

$\hat{z} = \texttt{new}\ C(\overline{\hat{p}})$ ⟅Rel-New⟆

$CT' \vdash \mathit{mresolve}(m,\, C) \in CT'_L$ ⟅Lemma 2⟆

$\hat{q}.m(\overline{\hat{d}}), \texttt{LPT} \twoheadrightarrow^* (\texttt{new}\ C(\overline{\hat{p}})).m(\overline{\hat{d}}), \texttt{LPT}$ ⟅RAC-Invk-Recv⟆
$\twoheadrightarrow \texttt{LIB}, \texttt{LPT} \cup \{\texttt{new}\ C(\overline{\hat{p}})\} \cup \overline{\hat{d}}$ ⟅RA-Lib-Invk⟆

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT} \cup \{\texttt{new}\ C(\overline{\hat{p}})\} \cup \overline{\hat{d}}$

$\texttt{new}\ C(\overline{p}) \propto \texttt{new}\ C(\overline{\hat{p}}), \texttt{LPT} \cup \{\texttt{new}\ C(\overline{\hat{p}})\} \cup \overline{\hat{d}}$ ⟅Lemma 5⟆
$\texttt{new}\ C(\overline{p}) \propto \texttt{LIB}, \texttt{LPT} \cup \{\texttt{new}\ C(\overline{\hat{p}})\} \cup \overline{\hat{d}}$ ⟅Rel-Lpt⟆

$\overline{d} \propto \overline{\hat{d}}, \texttt{LPT} \cup \{\texttt{new}\ C(\overline{\hat{p}})\} \cup \overline{\hat{d}}$ ⟅Lemma 5⟆
$\overline{d} \propto \texttt{LIB}, \texttt{LPT} \cup \{\texttt{new}\ C(\overline{\hat{p}})\} \cup \overline{\hat{d}}$ ⟅Rel-Lpt⟆

$\sigma\, e_0 \propto \texttt{LIB}, \texttt{LPT} \cup \{\texttt{new}\ C(\overline{\hat{p}})\} \cup \overline{\hat{d}}$ ⟅Lemma 4⟆

Letting $\hat{e}' = \texttt{LIB}$ and $\texttt{LPT}' = \texttt{LPT} \cup \{\texttt{new}\ C(\overline{\hat{p}})\} \cup \overline{\hat{d}}$
satisfies the conclusions of the lemma.

If $C \in CT_L$**:**

$\exists\, \hat{z}\,.\ \hat{q}, \texttt{LPT} \twoheadrightarrow^* \hat{z}, \texttt{LPT}$ and
$\texttt{new}\ C(\overline{p}) \propto \hat{z}, \texttt{LPT}$ is derived by Rel-Lib-New — ⟅Lemma 6⟆

$\hat{z} = \texttt{LIB}$ — ⟅Rel-Lib-New⟆

Let $m \in CT' \vdash dmethods(\hat{C})$
$\hat{C} \in CT_L$ — ⟅Ave-Method⟆
$\hat{C} \in CT'_L$ — ⟅lemma precondition on $\hat{y}$⟆

$\therefore CT' \vdash mresolve(m,\, \hat{C}) \in CT'_L$

$\hat{q}.m(\overline{d}), \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}.m(\overline{\hat{d}}), \texttt{LPT}$ — ⟅RAC-Invk-Recv⟆
$\twoheadrightarrow \texttt{LIB}.m(\overline{\hat{d}}), \texttt{LPT} \cup \{\texttt{new}\ \hat{C}(\overline{\texttt{LIB}})\}$ — ⟅RA-New⟆
$\twoheadrightarrow (\texttt{new}\ \hat{C}(\overline{\texttt{LIB}})).m(\overline{\hat{d}}), \texttt{LPT} \cup \{\texttt{new}\ \hat{C}(\overline{\texttt{LIB}})\}$ — ⟅RA-Return and RAC-Invk-Recv⟆
$\twoheadrightarrow \texttt{LIB}, \texttt{LPT} \cup \{\texttt{new}\ \hat{C}(\overline{\texttt{LIB}})\} \cup \overline{\hat{d}}$ — ⟅RA-Lib-Invk⟆

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT} \cup \{\texttt{new}\ \hat{C}(\overline{\texttt{LIB}})\} \cup \overline{\hat{d}}$

$\texttt{new}\ C(\overline{p}) \propto \texttt{LIB}, \texttt{LPT} \cup \{\texttt{new}\ \hat{C}(\overline{\texttt{LIB}})\} \cup \overline{\hat{d}}$ — ⟅Lemma 5⟆

$\overline{d} \propto \overline{\hat{d}}, \texttt{LPT} \cup \{\texttt{new}\ \hat{C}(\overline{\texttt{LIB}})\} \cup \overline{\hat{d}}$ — ⟅Lemma 5⟆
$\overline{d} \propto \texttt{LIB}, \texttt{LPT} \cup \{\texttt{new}\ \hat{C}(\overline{\texttt{LIB}})\} \cup \overline{\hat{d}}$ — ⟅Rel-Lpt⟆

$\sigma\, e_0 \propto \texttt{LIB}, \texttt{LPT} \cup \{\texttt{new}\ \hat{C}(\overline{\texttt{LIB}})\} \cup \overline{\hat{d}}$ — ⟅Lemma 4⟆

Letting $\hat{e}' = \texttt{LIB}$ and $\texttt{LPT}' = \texttt{LPT} \cup \{\texttt{new}\ \hat{C}(\overline{\texttt{LIB}})\} \cup \overline{\hat{d}}$
satisfies the conclusions of the lemma.

Subcase Rel-Lib-Invk **and** $CT \vdash mresolve(m,\, C) \in CT_A$**:**

$\hat{y} = \texttt{LIB}$
$\texttt{new}\ C(\overline{p}) \propto \texttt{LIB}, \texttt{LPT}$ — ⟅Rel-Lib-Invk⟆
$\overline{d} \propto \texttt{LIB}, \texttt{LPT}$

$C \in CT_A$ — ⟅Lemma 1⟆

$\exists\, \hat{z}\,.\ \texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* \hat{z}, \texttt{LPT}$ and
$\texttt{new}\ C(\overline{p}) \propto \hat{z}, \texttt{LPT}$ is derived by Rel-New
⟅Lemma 6⟆

$\hat{z} = \texttt{new}\ C(\overline{\hat{p}})$
⟅Rel-New⟆

$\exists\, \hat{C} \in CT'_L\,.\ m \in CT' \vdash \mathit{dmethods}(\hat{C})$
⟅Ave-Method⟆

$\begin{array}{ll}
\texttt{LIB}, \texttt{LPT} \twoheadrightarrow \texttt{LIB}, \texttt{LPT} \cup \{\texttt{LIB}.m(\overline{\texttt{LIB}})\} & \text{⟅RA-Invk⟆} \\
\quad \twoheadrightarrow \texttt{LIB}.m(\overline{\texttt{LIB}}), \texttt{LPT} \cup \{\texttt{LIB}.m(\overline{\texttt{LIB}})\} & \text{⟅RA-Return⟆} \\
\quad \twoheadrightarrow (\texttt{new}\ C(\overline{\hat{p}})).m(\overline{\texttt{LIB}}), \texttt{LPT} \cup \{\texttt{LIB}.m(\overline{\texttt{LIB}})\} & \text{⟅RAC-Invk-Recv⟆} \\
\quad \twoheadrightarrow \hat{\sigma}\, e_0, \texttt{LPT} \cup \{\texttt{LIB}.m(\overline{\texttt{LIB}})\} & \text{⟅R-Invk and RA-FJ⟆} \\
\quad \text{where } \hat{\sigma} = [\overline{\texttt{LIB}}/\overline{x},\ \texttt{new}\ C(\overline{\hat{p}})/\texttt{this}]
\end{array}$

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \hat{\sigma}\, e_0, \texttt{LPT} \cup \{\texttt{LIB}.m(\overline{\texttt{LIB}})\}$

$\overline{d} \propto \texttt{LIB}, \texttt{LPT} \cup \{\texttt{LIB}.m(\overline{\texttt{LIB}})\}$
⟅Lemma 5⟆

$\texttt{new}\ C(\overline{p}) \propto \texttt{new}\ C(\overline{\hat{p}}), \texttt{LPT} \cup \{\texttt{LIB}.m(\overline{\texttt{LIB}})\}$
⟅Lemma 5⟆

$\sigma\, e_0 \propto \hat{\sigma}\, e_0, \texttt{LPT} \cup \{\texttt{LIB}.m(\overline{\texttt{LIB}})\}$
⟅Lemma 3⟆

Letting $\hat{e}' = \hat{\sigma}\, e_0$ and $\texttt{LPT}' = \texttt{LPT} \cup \{\texttt{LIB}.m(\overline{\texttt{LIB}})\}$
satisfies the conclusions of the lemma.

Subcase Rel-Lib-Invk **and** $CT \vdash \mathit{mresolve}(m,\, C) \in CT_L$**:**

$\hat{y} = \texttt{LIB}$
$\texttt{new}\ C(\overline{p}) \propto \texttt{LIB}, \texttt{LPT}$
$\overline{d} \propto \texttt{LIB}, \texttt{LPT}$
⟅Rel-Lib-Invk⟆

$\sigma\, e_0 \propto \texttt{LIB}, \texttt{LPT}$
⟅Lemma 4⟆

Letting $\hat{e}' = \texttt{LIB}$ and $\texttt{LPT}' = \texttt{LPT}$ satisfies the conclusions of the lemma.

Case R-CAST**:**

$e = (D)(\texttt{new}\ C(\overline{p}))$
$e' = \texttt{new}\ C(\overline{p})$ ⟅$e \xrightarrow{CT} e'$ by R-CAST⟆
$C <: D$

From the form of e, the fact $e \propto \hat{y}, \texttt{LPT}$ could be derived by either REL-CAST or REL-LIB-CAST.

Subcase REL-CAST**:**

$\hat{y} = (D)\hat{q}$
$\texttt{new}\ C(\overline{p}) \propto \hat{q}, \texttt{LPT}$ ⟅REL-CAST⟆

C can either be in CT_A or CT_L ⟅e type-checks in CT⟆

If $C \in CT_A$**:**

$\exists\, \hat{z}\,.\ \hat{q}, \texttt{LPT} \twoheadrightarrow^* \hat{z}, \texttt{LPT}$ and
$\texttt{new}\ C(\overline{p}) \propto \hat{z}, \texttt{LPT}$ is derived by REL-NEW ⟅Lemma 6⟆

$\hat{z} = \texttt{new}\ C(\overline{\hat{p}})$ ⟅REL-NEW⟆

$(D)\hat{q}, \texttt{LPT} \twoheadrightarrow^* (D)(\texttt{new}\ C(\overline{\hat{p}})), \texttt{LPT}$ ⟅RAC-CAST⟆
$\twoheadrightarrow \texttt{new}\ C(\overline{\hat{p}}), \texttt{LPT}$ ⟅R-CAST and RA-FJ⟆

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \texttt{new}\ C(\overline{\hat{p}}), \texttt{LPT}$

Letting $\hat{e}' = \texttt{new}\ C(\overline{\hat{p}})$ and $\texttt{LPT}' = \texttt{LPT}$ satisfies the conclusions of the lemma.

If $C \in CT_L$**:**

$\exists\, \hat{z}\,.\ \hat{q}, \texttt{LPT} \twoheadrightarrow^* \hat{z}, \texttt{LPT}$ and
$\texttt{new}\ C(\overline{p}) \propto \hat{z}, \texttt{LPT}$ is derived by REL-LIB-NEW ⟅Lemma 6⟆

$\hat{z} = \texttt{LIB}$ ⟅REL-LIB-NEW⟆

$(D)\hat{q}, \texttt{LPT} \twoheadrightarrow^* (D)\texttt{LIB}, \texttt{LPT}$ ⟅RAC-CAST⟆
$\twoheadrightarrow \texttt{LIB}, \texttt{LPT}$ ⟅RA-CAST⟆

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}$

Letting $\hat{e}' = \texttt{LIB}$ and $\texttt{LPT}' = \texttt{LPT}$ satisfies the conclusions of the lemma.

Subcase Rel-Lib-Cast**:**

$\hat{y} = \texttt{LIB}$
$\texttt{new}\ C(\overline{p}) \propto \texttt{LIB}, \texttt{LPT}$ ⟅Rel-Lib-Cast⟆

Letting $\hat{e}' = \texttt{LIB}$ and $\texttt{LPT}' = \texttt{LPT}$ satisfies the conclusions of the lemma.

Case RC-Field**:**

$e = e_0.f$
$e' = e_0'.f$
$e_0 \xrightarrow{CT} e_0'$ ⟅$e \xrightarrow{CT} e'$ by RC-Field⟆

From the form of e, the fact $e \propto \hat{y}, \texttt{LPT}$ could be derived by either Rel-Field or Rel-Lib-Field.

Subcase Rel-Field**:**

$\hat{y} = \hat{e}_0.f$
$e_0 \propto \hat{e}_0, \texttt{LPT}$ ⟅Rel-Field⟆

There exists a pair $\hat{e}_0', \texttt{LPT}''$ such that
$\hat{e}_0, \texttt{LPT} \twoheadrightarrow^* \hat{e}_0', \texttt{LPT}''$ and
$e_0' \propto \hat{e}_0', \texttt{LPT}''$ ⟅induction hypothesis⟆

$\hat{e}_0.f, \texttt{LPT} \twoheadrightarrow^* \hat{e}_0'.f, \texttt{LPT}''$ ⟅RAC-Field⟆

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \hat{e}_0'.f, \texttt{LPT}''$

$e_0'.f \propto \hat{e}_0'.f, \texttt{LPT}''$ ⟅Rel-Field⟆

Letting $\hat{e}' = \hat{e}_0'.f$ and $\texttt{LPT}' = \texttt{LPT}''$ satisfies the conclusions of the lemma.

Subcase Rel-Lib-Field**:**

$\hat{y} = \texttt{LIB}$
$e_0 \propto \texttt{LIB}, \texttt{LPT}$
$CT \vdash \mathit{declaringclass}(f) \in CT_L$ ⟅Rel-Lib-Field⟆

There exists a pair $\hat{e}_0', \texttt{LPT}''$ such that
$\texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* \hat{e}_0', \texttt{LPT}''$ and
$e_0' \propto \hat{e}_0', \texttt{LPT}''$ ⟅induction hypothesis⟆

If $\hat{e}_0' = \texttt{LIB}$**:**
$\therefore {e_0}' \propto \texttt{LIB}, \texttt{LPT}''$ and
$\texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}''$

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}''$

$e_0'.f \propto \texttt{LIB}, \texttt{LPT}''$ ⟅Rel-Lib-Field⟆

Letting $\hat{e}' = \texttt{LIB}$ and $\texttt{LPT}' = \texttt{LPT}''$ satisfies the conclusions of the lemma.

If $\hat{e}_0' \neq \texttt{LIB}$**:**
There exists a set $\texttt{LPT}'''$ such that
${e_0}' \propto \texttt{LIB}, \texttt{LPT}'''$ and
$\texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}'''$ ⟅Lemma 10⟆

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}'''$

$e_0'.f \propto \texttt{LIB}, \texttt{LPT}'''$ ⟅Rel-Lib-Field⟆

Letting $\hat{e}' = \texttt{LIB}$ and $\texttt{LPT}' = \texttt{LPT}'''$ satisfies the conclusions of the lemma.

Case RC-Invk-Recv**:**
$e = e_0.m(\overline{d})$
$e' = {e_0}'.m(\overline{d})$
$e_0 \xrightarrow{CT} {e_0}'$ ⟅$e \xrightarrow{CT} e'$ by RC-Invk-Recv⟆

From the form of e, the fact $e \propto \hat{y}, \texttt{LPT}$ could be derived by either Rel-Invk or Rel-Lib-Invk.

Subcase Rel-Invk**:**
$\hat{y} = \hat{e}_0.m(\overline{\hat{d}})$
$e_0 \propto \hat{e}_0, \texttt{LPT}$ ⟅Rel-Invk⟆
$\overline{d} \propto \overline{\hat{d}}, \texttt{LPT}$

There exists a pair $\hat{e}_0', \texttt{LPT}''$ such that
$\hat{e}_0, \texttt{LPT} \twoheadrightarrow^* \hat{e}_0', \texttt{LPT}''$ and
${e_0}' \propto \hat{e}_0', \texttt{LPT}''$ ⟅induction hypothesis⟆

$\hat{e}_0.m(\overline{\hat{d}}), \texttt{LPT} \twoheadrightarrow^* \hat{e}_0'.m(\overline{\hat{d}}), \texttt{LPT}''$ ⟅RAC-Invk-Recv⟆

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \hat{e}_0'.m(\overline{\hat{d}}), \texttt{LPT}''$

$\texttt{LPT} \subseteq \texttt{LPT}''$ ⟅Lemma 8⟆

$\overline{d} \propto \overline{\hat{d}}, \texttt{LPT}''$ ⟅Lemma 5⟆

${e_0}'.m(\overline{d}) \propto \hat{e}_0'.m(\overline{\hat{d}}), \texttt{LPT}''$ ⟅Rel-Invk⟆

Letting $\hat{e}' = \hat{e}_0'.m(\overline{\hat{d}})$ and $\texttt{LPT}' = \texttt{LPT}''$ satisfies the conclusions of the lemma.

Subcase Rel-Lib-Invk**:**

$\hat{y} = \texttt{LIB}$
$e_0 \propto \texttt{LIB}, \texttt{LPT}$ ⟅Rel-Lib-Invk⟆
$\exists C \in CT_L\,.\ m \in CT \vdash \mathit{dmethods}(C)$

There exists a pair $\hat{e}_0', \texttt{LPT}''$ such that
$\texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* \hat{e}_0', \texttt{LPT}''$ and
${e_0}' \propto \hat{e}_0', \texttt{LPT}''$ ⟅induction hypothesis⟆

If $\hat{e}_0' = \texttt{LIB}$**:**
$\therefore {e_0}' \propto \texttt{LIB}, \texttt{LPT}''$ and
$\texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}''$

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}''$

$\texttt{LPT} \subseteq \texttt{LPT}''$ ⟅Lemma 8⟆

$\overline{d} \propto \texttt{LIB}, \texttt{LPT}''$ ⟅Lemma 5⟆

${e_0}'.m(\overline{d}) \propto \texttt{LIB}, \texttt{LPT}''$ ⟅Rel-Lib-Invk⟆

Letting $\hat{e}' = \texttt{LIB}$ and $\texttt{LPT}' = \texttt{LPT}''$ satisfies the conclusions of the lemma.

If $\hat{e}_0' \neq \texttt{LIB}$**:**
There exists a set $\texttt{LPT}'''$ such that
$e_0' \propto \texttt{LIB}, \texttt{LPT}'''$ and
$\texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}'''$ ⟅Lemma 10⟆

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}'''$

$\texttt{LPT} \subseteq \texttt{LPT}'''$ ⟅Lemma 8⟆

$\overline{d} \propto \texttt{LIB}, \texttt{LPT}'''$ ⟅Lemma 5⟆

$e_0'.m(\overline{d}) \propto \texttt{LIB}, \texttt{LPT}'''$ ⟅Rel-Lib-Invk⟆

Letting $\hat{e}' = \texttt{LIB}$ and $\texttt{LPT}' = \texttt{LPT}'''$ satisfies the conclusions of the lemma.

Case RC-Invk-Arg**:**

$e = e_0.m(\ldots, d_i, \ldots)$
$e' = e_0.m(\ldots, d_i', \ldots)$
$d_i \xrightarrow{CT} d_i'$ ⟅$e \xrightarrow{CT} e'$ by RC-Invk-Arg⟆

From the form of e, the fact $e \propto \hat{y}, \texttt{LPT}$ could be derived by either Rel-Invk or Rel-Lib-Invk.

Subcase Rel-Invk**:**

$\hat{y} = \hat{e}_0.m(\ldots, \hat{d}_i, \ldots)$
$e_0 \propto \hat{e}_0, \texttt{LPT}$
$d_i \propto \hat{d}_i, \texttt{LPT}$ ⟅Rel-Invk⟆

There exists a pair $\hat{d}_i', \texttt{LPT}''$ such that
$\hat{d}_i, \texttt{LPT} \twoheadrightarrow^* \hat{d}_i', \texttt{LPT}''$ and
$d_i' \propto \hat{d}_i', \texttt{LPT}''$ ⟅induction hypothesis⟆

$\hat{e}_0.m(\ldots, \hat{d}_i, \ldots), \texttt{LPT} \twoheadrightarrow^* \hat{e}_0.m(\ldots, \hat{d}_i', \ldots), \texttt{LPT}''$ ⟅RAC-Invk-Arg⟆

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \hat{e}_0.m(\ldots, \hat{d}_i', \ldots), \texttt{LPT}''$

$\texttt{LPT} \subseteq \texttt{LPT}''$ ⟅Lemma 8⟆

$e_0 \propto \hat{e}_0, \texttt{LPT}''$ ⟅Lemma 5⟆

$e_0.m(\ldots, {d_i}', \ldots) \propto \hat{e}_0.m(\ldots, \hat{d}_i', \ldots), \texttt{LPT}''$ ⟅Rel-Invk⟆

Letting $\hat{e}' = \hat{e}_0.m(\ldots, \hat{d}_i', \ldots)$ and $\texttt{LPT}' = \texttt{LPT}''$ satisfies the conclusions of the lemma.

Subcase Rel-Lib-Invk**:**

$\hat{y} = \texttt{LIB}$
$e_0 \propto \texttt{LIB}, \texttt{LPT}$ ⟅Rel-Lib-Invk⟆
$\exists C \in CT_L\,.\ m \in CT \vdash dmethods(C)$

There exists a pair $\hat{d}_i', \texttt{LPT}''$ such that
$\hat{d}_i, \texttt{LPT} \twoheadrightarrow^* \hat{d}_i', \texttt{LPT}''$ and
${d_i}' \propto \hat{d}_i', \texttt{LPT}''$ ⟅induction hypothesis⟆

If $\hat{d}_i' = \texttt{LIB}$**:**
$\therefore {d_i}' \propto \texttt{LIB}, \texttt{LPT}''$ and
$\texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}''$

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}''$

$\texttt{LPT} \subseteq \texttt{LPT}''$ ⟅Lemma 8⟆

$e_0 \propto \texttt{LIB}, \texttt{LPT}''$ ⟅Lemma 5⟆

$e_0.m(\ldots, {d_i}', \ldots) \propto \texttt{LIB}, \texttt{LPT}''$ ⟅Rel-Lib-Invk⟆

Letting $\hat{e}' = \texttt{LIB}$ and $\texttt{LPT}' = \texttt{LPT}''$ satisfies the conclusions of the lemma.

If $\hat{d}_i' \neq \texttt{LIB}$**:**
There exists a set $\texttt{LPT}'''$ such that
${d_i}' \propto \texttt{LIB}, \texttt{LPT}'''$ and
$\texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}'''$ ⟅Lemma 10⟆

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}'''$

$\texttt{LPT} \subseteq \texttt{LPT}'''$ ⟅Lemma 8⟆

$e_0 \propto \texttt{LIB}, \texttt{LPT}'''$ ⟅Lemma 5⟆

$e_0.m(\ldots, d_i', \ldots) \propto \texttt{LIB}, \texttt{LPT}'''$ ⟅Rel-Lib-Invk⟆

Letting $\hat{e}' = \texttt{LIB}$ and $\texttt{LPT}' = \texttt{LPT}'''$ satisfies the conclusions of the lemma.

Case RC-New-Arg**:**

$e = \texttt{new}\ C(\ldots, p_i, \ldots)$
$e' = \texttt{new}\ C(\ldots, \hat{p}_i, \ldots)$ ⟅$e \xrightarrow{CT} e'$ by RC-New-Arg⟆
$p_i \xrightarrow{CT} \hat{p}_i$

From the form of e, the fact $e \propto \hat{y}, \texttt{LPT}$ could be derived by either Rel-New or Rel-Lib-New.

Subcase Rel-New**:**

$\hat{y} = \texttt{new}\ C(\ldots, \hat{p}_i, \ldots)$
$C \in CT'$ ⟅Rel-New⟆
$p_i \propto \hat{p}_i, \texttt{LPT}$

There exists a pair $\hat{p}_i', \texttt{LPT}''$ such that
$\hat{p}_i, \texttt{LPT} \twoheadrightarrow^* \hat{p}_i', \texttt{LPT}''$ and
$p_i' \propto \hat{p}_i', \texttt{LPT}''$ ⟅induction hypothesis⟆

$\texttt{new}\ C(\ldots, \hat{p}_i, \ldots), \texttt{LPT} \twoheadrightarrow^* \texttt{new}\ C(\ldots, \hat{p}_i', \ldots), \texttt{LPT}''$ ⟅RAC-New-Arg⟆

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \texttt{new}\ C(\ldots, \hat{p}_i', \ldots), \texttt{LPT}''$

$\texttt{new}\ C(\ldots, p_i', \ldots) \propto \texttt{new}\ C(\ldots, \hat{p}_i', \ldots), \texttt{LPT}''$ ⟅Rel-New⟆

Letting $\hat{e}' = \texttt{new}\ C(\ldots, \hat{p}_i', \ldots)$ and $\texttt{LPT}' = \texttt{LPT}''$ satisfies the conclusions of the lemma.

Subcase Rel-Lib-New**:**

$\hat{y} = \texttt{LIB}$
$C \in CT_L$ ⟅Rel-Lib-New⟆
$p_i \propto \texttt{LIB}, \texttt{LPT}$

There exists a pair $\hat{p}_i', \texttt{LPT}''$ such that
$\texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* \hat{p}_i', \texttt{LPT}''$ and
${p_i}' \propto \hat{p}_i', \texttt{LPT}''$ ⟅induction hypothesis⟆

If $\hat{p}_i' = \texttt{LIB}$**:**
$\therefore {p_i}' \propto \texttt{LIB}, \texttt{LPT}''$ and
$\texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}''$

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}''$

$\texttt{new}\ C(\ldots, {p_i}', \ldots) \propto \texttt{LIB}, \texttt{LPT}''$ ⟅Rel-Lib-New⟆

Letting $\hat{e}' = \texttt{LIB}$ and $\texttt{LPT}' = \texttt{LPT}''$ satisfies the conclusions of the lemma.

If $\hat{p}_i' \neq \texttt{LIB}$**:**
There exists a set $\texttt{LPT}'''$ such that
${p_i}' \propto \texttt{LIB}, \texttt{LPT}'''$ and
$\texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}'''$ ⟅Lemma 10⟆

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}'''$

$\texttt{new}\ C(\ldots, {p_i}', \ldots) \propto \texttt{LIB}, \texttt{LPT}'''$ ⟅Rel-Lib-New⟆

Letting $\hat{e}' = \texttt{LIB}$ and $\texttt{LPT}' = \texttt{LPT}'''$ satisfies the conclusions of the lemma.

Case RC-Cast**:**
$e = (C)e_0$
$e' = (C){e_0}'$
$e_0 \xrightarrow{CT} {e_0}'$ ⟅$e \xrightarrow{CT} e'$ by RC-Cast⟆

From the form of e, the fact $e \propto \hat{y}, \texttt{LPT}$ could be derived by either Rel-Cast or Rel-Lib-Cast.

Subcase Rel-Cast**:**
$\hat{y} = (C)\hat{e}_0$
$e_0 \propto \hat{e}_0, \texttt{LPT}$ ⟅Rel-Cast⟆

There exists a pair $\hat{e}_0', \texttt{LPT}''$ such that
$\hat{e}_0, \texttt{LPT} \twoheadrightarrow^* \hat{e}_0', \texttt{LPT}''$ and
${e_0}' \propto \hat{e}_0', \texttt{LPT}''$ ⟅induction hypothesis⟆

$(C)\hat{e}_0, \texttt{LPT} \twoheadrightarrow^* (C)\hat{e}_0', \texttt{LPT}''$ ⟅RAC-Cast⟆

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \hat{e}_0'.f, \texttt{LPT}''$

$(C){e_0}' \propto (C)\hat{e}_0', \texttt{LPT}''$ ⟅Rel-Cast⟆

Letting $\hat{e}' = (C)\hat{e}_0'$ and $\texttt{LPT}' = \texttt{LPT}''$ satisfies the conclusions of the lemma.

Subcase Rel-Lib-Cast**:**

$\hat{y} = \texttt{LIB}$
${e_0} \propto \texttt{LIB}, \texttt{LPT}$ ⟅Rel-Lib-Cast⟆

There exists a pair $\hat{e}_0', \texttt{LPT}''$ such that
$\texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* \hat{e}_0', \texttt{LPT}''$ and
${e_0}' \propto \hat{e}_0', \texttt{LPT}''$ ⟅induction hypothesis⟆

If $\hat{e}_0' = \texttt{LIB}$**:**
$\therefore {e_0}' \propto \texttt{LIB}, \texttt{LPT}''$ and
$\texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}''$

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}''$

$(C){e_0}' \propto \texttt{LIB}, \texttt{LPT}''$ ⟅Rel-Lib-Cast⟆

Letting $\hat{e}' = \texttt{LIB}$ and $\texttt{LPT}' = \texttt{LPT}''$ satisfies the conclusions of the lemma.

If $\hat{e}_0' \neq \texttt{LIB}$**:**
There exists a set $\texttt{LPT}'''$ such that
${e_0}' \propto \texttt{LIB}, \texttt{LPT}'''$ and
$\texttt{LIB}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}'''$ ⟅Lemma 10⟆

$\therefore \hat{e}, \texttt{LPT} \twoheadrightarrow^* \texttt{LIB}, \texttt{LPT}'''$

$(C){e_0}' \propto \texttt{LIB}, \texttt{LPT}'''$ ⟅Rel-Lib-Cast⟆

Letting $\hat{e}' = \texttt{LIB}$ and $\texttt{LPT}' = \texttt{LPT}'''$ satisfies the conclusions of the lemma.

□

Lemma 12 relates the initial expression of an FJ program to its FJ_{AVE} counterpart.

Lemma 12. *Let e_0 be the initial expression of an FJ program. If $e_0 \xrightarrow{CT}{}^* e$, then $e_0, \{\} \twoheadrightarrow^* \hat{e}, \texttt{LPT}$ and $e \propto \hat{e}, \texttt{LPT}$ and $\mathcal{P}(\hat{e}, \texttt{LPT})$ holds.*

Proof. By induction on the number of steps n taken to derive $e_0 \xrightarrow{CT}{}^* e$

Case $n = 0$**:**

$e = \hat{e} = e_0$

$\mathcal{P}(e_0, \{\})$ holds ⟅e_0 is an application expression and by AVE-CT⟆

We will prove that the relation $e_0 \propto e_0, \{\}$ holds by structural induction on the form of e_0.

Subcase FIELD**:**

$e_0 = q.f$

$q \propto q, \{\}$ ⟅induction hypothesis⟆

$q.f \propto q.f, \{\}$ ⟅REL-FIELD⟆

$e_0 \propto e_0, \{\}$

Subcase NEW**:**

$e_0 = \texttt{new}\ C(\overline{p})$

$\overline{p} \propto \overline{p}, \{\}$ ⟅induction hypothesis⟆

$C \in CT'$ ⟅$\mathcal{P}(e_0, \{\})$ holds⟆

$\texttt{new}\ C(\overline{p}) \propto \texttt{new}\ C(\overline{p}), \{\}$ ⟅REL-NEW⟆

$e_0 \propto e_0, \{\}$

Subcase INVK**:**

$e_0 = q.m(\overline{d})$

$q \propto q, \{\}$
$\overline{d} \propto \overline{d}, \{\}$ ⟅induction hypothesis⟆

$q.m(\overline{d}) \propto q.m(\overline{d}), \{\}$ ⟅REL-INVK⟆

$e_0 \propto e_0, \{\}$

Subcase CAST**:**

$e_0 = (D)q$

$q \propto q, \{\}$ ⟅induction hypothesis⟆

$(D)q \propto (D)q, \{\}$ ⟅REL-FIELD⟆

$e_0 \propto e_0, \{\}$

Case $n > 0$**:**

$e_0 \xrightarrow{CT}{}^{*} e_i \xrightarrow{CT} e_{i+1}$ ⟅lemma precondition⟆

$e_0, \{\} \twoheadrightarrow^{*} \hat{e}_i, \mathtt{LPT}_i$
$e_i \propto \hat{e}_i, \mathtt{LPT}_i$ ⟅induction hypothesis⟆
$\mathcal{P}(\hat{e}_i, \mathtt{LPT}_i)$ holds

$e_0, \{\} \twoheadrightarrow^{*} \hat{e}_{i+1}, \mathtt{LPT}_{i+1}$
$e_{i+1} \propto \hat{e}_{i+1}, \mathtt{LPT}_{i+1}$ ⟅Lemma 11⟆
$\mathcal{P}(\hat{e}_{i+1}, \mathtt{LPT}_{i+1})$ holds

□

Theorem 1 shows that any call graph that is sound for the FJ_{AVE} program soundly over-approximates the application part of the call graph of the original FJ program. The theorem proves this property by showing that for any call edge that can occur in an execution of the application part of the original FJ program, a corresponding call edge can occur in an execution of the FJ_{AVE} program.

Theorem 1. *If* $e_0 \xrightarrow{CT}{}^{*} (\texttt{new}\ C(\overline{p})).m(\overline{d})$ *and* $CT \vdash mresolve(m, C) \in CT_A$, *then* $e_0, \{\} \twoheadrightarrow^{*} (\texttt{new}\ C(\overline{p})).m(\overline{\hat{d}}), \texttt{LPT}$.

Proof. We will prove this theorem using the conclusions of Lemma 6 and Lemma 12.

There exists a pair $\hat{e}, \texttt{LPT}$ such that
$e_0, \{\} \twoheadrightarrow^{*} \hat{e}, \texttt{LPT}$
$(\texttt{new}\ C(\overline{p})).m(\overline{d}) \propto \hat{e}, \texttt{LPT}$ ⟅Lemma 12⟆
$\mathcal{P}(\hat{e}, \texttt{LPT})$ holds

There exists an expression $\hat{y}$ such that
$(\texttt{new}\ C(\overline{p})).m(\overline{d}) \propto \hat{y}, \texttt{LPT}$ could be derived by Rel-Invk or Rel-Lib-Invk, and
$\hat{e}, \texttt{LPT} \twoheadrightarrow^{*} \hat{y}, \texttt{LPT}$ ⟅Lemma 6⟆

Case Rel-Invk**:**

$\hat{y} = (\texttt{new}\ C(\overline{\hat{p}})).m(\overline{\hat{d}})$ ⟅Rel-Invk⟆

$\therefore e_0, \{\} \twoheadrightarrow^{*} \hat{e}, \texttt{LPT} \twoheadrightarrow^{*} (\texttt{new}\ C(\overline{\hat{p}})).m(\overline{\hat{d}}), \texttt{LPT}$

Case Rel-Lib-Invk**:**

$\hat{y} = \texttt{LIB}$
$\texttt{new}\ C(\overline{p}) \propto \texttt{LIB}, \texttt{LPT}$ ⟅Rel-Lib-Invk⟆
$\overline{d} \propto \texttt{LIB}, \texttt{LPT}$

$\exists\, \hat{z}\,.\ \texttt{LIB}, \texttt{LPT} \twoheadrightarrow^{*} \hat{z}, \texttt{LPT}$ and ⟅Lemma 6⟆
$\texttt{new}\ C(\overline{p}) \propto \hat{z}, \texttt{LPT}$ could be derived by Rel-New or Rel-Lib-New

Subcase Rel-New**:**

$\hat{z} = \texttt{new}\ C(\overline{\hat{p}})$ ⟅Rel-New⟆

$\exists\, \hat{C} \in CT'_L\,.\ m \in CT' \vdash dmethods(\hat{C})$ ⟅Ave-Method⟆

$\texttt{LIB}, \texttt{LPT} \twoheadrightarrow \texttt{LIB}, \texttt{LPT} \cup \{\texttt{LIB}.m(\overline{\texttt{LIB}})\}$ ⟅RA-Invk⟆
$\twoheadrightarrow \texttt{LIB}.m(\overline{\texttt{LIB}}), \texttt{LPT} \cup \{\texttt{LIB}.m(\overline{\texttt{LIB}})\}$ ⟅RA-Return⟆
$\twoheadrightarrow (\texttt{new}\ C(\overline{\hat{p}})).m(\overline{\texttt{LIB}}), \texttt{LPT} \cup \{\texttt{LIB}.m(\overline{\texttt{LIB}})\}$ ⟅RAC-Invk-Recv⟆

$\therefore e_0, \{\} \twoheadrightarrow^* \hat{e}, \texttt{LPT} \twoheadrightarrow^* (\texttt{new}\ C(\overline{\hat{p}})).m(\overline{\texttt{LIB}}), \texttt{LPT} \cup \{\texttt{LIB}.m(\overline{\texttt{LIB}})\}$

Subcase Rel-Lib-New**:**

$C \in CT_L$ ⟅Rel-Lib-New⟆

$\therefore CT \vdash mresolve(m, C) \in CT_L$

However, $CT \vdash mresolve(m, C) \in CT_A$ ⟅theorem precondition⟆

Therefore, Rel-Lib-New cannot derive $\texttt{new}\ C(\overline{p}) \propto \hat{z}, \texttt{LPT}$ and this subcase can never occur.

□

Chapter 7

Conclusions

Call graphs are the building block of inter-procedural static analyses used in compilers, verification tools, and program understanding tools. The key problem that a call graph construction algorithm tries to solve is dynamic dispatch. Since the receiver of a call could have been created anywhere in the program, a precise call graph analysis generally has to analyze the whole program. However, the large sizes of common libraries (e.g., the Java standard library) makes it very expensive to analyze the whole program, even for a small program like a Java "Hello, World!" program. A common practice to overcome this excessive cost is to either completely ignore the code in those libraries, which yields unsound results, or conservatively over-approximate the side effects the unanalyzed libraries could have on the client/application code, which could yield very imprecise results.

In this dissertation, we proposed a novel approach that enables sound and precise call graph construction for the application part of a given program without analyzing its library dependencies.

7.1 The Separate Compilation Assumption

We have defined the *separate compilation assumption*, a realistic, yet useful, assumption that the library code can be compiled separate from the client code that uses it. From this assumption, we identified a set of constraints that enable precise call graph construction for the application part of a Java program without analyzing its library dependencies. The constraints model the side effects that the unanalyzed library code could have in terms of: class instantiation, method calls, field and array accesses, and exception handling.

We have proven the correctness of the separate compilation assumption in the context of call graph construction for Featherweight Java, a core calculus of the Java language. The main

theme of the proof revolves around proving two properties for an FJ program P whose AVERROES transformation is P'. First, the initial expression of P is related to the initial expression of P' by the abstraction relation $\propto$. Second, for any trace of P, there exists a corresponding trace in P' whose steps are related by the abstraction relation $\propto$

7.2 AVERROES

AVERROES enables the use of the separate compilation assumption in all Java whole-program analysis frameworks. Given any Java program, AVERROES generates an alternative placeholder library that models the constraints that follow from the separate compilation assumption. The placeholder library can then be used instead of the original library classes as input to a whole-program analysis.

We have empirically evaluated the soundness and precision of two AVERROES-based tools (SPARK$_{\text{AVE}}$ and DOOP$_{\text{AVE}}$). Call graphs generated by both SPARK$_{\text{AVE}}$ and DOOP$_{\text{AVE}}$ were found to be sound when compared to the dynamic call edges observed by *J during the execution of a given program. We have also compared the precision of the AVERROES-based tools and have shown that the imprecise library callback edges computed by SPARK$_{\text{AVE}}$ and DOOP$_{\text{AVE}}$ further caused some spurious edges to be computed between application methods and from application methods to the library.

Since AVERROES does not analyze the whole program, the overhead to construct the placeholder library was found to be minimal. The size of the placeholder library generated by AVERROES is typically on the order of 80 kB of class files compared to 25 MB of class files found in the Java standard library. We have also empirically shown that AVERROES improves the analysis time of whole-program call graph construction by a factor of 3.5x to 8x, while reducing the memory required to finish the analysis by a factor of 8.4x to 12x.

7.3 Future Work

We identify the following possible directions for future work.

7.3.1 AVERROES for frameworks

We plan to extend this work to generate placeholder libraries for various widely-used Java frameworks (e.g., Android, J2EE, Eclipse Plug-ins) using AVERROES. We hope that this will lead to an easier means of analyzing client applications developed in these frameworks without the need to analyze the framework itself. Like a library, a framework typically satisfies the separate

compilation assumption because it is developed without knowledge of the client applications that will be developed within it. One major difference is that in a framework, the main entry point to the program resides in the framework rather than in the client application. The application is then reflectively started by the framework code. We expect that with some changes, AVERROES will be applicable to these and other Java frameworks.

7.3.2 AVERROES for non-Java programs

The essence of the separate compilation assumption can be applied to languages other than Java to support partial program analyses for those languages. In our recent work [4], we have implemented some of the constraints of the separate compilation assumption in the context of analyzing Scala source files presented to the Scala compiler. In this case, the referenced library classes are provided to the compiler as bytecodes packaged in JAR files. Therefore, the separate compilation assumption was crucial to obtain sound and precise source-level analyses. We plan to expand on this work and fully apply the separate compilation assumption in the context of the Scala language. To achieve that, more constraints might be added to the separate compilation assumption to support the very rich set of features provided by Scala (e.g., traits, closures, and abstract type members). We also plan to apply the separate compilation assumption in the context of dynamic languages, like Javascript.

References

[1] Gagan Agrawal, Jinqian Li, and Qi Su. "Evaluating a Demand Driven Technique for Call Graph Construction". In: *Proceedings of the International Conference on Compiler Construction (CC)*. Vol. 2304. Springer, 2002, pp. 29–45.

[2] Karim Ali and Ondřej Lhoták. "Application-Only Call Graph Construction". In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Vol. 7313. Springer, 2012, pp. 688–712.

[3] Karim Ali and Ondřej Lhoták. "Averroes: Whole-Program Analysis without the Whole Program". In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Vol. 7920. Springer, 2013, pp. 378–400.

[4] Karim Ali, Marianna Rapoport, Ondrej Lhoták, Julian Dolby, and Frank Tip. "Constructing Call Graphs of Scala Programs". In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Vol. 8586. Springer, 2014, pp. 54–79.

[5] Frances E. Allen. "Interprocedural Data Flow Analysis". In: *Proceedings of the International Federation of Information Processing (IFIP)*. 1974, pp. 398–402.

[6] Paulo Sérgio Almeida. "Balloon Types: Controlling Sharing of State in Data Types". In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. 1997, pp. 32–59.

[7] Lars Ole Andersen. "Program Analysis and Specialization for the C Programming Language". (DIKU report 94/19). PhD thesis. DIKU, University of Copenhagen, May 1994.

[8] David F. Bacon and Peter F. Sweeney. "Fast Static Analysis of C++ Virtual Function Calls". In: *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 1996, pp. 324–341.

[9] Stephen M. Blackburn et al. "The DaCapo benchmarks: java benchmarking development and analysis". In: *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 2006, pp. 169–190.

[10] Eric Bodden. *Soot-list: Stack overflow when generating call graph*. http://www.sable.mcgill.ca/pipermail/soot-list/2008-July/001831.html. June 2014.

[11] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 241–250.

[12] Martin Bravenboer and Yannis Smaragdakis. "Strictly declarative specification of sophisticated points-to analyses". In: *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 2009, pp. 243–262.

[13] Dave Brosius, Torsten Curdt, Markus Dahm, and Jason van Zyl. *The Bytecode Engineering Library (Apache Commons BCEL™)*. http://commons.apache.org/bcel/. June 2014.

[14] David G. Clarke, John Potter, and James Noble. "Ownership Types for Flexible Alias Protection". In: *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 1998, pp. 48–64.

[15] Standard Performance Evaluation Corporation. *SPEC JVM98 Benchmarks*. http://www.spec.org/jvm98/. July 2014.

[16] Jeffrey Dean, David Grove, and Craig Chambers. "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis". In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Vol. 952. Springer, 1995, pp. 77–101.

[17] Greg DeFouw, David Grove, and Craig Chambers. "Fast Interprocedural Class Analysis". In: *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 1998, pp. 222–236.

[18] Werner Dietl and Peter Müller. "Universes: Lightweight Ownership for JML". In: *Journal of Object Technology (JOT)* 4.8 (2005), pp. 5–32.

[19] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. "Simple and Effective Analysis of Statically Typed Object-Oriented Programs". In: *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 1996, pp. 292–305.

[20] Bruno Dufour, Laurie J. Hendren, and Clark Verbrugge. "*J: a tool for dynamic analysis of Java programs". In: *Companion to the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA Companion)*. ACM, 2003, pp. 306–307.

[21] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. "Efficient construction of approximate call graphs for JavaScript IDE services". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE / ACM, 2013, pp. 752–761.

[22] Daniela Genius, Martin Trapp, and Wolf Zimmermann. "An Approach to Improve Locality Using Sandwich Types". In: *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation (TIC)*. Vol. 1473. Springer, 1998, pp. 194–214.

[23] Christian Grothoff, Jens Palsberg, and Jan Vitek. "Encapsulating objects with confined types". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29.6 (2007).

[24] David Grove and Craig Chambers. "A framework for call graph construction algorithms". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23.6 (2001), pp. 685–746.

[25] Nevin Heintze and Olivier Tardieu. "Demand-Driven Pointer Analysis". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2001, pp. 24–34.

[26] Michael Hind. "Pointer analysis: haven't we solved this problem yet?" In: *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. ACM, 2001, pp. 54–61.

[27] Michael Hind and Anthony Pioli. "Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses". In: *Proceedings of the International Static Analysis Symposium (SAS)*. Vol. 1503. Springer, 1998, pp. 57–81.

[28] Reid Holmes and David Notkin. "Identifying program, test, and environmental changes that affect behaviour". In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 371–380.

[29] IBM. *T.J. Watson Libraries for Analysis WALA*. http://wala.sourceforge.net/. June 2014.

[30] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. "Featherweight Java: a minimal core calculus for Java and GJ". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23.3 (2001), pp. 396–450.

[31] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. "The Soot framework for Java program analysis: a retrospective". In: *Cetus Users and Compiler Infrastructure Workshop*. Galveston Island, TX, Oct. 2011.

[32] The DOT Language. http://www.graphviz.org/content/dot-language. June 2014.

[33] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. "Preliminary design of JML: a behavioral interface specification language for java". In: *ACM SIGSOFT Software Engineering Notes (SEN)* 31.3 (2006), pp. 1–38.

[34] Yun Young Lee, Sam Harwell, Sarfraz Khurshid, and Darko Marinov. “Temporal code completion and navigation”. In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE / ACM, 2013, pp. 1181–1184.

[35] Ondřej Lhoták. “Comparing call graphs”. In: *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. ACM, 2007, pp. 37–42.

[36] Ondrej Lhoták and Laurie J. Hendren. “Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 18.1 (2008).

[37] Ondřej Lhoták and Laurie J. Hendren. “Scaling Java Points-to Analysis Using SPARK”. In: *Proceedings of the International Conference on Compiler Construction (CC)*. Vol. 2622. Springer, 2003, pp. 153–169.

[38] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.

[39] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S.-C. Lan. “An Empirical Study of Static Call Graph Extractors”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7.2 (1998), pp. 158–191.

[40] James Noble, Jan Vitek, and John Potter. “Flexible Alias Protection”. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Vol. 1445. Springer, 1998, pp. 158–185.

[41] JUnit Home Page. http://junit.sourceforge.net. June 2014.

[42] LogicBlox Home Page. http://logicblox.com/. June 2014.

[43] Atanas Rountev, Scott Kagan, and Thomas J. Marlowe. “Interprocedural Dataflow Analysis in the Presence of Large Libraries”. In: *Proceedings of the International Conference on Compiler Construction (CC)*. Vol. 3923. Springer, 2006, pp. 2–16.

[44] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. “Fragment Class Analysis for Testing of Polymorphism in Java Software”. In: *IEEE Transactions on Software Engineering (TSE)* 30.6 (2004), pp. 372–387.

[45] Atanas Rountev and Barbara G. Ryder. “Points-to and Side-Effect Analyses for Programs Built with Precompiled Libraries”. In: *Proceedings of the International Conference on Compiler Construction (CC)*. Vol. 2027. Springer, 2001, pp. 20–36.

[46] Atanas Rountev, Mariana Sharp, and Guoqing (Harry) Xu. “IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries”. In: *Proceedings of the International Conference on Compiler Construction (CC)*. Vol. 4959. Springer, 2008, pp. 53–68.

[47] Barbara G. Ryder. "Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages". In: *Proceedings of the International Conference on Compiler Construction (CC)*. Vol. 2622. Springer, 2003, pp. 126–137.

[48] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. "Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation". In: *Journal of Theoritical Computer Science* 167.1&2 (1996), pp. 131–170.

[49] Max Schäfer, Julian Dolby, Manu Sridharan, Emina Torlak, and Frank Tip. "Correct Refactoring of Concurrent Java Code". In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Vol. 6183. Springer, 2010, pp. 225–249.

[50] Olin Shivers. "Control-Flow Analysis in Scheme". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 1988, pp. 164–174.

[51] Graphviz - Graph Visualization Software. http://www.graphviz.org/. June 2014.

[52] Manu Sridharan and Rastislav Bodík. "Refinement-based context-sensitive points-to analysis for Java". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2006, pp. 387–400.

[53] Vijay Sundaresan, Laurie J. Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. "Practical virtual method call resolution for Java". In: *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 2000, pp. 264–280.

[54] Frank Tip and Jens Palsberg. "Scalable propagation-based call graph construction algorithms". In: *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 2000, pp. 281–293.

[55] Frank Tip, Peter F. Sweeney, Chris Laffra, Aldo Eisma, and David Streeter. "Practical extraction techniques for Java". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 24.6 (2002), pp. 625–666.

[56] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. "Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?" In: *Proceedings of the International Conference on Compiler Construction (CC)*. Vol. 1781. Springer, 2000, pp. 18–34.

[57] Jan Vitek and Boris Bokowski. "Confined Types". In: *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 1999, pp. 82–96.

[58] Jan Vitek and Boris Bokowski. "Confined types in Java". In: *Journal of Software: Practice and Experience (SPE)* 31.6 (2001), pp. 507–532.

[59] Frédéric Vivien and Martin C. Rinard. “Incrementalized Pointer and Escape Analysis”. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2001, pp. 35–46.

[60] Jyh-Shiarn Yur, Barbara G. Ryder, and William Landi. “An Incremental Flow- and Context-Sensitive Pointer Aliasing Analysis”. In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 1999, pp. 442–451.

[61] Weilei Zhang and Barbara G. Ryder. “Automatic construction of accurate application call graph with library call abstraction for Java”. In: *Journal of Software Maintenance* 19.4 (2007), pp. 231–252.

[62] Tian Zhao, Jens Palsberg, and Jan Vitek. “Type-based confinement”. In: *Journal of Functional Programming (JFP)* 16.1 (2006), pp. 83–128.