Pointer Analysis for Java Programs: Novel Techniques and Applications

by

Alexandru D. Sălcianu

Magistère, *Ecole Normale Supérieure de Cachan*, France (1999)
S.M., Massachusetts Institute of Technology (2001)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2006

Author .....
Department of Electrical Engineering and Computer Science
September 1, 2006

Certified by .....
Martin C. Rinard
Professor, Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Pointer Analysis for Java Programs:
Novel Techniques and Applications

by

Alexandru D. Sălcianu

Submitted to the Department of Electrical Engineering and Computer Science
on September 1, 2006, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This dissertation presents a pointer analysis for Java programs, together with several practical analysis applications.

For each program point, the analysis is able to construct a *points-to graph* that describes how local variables and object fields point to objects. Each points-to graph also contains escape information that identifies the objects that are reachable from outside the analysis scope.

Our pointer analysis can extract correct information by analyzing only parts of a whole program. First, our analysis analyzes a method without requiring information about its calling context. Instead, our analysis computes parameterized results that are later instantiated for each relevant call site. Second, our analysis correctly handles calls to unanalyzable methods (e.g., native methods). Hence, our analysis can trade precision for speed without sacrificing correctness: if the analysis of a call to a specific callee requires too much time, the analysis can treat that callee as unanalyzable.

The results of our analysis enable standard program optimizations like the stack allocation of local objects. More interestingly, this dissertation explains how to extend the analysis to detect pure methods. Our analysis supports a flexible definition of method purity: a method is pure if it does not mutate any object that exists in the program state before the start of the method. Therefore, our analysis allows pure methods to allocate and mutate temporary objects (e.g., iterators) and/or construct complex object structures and return them as a result.

Thesis Supervisor: Martin C. Rinard
Title: Professor, Electrical Engineering and Computer Science

Acknowledgments

As I write these words, exactly seven years have passed from the moment I came to MIT as a graduate student. Retrospectively, it is amazing how much I have changed since then. This period was full of learning opportunities and wonderful experiences, but also many difficulties. I would like to use this opportunity to thank the people who helped me overcome these difficulties: if I finish my dissertation today, it is largely due to them! I apologize in advance to all the wonderful people whose names I forgot to mention below.

First, I would like to thank my mother Maria and my late father Eugen for the education, love, and support they offered me. I still have a lot to learn from my mother's optimism! I would also like to thank my grandmother, Elisabeta Ohâi, who took care of me in my early childhood and taught me to value work.

Several true friends helped me overcome the stress of the past months: Mihai Anton, Iuliu Vasilescu, Darko Marinov, Emanuel Stoica, and Viktor Kuncak. I will always remember the long conversations we had together. With such friends, it was hard not to graduate! I will struggle to return to them the same care they showed me.

I greatly enjoyed being involved (as an officer or as a simple member) in three MIT student organizations: the International Film Club, the Romanian Student Association, and the MIT Ballroom Dance Team. The events they organized and the great friends I met there offered me many happy moments and prevented me from becoming a geek. Many thanks to the wonderful people who made these events possible: Thomas Gervais, Cansu Tunca, Krzysztof Gajos, Claudiu Stan, Mihai Ibănescu, Gheorghe Chistol, Alex Andoni, Virgil Petrea, Dan Iancu, Ioanid Roşu, Viorel Costeanu, Cătălin Frâncu, Silvia Şuteu, Florin Albeanu, Dumitru Daniliuc, and Filip Hsu. I would also like to thank my ballroom dance partner Stefana Stantcheva whose help and patience allowed me to discover the magic of ballroom dancing.

In the laboratory, I enjoyed working or simply chatting with Chandra Boyapati, Kostas Arkoudas, Maria-Cristina Marinescu, Karen Zee, Charles Bouillaguet, Huu Hai Nguyen, Enoch Peserico, Rodrigo Rodrigues, Mihai Pătraşcu, Suhabe Bugrara, Carlos Pacheco, and Adrian Corduneanu. Mary McDavitt helped me solve many administrative issues and proofread a draft of this page! Many thanks to all of them!

Several professors and researchers played an important role in my professional life. Since I was an undergraduate at the "Politehnica" University of Bucharest, Irina Athanasiu never ceased to impress me with her full dedication to her students. I would like to thank my advisor, Martin Rinard, for insisting that I come to MIT and for providing me with support during all these years. During my first term at MIT, David Gifford impressed me with his teaching skills. I greatly enjoyed being a student, and next a Teaching Assistant in 6.821, David's class. Manuel Fähndrich and Robert DeLine were my mentors during my intership at Microsoft Research. Thanks to them, that summer was probably the happiest part of my PhD! Finally, I would like to thank Arvind and Michael Ernst, for agreeing to be on my thesis committee, for interesting technical discussions, and for their help with the final draft of the dissertation.

Contents

List of Figures

Chapter 1

Introduction

The presence of pointers in a programming language significantly complicates the analysis of programs written in that language, because the analysis system cannot determine the memory locations pointed to by a pointer variable by a simple inspection of the program statements. In the absence of detailed knowledge about the memory locations manipulated by the program, compilers have to make very conservative assumptions about the objects that pointers may reference, and therefore very conservative assumptions about the effects of instructions that use these pointers. This limits the impact of compiler optimizations, and that of program understanding and testing tools. This problem is very important for programs written in modern object-oriented languages like Java [4], because these programs allocate all objects dynamically and access them using pointers.

The thesis of this dissertation is that one can use pointer analysis to perform interesting compiler optimizations and program understanding applications on significant Java programs. To prove our thesis, this dissertation presents a *pointer analysis* algorithm for Java programs, together with two pointer analysis applications: a program optimization (*stack allocation* of local objects) and a program understanding application (detection of *pure methods*). We briefly explain our analysis and its applications in the next paragraphs.

Overview of Our Pointer Analysis

Our pointer analysis is a combined points-to and escape analysis. Given a program point inside a method m, the analysis is able to construct a *points-to graph* that models the program state accessed by the execution of the *analysis scope* up to that point. The analysis scope consists of the method m and all its transitive callees. A points-to graph models the objects pointed to by m's local variables and the heap references[1] created by the analysis scope. Additionally, the points-to graph identifies the objects that *escape* from the analysis scope. An object escapes from the analysis scope if other parts of the program may access it.

[1]There exists a *heap reference* from object o_1 to object o_2, along field f, if the value of the field f of o_1 is the address of o_2.

The key feature that distinguishes our analysis from most other pointer analyses is its built-in ability to obtain correct information by analyzing only parts of a whole program:

- First, our analysis examines each method without knowing its *calling context*. Unless otherwise specified, the expression "calling context" denotes the objects transitively pointed to by the method parameters. Our analysis computes parameterized results (points-to graphs) that use special constructs to abstract over the calling context. These points-to graphs can later be instantiated for the calling context at each invocation of the analyzed method.

- Second, our analysis correctly handles *unanalyzable* method calls. Calls that invoke native methods are unanalyzable, because the analysis cannot access the source code of the native methods.[2] Additionally, the analysis can consider any call to be unanalyzable if, e.g., one of the invoked methods is too expensive to analyze. This operation loses precision, but preserves correctness.[3]

Another important feature of our analysis is its *compositional* nature. Our analysis uses the points-to graph from the end of a method as a summary of that method's execution. The inter-procedural analysis uses the summary of a method to process each call to that method. The inter-procedural analysis achieves context sensitivity by instantiating the method summary for the calling context at each relevant call site. This compositional strategy avoids the re-analysis of a method for each calling context.

To model the heap, our analysis uses an extension of the *object allocation site* model [18]. Our analysis uses one *inside node* to model all objects created by a specific instruction during the current execution of the analysis scope. Our analysis abstracts over the calling context by using *parameter* and *load* nodes. The parameter nodes model the objects pointed to by the actual arguments. The load nodes model the objects whose addresses are read from fields of escaped objects (e.g., an object read from a field of a parameter). Intuitively, the parameter/load nodes are placeholders for nodes from the unknown calling context. For each method invocation, the inter-procedural analysis computes a node map that disambiguates these placeholders, according to the current calling context. The analysis handles aliasing in the calling context by mapping aliased nodes to the same node.

In general, precise pointer analyses tend to have large worst-case time complexity, and our analysis is no exception. The asymptotic worst-case time complexity of our analysis is polynomial, but a very high polynomial: $\mathcal{O}(N^{10})$, where N is the size of the analysis scope. However, the ability to extract correct results by analyzing only parts of a whole program allows the analysis to trade precision for speed by reducing the

[2]However, the analysis can use manually-generated summaries for common native methods.

[3]The presence of unanalyzable calls requires an update of the definition of the analysis scope: during the analysis of the method m, the analysis scope consists of the method m and all the methods it transitively invokes *through analyzable calls*.

analysis scope until the analysis can terminate in a reasonable amount of time. E.g., if the processing of a method call instruction takes too much time, the analysis can simply consider it unanalyzable. In practice, our prototype implementation analyzes all methods from a large benchmark like the JDK 1.0.2 `javac` compiler (more than $50,000$ bytecode instructions) from the industry-standard SPECjvm98 bechmarks suite [76] in less than a minute, on a 2.8GHz Pentium 4 with 1G of RAM.

To avoid any confusion, we would like to clarify the distinction between the analysis of separate libraries and the analysis of parts of a whole program.

Any significant Java library contains virtual calls that may invoke unknown methods from outside the analyzed library. First, Java libraries use virtual calls to perform "upcalls" into the user code. E.g., hash-based implementations of sets invoke the virtual method `hashCode` on the set elements. Additionally, virtual methods allow the user to add new functionality by subclassing library classes and overriding certain methods. E.g., the class `java.util.AbstractMap` provides a skeleton implementation of an association map; library users are free to subclass this class and define customized maps by implementing the abstract `entrySet` method.

Theoretically, our analysis can analyze separate libraries by considering each virtual call as unanalyzable. However, virtual calls are ubiquitous in Java and currently, we do not have experimental evidence that the results of separate library analysis are sufficiently precise to be useful. Therefore, we do not claim that our analysis can analyze arbitrary groups of methods. Instead, we present our analysis as able to analyze *parts of a whole program*. Given a whole program, a compiler can use a relatively cheap whole-program analysis (e.g., Rapid Type Analysis [5]) to construct a static call graph that identifies all possible callees of each virtual call. Next, the compiler can use our more expensive pointer analysis to analyze only certain parts of the program. Vivien and Rinard [83] modified an early implementation of our analysis in order to analyze only the program parts "around" allocations sites that allocate a large number of objects (as indicated by profile data).

Other pointer analyses, although initially designed as whole-program analyses, were later modified to analyze only parts of a program. E.g., Rountev [69] presents a modification of Andersen's pointer analysis [3] that can analyze "program fragments".[4] In general, these modified analyses model the rest of the program using worst-case assumptions and next use the original whole-program analysis. In contrast, our analysis computes parameterized results that can be instantiated for any calling context.

First Analysis Application: Stack Allocation

Java programs allocate all objects in the garbage-collected heap. This technique is essential for Java's type safety, provides an elegant programming model, and eliminates hard-to-find programming bugs like dangling pointers. Unfortunately, garbage-

[4] In Rountev's analysis [69], a "program fragment" is an arbitrary collection of C procedures, not necessarily from the same program/library.

collection may incur a large runtime overhead. Our analysis detects allocation sites (i.e., `new` instructions) that allocate only objects that are unreachable from outside the enclosing method. When the method terminates, these objects become unreachable and can be deallocated. The compiler changes these allocation sites to allocate memory from a stack; our current implementation uses the execution stack, but a different, dedicated stack can be used too. When the method terminates, the stack pointer returns to its original value and all objects stack allocated inside the method are deallocated without any garbage-collection overhead.

One important reason for actually performing the stack allocation optimization is to empirically test the correctness of our analysis design and implementation: wrong stack allocation decisions by the analysis usually result in very visible runtime errors. Our implementation successfully performs the stack allocation optimization on a large set of twenty benchmarks, including the entire SPECjvm98 benchmark suite [76]. For the SPECjvm98 applications, our analysis stack allocates up to 95% of all dynamically allocated objects (32% on average). For each application, we manually checked that the optimized version produces the same results as the original version.

Second Analysis Application: Purity Analysis

Our analysis can detect *pure* methods. Our analysis supports a flexible definition of method purity: A method is pure if it does not mutate any object that exists in the *prestate*, i.e., the program state right before the method invocation. This is the same definition that the Java Modeling Language (JML) [55] uses. This definition allows a pure method to allocate and mutate temporary objects (e.g., iterators) and/or construct complex object structures and return them as a result.

The knowledge that a method is pure (according to our definition) is important for several tasks: Pure methods can safely be used in program assertions and specifications [55, 10]. Pure methods can safely be invoked at runtime by invariant-detection tools [30] and by specification-mining tools [27]. In program understanding tools, local invariants about existing objects can be propagated over a call to a pure method.

Intuitively, our purity analysis examines assignments "`v1.f = v2`" and uses the points-to information to detect the mutated objects (i.e., the objects that `v1` may point to). The purity analysis benefits from the fact that the inside nodes represent only new objects, i.e., only objects that do not exist in the prestate. The purity analysis ignores mutations on inside nodes. A method is pure if it does not mutate any other node.

In the absence of information about the objects pointed to by the program variables, other implemented side-effect analyses are unsound [17] or overly-conservative [53], e.g., by forbidding pure methods from mutating any object (not even local ones), thus preventing pure methods from using common programming constructs like iterators. Other researchers have used different pointer analyses to infer side effects [65, 44, 70, 20]. However, these analyses do not use special abstractions for the new objects. Therefore, they do not allow pure methods to mutate new objects. At most, these analyses allow pure methods to mutate captured objects [70]; still, they do not allow pure methods to allocate, initialize, and return new objects.

In practice, our purity analysis checked the purity of several complex data consistency predicates. Our purity analysis implementation is already used by several other researchers; e.g., Dallmeier *et al.* used our purity analysis in a published object behavior mining tool [27].

Contributions

This dissertation makes the following contributions:

- **Pointer Analysis:** We present a compositional pointer analysis for Java programs. Our analysis is able to extract correct information by analyzing only parts of a whole program. First, our analysis analyzes a method without requiring information about its calling context. Second, our analysis correctly handles unanalyzable call instructions (e.g., calls to native methods). This second feature allows our analysis to analyze large benchmarks by trading precision for speed *without sacrificing correctness*: if the processing of a specific call instruction takes too much time, the analysis can treat that call as unanalyzable.

  Our analysis uses many ideas from the analysis of Whaley and Rinard [85]. In order to formalize and prove the analysis correctness, we had to completely redesign the original analysis, obtaining a new analysis. For example, we had to modify the inter-procedural analysis in order to make it handle all aliasing situations correctly. Additionally, the presentation of our analysis is much clearer and easier to reason about.

  The analysis presented in this dissertation improves over the analysis presented in Sălcianu's SM dissertation [78]: First, we re-designed the inter-procedural analysis, obtaining a simpler and more intuitive analysis. Second, we designed several analysis optimizations that allow our prototype to analyze large benchmarks in reasonable time.

- **Purity Analysis:** We extend our pointer analysis to detect *pure* methods. We support a flexible definition of purity: a method is pure if it does not mutate any object that existed before the start of the method; a pure methods is free to allocate and mutate new objects. Our analysis uses special abstractions for the objects allocated by the analyzed method: hence, our analysis conservatively identifies and filters out mutations that occur only on new objects.

- **Correctness Proof:** The flexibility of our analysis — namely, the ability to obtain correct information by analyzing only parts of a whole program — comes at the cost of increased complexity in the analysis design. Due to this complexity, the correctness of the analysis is non-trivial, and requires a detailed proof.

  We define a concrete semantics for a non-trivial subset of the Java bytecode language (including threads, static fields, and virtual-calls). We use this concrete semantics to formalize a set of properties of the analysis results and prove that

these properties hold. These properties are strong enough to imply the correctness of the stack allocation optimization and the correctness of the detection of pure methods.

Our correctness proof is one of very few correctness proofs for pointer analyses. We presented a preliminary correctness proof in Sălcianu's SM dissertation [78]. In this dissertation, we updated the proof to cover the new inter-procedural analysis and the detection of pure methods.

During the correctness proof, we discovered and fixed several correctness problems in our initial analysis algorithm. Moreover, the intuition we gained as part of this process enabled us to find fundamental conceptual errors in the design of other previously published algorithms [85, 21]. Given this experience, we are extremely skeptical of the correctness of complex program analyses (such as compositional inter-procedural dataflow analyses) that do not come with a correctness proof.

- **Implementation:** A prototype implementation of our analysis is publicly available [79] and is currently used by other researchers. In a recent publication [27], Dallmeier *et al.* from *Universität des Saarlandes* (Germany) provide the following evaluation of our prototype implementation:

  > *"Currently, we use the purity analysis provided by Sălcianu and Rinard (2005), which is the only scalable implementation we are aware of. It is based on the FLEX compiler infrastructure, which unfortunately restricts analysis to programs compiled against the GNU Classpath API 0.08 [implementation of the Java standard library]. Besides this limitation, the analysis is sufficiently fast and produces reliable results."*

  Implementing program analyses requires an impressive infrastructure. Unfortunately, the program analysis community lacks generic and public implementations of many useful algorithms.[5] To address this problem, we designed and implemented `jpaul` [31], a stand-alone Java library of generic, high-level algorithms for program analysis: graph algorithms, constraint solving, finite state automata, etc. The `jpaul` library is publicly available as an open-source library under the Free BSD licence. Although `jpaul` is a very *niche* library, there were more than 150 downloads of `jpaul` during the first five months of 2006.

- **Experimental Validation:** To evaluate our analysis, we used our prototype implementation to conduct several experiments.

[5] As an anecdotal example, when we started implementing program analyses in late 1999, we had troubles finding something as simple as a flexible Java implementation of the construction of the strongly-connected components of a directed graph. Different compiler infrastructures were re-implementing basic graph algorithms for various graph representation.

First, we used our analysis to perform stack allocation, a classic application of pointer analysis. Stack allocation is a good empiric test for the correctness of the analysis implementation: incorrect stack allocation decisions usually lead to visible runtime errors. For the applications from the SPECjvm98 benchmark suite [76], our analysis stack allocates between 0% and 95% of all dynamically allocated objects (32% on average). Second, we used our analysis to detect several complex pure methods that are beyond the reach of previously published purity analyses. Third, as a scalability test, we applied our purity analysis to several large benchmarks, and measured the proportion of pure methods in these benchmarks.

Many previously published analyses are designed to enable compiler optimizations. However, presumably due to the difficulty of carrying the analysis information through the various stages of the compiler to perform the transformation that implements the optimization, many of these analyses [71, 60] do not actually come with an implementation of the corresponding optimization. One usual approach is to instrument the code to obtain statistics that characterize how successful the analysis was in enabling the application of the corresponding optimization. We note that when an optimization like stack allocation is really performed, incorrect analysis results often show up as runtime errors. With no correctness proof and no implemented application that may reveal potential analysis errors, we are even more skeptical about the correctness of a proposed complex analysis.

Dissertation Outline

The rest of this dissertation has the following structure. Chapter 2 introduces the basic concepts of our analysis through a non-trivial, self-contained example. Chapter 3 discusses the representation of the analyzed programs. Chapter 4 describes our pointer analysis. Chapter 5 presents two applications of our pointer analysis: the stack allocation of captured objects and the detection of pure methods. Chapter 6 presents a correctness proof for our analysis. More specifically, Chapter 6 defines a precise semantics for the analyzed language, formalizes several properties of the points-to graphs that the analysis computes, and proves that these properties hold. These properties imply the correctness of the stack allocation optimization and the purity analysis. Chapter 7 describes a few techniques for increasing the analysis speed and/or precision. Chapter 8 explains our implementation and presents experimental results. Chapter 9 discusses related work and Chapter 10 concludes our dissertation.

Chapter 2

Example

This chapter introduces the basic concepts of our analysis through a non-trivial, self-contained example. Chapter 4 contains a detailed description of our analysis.

Sample Program: Figure 2-1 presents a sample Java program that manipulates singly linked lists of bidimensional points. Class `List` implements a list using list cells of class `Cell`, and supports two operations: `add(e)` adds the object `e` to the front of the list; `iterator()` returns an iterator over the list elements. Class `ListItr` implements a list iterator: the field `cell` maintains a reference to the current list cell; each invocation of the method `next()` returns the data from the current cell, and also updates the field `cell` to point to the next cell.[1] Class `Point` implements a bidimensional point. The method `Main.sumX(list)` computes the sum of the `x` coordinates of all points from the list `list`. The method `sumX` uses an iterator to explore all the list elements.

We explain how our analysis works on the sample program in the next two sections. For the moment, we describe two analysis uses for this sample program:

Stack Allocation: In line 50, the method `Main.sumX` invokes the method `List.iterator` that allocates an iterator object (in line 7) and returns it. Our analysis detects that the iterator object is *captured* inside `sumX`, i.e., it is unreachable from outside `sumX`. Once `sumX` terminates, the iterator object becomes unreachable from the entire program and can be deallocated. Therefore, instead of allocating the iterator object in the garbage-collected heap, the program can allocate it in the stack frame of the method `sumX`. Stack allocated objects are deallocated without any garbage collection overhead, when the method terminates and its stack frame is discarded. The stack allocation optimization (1) inlines the call from line 50, and (2) changes the instruction "`new ListItr(head)`" from the inlined copy of `List.iterator` to allocate the iterator object on the stack. Stack allocation requires a simple adjustment

[1] In a real implementation, the classes `Cell` and `ListItr` would be implemented as inner classes of `List`; we use a flat format for simplicity. As another simplification, we wrote the body of `ListItr.next()` using simple instructions.

```
 1  class List {
 2    Cell head = null;
 3    void add(Object e) {
 4      head = new Cell(e, head);
 5    }
 6    Iterator iterator() {
 7      return new ListItr(this.head);
 8    }
 9  }
10
11  class Cell {
12    Cell(Object d, Cell n) {
13      this.data = d; this.next = n;
14    }
15    Object data;
16    Cell   next;
17  }
18
19  interface Iterator {
20    boolean hasNext();
21    Object next();
22  }
23
24  class ListItr implements Iterator {
25    ListItr(Cell head) {
26      this.cell = head;
27    }
28    Cell cell;
29    public boolean hasNext() {
30      return this.cell != null;
31    }
32    public Object next() {
33      Cell c = this.cell;
34      Object result = c.data;
35      Cell c2 = c.next;
36      this.cell = c2;
37      return result;
38    }
39  }
40  class Point {
41    Point(float x, float y) {
42      this.x = x; this.y = y;
43    }
44    float x, y;
45  }
46
47  class Main {
58    static float sumX(List list) {
59      float s = 0;
50      Iterator it = list.iterator();
51      while(it.hasNext()) {
52        Point p = (Point) it.next();
53        s += p.x;
54      }
55      return s;
56    }
57
58    public static void main(String args[]) {
59      List list = new List();
60      /* add some points to the list */
61      list.add(new Point(1,2));
62      list.add(new Point(1,3));
63      list.add(new Point(2,7));
64      /* compute sum of the x coordinates */
65      int s = sumX(list);
66    }
67  }
```

Figure 2-1: Example source code.

of the stack pointer to make space for the new object.[2] The inlining step brings the allocation site into a method where it is captured. Notice that the iterator object is not captured in the method `List.iterator` that allocates it, because `List.iterator` returns the iterator object.

Purity Analysis: The method `sumX` uses the iterator method `next` in order to explore the list elements. The method `next` mutates the iterator state; in our implementation, it mutates the field `cell` of the iterator object. However, the iterator object did not exist at the beginning of `sumX`. In spite of the mutation on the iterator, our analysis infers that `sumX` is pure, i.e., `sumX` does not mutate objects that existed before its invocation. Notice that a simple purity analysis that just checks the absence of field write instructions would not be able to detect the purity of `sumX`.

[2]E.g., if the stack grows up, we add to the stack pointer the size of the object we want to allocate.

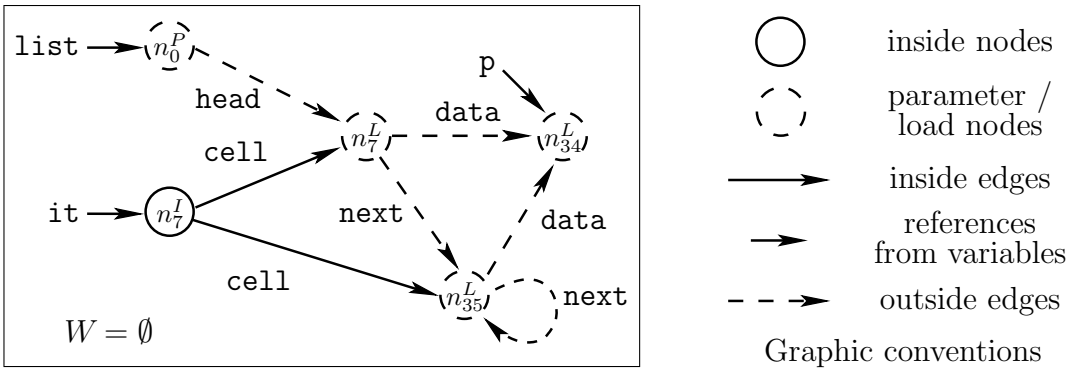


Figure 2-2: Points-To Graph for the end of `Main.sumX(list)`

2.1 Analysis Overview

The analysis examines individual methods from the analyzed program. For each program point inside an analyzed method m, our analysis constructs a points-to graph that models the execution of the *analysis scope* up to that program point. During the analysis of the method m, the analysis scope consists of the method m and all its transitive callees.[3] The analysis scope does not include the method callers: our analysis examines a method without knowing the calling context.

A points-to graph describes how local variables and object fields point to objects. Additionally, a points-to graph identifies the objects that *escape* from the analysis scope, i.e., the objects that are reachable from other parts of the program. Both the points-to information and the escape information are *may*-approximations;[4] therefore, the authoritative information is the negative one: e.g., an object does *not* escape.

The escape information allows the stack allocation optimization to detect the captured (i.e., non-escaped) objects. Knowing which objects are pointed to by local variables allows the purity analysis to identify mutated objects. Additionally, the escape information and the points-to information are important for the analysis inner-working.

Figure 2-2 presents the points-to graph for the end of the method `Main.sumX`. This points-to graph models the execution of the analysis scope consisting of the method `sumX` and all its transitive callees. The rest of this section describes the meaning of this points-to graph. Later, Section 2.2 explains how the analysis constructs such points-to graphs.

In a points-to graph, the nodes model heap objects and the edges model heap references.[5] Each edge is labeled with the field it corresponds to. We write $\langle n_1, f, n_2 \rangle$

[3]During this example, we ignore the possibility of unanalyzable CALLs.

[4]As a trivial application of Rice's theorem [67], points-to and escape information is undecidable; therefore, any static analysis computes an approximation of this information.

[5]There exists a heap reference from object o_1 to object o_2, along field f, if the value of the field f of o_1 is the address of o_2.

to denote an edge from node n_1 to node n_2, labeled with the field f. The edge $\langle n_1, f, n_2 \rangle$ models a heap reference from an object that n_1 models to a node that n_2 models, along field f. Our analysis uses several kinds of nodes and edges:

- *Inside nodes* model objects created by the analysis scope.[6] There is one inside node for each allocation site (i.e., each `new` instruction); this node models *all* objects allocated at that site during the current execution of the analyzed method. We write n_l^I to denote the inside node for the allocation site from line l. In Figure 2-2, the inside node n_7^I models the iterator object allocated by the `new` instruction from line 7.

- *Parameter nodes* model objects passed as arguments; there is one parameter node for each formal parameter of object type (i.e., not an `int`, `boolean`, etc.). We write n_i^P to denote the parameter node for the ith formal parameter of the analyzed method. In Figure 2-2, the parameter node n_0^P models the `List` object pointed to by the formal parameter `list` of the method `sumX`.

- *Load nodes and outside edges:* The analysis scope may read heap references from fields of objects that *escape*, i.e., objects that are reachable from outside the analysis scope. As an example, the method `sumX` reads the field `head` of its list parameter.[7] The object whose address is read depends on the particular (unknown) calling context.

  In such situations, the analysis abstracts over the unknown calling context by using a *load node* to model the unknown object. There is at most one load node for each load instruction. We write n_l^L to denote the load node for the load instruction from line l. Additionally, the analysis uses *outside edges* to record where each load node is read from.

  In Figure 2-2, the outside edge $\langle n_0^P, \texttt{head}, n_7^L \rangle$ indicates that the load node n_7^L models an object that was read from the field `head` of the list object (modeled by the parameter node n_0^P).

- *Inside edges* model heap references created by the analyzed method. In Figure 2-2, the inside edges $\langle n_7^I, \texttt{cell}, n_7^L \rangle$ and $\langle n_7^I, \texttt{cell}, n_{35}^L \rangle$ model the references created by `sumX`[8] from the iterator n_7^I to the first (respectively, to the next) list cells.

Points-to graphs also indicate the nodes that each local variable may point to, and the nodes that the analyzed method may return. E.g., in Figure 2-2, the local variable `it` points to the node n_7^I (the node that models the iterator object); `sumX` does not return any node (it returns an integer, not an object).

For the purpose of purity analysis, each points-to graph contains a set W of *modified prestate abstract fields*. An abstract field is a field of a specific node, i.e., a

[6] We use the adjective "inside" for entities created by the analysis scope.

[7] Indirectly, in line 7 of the callee `List.iterator`.

[8] Indirectly, through the iterator-related methods it invokes.

pair of the form $\langle n, f \rangle$. Our purity analysis studies only mutations on *prestate objects* (i.e., objects that already existed when the analyzed method was invoked). As inside nodes model only new objects, the analysis ignores mutations on inside nodes. There are no modified prestate abstract fields for the method `sumX`: $W = \emptyset$; hence, `sumX` is pure.

Finally, a points-to graph records the nodes pointed to by the arguments of unanalyzable method calls (none in this example). These nodes escape from the analysis scope. Our analysis conservatively approximates object escapability by reachability from parameter nodes, returned nodes, and nodes passed as arguments to unanalyzable calls. In Figure 2-2, the inside node n_7^I is unreachable from any such node. Therefore, the iterator object that n_7^I models is captured and can be stack allocated.

A few important observations are in order:

- For each analysis scope, the number of nodes is bounded, ensuring the termination of our fixed-point computations.
- "Loop" edges like $\langle n_{35}^L, \texttt{next}, n_{35}^L \rangle$ are typical for methods that manipulate recursive data structures.
- The analysis does not need load nodes and outside edges to process reads from fields of captured objects. As a captured object is not accessible from outside the analysis scope, the analysis has "seen" all relevant field assignments and modeled the created heap references using inside edges. To process such a read, the analysis simply "follows" the inside edges. E.g., if method `sumX` read the field `it.cell`, the loaded object is modeled by one of the nodes n_7^L and n_{35}^L, the two nodes targeted by the `cell`-labeled inside edges that start in n_7^I, the only node pointed to by the local variable `it`.
- The parameter nodes, load nodes, and the outside edges allow our analysis to abstract over the unknown calling context. Intuitively, the parameter and the load nodes are "placeholders" for nodes from the caller analysis scope. The inter-procedural analysis (Section 2.2.2) replaces these placeholders with appropriate nodes, function of the calling context of each relevant call site.

Section 4.1 presents the analysis abstractions more formally.

2.2 Analysis of the Example

This section explains how the analysis examines the program from Figure 2-1 and computes points-to graphs similar to the one from Figure 2-2.

Intra-procedurally, our analysis is a dataflow analysis: it starts with an initial points-to graph for the beginning of the analyzed method. Next, the analysis applies the transfer functions for the method instructions and computes the points-to graphs for the other points inside the method.

Inter-procedurally, the analysis propagates information from the callees into the caller. Therefore, the analysis of method m requires the analysis of m's transitive callees.[9] E.g., the analysis of the method `sumX` requires the analysis of the method `ListItr.next` (invoked by `sumX` in line 52).

Section 2.2.1 below illustrates the intra-procedural analysis on the example of method `ListItr.next()`, a method that does not contain any method invocation. Next, Section 2.2.2 explains how the inter-procedural analysis handles method invocations.

2.2.1 Intra-procedural Analysis

Figure 2-3 presents the Java source code of `ListItr.next()` (a fragment of the code from Figure 2-1), and the points-to graphs that the analysis computes for the program points inside `ListItr.next()`.

At the beginning of the method, the formal parameter `this` points to n_0^P, the parameter node that models the receiver object (of class `ListItr`). The first instruction (line 33) reads the field `this.cell` to obtain the address of the current list cell. The analysis does not know what the field `cell` of n_0^P points to because the analysis examines each method without knowing the calling context. Instead, the analysis sets the local variable `c` to point to the load node n_{33}^L, and adds the outside edge $\langle n_0^P, \texttt{cell}, n_{33}^L \rangle$ to record where it read n_{33}^L from. The next instruction (line 34) is similar: it reads the field `data` of the current list cell (the obtained object will be returned as the result of this method); the analysis generates the outside edge $\langle n_{33}^L, \texttt{data}, n_{34}^L \rangle$.

In line 35, the program reads `c.next` to obtain the address of the next list cell; accordingly, the analysis generates the outside edge $\langle n_{33}^L, \texttt{next}, n_{35}^L \rangle$ and sets local variable `c2` to point to n_{35}^L. Next, in line 36, the program sets `this.cell` to point to the newly read object; accordingly, the analysis adds the inside edge $\langle n_0^P, \texttt{cell}, n_{35}^L \rangle$ to model the new heap reference. This instruction mutates the field `cell` of the parameter object; accordingly, the analysis adds the pair $\langle n_0^P, \texttt{cell} \rangle$ to the set W of modified prestate abstract fields.

The analysis processes the final instruction, "`return result`", by recording the fact that the method returns the node n_{35}^L (the only node pointed to by `result`). In addition, as the local variables no longer exist once the method terminates, the points-to graph for the end of the method does not contain any information about the local variables.

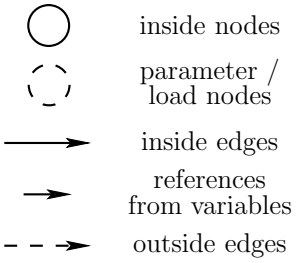
2.2.2 Inter-procedural Analysis

Given the points-to graph for the program point before a call instruction, the analysis needs to produce the points-to graph for the program point right after the call. For this purpose, the analysis needs to model the effect of the callee execution on the points-to graph before the call.

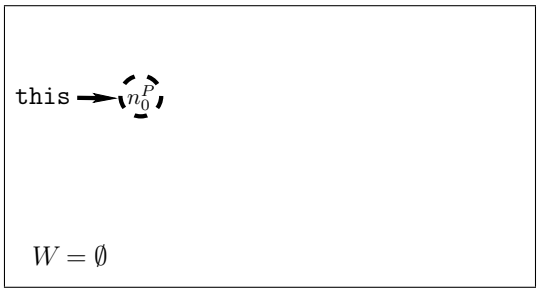
[9] Except those invoked through unanalyzable CALLs.

```
24  class ListItr implements Iterator {
      ...
32    public Object next() {
33      Cell c = this.cell;
34      Object result = c.data;
35      Cell c2 = c.next;
36      this.cell = c2;
37      return result;
38    }
39  }
```

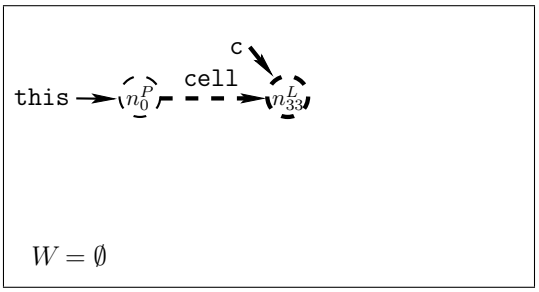
a. Java source



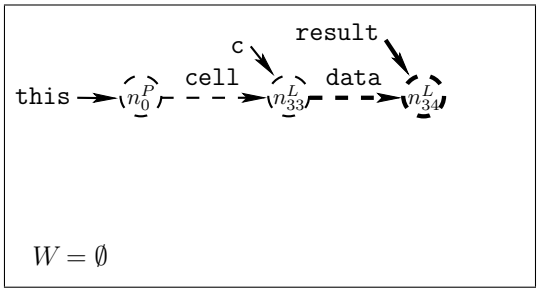
b. Graphic conventions



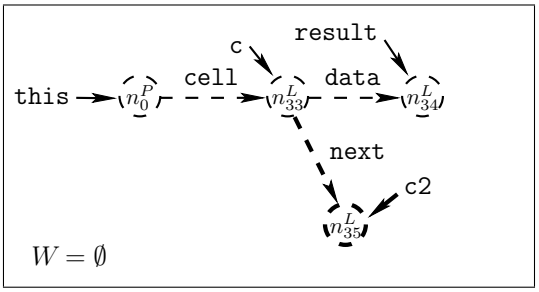
c. Points-to graph at method beginning.



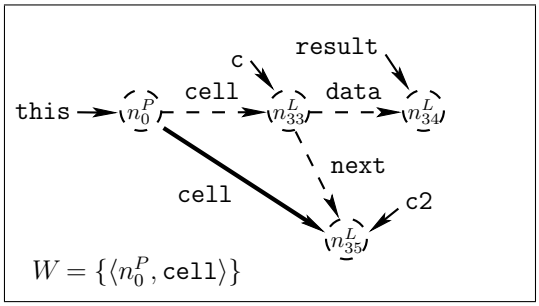
d. Points-to graph after line 33.



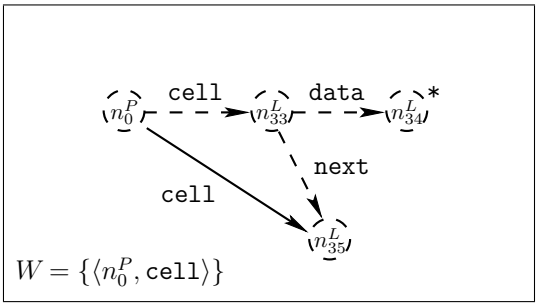
e. Points-to graph after line 34.



f. Points-to graph after line 35.



g. Points-to graph after line 36.



h. Points-to graph at method end. We use "*" to indicate nodes that may be returned.

Figure 2-3: Intra-procedural analysis for method `ListItr.next()`. In the points-to graphs, we depict new elements in bold.

Intuition

The key idea behind the inter-procedural analysis is that the points-to graph from the end of a method is an abstraction not only of the heap manipulated by the method, but also of the execution of that method: e.g., inside edges abstract store instructions[10] and outside edges abstract load instructions. The analysis assigns an operational meaning to the elements of the callee points-to graph (e.g., inside/outside edges). The analysis uses this operational meaning to "execute" the callee on the points-to graph before the call.

Some of the actions of the callee may involve parameter/load nodes, that are just placeholders for nodes that were unknown during the analysis of the callee. To interpret such actions in the context of the caller points-to graph before the call, the analysis maintains a *node map* that maps each callee node to the nodes it may represent. Initially, each parameter node is mapped to the node(s) pointed to by the corresponding actual argument; the analysis discovers mappings for the load nodes during the execution of the operational meaning of the callee points-to graph.

The points-to graph for the end of the callee does not contain information about the instruction ordering, nor about the number of times each instruction executes. Hence, the inter-procedural analysis has to assume that any ordering/repetition is possible: the analysis repeatedly executes the elements of the callee points-to graph until it reaches a fixed-point. The execution of each element may add new inside/outside edges to the callee points-to graph and may also extend the node map with new mappings for the load nodes.

Example

Consider the call `it.next()` from line 52 inside method `Main.sumX(list)`. The only possible callee is the method `ListItr.next()`.[11] Line 52 is inside a `while` loop; as with any dataflow analysis, our analysis iterates over the loop body until it reaches a fixed-point. We consider the processing for the call from line 52, in the first analysis iteration over the loop body.

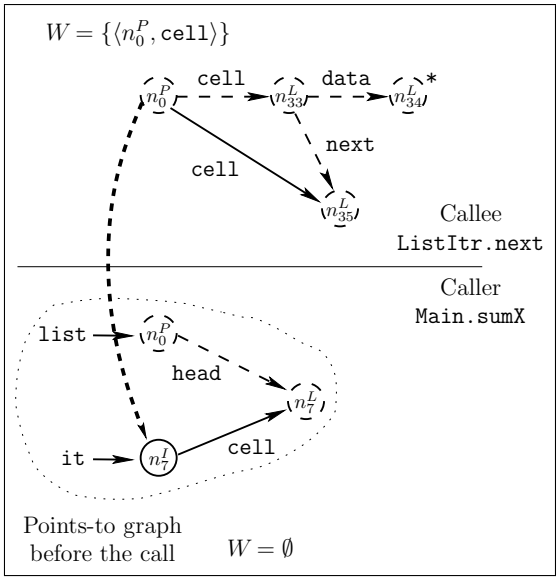
Figure 2-4 contains a graphic representation of four steps from a possible execution of the inter-procedural analysis in this case. For this example, ignore the particular way we computed the points-to graph before the call. Here are a few steps from a possible execution of the inter-procedural analysis:

1. Initially, the analysis maps the parameter node n_0^P to n_7^I, the only node pointed to by the actual argument `it`.

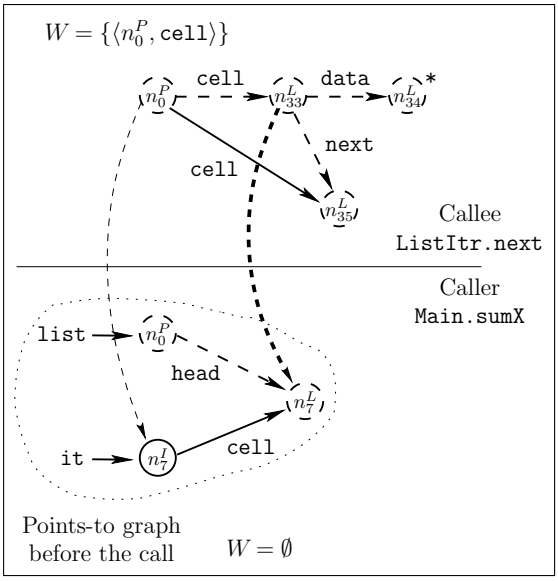
2. Consider the callee outside edge $\langle n_0^P, \texttt{cell}, n_{33}^L \rangle$. This outside edge abstracts a

[10] A store instruction is an instruction that creates a heap reference, i.e., an instruction of the form "$v_1.f = v_2$", where f is a field of object type.

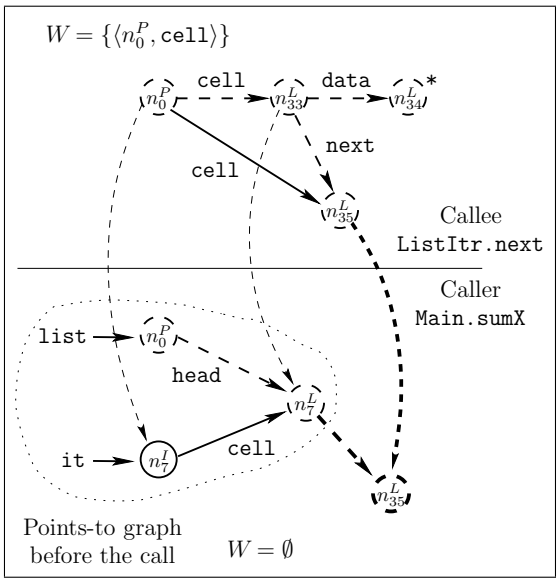
[11] The details of the call graph construction are beyond the scope of this example. For the moment, notice that in our analyzed program, the only instantiated iterator is of class `ListItr` (see line 7). Therefore, the only possible callee for the call from line 52 is `ListItr.next()`.



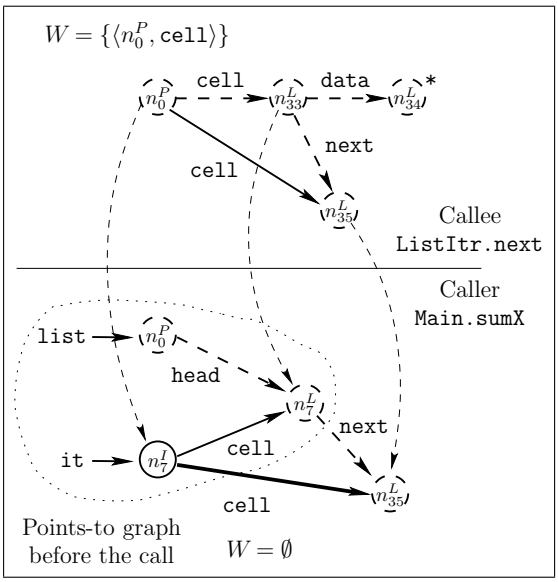
1. Initial map.



2. Execution of the callee outside edge $\langle n_0^P, \texttt{cell}, n_{33}^L\rangle$.



3. Execution of the callee outside edge $\langle n_{33}^L, \texttt{next}, n_{35}^L\rangle$.



4. Execution of the callee inside edge $\langle n_0^P, \texttt{cell}, n_{35}^L\rangle$.

Figure 2-4: Graphic representation of the first 4 steps from the inter-procedural analysis for the call to `ListItr.next()` in line 52 of `Main.sumX()`. We use the same graphic conventions as in Figure 2-2 on page 23. Additionally, dashed curved arrows represent node mappings and bold graphic elements represent new node mappings and new edges and nodes added to the points-to graph after the call. *Note:* both the analysis of the callee and the analysis of the caller use the parameter node n_0^P: in the callee points-to graphs, n_0^P models the iterator object pointed to by the (implicit) formal parameter `this`; in the caller points-to graphs, n_0^P models the list object pointed to by the formal parameter `list`.

load instruction. As n_0^P may represent n_7^I,[12] this load instruction may read the reference modeled by the inside edge $\langle n_7^I, \texttt{cell}, n_7^L \rangle$. As a consequence, n_{33}^L may represent n_7^L; the analysis extends the map to record this fact.

3. Consider the callee outside edge $\langle n_{33}^L, \texttt{next}, n_{35}^L \rangle$. n_{33}^L may represent n_7^L, a node that escapes even in the caller (it is reachable from the parameter `list`). Therefore, even the analysis of the caller does not know what the corresponding load instruction reads.

   Notice the difference between this case and the previous one: in the previous case, the analysis was able to "resolve" the load, i.e., the analysis was able to find the node(s) that n_{33}^L represents. In the current case, the analysis cannot find the node(s) that n_{35}^L represents. Hence, the analysis has to represent this unresolved load in the caller points-to graph, using an outside edge.

   The analysis adds the caller outside edge $\langle n_7^L, \texttt{next}, n_{35}^L \rangle$ to record the fact that n_{35}^L is a placeholder for the nodes that may be read from the escaped node n_7^L, along field `cell`. The analysis also maps n_{35}^L to itself to indicate that n_{35}^L is present in the points-to graph after the call.

4. The callee inside edge $\langle n_0^P, \texttt{cell}, n_{35}^L \rangle$ models a store instruction between the node(s) that n_0^P represents (i.e., n_7^I) and the node(s) that n_{35}^L models (i.e., n_{35}^L). Therefore, the analysis adds the caller inside edge $\langle n_7^I, \texttt{cell}, n_{35}^L \rangle$.

The inter-procedural analysis executes a few more similar steps. As the set of nodes is finite, the inter-procedural analysis will eventually reach a fixed-point.

At the end of the inter-procedural analysis, the analysis uses the final node map to project the set of mutated abstract fields in the context of the caller. The callee `ListItr.next()` mutates the field `cell` of the parameter node n_0^P. As n_0^P represents only the inside node n_7^I, a node that models only new objects allocated after the start of `sumX`, the analysis of the caller can ignore this mutation. This example illustrates two aspects of our purity analysis: (1) the analysis uses the inter-procedural node map to project the mutated abstract fields from the callee to the caller, and (2) the analysis ignores the mutation on inside nodes.

Discussion

A natural question is whether the inter-procedural analysis described above has any advantage over a solution based on inlining the callee and next using standard intra-procedural analysis. Besides being able to deal with recursive methods, the current solution has the advantage that the points-to graph for the end of the callee is a *simplified* model of the callee execution: less precise (e.g., no information about the instruction ordering) but smaller and faster to execute. Intuitively, the points-to

[12] In this particular example, n_0^P represents only n_7^I. We write "n_0^P *may* represent n_7^I" because, in general, a parameter/load node may be mapped to more than one node; e.g., just consider the case where the actual argument `it` points to two nodes, each allocated on a separate branch of the program.

graph for the end of the callee is a pre-processed form of the analysis of an inlined copy of the callee. We discuss this issue in more detail in Section 4.6.1.

To summarize, here are the main features of the inter-procedural analysis:

- The inter-procedural analysis interprets the points-to graph for the end of the callee as an abstraction of the callee execution.
- The inter-procedural analysis maintains a map that disambiguates the callee parameter/load nodes for the calling context from the currently analyzed call. This map allows the analysis to interpret the actions of the callee in the context of the caller.

Section 4.4 contains a detailed presentation of the inter-procedural analysis.

Chapter 3

Program Representation

3.1 General Mathematical Notations

This thesis uses the following notations: "$\{a_0, a_1, \ldots, a_k\}$" represents the set of distinct elements $a_0, a_1, \ldots, a_k$, and $\emptyset$ denotes the empty set. For any set A, $\mathcal{P}(A)$ is the set of all subsets of A, i.e., $\mathcal{P}(A) = \{B \mid B \subseteq A\}$. "$[a_0, a_1, \ldots, a_k]$" is the list whose elements are, in order, $a_0, a_1, \ldots, a_k$. In a list, the order is important, and the same element can appear multiple times. "`list of` A" denotes the set of all the lists with elements from the set A. "$a : l$" is the list obtained by adding the element a at the head of list l, and "$l_1 @ l_2$" is the list obtained by appending list l_2 at the end of list l_1.

"$\{a_i \mapsto b_i\}_{i \in I}$" denotes a partial function f such that $f(a_i) = b_i, \forall i \in I$, and f is undefined in the other points; in particular, "$\{\}$" denotes a partial function that is not defined in any point. If $f : A \to B$ is a function from A to B, $a \in A$, and $b \in B$, "$f\,[a \mapsto b]$" denotes the function that has the value b in the point a, and behaves exactly like f in the other points of the domain A.

If $\langle L, \sqsubseteq \rangle$ is a lattice and $a, b \in L$, we write $a \sqsupseteq b$ iff $b \sqsubseteq a$; we write $a \sqsubset b$ iff $a \sqsubseteq b$ and $a \neq b$; similarly, $a \sqsupset b$ iff $a \sqsupseteq b$ and $a \neq b$.

If $\mu \subseteq A \times B$ is a relation between the sets A and B, and $a \in A$, then $\mu(a) = \{b \mid \langle a, b\rangle \in \mu\}$. If $S \subseteq A$, $\mu(S) = \bigcup_{a \in S} \mu(a)$. We write $a\ \mu\ b$ for $\langle a, b\rangle \in \mu$. Finally, μ^{-1} denotes the reverse relation: $\mu^{-1} = \{\langle b, a\rangle \mid \langle a, b\rangle \in \mu\}$.

3.2 Program Representation

We present our analysis in the context of SmallJava, a language similar to a small but non-trivial subset of the Java bytecode. It is easy to extend the analysis to handle most of the Java bytecode language. Section 8.1.2 explains how our implementation deals with the most complex features of Java (exceptions, reflection, dynamic loading, etc.)

Figure 3-1 presents the sets and the notations for the representation of the analyzed program. A program consists of a set of classes, *Class*, and a set of methods, *Method*. Each method has a list of parameters, a set of local variables, and

a body consisting of a list of instructions. Method m has $k = \mathit{arity}(m)$ parameters: $p_0, p_1, \ldots, p_{k-1}$. The first parameter p_0 is the "`this`" parameter. For simplicity, we consider that all parameters, local variables, object fields and return values have object type; the few integers that appear in our intermediate representation are all constants.

The execution of the program starts with the first instruction from the special method $m_{main} \in \mathit{Method}$. Each instruction from the body of a method m has a unique label $lb = \langle m, a \rangle$, where a is the index/address into m's list of instructions; the first instruction of method m has the label $\langle m, 0 \rangle$. $P(lb)$ denotes the instruction associated with the label lb. Each class $C \in \mathit{Class}$ has a set of fields $\mathit{fields}(C) = \{f_0, f_1, \ldots, f_{q-1}\}$. Some fields are static, i.e., attached to a class C, not to a specific instance of C; static fields act as global variables. We distinguish between static and non-static fields by using different instructions for manipulating them. The special field "[$*$]" cannot appear in a program; the analysis uses it internally to model array cells.

$$
\begin{array}{rcl}
C & \in & \mathit{Class} = \{C_0, C_1, \ldots\} \\
m & \in & \mathit{Method} = \{m_0, m_1 \ldots\} \\
s & \in & \mathit{MethodName} = \{\text{“foo”}, \text{“bar”}, \ldots\} \\
lb & \in & \mathit{Label} = \mathit{Method} \times \mathit{Address} \\
a & \in & \mathit{Address} = \mathbb{N} \\
P & : & \mathit{Label} \rightarrow \mathit{Instruction} \quad \text{(see Figure 3-2 for the list of instructions)} \\
f & \in & \mathit{Field} = \{f_0, f_1, \ldots\} \cup \{[*]\} \\
\mathit{fields} & : & \mathit{Class} \rightarrow \mathcal{P}(\mathit{Field}) \\
v, p & \in & \mathit{Var} = \{v_0, v_1, \ldots\} \cup \{p_0, p_1, \ldots\} \\
\mathit{next} & : & \mathit{Label} \rightarrow \mathit{Label} = \lambda\langle m, a \rangle.\langle m, a+1 \rangle \\
\mathit{arity} & : & \mathit{Method} \rightarrow \mathbb{N}
\end{array}
$$

Figure 3-1: Sets and notations for the program representation.

We suppose that prior to the analysis, the program is converted into an intermediate representation that contains only the instructions from Figure 3-2; e.g., we convert a complex instruction like "$v_2 = v_1.f_1.f_2$" into a sequence of two simple LOAD instructions: "$v_{\text{temp}} = v_1.f_1$" followed by "$v_2 = v_{\text{temp}}.f_2$". The informal semantics of our instructions is presented in the last column of Figure 3-2; we give the formal semantics in Section 6.1.

Normally, the intra-procedural control flow goes from label $lb = \langle m, a \rangle$ to label $\mathit{next}(lb) = \langle m, a+1 \rangle$. An IF instruction may alter this normal flow of control by branching to a specific address in the same method. CALL invokes a virtual method: "$v_R = v_0.s(\ldots)$" calls the method named s from the class C of the object pointed to

Instruction Name	Instruction Format	Informal Semantics
COPY	$v_1 = v_2$	copy one local variable into another
NEW	$v =$ `new` C	create a new object of class C; all fields of the new object are initialized to `null`
NEW ARRAY	$v =$ `new` $C[k]$	create an array of k references to objects of class C; all array cells are initialized to `null`
NULLIFY	$v =$ `null`	assign `null` to v
STORE	$v_1.f = v_2$	store a reference into an object field
STATIC STORE	$C.f = v$	store a reference into a static field
ARRAY STORE	$v_1[i] = v_2$	store a reference into an array cell
LOAD	$v_2 = v_1.f$	load a reference from an object field
STATIC LOAD	$v = C.f$	load a reference from a static field
ARRAY LOAD	$v_2 = v_1[i]$	load a reference from an array cell
IF	`if` $(\ldots)$ `goto` a_t	conditional transfer of control (the condition is irrelevant for our analysis; we just require it has no heap side effects)
CALL	$v_R = v_0.s(v_1, \ldots, v_j)$	call method named s of object pointed to by v_0
RETURN	`return` v	return from the currently executing method with the result v
THREAD START	`start` v	start the thread pointed to by v
NOP	`nop`, other instructions that do not manipulate pointers	

Figure 3-2: SmallJava instructions.

by v_0. The parameter passing semantics is call-by-value. Although we did not give any mechanism for calls to native methods, the analysis handles the more general case of *unanalyzable* calls, i.e., calls to methods whose code is unavailable or too expensive to analyze.

In Java, threads are instances of the `java.lang.Thread` class; a thread is started by calling a special native method (`java.lang.Thread.start()`) on the thread object. The body of the newly started thread is the `run` method of the thread object. Equivalently, in our language, we start a thread by executing the THREAD START instruction "`start` v." No thread object can be started twice.

The analysis does not interpret the explicit synchronization instructions (these instructions are treated as NOPs). Thread synchronization restricts the set of possible thread interleavings. As our analysis ignores these restrictions, the analysis results

are valid for a conservative superset of the possible program executions. This choice may affect the analysis precision, but preserves its correctness.

The *control flow graph* of a method m, denoted CFG_m, is a directed graph whose nodes are the labels of the instructions from the body of m. For every two labels lb_1 and lb_2 that might be consecutive on an execution path inside method m, there is an arc from lb_1 to lb_2. We add the special label $entry_m$ to guarantee that CFG_m contains an isolated entry point (no arc points to it); similarly, we add the special label $exit_m$ to guarantee that CFG_m has a single exit point. Given a label lb from a method m, $pred(lb)$ is the set of direct predecessors of lb in CFG_m, and $succ(lb)$ is the set of direct successors of lb in CFG_m.

Our analysis requires a *conservative static call graph* CG. Constructing a call graph for a language like Java is very delicate, because of the ubiquity of dynamic dispatch (i.e., virtual method invocations) and the dynamic loading of classes that are unknown at compile time. Describing a precise algorithm for call graph construction is beyond the scope of this thesis.

Instead, we assume that we already have a static call graph such that for any given CALL from label lb, either (1) the call graph reports that that CALL is unanalyzable (e.g., because it may invoke a method from an unknown dynamically-loaded class); or (2) $CG(lb)$ contains *all* methods that may be called by that CALL (and possibly some other methods). The particular call graph construction algorithm is irrelevant for the correctness of the analysis (as long as the call graph is conservative); still, the precision of the call graph is important for the precision and the running time of the analysis.[1] Section 8.1.2 contains a brief description of the call graph construction algorithm that we use in our implementation.

A program terminates when all its threads terminate. It is possible to have infinite executions. We did not mention anything about failure modes: we require our programs to treat exceptional situations explicitly; e.g., there is a null pointer check before each pointer dereferencing.

[1]For example, an imprecise call graph could report very large groups of (allegedly) mutually recursive methods. This imprecision increases the analysis execution time, because the analysis needs to iterate over larger sets of methods in order to reach a fixed-point.

Chapter 4

Pointer Analysis

This chapter provides a detailed description of our pointer analysis. We encourage the reader to read the example from Chapter 2 before reading this chapter.

Analysis Overview: Our pointer analysis processes individual methods from the analyzed program. For each program point inside a method m, the analysis computes a *points-to graph* that models the parts of the heap that the *analysis scope* accesses up to that point. The analysis scope consists of the method m plus the methods it transitively invokes using analyzable CALL instructions. A CALL instruction is analyzable only if the following two conditions hold: (1) The static call graph identifies all possible callees for that CALL; and (2) For each possible callee, the analysis can analyze the callee code, or the analysis user provides a manually-generated summary for that callee. Hence, a CALL to a native method is unanalyzable, unless the analysis user supplies a summary for that native method.[1] Additionally, the analysis can consider any CALL to be unanalyzable, in order to reduce the analysis scope; this operation sacrifices precision for speed, but preserves correctness.

Our analysis analyzes each method m without knowing its calling context, i.e., without knowing the objects transitively pointed to by the formal parameters at the start of m's execution. Instead, the analysis uses special constructs to abstract over the calling context.

Inter-procedurally, when the analysis of method m processes an analyzable CALL instruction, the analysis uses the points-to graph G_{callee} for the end of the callee as a summary of the callee's execution. The inter-procedural analysis uses this summary in conjunction with the calling context at the current CALL instruction. Therefore, inter-procedurally, our analysis propagates information from callees to callers.

The rest of this chapter is organized as follows. Section 4.1 formally defines the points-to graphs and the other analysis abstractions. Section 4.2 describes the intended meaning of the analysis results. This meaning is essential for understanding the correctness of the analysis applications. Section 4.3 presents the intra-procedural

[1]For precision reasons, our analysis implementation uses manually-generated summaries for several common native methods (see Section 8.1.2).

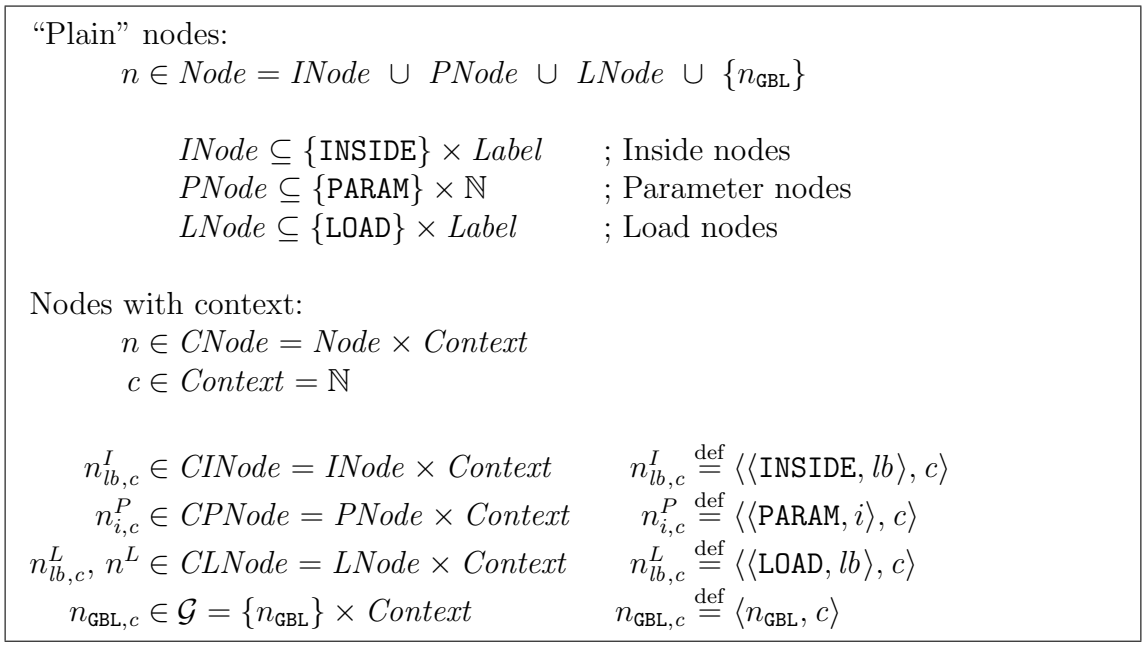


Figure 4-1: Pointer Analysis Abstractions - Part 1 of 2: Nodes.

analysis, while Section 4.4 presents the inter-procedural analysis. Section 4.5 discusses algorithms for computing the analysis results. Finally, Section 4.6 discusses several aspects of the analysis design.

4.1 Analysis Abstractions

Nodes: Our analysis models the potentially unboundedly many objects from the program execution using a statically bounded number of *nodes*.

Figure 4-1 presents the formal definitions for the analysis nodes. Our analysis uses an extension of the *object allocation site* model [18]. Our analysis uses one *inside* node to model *all* objects allocated at the same *allocation site*; an allocation site is a NEW / ARRAY NEW instruction.

Our analysis uses several other kinds of nodes in order to analyze methods without knowing their calling context. First, our analysis uses one *parameter* node for each formal parameter of the analyzed method; a parameter node models the object that the corresponding formal parameter points to.

Some LOAD instructions read references from fields of escaped objects, i.e., objects accessible from outside the analyzed scope. The analysis does not know what these fields point to (e.g., consider the case when we read a field of a parameter). Instead, for each label *lb* that corresponds to such a LOAD / ARRAY LOAD instruction, our analysis introduces a *load* node that models the objects read at label *lb* from fields of escaped objects.

The parameter/load nodes are essential for our ability to analyze the method

m without knowing its calling context. Intuitively, a parameter/load node n is a placeholder for the inside nodes associated with the objects that n models. For each call to method m, the inter-procedural analysis computes a node map that disambiguates these placeholders, according to the current calling context.

The special *global* node $n_{\texttt{GBL}}$ models objects that are read from a static field and objects returned by unanalyzable CALLs. We use the term "global" because these objects may be accessed by the entire program.

A node may model multiple objects: e.g., consider an inside/load node for an instruction inside a loop. Similarly, several nodes may model the same objects: e.g., if several formal parameters point to the same object, the corresponding parameter nodes model the same object.

Sometimes in the analysis and in the correctness proof, we need to distinguish between several versions of the same node. For this purpose, we use *nodes with context*. A node with context is a pair $\langle n, c\rangle$ of a node n and a numeric *context* $c \in \mathit{Context} = \mathbb{N}$. We use the term "node" for both plain nodes and nodes with context; the distinction is usually clear from the context.

Notation-wise, $n^I_{lb,c}$ denotes the inside node for the allocation site from label lb, with context c. $n^P_{i,c}$ denotes the parameter node for the i-th formal parameter, with context c. $n^L_{lb,c}$ denotes the load node for the LOAD instruction from label lb, with context c; occasionally, we write n^L to denote a generic load node. $n_{\texttt{GBL},c}$ is the special node $n_{\texttt{GBL}}$ with context c.

General Points-to Graphs: Figure 4-2 presents the formal definitions for the points-to graph abstraction. A general points-to graph $G \in \mathit{PTGraph}$ is a tuple $G = \langle L : J, I, O, E, R\rangle$, consisting of an *abstract stack* $L : J$ with at least one element, a set I of *inside* edges, a set O of *outside* edges, a set E of *directly globally escaped* nodes, and a set R of *returned nodes*.

The *abstract stack* $L : J$ models the state of the local variables. The symbol ":" denotes the addition of an element at the head of a list (see Section 3.1); hence, $L : J$ is a stack with the top element L and the stack tail J. In the points-to graphs that the analysis manipulates, the abstract state has exactly one element, i.e. $J = []$. The only element, the *abstract local variable state* L, maps each local variable v to the set $L(v)$ of nodes; these nodes model the objects that v may point to during program execution. In the special abstract local variable state $L_{\text{all–empty}}$, each local variable v points to an empty set of nodes.

During the analysis correctness proof (Chapter 6), we define an *abstract semantics* that uses multi-element abstract stacks in order to model the values of the callee variables.

Example 1. Consider a method m with two parameters `p0` and `p1`. In the points-to graph that the analysis computes for the beginning of m, the abstract stack is $L : []$, where

$$L = L_{\text{all–empty}}\left[\texttt{p0} \mapsto \{n^P_{0,0}\},\ \texttt{p1} \mapsto \{n^P_{1,0}\}\right]$$

General points-to graphs:

$$G \in PTGraph = \{\ \langle L : J, I, O, E, R\rangle \ \mid\ L : J \in AStack;\ I \in IEdges;\ O \in OEdges;\ E, R \in \mathcal{P}(Node)\ \}$$

$J \in AStack = \texttt{list of}\ ALocVar$; Abstract stacks

$L \in ALocVar = Var \to \mathcal{P}(Node)$; Abstract local variable states

$L_{\text{all-empty}} = \lambda v.\emptyset$

$I \in IEdges = \mathcal{P}(CNode \times Field \times CNode)$; Sets of inside edges

$O \in OEdges = \mathcal{P}(CNode \times Field \times CLNode)$; Sets of outside edges

Analysis points-to graphs:

$$PTGraph^a = \{\ G \in PTGraph \ \mid\ G = \langle L : [], I, O, E, R\rangle, \forall \langle n, c\rangle \in nodes(G).\ c = 0\ \}$$

Set of nodes that appear in a node-based structure:

$$nodes(G = \langle J, I, O, E, R\rangle) = nodes(J) \cup nodes(I) \cup nodes(O) \cup E \cup R$$

$$nodes(J = [L_1, \ldots, L_k]) = \bigcup_{i=1}^{k} nodes(L_i) \qquad nodes(L) = \bigcup_v L(v)$$

$$nodes(I) = \bigcup_{\langle n_1, f, n_2\rangle\ \in\ I} \{n_1, n_2\} \qquad nodes(O) = \bigcup_{\langle n, f, n^L\rangle\ \in\ O} \{n, n^L\}$$

Figure 4-2: Pointer Analysis Abstractions - Part 2 of 2: Points-To Graphs.

Recall from Section 3.1 that the notation $f\,[a_1 \mapsto b_1, \ldots, a_k \mapsto b_k]$ denotes a function that is equal to f in any point, except that $f(a_i) = b_i, \forall i \in \{1, 2, \ldots k\}$. Therefore, in L, parameter `p0` points to the parameter node $n^P_{0,0}$, `p1` points to $n^P_{1,0}$, and other local variables do not point to any node. Both parameter nodes have context 0. △

The *inside edges* from I model the heap references created by the analyzed scope: the inside edge $\langle n_1, f, n_2\rangle$ models the fact that the field f of an object that n_1 models may point to an object that n_2 models.

The *outside edges* from O model read actions of the analyzed scope: the outside edge $\langle n, f, n^L_{lb}\rangle$ models the fact that, at label lb, the analyzed scope reads the f field of an object that (1) n models, and (2) is reachable from outside the analyzed scope. An outside edge always ends in a load node. We use the outside edges in the inter-procedural analysis (Section 4.4).

Arrays are just a special kind of objects; hence, the analysis uses nodes to model the array objects. If an array has elements of a non-primitive type, the values stored in the array cells are addresses of objects. We use edges to represent these heap references. We do not distinguish between individual array cells: the special field $[*]$

represents *all* cells of an array.

The fourth component of a points-to graph, the set E of *directly globally escaped nodes* contains: (1) nodes that are stored in static fields, (2) nodes that correspond to started threads, and (3) nodes passed as arguments to unanalyzable CALLs. These nodes model objects that are potentially reachable from the entire program (hence the name "globally escaped").

The last component of a points-to graph, the set R, contains the nodes that may have been returned from the analyzed method m. This information is used by the inter-procedural analysis while processing calls to method m. This component is empty for almost all points-to graph, except those from the end of a method.[2]

The function *nodes* takes a node-based structure and returns the set of nodes that appear inside that structure: e.g., $nodes(I)$ returns the set of nodes that appear as endpoints of the inside edges from I, $nodes(G)$ returns the set of nodes that appear in the points-to graph G, etc. The bottom part of Figure 4-2 contains the straightforward definition of *nodes*.

Escape information: Intuitively, a node *escapes* if some of the objects it models may be reachable from outside the analyzable scope. We have already encountered several nodes that escape: the nodes from the set E of directly globally escaped nodes and the nodes from the set R of returned nodes (obviously reachable from the caller). Other escaped nodes include the parameter nodes, the load nodes (they model objects read from objects reachable from outside the analyzed scope) and the global nodes $n_{\texttt{GBL},c} \in \mathcal{G}$ (nodes of the form $n_{\texttt{GBL},c}$ model objects returned from unanalyzable CALLs and/or read from a static field). In addition, escapability propagates along edges: nodes reachable from an escaped node escape too. More rigorously, we use the following definitions:

Definition 1 (General graph reachability). *Consider a general directed graph (not necessarily a points-to graph); if R is a set of "root" vertices, and A is a set of arcs, then $reachable(R, A)(v)$ is true iff there exists a (possibly empty) path of arcs from A that starts in a vertex from R and ends in the vertex v.*

Definition 2 (Escape predicate). *Given a points-to graph $G = \langle L : J,\ I,\ O,\ E,\ R \rangle$, we define the escape predicate on nodes, $e(G) : CNode \to \{\texttt{true}, \texttt{false}\}$, as follows:*

$$e(G)(n) = reachable(CPNode \cup CLNode \cup \mathcal{G} \cup E \cup R,\ I)(n)$$

$e(G)(n)$ checks whether there exists a (possibly empty) path of inside edges that reaches n from (1) a parameter node, (2) a load node, (3) a global node from $\mathcal{G}$,

[2]Hence, we need only one set of returned nodes, for the end of the method, instead of one for each program point. We use the current flow-sensitive formalism due to its uniformity. Our implementation uses only one set of returned nodes per method.

(4) a globally escaped node, or (5) from a returned node. A node n escapes from G *iff $e(G)(n)$ is true. Otherwise, n is* captured in G.

Notice that the predicate $e(G)$ does not use all components of G: it uses only I, E and R. Moreover, for all points-to graphs except those for the end of a method, R is empty. This motivates us to define a simplified escape predicate, that we use later in the analysis presentation and in the correctness proof:

Definition 3 (Simplified escape predicate).

$$e_2(E, I)(n) = \mathit{reachable}(\mathit{CPNode} \cup \mathit{CLNode} \cup \mathcal{G} \cup E,\ I)(n)$$

If $e_2(E, I)(n)$ holds, we say that n escapes according to $\langle E, I\rangle$.

Clearly, for any points-to graph G of the form $\langle L : J,\ I,\ O,\ E,\ \emptyset\rangle$, $e(G) = e_2(E, I)$.

Analysis Points-to Graphs: The pointer analysis computes only *analysis points-to graphs*. Analysis points-to graphs are general points-to graphs with the following two simplifications:

1. The abstract stack has only one element, i.e., G has the form $\langle L : [], I, O, E, R\rangle$.
2. All nodes that appear in G have context 0.

$\mathit{PTGraph}^a \subseteq \mathit{PTGraph}$ denotes the set of analysis points-to graphs.[3] We define an order relation between analysis points-to graphs, using standard techniques from lattice theory: the order between tuples is defined component-wise,[4] the order between functions is defined element-wise,[5] and the order between sets is the inclusion order.

Definition 4 (Order relation between analysis points-to graphs).

$$\begin{aligned}&\langle L_1 : [], I_1, O_1, E_1, R_1\rangle \sqsubseteq \langle L_2 : [], I_2, O_2, E_2, R_2\rangle \ \overset{\text{def}}{\leftrightarrow}\\ &\qquad L_1 \sqsubseteq L_2\ \wedge\ I_1 \subseteq I_2\ \wedge\ O_1 \subseteq O_2\ \wedge\ E_1 \subseteq E_2\ \wedge\ R_1 \subseteq R_2\\ &\textit{where}\ \ L_1 \sqsubseteq L_2 \ \overset{\text{def}}{\leftrightarrow}\ \ \forall v \in \mathit{Var}.\ L_1(v) \subseteq L_2(v)\end{aligned}$$

By standard lattice theory [66, Appendix A.2], $\langle \mathit{PTGraph}^a, \sqsubseteq\rangle$ is a lattice, with the associated join operation:

$$\begin{aligned}&\langle L_1 : [],\ I_1,\ O_1,\ E_1,\ R_1\rangle\ \sqcup\ \langle L_2 : [],\ I_2,\ O_2,\ E_2,\ R_2\rangle =\\ &\qquad \langle (L_1 \sqcup L_2) : [],\ I_1 \cup I_2,\ O_1 \cup O_2,\ E_1 \cup E_2,\ R_1 \cup R_2\rangle\end{aligned}$$

where

$$L_1 \sqcup L_2 = \lambda v.\ L_1(v) \cup L_2(v)$$

The least element of $\mathit{PTGraph}^a$ is $\bot_{\mathit{PTGraph}^a} = \langle L_{\text{all-empty}} : [],\ \emptyset,\ \emptyset,\ \emptyset,\ \emptyset\rangle$.

[3] Not surprisingly, the superscript a stands for "analysis".
[4] I.e., $\langle a_1, a_2, \ldots, a_k\rangle \sqsubseteq \langle b_1, b_2, \ldots, b_k\rangle$ iff $a_i \sqsubseteq b_i$, $\forall i$.
[5] I.e., $f \sqsubseteq g$ iff $\forall x. f(x) \sqsubseteq g(x)$.

The size of an analysis scope is the number of instructions from the scope plus the maximum number of parameters of a method from the scope. Consider an analysis scope of size N. The number of plain nodes (i.e., nodes without context) is $\mathcal{O}(N)$: there is at most one inside or load node for each label from the program, P parameter nodes, where $P < N$ is the maximum number of parameters of a method, and the special node $n_{\texttt{GBL}}$. Therefore, the number of nodes with context 0 is $\mathcal{O}(N)$. The number of local variables and fields used by the analysis scope is also $\mathcal{O}(N)$. Hence, we can prove the following lemma:

Lemma 1. *For each analysis scope of size N, the lattice $\langle PTGraph^a, \sqsubseteq \rangle$ has depth $\mathcal{O}(N^3)$: any strictly ascending chain $G_1 \sqsubset G_2 \sqsubset \ldots$ has length $\mathcal{O}(N^3)$.*

Proof. As $G_i \sqsubset G_{i+1}$, G_{i+1} contains all elements of G_i plus (at least) a new node pointed to by a variable, a new inside edge, a new outside edge, a new globally escaped node or a new returned node. There are $\mathcal{O}(N^3)$ possible inside/outside edges, $\mathcal{O}(N)$ nodes, and $\mathcal{O}(N^2)$ pairs of variables and nodes. Hence, the length of any strictly ascending chain in $PTGraph^a$ is $\mathcal{O}(N^2 + N^3 + N^3 + N + N) = \mathcal{O}(N^3)$. □

We use this lemma to prove the termination of our analysis in Section 4.5.

4.2 Intended Meaning of the Analysis Results

This section presents the intended meaning of the points-to graphs that the analysis computes. The style of this section is informal; we formalize our ideas during the correctness proof from Chapter 6, when we prove that the analysis results satisfy the properties presented in this section. The list of properties we present is not exhaustive: for brevity, we focus on those properties that are important for the correctness of the two analysis applications that we present later in this thesis: the stack allocation optimization and the purity analysis.

Consider a method m and let lb be either (1) a label inside m's code, or (2) the special label $exit_m$ for the end of m. In each case, we select a program state and an *activation*, i.e., an execution of m: In the first case, we consider one of the possible states of the program right before executing the instruction from label lb. Let $A(m)$ be the current activation of method m; in case of recursion, we take the inner-most activation. In the second case, we consider one program state right after an activation $A(m)$ terminates.

The activation $A(m)$ contains all instructions executed between the CALL that starts $A(m)$ and the RETURN that finishes $A(m)$, including the instructions from all transitive callees invoked through analyzable CALLs.

We say that an object o *escapes* from $A(m)$ if o is reachable from outside $A(m)$: from a static field, from the caller, from an unanalyzed method, or from the local variables from the execution stack of a parallel thread. An object is *captured* in $A(m)$ if it does not escape.

Let $G = \langle L : [], I, O, E, R \rangle = \circ A(lb)$ be the points-to graph that the analysis computes for the program point before lb.

With these notations, our analysis guarantees that there exists a *modeling relation* between nodes and objects $\rho \subseteq CNode \times Object$ such that the following properties hold:

Property 1. *If $lb \neq exit_m$, then the abstract state of local variables L conservatively models the state of the local variables of method m: if in the program state, a local variable v of m points to an object o, then there exists a node n such that n models o (i.e., $n \ \rho \ o$) and $n \in L(v)$.*

The case $lb = exit_m$ is irrelevant for the property above, because after the end of $A(m)$, the local variables of m no longer exist.

Property 2. *The set I of inside edges conservatively models all heap edges created by $A(m)$: if o_1 and o_2 are reachable objects and $A(m)$ created the heap edge $\langle o_1, f, o_2 \rangle$, then there exists nodes n_1 and n_2 such that n_1 models o_1 (i.e., $n_1 \ \rho \ o_1$), n_2 models o_2 (i.e., $n_2 \ \rho \ o_2$), and $\langle n_1, f, n_2 \rangle \in I$.*

The above property refers only to edges between objects that are reachable in the program state. For efficiency purposes, the analysis reserves the right to ignore the references between objects that the program can no longer access.[6]

Property 3. *Any inside node $n^I_{lb,0}$ models only objects allocated by $A(m)$ by executing the NEW / ARRAY NEW instruction from label lb.*

Property 4. *Let o be an object allocated by $A(m)$ by executing the NEW / ARRAY NEW instruction from label lb. If the inside node $n^I_{lb,0}$ is captured, then*

1. *The object o is captured in $A(m)$.*
2. *The inside node $n^I_{lb,0}$ is the only node that models the object o.*

The modeling relation ρ is specific to the program state we selected: we may have different modeling relations for distinct program states. A node may model several objects: e.g., the inside node for a NEW inside a loop models all objects allocated there by $A(m)$. Conversely, several nodes may model the same object (e.g., two parameter nodes may model the same object, in the case of aliased arguments); as a consequence, the information about an object may be "scattered" among different nodes. For example, suppose the local variable v points to the object o_1, whose field f points to o_2. By Property 1, there exists a node n_1 such that n_1 models o_1 and v points to n_1 in L. By Property 2, there exists nodes n'_1, n_2 such that n'_1 models o_1, n_2 models o_2, and $\langle n'_1, f, n_2 \rangle \in I$. However, there is no guarantee that $n_1 = n'_1$! In general, reachability in the program state (e.g., in our example, reachability of the object o_2 from the local variable v), does not translate into reachability in the points-to graphs that the analysis computes.

Fortunately, Part 2 of Property 4 guarantees that if o is an object allocated by $A(m)$ at the allocation site from label lb and the corresponding inside node $n^I_{lb,0}$ is captured, then all information about o is concentrated at the level of $n^I_{lb,0}$; the information about the other objects may be distributed among different nodes.

[6] In a Java Virtual Machine, such objects are eventually deallocated, modulo the imprecision of the garbage collector.

We use the above properties to prove the correctness of our two analysis applications: Part 1 of Property 4 guarantees us that if $n^I_{lb,0}$ is captured in the points-to graph for the end of m, then all objects allocated by $A(m)$ by executing the NEW / ARRAY NEW instruction from lb are unreachable from outside $A(m)$. Hence, the program cannot access these objects after the end of $A(m)$. Therefore, we can change the instruction from lb to allocate objects on the stack, instead of allocating them in the garbage-collected heap.

Property 3 is directly relevant for the purity analysis. As an inside node models only objects allocated by $A(m)$, i.e., objects that did not exist when $A(m)$ started, we can ignore mutations on inside nodes. Hence, Property 3 increases the flexibility of our purity analysis: we support pure methods that allocate and mutate new objects, as long as they do not mutate "old" objects allocated before the invocation of the method.

Properties 1 and 2 are useful for understanding the points-to relation abstracted by the abstract state of local variables L and the set of inside edges I. In addition, during the correctness proof from Chapter 6, Properties 1 and 2 are useful for proving Properties 3 and 4.

4.3 Intra-procedural Analysis

The analysis of method m computes an analysis points-to graph for each program point inside method m. More specifically, for each label lb from method m, the analysis of method m computes the analysis points-to graph $\circ A(lb) \in PTGraph^a$ for the program point right before lb, and the analysis points-to graph $A\circ(lb) \in PTGraph^a$ for the program point right after lb.[7] During the analysis of a method m, the scope of the analysis is the method m plus the methods it transitively invokes using analyzable CALLs.

We express the analysis of method m as a set of forward dataflow constraints:

$$\begin{aligned} \circ A(entry_m) &\sqsupseteq G^m_{\text{init}} \\ \circ A(lb) &\sqsupseteq \bigsqcup\{A\circ(lb') \mid lb' \in pred(lb)\} \\ A\circ(lb) &\sqsupseteq [\![lb]\!]^a(\circ A(lb)) \end{aligned} \tag{4.1}$$

Intuitively, the analysis points-to graph for the beginning of the method m is (at least) G^m_{init}. In control-flow join points, the analysis joins the points-to graphs for the predecessors using the join operator $\sqcup$ for the lattice of analysis points-to graphs $\langle PTGraph^a, \sqsubseteq \rangle$ (see Section 4.1). The transfer function $[\![lb]\!]^a$ propagates information across the instruction from label lb. There is one transfer function for each program label. The transfer function $[\![lb]\!]^a$ takes as argument the points-to graph for the

[7] We use the symbol "$\circ$" to indicate the position of the program point relative to the label lb: right before or right after.

program point before label lb and returns the points-to graph for the program point after lb.

Constraints 4.1 use inequality instead of equality because upper approximations (i.e., bigger points-to graphs) preserve the analysis properties from Section 4.2. Additionally, inequality constraints allow additional flexibility to the constraint solver; e.g., the above constraints have a solution even for non-monotonic transfer functions (see Section 4.5.1).

The points-to graph for the beginning of m is:

$$G^m_{\text{init}} = \langle\ (L_{\text{all–empty}}\left[p_i \mapsto \{n^P_{i,0}\}\right]_{0\le i\le k-1}) : [],\ \emptyset,\ \emptyset,\ \emptyset,\ \emptyset\ \rangle$$

where $p_0, p_1, \ldots, p_{k-1}$ are the k parameters of m. Each parameter p_i points to the corresponding parameter node n^P_i; G^m_{init} is otherwise empty: no inside/outside edges, globally escaped nodes, etc.

Instead of directly describing the analysis transfer functions, we first introduce a generalization of them, the *abstract semantics transfer functions*. Unlike the analysis transfer functions $[\![.]\!]^a$: $Label \to PTGraph^a \to PTGraph^a$, that work only with analysis points-to graphs (points to graphs with a single-element abstract state), the abstract semantics transfer functions $[\![.]\!]$: $Label \to PTGraph \to PTGraph$ work with general points-to graphs, including points-to graphs whose abstract stack has more than one element. The analysis transfer functions are just a special case of the abstract semantics transfer functions. The correctness proof (Chapter 6) defines and uses an *abstract semantics* that is very similar to the analysis, and we wanted to avoid having two sets of almost identical definitions.

We define the intra-procedural analysis transfer functions as the restriction of the abstract semantics transfer functions to the set $PTGraph^a$. More specifically,

$$[\![lb]\!]^a(G) = \begin{cases} [\![lb]\!](G), \text{ if the instruction from } lb \text{ is not an analyzable CALL.} \\ \textit{the case of analyzable CALL presented later in Sec. 4.4} \end{cases}$$

We prove later in this section that this definition respects the type signature of $[\![lb]\!]$, i.e., if $G \in PTGraph^a$ and lb is the label of an instruction that is not an analyzable CALL, then $[\![lb]\!](G) \in PTGraph^a$.

Figure 4-3 presents the definition of the abstract semantics transfer functions, on a case by case basis, based on the kind of the instruction from label lb. During the pointer analysis, the abstract stack tail J for the argument points-to graph is always empty: $J = []$; equivalently, $|J| = 0$. Hence, all nodes explicitly mentioned in Figure 4-3 (e.g., the inside node $n^I_{lb,|J|}$ for a NEW instruction) have context 0.

Figure 4-4 presents a graphic representation of several transfer functions.

As a general rule, assignments to variables are *destructive*, i.e., assigning something to v "removes" all previous values of $L(v)$, while assignments to node fields are *non-*

$[\![.]\!] : \mathit{Label} \to \mathit{PTGraph} \to \mathit{PTGraph}$

$[\![\mathit{entry}_m]\!]$ and $[\![\mathit{exit}_m]\!]$ are the identity function; see other cases below:

$P(lb)$	$[\![lb]\!](G = \langle L : J,\ I,\ O,\ E,\ R\rangle)$
$v_1 = v_2$	$\langle L\,[v_1 \mapsto L(v_2)] : J,\ I,\ O,\ E,\ R\rangle$
$v = \texttt{new}\ C$	$\langle L[v \mapsto \{n^I_{lb,\lvert J\rvert}\}] : J,\ I,\ O,\ E,\ R\rangle$
$v = \texttt{new}\ C[k]$	$\langle L[v \mapsto \{n^I_{lb,\lvert J\rvert}\}] : J,\ I,\ O,\ E,\ R\rangle$
$v = \texttt{null}$	$\langle L[v \mapsto \emptyset] : J,\ I,\ O,\ E,\ R\rangle$
$v_1.f = v_2$	$\langle L : J,\ I \cup (L(v_1) \times \{f\} \times L(v_2)),\ O,\ E,\ R\rangle$
$v_1[i] = v_2$	$\langle L : J,\ I \cup (L(v_1) \times \{[*]\} \times L(v_2)),\ O,\ E,\ R\rangle$
$C.f = v$	$\langle L : J,\ I,\ O,\ L,\ E \cup L(v),\ R\rangle$
$v_2 = v_1.f$	$\mathit{process\_load}(G,\ v_2,\ v_1,\ f,\ lb)$ (see Fig. 4-5)
$v_2 = v_1[i]$	$\mathit{process\_load}(G,\ v_2,\ v_1,\ [*],\ lb)$ (see Fig. 4-5)
$v = C.f$	$\langle L[v \mapsto \{n_{\texttt{GBL},\lvert J\rvert}\}] : J,\ I,\ O,\ E,\ R\rangle$
$\texttt{if}\ (\ldots)\ \texttt{goto}\ a_t$	$\langle L : J,\ I,\ O,\ L,\ E,\ R\rangle$ (unmodified)
$\texttt{start}\ v$	$\langle L : J,\ I,\ O,\ E \cup L(v),\ R\rangle$
$\texttt{nop}$	$\langle L : J,\ I,\ O,\ L,\ E,\ R\rangle$ (unmodified)
$v_R = v_0.s(v_1, \ldots, v_j)$	1. unanalyzable CALL $\langle L[v_R \mapsto n_{\texttt{GBL},\lvert J\rvert}] : J,\ I,\ O,\ E \cup \bigcup_{i=0}^{j} L(v_i),\ R\rangle$ 2. analyzable CALL — presented later in Fig. 6-5.
$\texttt{return}\ v$	1. if $J = [],\ \langle L : [],\ I,\ O,\ E,\ L(v)\rangle$ 2. otherwise — presented later in Fig. 6-5.

Figure 4-3: Definition of abstract semantics transfer functions $[\![lb]\!], lb \in \mathit{Label}$. The transfer functions $[\![lb]\!]^a$ from the intra-procedural part of the pointer analysis are restrictions of the abstract semantics transfer functions $[\![lb]\!]$ to analysis points-to graphs. During the analysis, J is always empty (equivalently, $|J| = 0$): the points-to graphs manipulated by the analysis have exactly one stack frame, corresponding to the state of the local variables from the analyzed method.

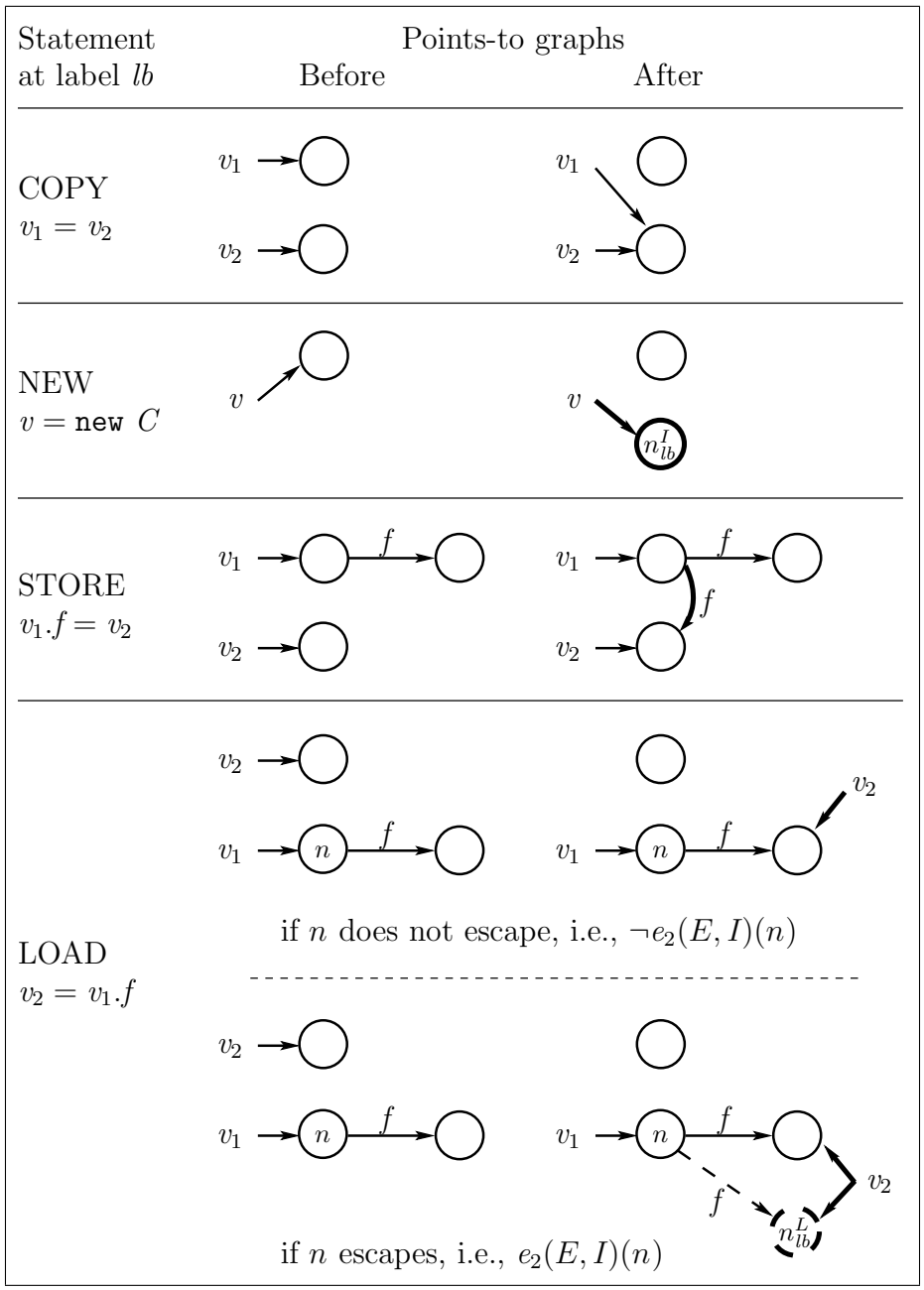


Figure 4-4: Graphic representation of several intra-procedural transfer functions. Circles represent general nodes, dashed circles represent load and parameter nodes, solid arrows represent inside edges, and dashed arrows represent outside edges. Bold circles and arrows indicate potentially new nodes and edges. All nodes have context 0; for brevity, we omit this constant context.

$$
\begin{array}{l}
\mathit{process\_load}\colon \mathit{PTGraph} \times \mathit{Var} \times \mathit{Var} \times \mathit{Field} \times \mathit{Label} \to \mathit{PTGraph} \\
\quad = \lambda\langle\langle L : J,\ I,\ O,\ E,\ R\rangle,\ v_2,\ v_1,\ f,\ lb\rangle. \\
\quad\quad \textbf{let}\ \ A = \{n \in \mathit{Node} \mid \exists n_1 \in V(v_1), \langle n_1, f, n\rangle \in I\} \\
\quad\quad\quad\ \ B = \{n \in V(v_1) \mid e_2(E, I)(n)\}\ \ \textbf{in} \\
\quad\quad\ \textbf{if}\ (B = \emptyset) \\
\quad\quad\quad\ \textbf{then}\ \langle L\,[v_2 \mapsto A] : J,\ I,\ O,\ E,\ R\rangle \\
\quad\quad\quad\ \textbf{else}\ \ \langle L\left[v_2 \mapsto \left(A \cup \{n^L_{lb,|J|}\}\right)\right] : J,\ I,\ O \cup (B \times \{f\} \times \{n^L_{lb,|J|}\}),\ E,\ R\rangle \\
\quad\quad\ \textbf{fi}
\end{array}
$$

Figure 4-5: Definition of the function *process_load*. Its arguments are, in order, the points-to graph before the load ($G = \langle L : J,\ I,\ O,\ E,\ R\rangle$), the variable v_2 we load into, the variable v_1 we load from, the loaded field f, and the label lb of the LOAD instruction "$v_2 = v_1.f$." It returns the points-to graph after the instruction.

destructive:[8] assigning something to $n_1.f$ does not remove the existing edges that start from n_1. The reason is that a node might represent multiple objects and so, updating $n_1.f$ might not overwrite the edge $\langle n_1, f, n_2\rangle$ because the update instruction and the edge might concern different objects.

The two special labels $entry_m$ and $exit_m$ do not correspond to any concrete instruction. The transfer function for them is naturally the identity function. This is also the case for the labels that correspond to IF, NOP, or any other instruction that does not manipulate pointers.

A COPY instruction "$v_1 = v_2$" makes v_1 point to all nodes that v_2 points to. As previously mentioned, the analysis "forgets" the previous value of $L(v_1)$. The transfer function for a label lb that corresponds to a NEW instruction "$v =$ `new` C" makes v point to the inside node attached to the label lb, $n^I_{lb,0}$. The case of ARRAY NEW is identical. A NULLIFY instruction "$v =$ `null`" sets v to point to an empty set of nodes.

For a STORE instruction "$v_1.f = v_2$," the analysis introduces an f-labeled inside edge between each node pointed to by v_1 and each node pointed to by v_2. The case of an ARRAY STORE instruction "$v_1[i] = v_2$" is similar, except that we use the special field "$[*]$" that models the references coming from all the cells of the array. For a STATIC STORE "$C.f = v$," we add all nodes pointed to by v to the set of globally escaped nodes E.

The transfer function for a LOAD instruction "$v_2 = v_1.f$" uses the auxiliary function *process_load* from Figure 4-5. Before the instruction, v_1 points to the nodes from the

[8] An equivalent term is *weak updates*; the opposite term, *strong updates*, denotes the updates that remove the previous edges.

set $L(v_1)$; some of these nodes are the starting points for f-labeled inside edges. Let A be the set of target nodes of these inside edges. After the instruction, v_2 points to all nodes from A.

In addition, if we load from nodes that escape the analyzed scope, i.e., $B \neq \emptyset$ in Figure 4-5, v_2 also points to the load node $n^L_{lb,0}$. In this case, for each escaped node n we load from, the analysis introduces an f-labeled outside edge from n to $n^L_{lb,0}$. Later, when we analyze calls to m, we use these outside edges to detect the nodes that the placeholder $n^L_{lb,0}$ stands for. To identify the nodes that escape the analyzed scope, we use the predicate $e_2(E, I)$. The transfer function for an ARRAY LOAD instruction is similar to the one for a LOAD, except that it uses the special field $[*]$.

A STATIC LOAD "$v = C.f$" sets v to point to the node $n_{\texttt{GBL},0}$, a node that models unknown objects that may be accessed from the entire program.

We describe the transfer function for an analyzable CALL when we present the inter-procedural analysis in Section 4.4. An unanalyzable CALL "$v_R = v_0.s(v_1, \ldots, v_j)$" makes its argument objects reachable from unanalyzed parts of the program. Therefore, the analysis adds all nodes pointed to by $v_0, \ldots, v_j$ to E, the set of globally escaped nodes. Also, in the points-to graph after the unanalyzable CALL, we set v_R to point to the node $n_{\texttt{GBL},0}$. Similarly, for a START THREAD instruction "`start` v," the analysis adds all nodes pointed to by v to E.

Figure 4-3 contains two cases for a RETURN instruction "`return` v." The pointer analysis works only with points-to graphs with abstract stacks of length one. Hence, it suffices to examine the first case: $J = []$. The transfer function records the fact the method returns the nodes pointed to by v, i.e., the nodes from the set $L(v)$.

Example 2. We encourage the reader to revisit the example from Section 2.2.1. That example does not take node contexts into account; the reader can assume that all missing node contexts are 0. The reader can also ignore the sets W of mutated abstract fields from that example; we discuss purity analysis in Section 5.2. △

The following lemma proves that for each label lb that does not correspond to an analyzable CALL, our definition of the analysis transfer function $[\![lb]\!]^a = [\![lb]\!]$ respects its signature: if given an analysis points-to graph, $[\![lb]\!]^a$ returns an analysis points-to graph too. In the next section, we extends this lemma to cover the analyzable CALLs too, thus proving that the pointer analysis works only with analysis points-to graphs.

Lemma 2. *Consider an arbitrary label $lb \in Label$ that does not correspond to an analyzable CALL, and an arbitrary analysis points-to graph $G \in PTGraph^a$. Then, $[\![lb]\!](G) \in PTGraph^a$ too.*

Proof. Case analysis on the kind of the instruction from label lb, followed by a simple inspection of the definitions from Figure 4-3. We notice that (1) none of the instructions increases the size of the stack, and (2) the nodes from $[\![lb]\!](G)$ are either nodes that appear in G or new, explicitly mentioned nodes with context $|J|$, where J is the empty tail of the abstract stack from $G \in PTGraph^a$; hence, all nodes from $[\![lb]\!](G)$ have context 0. □

4.4 Inter-procedural Analysis

This section presents the transfer functions for the labels that correspond to analyzable CALLs. Consider a CALL instruction at label lb, and let G be the points-to graph for the program point right before the CALL. The analysis needs to compute $[\![lb]\!]^a(G)$, the points-to graph for the program point right after the CALL.

4.4.1 Intuition

Method Summaries: One naive solution for the inter-procedural analysis is to inline the callee(s) and to perform the intra-procedural analysis. Besides having problems with recursive methods, this solution is very inefficient even for non-recursive methods, as it may inline and re-analyze a method multiple times. Instead, our analysis computes a single *method summary* for each method, and uses this summary for all CALLs that may invoke that method. Intuitively, the summary of a method is a pre-computed, simplified version of the pointer analysis of an inlined copy of the method. We discuss the advantages of method summaries in more detail in Section 4.6.1.

Let *callee* be one of the methods that the CALL from label lb may invoke.[9] The summary of *callee* should allow us to compute the points-to graph after the CALL, in the case when *callee* is invoked: the new inside and outside edges, the new globally escaped nodes, and the nodes that v_R points-to after the CALL.

Our idea is to compute *callee*'s summary based on the points-to graph G_{callee} for the end of *callee*. Notice that G_{callee} is not only an abstraction of the heap manipulated by the execution of *callee*, but also an abstraction of the execution of *callee*: the inside edges abstract the STORE instructions, the outside edges abstract the LOAD instructions, and the set of globally escaped nodes abstracts all instructions that globally escape nodes (e.g., STATIC STOREs, unanalyzable CALLs, etc.). Finally, G_{callee} contains information about the nodes that may be returned from *callee*.

The core of *callee*'s summary is an *inter-procedural transformer* that "executes" the instructions of *callee* on the points-to graph G before the CALL. To construct this transformer, the analysis assigns an atomic transformer to each inside edge, each outside edge, and each globally escaped node from G_{callee}. Next, the analysis combines these atomic transformers into a transformer for the entire *callee*.

Inter-procedural Node Maps: The instructions of *callee* may involve parameter and load nodes, which are placeholders for nodes from the calling context. To interpret these instructions for a particular CALL site, the analysis needs to know what the callee parameter / load nodes stand for. E.g., assume that G_{callee} contains an inside edge $\langle n_1, f, n_2 \rangle$, where n_1 and n_2 may be parameter/load nodes. This inside edge corresponds to a STORE instruction executed by *callee*; to understand what inside

[9] Due to dynamic dispatch, different executions of the same CALL may invoke different methods. A static analysis like ours must consider all possible callees.

edges this instruction may create in the caller, the analysis needs to know the nodes that n_1 and n_2 may represent.

To address this problem, each transformer maintains a map between *callee*'s nodes and the nodes they stand for:

$$\mu \in Map = \mathcal{P}(CNode \times CNode)$$

At the beginning of *callee*, each parameter node is mapped to the nodes pointed to by the corresponding argument. Some atomic transformers may add new node mappings for the load nodes. The analysis always maps inside and global nodes to themselves: in our transformers, inside/global nodes are similar to the constants from a classic programming language, while parameter/load nodes are similar to the variables.

4.4.2 Putting the Intuition to Work

The analysis transfer function for a label *lb* that corresponds to an analyzable CALL instruction is

$$[\![lb]\!]^a(G) = \bigsqcup_{callee \in CG(lb)} interproc(G,\ \circ A(exit_{callee}),\ P(lb)) \tag{4.2}$$

For each possible callee $callee \in CG(lb)$, the analysis uses the function *interproc* to compute a points-to graph for the program point after the CALL, for the case when *callee* is called. Next, the analysis joins the resulting points-to graphs.

Figure 4-6 presents the definition of the function *interproc*. The arguments of *interproc* are, in order, (1) the points-to graph G for the program point before the CALL, (2) the points-to graph G_{callee} for the end of the callee, and (3) the CALL instruction. For the moment, assume that the functions ρ, gc, τ, and α_0 are identities; in particular, $\tau(gc(\rho(G_{callee}))) = G_{callee}$. We explain these functions in Section 4.4.3.

Function *interproc* corresponds to the following algorithm:

1. Use G_{callee} to obtain the method summary for *callee*. The summary of *callee* consists of two elements:

   (a) An *inter-procedural transformer* T_{callee} that executes (a simplified model of) *callee*'s instructions in order to update the set of inside edges, the set of outside edges, the set of globally escaped nodes, and the node map:
   $T_{callee} \in IPTransformer = IPState \rightarrow IPState$, where
   $$IPState = IEdges \times OEdges \times \mathcal{P}(CNode) \times Map$$
   We explain later in this section how to construct T_{callee}.

   (b) The set of nodes $R_{callee} \subseteq CNode$ that *callee* returns (the last component of G_{callee}).

2. Construct an initial map μ_0: μ_0 maps each parameter node $n^P_{i,0}$ to the nodes

$$
\begin{array}{l}
\textit{interproc}\colon\ \textit{PTGraph}^a \times \textit{PTGraph}^a \times \textit{Instruction} \to \textit{PTGraph}^a \\
\textit{interproc}(G,\ G_{\textit{callee}},\ \text{“}v_R = v_0.s(v_1, \ldots, v_j)\text{”}) \ = \\
\quad \textbf{let}\ \ \langle L : [],\ I,\ O,\ E,\ R\rangle = G \\
\qquad \langle T_{\textit{callee}}, R_{\textit{callee}}\rangle = \textit{summary}(\tau(gc(\rho(G_{\textit{callee}})))) \\
\qquad \mu_0 = \left(\bigcup_{i=0}^{k} \{n_{i,1}^P\} \times L(v_i)\right) \ \cup\ \{\langle n, n\rangle \mid n \in \textit{CINode} \cup \mathcal{G}\} \\
\qquad \langle I_a,\ O_a,\ E_a,\ \mu_a\rangle = T_{\textit{callee}}(I,\ O,\ E,\ \mu_0) \\
\qquad L_a = L\,[v_R \mapsto \mu_a(R_{\textit{callee}})]\ \ \textbf{in} \\
\quad \alpha_0(\langle L_a : [],\ I_a,\ O_a,\ E_a,\ \emptyset\rangle)
\end{array}
$$

$$
\begin{array}{l}
\textit{summary}\colon\ \textit{PTGraph} \to \textit{IPTransformer} \times \mathcal{P}(\textit{CNode}) \\
\textit{summary}(G_{\textit{callee}}) = \textbf{let}\ \langle \_, \_, \_, \_, R_{\textit{callee}}\rangle = G_{\textit{callee}}\ \textbf{in}\ \langle \textit{mct}(G_{\textit{callee}}), R_{\textit{callee}}\rangle
\end{array}
$$

Notation: we use a tuple of identifiers on the left-hand side of a let-definition to name the components of a tuple. We use _ (“don’t care”) for irrelevant tuple components.

Figure 4-6: Definition of the function *interproc*. We define the auxiliary functions ρ, gc, τ, and α_0 in Section 4.4.3. On a first reading, one may assume that these four auxiliary functions are all identities.

from $L(v_i)$, where v_i is the corresponding argument.[10] Additionally, μ_0 maps each inside or global node to itself.

3. Apply $T_{\textit{callee}}$ to obtain the sets of inside edges, outside edges and globally escaped nodes after the CALL, and also a final map:

$$\langle I_a, O_a, E_a, \mu_a\rangle = T_{\textit{callee}}(I, O, E, \mu_0)$$

4. Project $R_{\textit{callee}}$ through μ_a to obtain the nodes that the caller variable v_R points to after the CALL. This projection is necessary because $R_{\textit{callee}}$ may contain parameter/load nodes. Set v_R to point to the resulting nodes in the points-to graph after the CALL.

The difficult part is the construction of the inter-procedural transformer $T_{\textit{callee}}$. First, we define one *atomic transformer* for each basic element from $G_{\textit{callee}}$: there is one atomic transformer for each inside edge, outside edge and globally escaped node from $G_{\textit{callee}}$. Next, we present how to “assemble” the atomic transformers to obtain $T_{\textit{callee}}$.

[10] The meticulous reader may notice that in Figure 4-6, we use $n_{i,1}^P$ instead of $n_{i,0}^P$. We clarify this node context difference in Section 4.4.3, when we explain the use of the functions τ and α_0.

Atomic Inter-procedural Transformers

Figure 4-7 defines the set $\mathcal{AT}(G_{callee})$ of atomic transformers for the points-to graph G_{callee}. For each node $n \in E_{callee}$ that *callee* escapes globally, the associated transformer $\texttt{gesc}(n)$ ensures that all nodes that n stands for (i.e., the nodes from $\mu(n)$) escape globally in the caller. For each inside edge $\langle n_1, f, n_2\rangle$ from G_{callee}, the associated transformer $\texttt{store}(n_1, f, n_2)$ creates inside edges between all nodes that n_1 stands for and all nodes that n_2 stands for. Figure 4-8.a contains a graphic representation of $\texttt{store}(n_1, f, n_2)$.

The transformer $\texttt{load}(n, f, n^L)$ associated to an outside edge $\langle n, f, n^L\rangle$ is more complex. Figure 4-8.b contains a graphic representation of this transformer. If n represents node n_1 (i.e., $n_1 \in \mu(n)$) and there exists an inside edge $\langle n_1, f, n_2\rangle \in I$, then the LOAD instruction that created the outside edge may read the inside edge $\langle n_1, f, n_2\rangle$. Hence, the transformer extends the map μ to record the fact that n^L may represent n_2. However, nodes like n_2 above may not be the only nodes that n^L represents: if n_1 escapes in the caller (i.e., if $e_2(E, I)(n_1)$ holds), then even the analysis of the caller may not know all the nodes that $n_1.f$ points to. In this case, the analysis still needs the load node n^L to represent the unknown nodes that the program may read from the escaped node n_1. Hence, the transformer adds the mapping $\langle n^L, n^L\rangle$. Also, in this case, the points-to graph for the program point after the CALL contains an outside edge $\langle n_1, f, n^L\rangle$, to record that n^L models nodes loaded from $n_1.f$.

Example 3. We encourage the reader to revisit the example from Section 2.2.2 on page 26. Steps 2, 3 and 4 from that example correspond to the execution of the atomic transformers $\texttt{load}(n_0^P, \texttt{cell}, n_{33}^L)$, $\texttt{load}(n_{33}^L, \texttt{next}, n_{35}^L)$, respectively $\texttt{store}(n_0^P, \texttt{cell}, n_{35}^L)$ (the example ignores the node contexts, the reader can assume they are all 0). The graphic representation of the inter-procedural analysis from Figure 2-4 on page 29 displays only the non-trivial mappings: for brevity, we ignored the mapping of each inside or global node to itself. △

Inter-procedural Transformer for the Entire Callee

Figure 4-9 defines the set $\mathit{Trans}(G_{callee})$ of inter-procedural transformers that one can build with the atomic transformers from $\mathcal{AT}(G_{callee})$ (see Figure 4-7), using functional composition "$\circ$", join "$\sqcup$", and transitive closure "$\texttt{star}$." For technical reasons, we also introduce the identity inter-procedural transformer $\texttt{id}$.

To obtain the inter-procedural transformer T_{callee} for the entire *callee*, the analysis combines the atomic transformers from $\mathcal{AT}(G_{callee})$. In the presence of ordering information, the analysis could simply compose the atomic transformers. E.g., suppose G_{callee} contains only an inside edge and an outside edge, and suppose the analysis has information that the STORE that created the inside edge always executes before the LOAD that created the outside edge. In this case, the analysis could define T_{callee} to be the composition of the $\texttt{load}$ for the outside edge with the $\texttt{store}$ for the inside edge.

However, the points-to graph G_{callee} does not contain ordering information. Additionally, G_{callee} does not contain information about the number of executions of

$$IPState = IEdges \times OEdges \times \mathcal{P}(CNode) \times Map$$
$$\mu \in Map = \mathcal{P}(CNode \times CNode)$$
$$T \in IPTransformer = IPState \to IPState$$

Atomic inter-procedural transformers for a points-to graph:

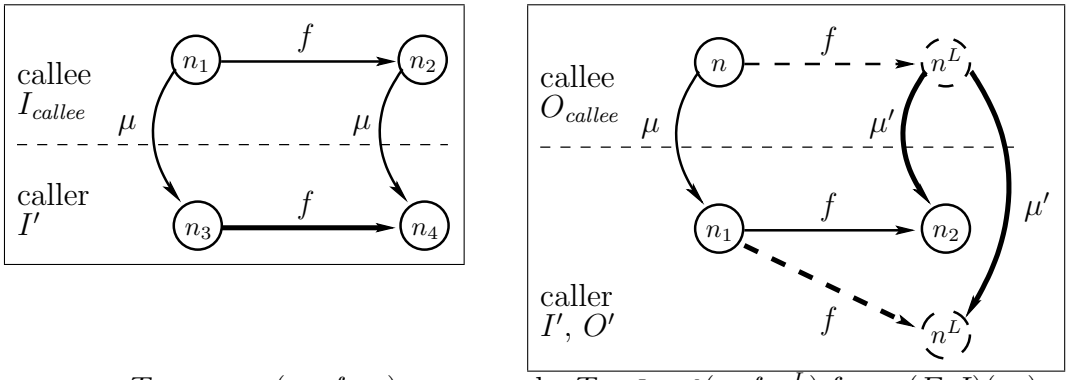
$$\mathcal{AT} : PTGraph \to \mathcal{P}(IPTransformer)$$
$$\mathcal{AT}(\langle J_{callee},\ I_{callee},\ O_{callee},\ E_{callee},\ R_{callee}\rangle) =$$
$$\{\ \texttt{gesc}(n)\ \mid\ n \in E_{callee}\ \}$$
$$\{\ \texttt{store}(n_1, f, n_2)\ \mid\ \langle n_1, f, n_2\rangle \in I_{callee}\ \}\ \cup$$
$$\{\ \texttt{load}(n, f, n^L)\ \mid\ \langle n, f, n^L\rangle \in O_{callee}\ \}\ \cup$$

$$\texttt{gesc}(n)(I, O, E, \mu) = \langle I,\ O,\ E \cup \mu(n),\ \mu\rangle$$

$$\texttt{store}(n_1, f, n_2)(I, O, E, \mu) = \langle I \cup (\mu(n_1) \times \{f\} \times \mu(n_2)),\ O,\ E,\ \mu\rangle$$

$$\texttt{load}(n, f, n^L)(I, O, E, \mu) =$$
$$\textbf{let}\ \ \mu_2 = \mu\ \cup\ \{\langle n^L, n_2\rangle \mid n_1 \in \mu(n).\ \langle n_1, f, n_2\rangle \in I\}$$
$$A = \{n_1 \in \mu(n) \mid e_2(E, I)(n_1)\}\ \textbf{in}$$
$$\textbf{if}\ A = \emptyset\ \ \textbf{then}\ \ \langle I,\ O,\ E,\ \mu_2\rangle$$
$$\textbf{else}\ \ \langle I,\ O \cup \left(A \times \{f\} \times \{n^L\}\right),\ E,\ \mu_2 \cup \{\langle n^L, n^L\rangle\}\rangle$$

Figure 4-7: Inter-procedural transformers - Part I: Atomic transformers.



a. $T = \texttt{store}(n_1, f, n_2)$. b. $T = \texttt{load}(n, f, n^L)$ for $e_2(E, I)(n_1)$.

Figure 4-8: Graphic representation of the atomic inter-procedural transformers `store` and `load`. In each case, if T is the transformer, $\langle I', O', E', \mu'\rangle = T(I, O, E, \mu)$. Circles represent general nodes, dashed circles represent load nodes, solid straight arrows represent inside edges, dashed straight arrows represent outside edges, and curved arrows represent node mappings. Bold arrows represent possibly new edges and node mappings; an inside edge is new if it appears in $I' \setminus I$, an outside edge is new if it appears in $O' \setminus O$, and a node mapping is new if it appears in $\mu' \setminus \mu$.

Compound inter-procedural transformers for a points-to graph:

$$\begin{aligned}
&\mathit{Trans} : \mathit{PTGraph} \to \mathcal{P}(\mathit{IPTransformer}) \\
&\mathit{Trans}(G_{\mathit{callee}}) = \ \mathcal{AT}(G_{\mathit{callee}}) \ \cup \ \{\ \mathtt{id}\ \} \ \cup \\
&\qquad\qquad \{\ T_1 \circ T_2,\ T_1 \sqcup T_2 \mid \ T_1, T_2 \in \mathit{Trans}(G_{\mathit{callee}})\ \} \ \cup \\
&\qquad\qquad \{\ \mathtt{star}(T) \ \mid \ T \in \mathit{Trans}(G_{\mathit{callee}})\ \}
\end{aligned}$$

$$\mathtt{id}(I, O, E, \mu) \ = \ \langle I, O, E, \mu \rangle$$

$$(T_1 \circ T_2)(I, O, E, \mu) \ = \ T_1(T_2(I, O, E, \mu))$$

$$(T_1 \sqcup T_2)(I, O, E, \mu) \ = \ T_1(I, O, E, \mu) \ \sqcup \ T_2(I, O, E, \mu)$$

$$\mathtt{star}(T)(I, O, E, \mu) \ = \ \bigsqcup_{k \geq 0} T^k(I, O, E, \mu)$$

where $T^0 = \mathtt{id}$ and $T^{i+1} = T \circ T^i$, $\forall i \geq 0$.

Most conservative transformer for a points-to graph:

$$\begin{aligned}
&\mathit{mct}\colon \mathit{PTGraph} \to \mathit{IPTransformer} \\
&\mathit{mct}(G_{\mathit{callee}}) = \mathtt{star}(\bigsqcup \mathcal{AT}(G_{\mathit{callee}}))
\end{aligned}$$

Figure 4-9: Inter-procedural transformers - Part II: Compound transformers.

each action. For example, the analysis has no information whether the LOAD instruction corresponding to an outside edge is executed once, or several times. To be conservative, our analysis uses the *most conservative transformer* from $\mathit{Trans}(G_{\mathit{callee}})$, $\mathit{mct}(G_{\mathit{callee}})$. $\mathit{mct}(G_{\mathit{callee}})$ is the transitive closure of the join of all atomic transformers for G_{callee}:

$$\mathit{mct}(G_{\mathit{callee}}) = \mathtt{star}(\bigsqcup \mathcal{AT}(G_{\mathit{callee}}))$$

As we prove in Lemma 19 on page 170, $\mathit{mct}(G_{\mathit{callee}})$ is bigger than any other transformer from $\mathit{Trans}(G_{\mathit{callee}})$. Therefore, it conservatively approximates any ordering of the atomic transformers (even with possible repetitions).

4.4.3 Additional Elements

This section explains the functions ρ, gc, τ, and α_0, and their use in the inter-procedural analysis (see definition of *interproc* in Figure 4-6).

The size of the method summary for *callee* is proportional with the size of the points-to graph G_{callee} for the end of *callee*. The inter-procedural analysis uses the function gc ("garbage collection") to remove from G_{callee} all captured nodes. Intuitively, the captured nodes model objects that are unreachable from the caller. The correctness proof from Chapter 6 shows that this simplification preserves the analysis correctness.

Figure 4-10 presents the definition of gc. The definition is more general than

$gc : PTGraph \to PTGraph$
$gc(G) = del_{garbage(G)}(G)$

$garbage : PTGraph \to \mathcal{P}(CNode)$
$garbage(G = \langle J, I, O, E, R\rangle) = \{n \in nodes(G) \mid \neg reachable(A, I)(n)\}$,
where $A = CPNode \cup CLNode \cup \mathcal{G} \cup E \cup R \cup nodes(J)$

Note: for a points-to graph G where local variables do not point to any node, $garbage(G) = \{n \in nodes(G) \mid \neg e(G)(n)\}$

The overloaded symbol del_S denotes a transformation that takes a node-based structure and removes all nodes from the set S:

$del_S(\langle J, I, O, E, R\rangle) = \langle del_S(J),\ del_S(I),\ del_S(O),\ del_S(E),\ del_S(R)\rangle$
$del_S([L_1, L_2, \ldots, L_k]) = [del_S(L_1),\ del_S(L_2),\ \ldots, del_S(L_k)]$
$del_S(L) = \lambda v.\ L(v) \setminus S$
$del_S(I) = \{\langle n_1, f, n_2\rangle \in I \ \mid\ n_1 \notin S \ \wedge\ n_2 \notin S\}$
$del_S(O) = \{\langle n, f, n_L\rangle \in O \ \mid\ n \notin S \ \wedge\ n_L \notin S\}$
$del_S(A) = A \setminus S$, for any set of nodes $A \subseteq CNode$.

$\rho : PTGraph^a \to PTGraph^a$
$\rho(\langle L : [], I, O, E, R\rangle) = \langle L_{\text{all–empty}} : [], I, O, E, R\rangle$

Figure 4-10: Definition of the function gc. $gc(G)$ preserves only the nodes that are reachable, along (possibly empty) paths of inside edges, from nodes pointed to by local variables or from escaped nodes; $gc(G)$ removes all other nodes from G. The auxiliary function ρ sets each local variable to point to an empty set of nodes.

what the pointer analysis requires: gc handles general points-to graphs, and it does not remove nodes that are transitively reachable, along (possibly empty) paths of inside edges, from nodes pointed to by local variables. This feature is irrelevant for the analysis: $\rho(G_{callee})$ is a points-to graph identical to G_{callee}, except that each local variable points to an empty set of nodes (see definition of ρ in Figure 4-10). We use the full definition of gc during the correctness proof.

To summarize, $gc(\rho(G_{callee}))$ is a simplified version of G_{callee} where each local variable points to an empty set of nodes, and all captured nodes have been removed.

Figure 4-11 presents the formal definitions of the functions τ and α_0. The function τ takes a structure built with nodes (e.g., a points-to graph) and produces a similar structure, but with all node contexts incremented by one. If $k \in \mathbb{N}$, function α_k takes a structure built with nodes and produces a similar structure, but where all node contexts bigger than k have been replaced with k (the other node contexts are unaffected). The *interproc* function (Figure 4-6) uses τ and α_0 as follows:

1. *interproc* uses $\tau(gc(\rho(G_{callee})))$ to construct the summary for the callee. The

$\tau : CNode \to CNode$
$\tau(\langle n, c\rangle) = \langle n, c+1\rangle$

$\alpha_k : CNode \to CNode,\ k \in \mathbb{N}$
$\alpha_k(\langle n, c\rangle) = \langle n, \min(k, c)\rangle$

We overload the symbols τ and α_k to denote functions that apply not only to nodes, but to any structures built with nodes. The functions τ and α_k propagate deep inside such structures; they preserve the structure, but may change the encountered nodes. E.g.,
$\tau(\langle J,\ I,\ O,\ E,\ R\rangle) = \langle \tau(J),\ \tau(I),\ \tau(O),\ \tau(E),\ \tau(R)\rangle$,
$\tau(I) = \{\ \langle \tau(n_1), f, \tau(n_2)\rangle \mid \langle n_1, f, n_2\rangle \in I\ \}$, etc.

Figure 4-11: Definitions of the functions τ and α_k, $k \in \mathbb{N}$.

points-to graph $\tau(gc(\rho(G_{callee})))$ is similar to $gc(\rho(G_{callee}))$, except that all nodes have context 1, instead of 0. In particular, the context of the parameter nodes from $\tau(gc(\rho(G_{callee})))$ is 1, explaining our use of $n^P_{i,1}$ (instead of $n^P_{i,0}$) in the definition of the initial map μ_0 (Figure 4-6).

2. *interproc* uses α_0 to ensure that the context of all nodes from the points-to graph after the CALL is 0, as it should be in an analysis points-to graph.

The use of τ and α_0 has two purposes:

- To obtain additional context sensitivity during the inter-procedural analysis: G and $\tau(gc(\rho(G_{callee})))$ have disjoint sets of nodes (G has only nodes with context 0, while $\tau(gc(\rho(G_{callee})))$ has only nodes with context 1).
- To allow the correctness proof from Chapter 6; we comment more on this topic in Section 6.4.

The computation of $\tau(gc(\rho(G_{callee})))$ is expensive (due to gc). Our analysis implementation caches this result and uses it at each CALL that may invoke *callee*.

4.4.4 Computational Aspects

The transfer function for an analyzable CALL is far more complex than any other transfer function. Let $G_2 = \tau(gc(\rho(G_{callee})))$ and $F = \bigsqcup \mathcal{AT}(G_2)$. The function *interproc* uses the transformer $T_{callee} = mct(G_2) = \texttt{star}(F)$. The definition of $\texttt{star}(F)$ involves a join operation over an infinite set: $\texttt{star}(F) = \bigsqcup_{i\geq 0} F^i$. Therefore, it is not obvious that $T_{callee}(I, O, E, \mu_0)$ exists and that it can be computed. In the next paragraphs, we show that $T_{callee}(I, O, E, \mu_0)$ exists and that the analysis can compute it in polynomial time in the size N of the analysis scope.

First, notice that *interproc* manipulates only nodes with context 0 or 1. Hence, it suffices to consider only inter-procedural transformers that operate on *IPState2*, the restriction of

$$IPState = IEdges \times OEdges \times \mathcal{P}(CNode) \times Map$$

to nodes with context ≤ 1. For any analysis scope of size N, there are $\mathcal{O}(N)$ nodes with contexts ≤ 1 (see the discussion before Lemma 1 on page 43). As the analysis scope uses $\mathcal{O}(N)$ fields, the depth of the lattice *IPState2* is $\mathcal{O}(N^3 + N^3 + N + N^2) = \mathcal{O}(N^3)$.

Next, consider the following result:

Lemma 3. *Each atomic transformer is extensive*[11] *and monotonic.*[12]

Proof. Simple inspection of the definitions from Figure 4-7. The only non-trivial case is the monotonicity of $\texttt{load}(n, f, n^L)$ transformers: still, as graph reachability is monotonic in the set of roots and the set of edges, the predicate $e_2(E, I)$ is monotonic in E and I. □

By consequence, the transformer $F = \bigsqcup \mathcal{AT}(G_2)$ is extensive and monotonic. As F is extensive, the series $(F^i(I, O, E, \mu_0))_{i\geq 0}$ constitutes an ascending chain in *IPState2*. This chain stabilizes after some finite number of steps $k = \mathcal{O}(N^3)$, with the value $F^k(I, O, E, \mu_0)$. Hence, $T_{callee}(I, O, E, \mu_0) = \bigsqcup_{i\geq 0} F^i(I, O, E, \mu_0)$ exists and has the value $F^k(I, O, E, \mu_0)$, for some finite $k = \mathcal{O}(N^3)$.

A naive algorithm for computing $T_{callee}(I, O, E, \mu_0)$ applies F repeatedly on $\langle I, O, E, \mu_0\rangle$, until it reaches a fixed point. Such an algorithm requires $\mathcal{O}(N^3)$ iterations; each iteration applies all the atomic transformers from $\mathcal{AT}(G_2)$ and joins the results. There are $\mathcal{O}(N^3)$ such transformers: there is one atomic transformer for each of the $\mathcal{O}(N^3)$ inside edges, $\mathcal{O}(N^3)$ outside edges, and $\mathcal{O}(N)$ globally escaped nodes from G_{callee} (see the proof of Lemma 1 on page 43.) A naive implementation of the atomic transformers requires $\mathcal{O}(N)$ time for a `gesc` transformer, $\mathcal{O}(N^2)$ for a `store` transformer, and $\mathcal{O}(N^3)$ for a `load` transformer (because `load` uses the predicate $e_2(E, I)$ that involves reachability over the $\mathcal{O}(N^3)$ edges from I). The overall complexity of this naive algorithm is polynomial, but huge, $\mathcal{O}(N^9)$.

The naive algorithm "blindly" executes all atomic transformers in each iteration. Our implementation uses an optimized algorithm that keeps track of the changes (new mappings, new inside edges and new escaped nodes) and executes only those atomic transformers that have a chance of producing new information. For example, the optimized algorithm executes a $\texttt{gesc}(n)$ transformer only if there are new mappings for n. The optimized algorithm has complexity $\mathcal{O}(N^5)$. Appendix A presents our optimized algorithm and proves its correctness and its asymptotic complexity.

The other steps of *interproc* are far less time-intensive than the computation of $T_{callee}(I, O, E, \mu_0)$. Therefore, *interproc* has time complexity $\mathcal{O}(N^5)$.

[11] *If $\langle A, \sqsubseteq\rangle$ is a lattice, a function $f : A \to A$ is extensive iff $\forall a \in A.\ a \sqsubseteq f(a)$.*

[12] *If $\langle A, \sqsubseteq_A\rangle$ and $\langle B, \sqsubseteq_B\rangle$ are lattices, a function $f : A \to B$ is monotonic iff $\forall a_1, a_2 \in A.\ a_1 \sqsubseteq_A a_2 \to f(a_1) \sqsubseteq_B f(a_2)$.*

Lemma 2 on page 50 already proved that the analysis transfer functions for labels that do not correspond to analyzable CALLs produce only analysis points-to graphs. For analyzable CALLs, a simple inspection of the definition of *interproc* (Figure 4-6) reveals that in the resulting graph (1) the abstract stack has exactly one element, and (2) each node has context 0 (due to the application of the function α_0). Hence, we extend Lemma 2 to cover all labels, showing that the pointer analysis manipulates only analysis points-to graphs:

Lemma 4. $\forall lb \in Label.\ \forall G \in PTGraph^a.\ [\![lb]\!](G) \in PTGraph^a.$

4.5 Computation of the Analysis Results

Previous sections presented the analysis in a declarative style, as a set of dataflow constraints. This section explains how to compute the analysis results, i.e., how to solve the dataflow constraints. Section 4.5.1 presents a general algorithm. Section 4.5.2 presents an algorithm that reduces the analysis memory consumption.

4.5.1 General Algorithm

Assume that an analysis client requires the analysis points-to graph for a program point from a method m. The dataflow constraints for method m (Equations 4.1 on page 45) use the points-to graphs for the end of the methods that m may transitively invoke through analyzable CALLs. Therefore, the analysis needs to examine all methods that m may transitively invoke through analyzable CALLs. Let S be the set of such methods (including m). The algorithm from this section computes points-to graphs for all program points from the methods in the set S. The algorithm generates the dataflow constraints for all labels inside the methods in the set S and next uses a constraint solver. We describe these two steps below.

Constraint Generation

Our analysis generates constraints of the general form $\mathrm{v} \sqsupseteq f(\mathrm{v}_1, \mathrm{v}_2, \ldots, \mathrm{v}_k)$, where v, $\mathrm{v}_1, \ldots, \mathrm{v}_k$ are *flow variables*, and f is a function (not necessarily monotonic). The flow variables represent the values (i.e., the points-to graphs) that our analysis computes; they are unrelated to the variables from the analyzed program. Our analysis uses the flow variable v^b_{lb} for the points-to graph before the label lb, and the flow variable v^a_{lb} for the points-to graph after the label lb. The constraints between flow variables are an almost straightforward transcription of the Constraints 4.1, replacing $\circ A(lb)$ with v^b_{lb} and $A\circ(lb)$ with v^a_{lb}.[13] A solution for the resulting system of constraints is a *valuation* ψ that assigns to each variable v an analysis points-to graph $\psi(\mathrm{v}) \in PTGraph^a$, such that all constraints are satisfied. Each solution ψ corresponds to a set of valid analysis results: $\circ A(lb) = \psi(\mathrm{v}^b_{lb})$ and $A\circ(lb) = \psi(\mathrm{v}^a_{lb})$.

[13] Conceptually, the difference between v^b_{lb} and $\circ A(lb)$ is the difference between an equation unknown (e.g., x in $x - 1 = 0$), and its value in a valid solution (e.g., 1).

For each method $m_2 \in S$, the analysis generates a constraint $\mathrm{v}^b_{entry_{m_2}} \sqsupseteq G^{m_2}_{\text{init}}$. For each label lb inside a method from S and for each predecessor $lb' \in pred(lb)$, the analysis generates a constraint $\mathrm{v}^b_{lb} \sqsupseteq \mathrm{v}^a_{lb'}$. These simple constraints encode the analysis constraint

$$\circ A(lb) \sqsupseteq \bigsqcup \{A\circ(lb') \mid lb' \in pred(lb)\}$$

A complex constraint of the form $\mathrm{v} \sqsupseteq t_1 \sqcup t_2 \sqcup \ldots t_k$ (where t_i's are some terms) is equivalent to the set of simpler constraints $\mathrm{v} \sqsupseteq t_1$, $\mathrm{v} \sqsupseteq t_2$, $\ldots$, $\mathrm{v} \sqsupseteq t_k$.

For each label lb that does not correspond to an analyzable CALL, the analysis generates the constraint $\mathrm{v}^a_{lb} \sqsupseteq [\![lb]\!]^a(\mathrm{v}^b_{lb})$. For each label lb that corresponds to an analyzable CALL, we use the definition of the transfer function for an analyzable CALL (Equation 4.2 on page 52) to expand the analysis constraint $A\circ(lb) \sqsupseteq [\![lb]\!]^a(\circ A(lb))$ as follows:

$$A\circ(lb) \sqsupseteq \bigsqcup_{callee \in CG(lb)} interproc(\circ A(lb),\ \circ A(exit_{callee}),\ P(lb))$$

Accordingly, for each possible callee $callee \in CG(lb)$, the analysis generates the constraint

$$\mathrm{v}^a_{lb} \sqsupseteq interproc(\mathrm{v}^b_{lb}, \mathrm{v}^a_{exit_{callee}}, P(lb))$$

Constraint Solving

Solving constraints over a finite-depth lattice (like $PTGraph^a$; see Lemma 1) is a well-understood problem. To provide the background for the discussion on the analysis termination and complexity, we describe below a simple worklist-based solver, the "Chaotic Iteration Algorithm." Reference [66, Chapter 6] discusses several other worklist-based constraint solvers that have the same asymptotic complexity as the "Chaotic Iteration Algorithm," but are much faster in practice.

Chaotic Iteration Algorithm: The algorithm maintains a worklist W of potentially unsatisfied constraints. Initially, W contains all the constraints, and the valuation ψ assigns to each variable v the bottom value $\bot_{PTGraph^a}$. The solver iterates as long as W is not empty. In each iteration, the solver extracts one constraint "$\mathrm{v} \sqsupseteq f(\mathrm{v}_1, \mathrm{v}_2, \ldots, \mathrm{v}_k)$" from W and increases the value of v in order to satisfy the constraint. More precisely, the solver computes the value of the right-hand side of the constraint, with respect to the current valuation ψ. Let l be this value; the solver joins l to the current value for the variable v, i.e., the solver updates the valuation ψ as follows:

$$\psi \leftarrow \psi\,[\mathrm{v} \mapsto \psi(\mathrm{v}) \sqcup l] \tag{4.3}$$

If this update causes $\psi(\mathrm{v})$ to increase strictly,[14] then the constraints that use v on their right-hand side may become unsatisfied. Accordingly, the solver adds all such

[14] As we use $\sqcup$, the value of v can either remain unchanged, or increase strictly.

constraints to the worklist W. The solver terminates when W is empty. The valuation ψ at the end of the algorithm is a solution for the system of constraints.

Correctness: One can easily prove that at the beginning and at the end of each iteration, the worklist W contains all constraints that are not satisfied by the current valuation ψ. This property is trivially true at the beginning of the first iteration, when W contains all the constraints. In each iteration, if the value for a variable v has strictly increased, the algorithm adds to W all constraints that use v on their right-hand side. These are the only constraints that may become unsatisfied after the increase of v's value: constraints that do not use v are unaffected, and constraints that use v on their left-hand side remain valid. Therefore, if the algorithm terminates (i.e., if W becomes empty), it terminates with a valuation ψ that satisfies all the constraints.

Termination and Complexity: The algorithm adds a constraint to the worklist W in two cases: (1) in the initialization step, and (2) when the value for a right-hand side variable strictly increases. As the value of a variable never decreases during the algorithm, the value of a variable can increase at most d times, where d is the finite depth of the lattice $PTGraph^a$. Therefore, the algorithm processes each constraint at most $1 + d \cdot M$ times, where M is the maximum number of distinct variables that may appear of the right side of a constraint. Let R be the time required for computing the value of the right-hand side of a constraint, and J be the time for the join operation. Each time the algorithm processes a constraint extracted from the worklist W, the algorithm spends time $R + J$. For a system with C constraints, the asymptotic complexity of the entire algorithm is $\mathcal{O}(d \cdot M \cdot C \cdot (R + J))$.

Asymptotic Complexity of the Pointer Analysis

We use the notations from the complexity analysis for the constraint solver. For our analysis, $d = \mathcal{O}(N^3)$ (Lemma 1 on page 43), $M = 2$ (due to the *interproc* constraints), $R = \mathcal{O}(N^5)$ (due to the complexity of *interproc*), and $J = \mathcal{O}(N^3)$ (points-to graphs are structures of size $\mathcal{O}(N^3)$). To estimate C, the number of constraints, notice that apart from the *interproc* constraints, there is one constraint for the beginning of each method and one constraint for each control flow edge. There are $\mathcal{O}(N)$ methods and $\mathcal{O}(N)$ control flow edges: each label has at most two successors.[15] In addition, there are $\mathcal{O}(N^2)$ *interproc* constraints: there are $\mathcal{O}(N)$ analyzable CALLs and each of them may have $\mathcal{O}(N)$ callees. This accounts for $C = \mathcal{O}(N^2)$ constraints. Hence, the entire algorithm has time complexity $\mathcal{O}(N^3 \cdot 2 \cdot N^2 \cdot (N^3 + N^5)) = \mathcal{O}(N^{10})$.

Discussion: The asymptotic complexity of our analysis is polynomial, but huge ... Part of the reason is the coarseness of the complexity analysis and the fact that we

[15] Even if the analyzed program contained `switch`-like instructions with arbitrarily many cases, notice that each `switch` case has exactly one predecessor.

analyze the worst-case complexity. Fortunately, our analysis can obtain correct information by analyzing only parts of the program. Hence, our analysis can decrease N (the size of the analysis scope) in order to decrease the analysis time. For example, the analysis can consider as unanalyzable all CALLs to very large methods. Also, the analysis can impose a constant bound on the number of callees at each analyzable CALL. This modification reduces the number of the *interproc* constraints to $\mathcal{O}(N)$, reducing the overall analysis complexity to $\mathcal{O}(N^9)$. Chapter 7 describes a few techniques for improving the analysis speed. In practice, our implemented prototype analyzes applications on the scale of `javac` (1960 analyzed methods from 332 classes, with more than $50,000$ bytecode instructions) in less than one minute.

4.5.2 Reducing Memory Consumption

The algorithm from Section 4.5.1 simultaneously maintains in memory the points-to graphs for all program points from all methods from the set S. If this is a concern, then one can use the following algorithm:

1. Apply a simple reachability algorithm to determine the set S of all methods that m may transitively invoke through analyzable CALLs.
2. Compute the strongly-connected components of the caller-callee relation between methods from S. Each strongly-connected component corresponds to a set of mutually recursive methods.
3. Process the strongly-connected components bottom-up, from the callees toward the callers. For each strongly-connected component *scc* of methods:
   (a) Initialize the points-to graphs for the end of all methods from *scc* to $\bot_{PTGraph^a}$.
   (b) Add all methods from *scc* to a worklist W_{inter}.
   (c) As long as the worklist W_{inter} is non-empty:
      i. Extract a method m_2 from W_{inter}.
      ii. Use a worklist-based algorithm [66, Chapter 6] to solve the dataflow equations for the labels inside m_2. The constraints for *interproc* use the currently-available points-to graphs for the end of the callees.
      iii. Consider the points-to graph for the end of m_2 and ignore the points-to graphs for the other program points inside m_2. If the points-to graph for the end of m_2 has changed, we add to the worklist W_{inter} all methods from *scc* that call m_2 through an analyzable CALL.

At any moment, the algorithm keeps in memory at most the points-to graphs for the end of all methods, and the points-to graphs for the program points inside the currently analyzed method m_2. The algorithm computes the points-to graphs for the end of all methods from S. The analysis can compute the points-to graph for any program point inside any method from S: the analysis (re)computes the solution of

the dataflow equations for that method only (the analysis already knows the points-to graphs for the end of any callee).

Notice that the analysis of method m performs the analysis of any method from S. Hence, if we apply the algorithm above repeatedly, for different methods m, then the computation of the set S from step 1 can ignore the already analyzed methods: the analysis already knows the points-to graphs for the end of these methods and for the end of their callees.

4.6 Discussion

This section discusses a few aspects of the analysis design and presentation.

4.6.1 Inter-procedural Analysis

The inter-procedural analysis performs an abstract execution of a model of the callee. It is natural to inquire whether this solution is really better than simply inlining the callee and performing standard intra-procedural analysis. We list below some of the advantages of the current solution:

- For each callee, the analysis constructs the inter-procedural transformer on the basis of the analysis points-to graph for the end of the callee. As the set $PTGraph^a$ of analysis points-to graphs is finite (for each analysis scope), our inter-procedural analysis terminates even for recursive methods.

- The inter-procedural transformer for a callee is a simplified model of the callee, less detailed than the callee's source code:

    1. The inter-procedural transformer ignores all callee's instructions that do not manipulate pointers.
    2. The inter-procedural transformer is a pre-computed form of the analysis of an inlined copy of the callee. For example, the inter-procedural transformer does not deal with the flow of nodes between local variables. This flow has already been resolved during the intra-procedural analysis for the callee. Also, the LOAD instructions that read only from captured nodes do not generate any outside edge; as such, they do not generate atomic transformers. The intra-procedural analysis of the callee already discovered all nodes loaded by these instructions.
    3. The transfer function for RETURN (see Figure 4-3 on page 47) eliminates captured nodes from the points-to graph for the end of the callee. Therefore, the inter-procedural transformer ignores all captured nodes from the callee, and all instructions that manipulate these nodes.
    4. As we explain in Section 7.3, in addition to eliminating the captured nodes, the analysis can perform other simplifications on the points-to graph from the end of the callee, to reduce its size, and, implicitly, the cost of executing the corresponding inter-procedural transformer.

4.6.2 Strong vs. Weak Updates

The analysis transfer functions perform destructive updates on the local variables, and weak updates on the node fields (see Figure 4-3): adding a new inside edge $\langle n_1, f, n_2 \rangle$ does not remove the previous inside edges from $n_1.f$. The reason is that a node may represent multiple objects and so, updating $n_1.f$ might not overwrite the edge $\langle n_1, f, n_2 \rangle$ because the update instruction and the edge may concern different objects. Still, it is conceivable to extend the analysis to do strong updates in the case of STORE statements "v_1`.f =` v_2" that definitely write exactly one object: i.e., v_1 points to exactly one node, and that node is guaranteed to represent only one object in each activation of the analyzed method. Examples of such nodes include the parameter nodes and the inside nodes corresponding to allocation sites outside loops. Adding support for strong updates is an interesting direction for future work.

4.6.3 Handling of the Static Fields

Our analysis treats static fields very conservatively: when a reference to a node n is stored in a static field, the analysis records the fact that n escapes globally (by adding n to the E component of the points-to graph) and ignores other information about n, e.g., the analysis does not record the static field that points to n. Similarly, reading a static field (the STATIC LOAD instruction) produces the general, globally-escaping node $n_{\texttt{GBL},0}$, that we also use to model the result of an unanalyzable CALL.

The main reason for this design choice is our desire to reduce the size of the points-to graphs. In addition, obtaining complete information about the nodes reachable from the static fields would require examining the entire source code (including all the native methods). For example, the analysis would need to check that no parallel thread mutates a static field. In our analysis, we focus on the information that can be detected by analyzing only parts of a whole program.

Most likely, a more precise treatment of the static fields would not benefit the stack allocation optimization: in general, programmers use static fields to store references to long-lived, non-local data.

Still, some other analysis clients may benefit from a more precise treatment of static fields. For example, consider the case of the purity analysis, and assume that the analysis analyzes a method whose body consists of the following two instructions:

```
v = C.f;
v.f2 = null;
```

Currently, the analysis can report to the user only that the method is impure because it mutates the field `f2` of some unknown object that is either read from an unspecified static field, or returned from an unanalyzable CALL (the analysis uses $n_{\texttt{GBL},0}$ in both cases). Imagine that we change the analysis to process a STATIC LOAD similar to the way we process a LOAD from an escaped node, i.e., using a load node and an

outside edge to tell us where the load node is read from.[16] Such a modification would allow us to provide more information to the user: e.g., "*the method is impure because it mutates the field* `f2` *of the object read from* `C.f`*, i.e., the field* `C.f.f2`".

Our current implementation is optimized toward reducing the size of the points-to graphs. Still, a more precise processing for the static fields is an interesting direction for future work.

4.6.4 Flow Sensitivity

We presented the analysis in a flow-sensitive framework: the analysis takes into account the order of the instructions[17] and computes one points-to graph for each program point. One advantage of flow-sensitivity is that it simplifies reasoning about the analysis: the execution of the analysis is a simulation of the program execution, with one state at each point. Another advantage is the possible increase in analysis precision (unproven experimentally yet). The disadvantage is the size of the data that the analysis manipulates, which influences negatively the analysis speed.

Notice that the analysis does not need to maintain all components of the points-to graphs flow-sensitively:

Outside Edges: During the intra-procedural analysis of a method m, some transfer functions may generate new outside edges, but no transfer function uses these edges. Only after the intra-procedural analysis of m terminates, the inter-procedural analysis uses the outside edges from the end of m, while processing CALLs to m. Hence, for each method, it suffices to maintain only one set of outside edges: the set for the end of the method. This set collects all outside edges that LOAD/CALL instructions may generate.

Set of Returned Nodes: For similar reasons, the analysis can use a single set of returned nodes for each method. After this modification, each RETURN instruction adds nodes to the method-wide set of returned nodes.

Abstract State of Local Variables: The analysis can use the SSA form [26] to maintain a single, flow-insensitive state of the local variables: in the SSA form, each variable is assigned by a single instruction.

The SSA form introduces ϕ instructions in the control-flow join points. A ϕ instruction "$v = \phi(v_1, v_2, \ldots, v_k)$" copies into v one of the v_i's, depending on which predecessor the program executes right before the ϕ instructions.[18] The analysis processes a ϕ instruction by setting v to point to all nodes pointed

[16] We may also add a special node, different from $n_{\texttt{GBL},0}$, to serve as an "envelope" for all static fields. The updated analysis could interpret all static fields as fields of this special node.

[17] E.g., consider the use of the *pred* function in the dataflow equations from Section 4.3.

[18] A detailed presentation of the SSA form is beyond the scope of this dissertation. The interested reader should consult Reference [26] for more information on the SSA form.

to by any of the variables v_i; the other pieces of information that the analysis maintains (e.g., inside edges) do not change.

Our prototype implementation (see Section 8.1) uses these optimizations to reduce the size of the manipulated data. The remaining flow-sensitive pieces of information are the set of inside edges and the set of globally escaped nodes.

More Flow-Insensitivity

An interesting question is whether an analysis that maintains flow-insensitive sets of inside edges and globally escaped nodes can achieve similar precision and run much faster. We experimented with a completely flow-insensitive version of our analysis. This version computes a single points-to graph for each method and completely disregards the control-flow information (e.g., the analysis computes the same result for different permutations of the method instructions). We describe the completely flow-insensitive analysis below and explain why it is much slower in practice than the flow-sensitive analysis. We close this section by sketching an improved flow-insensitive version of the analysis that computes a single points-to graph for each method, but takes into account the control-flow graph. Due to time constraints, the implementation and evaluation of the improved flow-insensitive analysis is left for future work.

Completely Flow-Insensitive Analysis (CFIA): For each analyzed method m, CFIA computes a single points-to graph G, such that the pair $\langle \circ A, A \circ \rangle$ with $\circ A(lb) = A\circ(lb) = G$ for any label lb from m, satisfies the Constraints 4.1. Equivalently, CFIA computes a points-to graph G that satisfies the following constraints:

$$\begin{array}{lcl} G & \sqsupseteq & G^m_{\text{init}} \\ G & \sqsupseteq & [\![lb]\!]^a(G), \quad \forall \text{ label } lb \text{ from } m \end{array} \tag{4.4}$$

The constraints for the control-flow join points, $\circ A(lb) \sqsupseteq \bigsqcup\{A\circ(lb') \mid lb' \in pred(lb)\}$, $\forall lb$, are trivially satisfied if $\circ A(lb) = A\circ(lb) = G$, $\forall lb$. The control-flow graph of the analyzed method m (i.e., the ordering of the instructions from m) is irrelevant for the constraints above.

Intuitively, CFIA computes a particular solution for the flow-sensitive analysis, where the points-to graphs for all program points inside m are equal to G. Therefore, the flow-insensitive solution G satisfies the properties from Section 4.2.

The CFIA algorithm is the same as the algorithm for the flow-sensitive analysis (see Section 4.5), except that it uses the same flow variable v for all program points inside an analyzed method, i.e., $\mathrm{v}^b_{lb} = \mathrm{v}^a_{lb} = \mathrm{v}$, $\forall lb$.

Unfortunately, CFIA introduces data dependencies that force the constraint solver to iterate even for loop-free methods. Every time the method-wide points-to graph G changes, the constraint solver needs to re-evaluate all constraints.[19] This problem persists even if the analysis represents data at sub-points-to graph granularity.

[19] Except the constraint for the beginning of the analyzed method m.

1. Initialize $G \leftarrow G^m_{\text{init}}$.
2. Compute the strongly-connected components of m's control-flow graph.
3. Examine the strongly-connected components in topological order, from the beginning of the method toward the end. For each component scc,
   (a) If scc contains a single label lb, and there is no control-flow loop arc from lb to lb, then update $G \leftarrow G \sqcup [\![lb]\!]^a(G)$.
   (b) Otherwise, as long as there exists a label lb in scc such that $G \sqcup [\![lb]\!]^a(G) \sqsupset G$, update $G \leftarrow G \sqcup [\![lb]\!]^a(G)$.

Figure 4-12: Sketch of an algorithm for the Improved Flow-Insensitive Analysis (IFIA). For brevity, we present only the algorithm for the intra-procedural analysis of a method m. If the method m invokes other methods, the algorithm from Section 4.5.2 can serve as a top-level driver for the analysis, using the algorithm from this figure in Step 3(c)ii.

Consider the case of a straight-line method containing two CALL instructions: the transfer function for each CALL reads and updates the method-wide set of inside edges. Therefore, the analysis needs to iterate the two (expensive) transfer functions in order to reach a fixed-point that satisfies the flow-insensitive constraints. In contrast, the flow-sensitive analysis executes each transfer function exactly once: first the transfer function for the first CALL, and next the transfer function for the second CALL. In practice, CFIA behaved much slower than the flow-sensitive analysis, prompting us to abandon CFIA as a viable idea.

Improved Flow-Insensitive Analysis (IFIA): As the CFIA algorithm, IFIA computes a single points-to graph for each method. Unlike CFIA, IFIA uses the control-flow graph of the analyzed method m to eliminate spurious data dependencies.

Figure 4-12 presents the sketch of an algorithm for IFIA. The key element of IFIA is that it iterates only over the loops from the control-flow graph.

One can prove the existence of a solution $\langle \circ A, A \circ \rangle$ for the flow-sensitive analysis (i.e., Constraints 4.1) such that $G \sqsupseteq \circ A(lb)$ and $G \sqsupseteq A\circ(lb)$, for any label lb inside the analyzed method m.

Proof. Let $scc_1, scc_2, \ldots, scc_k$ be the strongly-connected components of m's control-flow graph, in the topological order the algorithm from Figure 4-12 explores them. For each strongly-connected component scc_i, let G_i be the version of the points-to graph G immediately after the algorithm processed scc_i. As G never decreases during the algorithm, $G^m_{\text{init}} \sqsubseteq G_i \sqsubseteq G_j$, $\forall i < j$. For each label lb, let scc_l be the strongly-connected component that lb is an element of. Let $\circ A(lb) = A\circ(lb) = G_l$. This flow-sensitive solution satisfies Constraints 4.1: First, $\circ A(entry_m) = G_0 \sqsupseteq G^m_{\text{init}}$.

Second, $\forall lb, lb'$ such that $lb' \in pred(lb)$, lb' and lb belong to the strongly-connected components scc_i, respectively scc_j, such that $i \leq j$. Therefore, $A\circ(lb') = G_i \sqsubseteq \circ A(lb)$. Third, $A\circ(lb) \sqsupseteq [\![lb]\!]^a(\circ A(lb))$, due to the termination condition of the Step 3b from the algorithm in Figure 4-12. $\square$

Therefore, the flow-insensitive result G of the algorithm from Figure 4-12 is an upper-approximation of a flow-sensitive solution. Hence, G satisfies the properties from Section 4.2 (those properties remain valid for bigger points-to graphs).

Chapter 5

Analysis Applications

This chapter presents two applications of our analysis: the stack allocation optimization (Section 5.1) and the purity analysis (Section 5.2).

5.1 Stack Allocation Optimization

Java programs allocate all objects in a garbage-collected heap. This technique is essential for Java's type safety, provides an elegant programming model, and eliminates hard-to-find programming bugs like dangling pointers. Unfortunately, garbage-collection may incur a significant runtime overhead.

Our analysis detects allocation sites (i.e., NEW / ARRAY NEW instructions) that allocate only captured objects. Captured objects are unreachable from outside the enclosing method. When the method terminates, these objects become unreachable and can be deallocated. The compiler changes these allocation sites to allocate memory from a stack. Our current implementation uses the execution stack, but a different, dedicated stack can be used too. Allocating an object in a stack is very cheap: it requires a simple adjustment of the stack pointer. When the method terminates, the stack pointer returns to its original value and all objects stack allocated inside the method are implicitly deallocated without any garbage-collection overhead.

The biggest advantage of the stack allocation optimization is the reduction of the garbage-collection overhead. Additionally, we use the stack allocation optimization as an empirical test of the correctness of our analysis design and implementation: generally, incorrect stack allocation decisions result in severe, visible runtime errors.

Basic Stack Allocation Strategy

Consider a method m and let $G = \circ A(exit_m)$ be the points-to graph that the analysis computes for the end of method m. In the basic stack allocation strategy, the compiler examines the allocation instructions from the method m. Consider an allocation instruction at label lb. If the corresponding inside node $n^I_{lb,0}$ is captured in G, i.e., $\neg e(G)(n^I_{lb,0})$, the compiler changes the allocation instruction to allocate memory from the stack.

Correctness: Consider an activation (i.e., execution) $A(m)$ of the method m. Let o be one of the objects that $A(m)$ allocates by executing the instruction from label lb. As $n^I_{lb,0}$ is captured in G, by Part 1 of Property 4 on page 44, the object o is captured in $A(m)$. Therefore, after the end of $A(m)$, o is unreachable from the entire program. Hence, the lifetime of the object o is included in the lifetime of method m's stack frame. This fact ensures the safety of allocating the object o in method m's stack frame.

Use of Method Inlining for Enhancing the Stack Allocation Optimization

The compiler can significantly improve the effectiveness of stack allocation by using method inlining. Inlining extends the lifetime of a method stack frame by merging the method stack frame into the stack frame of the caller. Therefore, more objects are likely to have their lifetime included in the lifetime of the stack frame.

With our previous notations, suppose $n^I_{lb,0}$ escapes from m, but is captured in m_2, one of m's callers. In this case, the compiler can inline m in m_2 and change the inlined copy of the NEW instruction from label lb to allocate memory from the stack. This technique performs only inlining that is potentially beneficial for the stack allocation: e.g., if no inside node that escapes from m is captured in m_2, the compiler does not need to inline m into m_2. This technique can be extended to arbitrarily long call chains. The correctness argument is the same as in the case of the basic stack allocation strategy: Property 4 on page 44 refers to all objects allocated inside $A(m)$, including those allocated by transitive callees of the method m.

Additional Practical Considerations

In the presence of stack allocation, the garbage collector scans stack allocated objects as any other objects, but does not attempt to move or collect them. The garbage collector can identify stack allocated objects by using a simple address-based mechanism.

The largest potential drawback of stack allocation is that it may increase memory consumption by extending the lifetime of the objects that are allocated on the stack. This problem may be especially acute for the allocation sites that create a statically unbounded number of objects, i.e., allocation sites inside statically unbounded loops. To alleviate this problem, our implementation does not stack allocate allocation sites inside loops and does not inline calls inside loops. Another solution, left for future work, consists of extending the analysis to recognize loop-local objects.

5.2 Purity Analysis

The knowledge that a method is *pure*, or has no externally visible side-effects, is important for several program understanding and verification tasks. Pure methods can safely be invoked by program instrumentation code whose purpose is to "observe" the execution of a program without changing the program semantics. Therefore, pure methods can safely be used in program assertions and specifications [55, 10]. Similarly,

pure methods can safely be invoked at runtime by invariant-detection tools [30] and by specification-mining tools [27]. As an additional example, in program understanding tools, local invariants about existing objects can be propagated over a call to a pure method.

This section extends our pointer analysis from Chapter 4 to detect *pure* methods. We refer to the extended analysis as the "purity analysis". Section 5.2.1 explains how our purity analysis detects pure methods. Section 5.2.2 explains additional information that our purity analysis can compute for methods that are not pure.

5.2.1 Detection of Pure Methods

Our purity analysis supports a flexible definition of method purity:

Definition 5. *A method is* pure *iff each execution of the method (1) does not perform I/O operations, (2) does not mutate any static field, and (3) does not mutate any* prestate object*. A prestate object is an object allocated before the start of the method. The execution of a method includes the execution of the instructions from the transitive callees.*

This is the same definition that is used in the Java Modeling Language (JML) [55]. This definition allows a pure method to allocate and mutate new objects. Hence, pure methods can use common programming idioms like iterators. Pure methods can also allocate, initialize, and return complex object structures.

Pure methods can allocate and throw exceptions. In Java, virtually every instruction can throw an exception (e.g., each field dereference may throw a `NullPointerException`). Hence, requiring pure methods not to throw any exception would result in a definition that is too strict to be useful.

Our purity analysis is conservative: if the analysis reports that a method is pure, the method satisfies Definition 5. However, the analysis can report false negatives, i.e., the analysis can fail to identify certain pure methods.

Challenges

Checking the absence of I/O operations and static field mutations is simple once we have a static call graph.[1] Therefore, this section focuses on the last condition from Definition 5, the absence of mutations on prestate objects.

To detect mutations on prestate objects, the analysis needs to process store instructions of the form "$v_1.f = v_2$". More specifically, the analysis needs to detect whether v_1 may point to a prestate object. Assuming that any assignment instruction may mutate a prestate object would prevent the analysis from detecting pure methods that mutate newly-allocated objects.

[1] This does not imply that computing a static call graph for Java programs is an easy task; it is not! However, the construction of static call graphs for Java programs is beyond the scope of this thesis.

Inter-procedurally, the analysis needs to propagate mutations from callees into callers. A challenging fact is that objects that are prestate objects for the callee may be new objects for the caller (i.e., the caller allocated those objects before invoking the callees). Therefore, it is possible to have pure methods that invoke impure methods. For example, the pure method `sumX` from Chapter 2 uses a newly-allocated iterator to iterate over a list. The method that advances the iterator through the list is impure: it mutates the iterator state to update the iterator position into the list.

Purity Analysis Overview

The key property that makes our pointer analysis suitable for purity analysis is the fact that inside nodes are guaranteed to model only new objects, i.e., objects allocated in the current execution of the analyzed method (Property 3 on page 44). Additionally, the pointer analysis computes conservative points-to information: assume that the local variable v_1 points to the object o before a store instruction of the form "$v_1.f = v_2$". Then, in the points-to graph that the analysis computes for that program point, v_1 points to at least one node that models the object o (Property 1 on page 44).

Intra-procedurally, our purity analysis uses the points-to information to collect the set of non-inside nodes that are mutated by store instructions. Inter-procedurally, our purity analysis uses the inter-procedural node map to project mutations from the callee into the caller; the purity analysis ignores the inside nodes from the projected set of mutated nodes. To detect pure methods, the purity analysis checks the absence of mutations on non-inside nodes.

Technical Details

We extend the points-to graphs with a new component, a set of *mutated abstract fields*. An abstract field is a field of a specific node, i.e., a pair of the form $\langle n, f \rangle$. The abstract field $\langle n, f \rangle$ records a mutation on the field f of an object modeled by the node n. Because our analysis studies only mutations on prestate objects, it considers only abstract fields of non-inside nodes. Formally,

$$\begin{aligned} G \in\ PTGraph = \{\ \langle L{:}J,\ I,\ O,\ E,\ R,\ W\rangle\ \mid\ & L{:}J \in AStack, \\ & I \in IEdges,\ O \in OEdges,\ E, R \in \mathcal{P}(Node), \\ & W \in \mathcal{P}(AbstrField)\ \} \\ AbstrField = (CNode \setminus CINode) \times Field & \end{aligned}$$

To check whether a method is pure, the analysis checks that: (1) the set of mutated abstract fields is empty at the end of the method, and (2) the method does not transitively contain any unanalyzable CALL. The analysis uses the static call graph to check condition (2).

Discussion: Our purity analysis uses sets of abstract fields for the following reasons:

- An analysis that uses just a boolean flag *pure/impure* has to assume that the caller of an impure method is impure itself. This choice is sometimes too conservative. For example, it would prevent our analysis from detecting the pure method `sumX` in the example from Chapter 2.

  Instead, our analysis keeps track of the mutated nodes. The inter-procedural analysis may discover that a parameter/load node from the callee represents only inside nodes from the caller. In this case, the analysis of the caller ignores the mutations that the callee performs on the parameter/load node.

- Using abstract fields instead of nodes does not gain any precision in the purity analysis. However, recording the mutated field(s) for each node is useful for other purposes, such as generating informative messages about why the analysis considers a method impure (see Section 5.2.2).

Intra-procedural Analysis: We update the transfer functions from Figure 4-3 to maintain the set of mutated abstract fields. In the case of a STORE instruction "$v_1.f = v_2$" at label lb, the updated transfer function records the mutation of the field f of all non-inside nodes pointed to by v_1:

$$[\![lb]\!](\langle L : J,\ I,\ O,\ E,\ R,\ \underline{W}\rangle) = \\ \langle L : J,\ I',\ O,\ E,\ R,\ \underline{W \cup ((L(v_1) \setminus \mathit{CINode}) \times \{f\})}\rangle$$

The underlined parts of the formula above correspond to the new elements on top of the plain pointer analysis from Chapter 4. As in the plain pointer analysis, if f is a field of object type, then $I' = I\ \cup (L(v_1) \times \{f\} \times L(v_2))$. Otherwise, $I' = I$.

The other intra-procedural transfer functions simply propagate the set of mutated abstract fields unchanged.

Inter-procedural Analysis: We update the inter-procedural analysis to propagate the mutations from the callee into the caller. Figure 5-1 presents the updated definition of the function *interproc*. The set of mutated abstract fields for the program point after the CALL contains (1) the abstract fields mutated before the CALL (i.e., the abstract fields from the set W), and (2) the mutated abstract fields from the callee, projected through the inter-procedural node map μ_a. The purity analysis "filters out" the projected abstract fields that refer to inside nodes.

Optimization: As presented above, the purity analysis maintains the sets of mutated abstract fields flow-sensitively: there is one such set for each program point. However, for each method m, the purity analysis uses only the set of modified abstract fields for the end of the method. The analysis uses that set for two purposes: (1) to detect the absence of prestate mutation (i.e., method purity), and (2) to propagate m's mutations to m's callers. Therefore, it suffices to maintain only one set W_m of mutated abstract fields for each method m. With this modification the analysis of STORE/CALL instructions adds each new mutated abstract fields directly to W_m.

$$
\begin{array}{l}
\textit{interproc}:\ \textit{PTGraph}^a \times \textit{PTGraph}^a \times \textit{Instruction} \rightarrow \textit{PTGraph}^a \\
\textit{interproc}(G,\ G_{callee},\ \text{“}v_R = v_0.s(v_1, \ldots, v_j)\text{”}) \ = \\
\quad \textbf{let}\ \ \langle L : [],\ I,\ O,\ E,\ R,\ W\rangle = G \\
\qquad \langle T_{callee}, R_{callee}, W_{callee}\rangle = \textit{summary}(\tau(gc(\rho(G_{callee})))) \\
\qquad \mu_0 = \{\langle n, n\rangle \mid n \in \textit{CINode} \cup \mathcal{G}\} \ \cup\ \left(\bigcup_{i=0}^{k} \{n_{i,1}^P\} \times L(v_i)\right) \\
\qquad \langle I_a,\ O_a,\ E_a,\ \mu_a\rangle = T_{callee}(I,\ O,\ E,\ \mu_0) \\
\qquad L_a = L\,[v_R \mapsto \mu_a(R_{callee})] \\
\qquad \underline{W_a = W \ \cup\ \bigcup_{\langle n,f\rangle \in W_{callee}} (\mu_a(n) \setminus \textit{CINode}) \times \{f\}}\ \ \textbf{in} \\
\quad \alpha_0(\langle L_a : [],\ I_a,\ O_a,\ E_a,\ \emptyset,\ W_a\rangle) \\
\\
\textit{summary}:\ \textit{PTGraph} \rightarrow \textit{IPTransformer} \times \mathcal{P}(\textit{CNode}) \\
\textit{summary}(G_{callee} = \langle \_, \_, \_, \_, R_{callee}, W_{callee}\rangle) = \langle \textit{mct}(G_{callee}),\ R_{callee},\ W_{callee}\rangle
\end{array}
$$

Figure 5-1: Updated definition of *interproc* for the purity analysis. We underline the most significant addition, the equation for computing W_a.

5.2.2 Additional Results of the Purity Analysis

For the impure methods,[2] our purity analysis can compute information that allows the programmer to bound the side-effects of the method. Unlike the stack allocation optimization and the detection of the pure methods, the additional information described in this section is unsound. Therefore, it can be used for program understanding and debugging tasks, but not for program optimizations.

Informative Messages

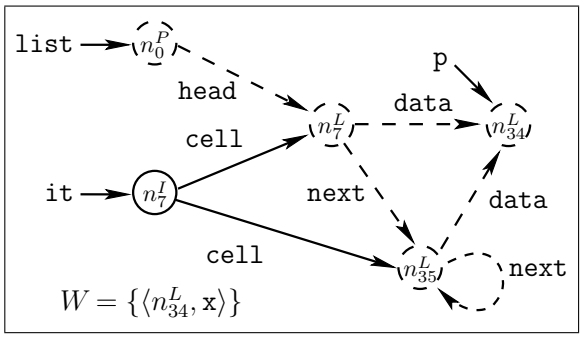
The programmer may benefit from knowing why the analysis considers a method impure. Generating an informative message for a method that transitively executes an I/O operation or contains an unanalyzable CALL is simple. Generating informative messages for mutated abstract fields is more complex. Simply reporting that a load node is mutated is not informative, because a load node is an internal analysis abstraction. Instead, our analysis generates a regular expression that models heap paths that start in the method parameters and end in potentially mutated prestate locations. This task is facilitated by the fact that our analysis uses outside edges to keep track of the references read from escaped objects.

Example 4. Figure 5-2.a presents the code of the method `Main.sumX` from Chapter 2-1, modified to mutate the `x` field of each point from the list. Figure 5-2.b presents

[2]The impure methods are those methods that are not pure.

```
47  class Main {
58    static float sumX(List list) {
59      float s = 0;
50      Iterator it = list.iterator();
51      while(it.hasNext()) {
52        Point p = (Point) it.next();
53        s += p.x; p.x = 0;
54      }
55      return s;
56    }
      ...
    }
```

a. Modified code: added mutation "`p.x = 0`" in line 53



b. Points-to graph for the end of the method `Main.sumX`. For brevity, we ignore the node contexts: they are all 0.

Figure 5-2: Example for the generation of regular expressions.

the points-to graph for the end of the modified method. Notice that the method `sumX` is no longer pure, due to the mutation of the abstract field $\langle n_{34}^L, \mathtt{x}\rangle$. The load node n_{34}^L models objects read from the parameter `list`, along paths modeled by the outside edges. The paths of outside edges from the only parameter node n_0^P to the mutated node n_{34}^L are described by the regular expression `head.next*.data`. Our analysis reports that `Main.sumX` may mutate the locations reachable along the heap path `list.head.next*.data.x`. △

Technically, let G be the points-to graph for the end of a method m. First, the analysis constructs a finite state automaton with the following states: (1) all the nodes from G, (2) an initial state s, and (3) an accepting state t. Each outside edge from G generates a transition in G, labeled with the field of the outside edge. For each parameter, the analysis adds a transition from s to the corresponding parameter node, and labels it with the parameter name. For each mutated abstract field $\langle n, f\rangle$, the analysis adds a transition from n to the accepting state t, and labels it with the field f. Next, the analysis converts the finite state automaton into a regular expression [74], and reports the regular expression to the user.

Read-Only Parameters

Another possibility of bounding the side-effects of an impure method is to identify *read-only* method parameters. The read-only parameters of a method m have the property that the method m mutates only prestate objects that are reachable from the other (non-read-only) parameters or from the static fields.

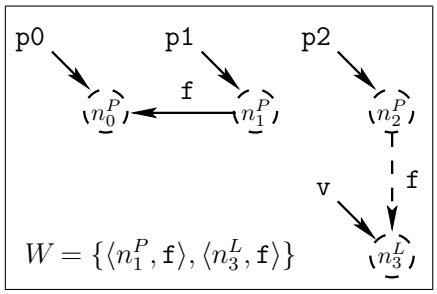
Example 5. Consider the method `put` of the class `java.util.HashMap` (a hash-based implementation of an association map, from the Java standard library). The method `put` has the signature:

```
Object put(Object key, Object value);
```

```
class C {
  C f;
}

1 static void m(C p0, C p1, C p2) {
2   p1.f = p0;
3   v = p2.f;
4   v.f = null;
5 }
```

a. Sample code



b. Points-to graph for the end of the method `m`. For brevity, we ignore the node contexts; they are all 0.

Figure 5-3: Example of the unsoundness of the read-only parameter detection.

The method `put` is impure: it mutates its receiver object (the map), to store the association between `key` and `value`. However, it does not mutate the objects transitively reachable from the parameters `key` and `value`. Hence, the parameters `key` and `value` are read-only. The implicit parameter `this` is not read-only. △

Consider a method m. The analysis detects the read-only parameters of m as follows. Let $G = \langle J, I, O, E, R, W\rangle$ be the points-to graph for the end of m. For each parameter p of m, the analysis computes the set S_p of nodes transitively and reflexively reachable from the corresponding parameter node, along outside edges from O. p is a read-only parameter if any node $n \in S_p$ satisfies the following conditions:

1. n is not mutated: $\forall f \in \mathit{Field}.\ \langle n, f\rangle \notin W$.
2. n is not transitively and reflexively reachable from a directly globally escaped node from $E \cup \{n_{\mathtt{GBL},0}\}$, along inside and outside edges from $I \cup O$. The nodes $E \cup \{n_{\mathtt{GBL},0}\}$ model objects that are potentially reachable from outside the analysis scope (e.g., from an unanalyzable callee). All objects transitively reachable from these objects may by mutated.

The next example illustrates a situation when the above algorithm for detecting read-only parameters is unsound.

Example 6. Consider the code from Figure 5-3.a. The method `m` has three parameters of class `C`. First, `m` creates a reference from its second parameter to the first one: `p1.f = p0`. Next, `m` reads the field `f` of the third parameter and mutates the resulting object. Figure 5-3.b presents the points-to graph for the end of `m`. There are no outside edges starting in the parameter node n_0^P; hence $S_{\mathtt{p0}} = \{n_0^P\}$. The parameter n_0^P is not mutated and does not escape globally. Therefore, the analysis reports that `p0` is a read-only parameter. The other two parameters are not read-only, due to the mutations on the nodes n_1^P and n_3^L.

Consider the following invocation of the method `m`:

```
v1 = new C(); v2 = new C();
m(v1, v2, v2);
```

Notice that the last two parameters of `m` point to the same object. The only object transitively reachable from the first parameter is unreachable from the other two (not read-only) parameters. However, this invocation of `m` does mutate the object pointed to by `v1`! As the parameters `p1` and `p2` point to the same object, the reference read in line 3 is the reference created in line 2, i.e., `v` points to the object pointed to by the "read-only" parameter `p0`. In the points-to graph from Figure 5-3.b, the mutated load node n_3^L may represent the same object as the parameter node n_0^P. △

There are two reasons for the unsoundness from the previous example:

1. The method creates a new reference toward the object pointed to by the "read-only" parameter `p0`. Due to this reference, the object pointed to `p0` becomes reachable from the non-read-only parameters.

2. More importantly, in our analysis, the same escaped object may be represented by several nodes. E.g., the two parameter nodes n_0^P and n_1^P model the same object. As a consequence, the analysis does not detect that the load instruction from line 3 reads the reference created by the instruction from line 2.

   Assuming maximal aliasing between objects transitively pointed to parameters leads to results that are too conservative to be useful. For example, for the method `put` of `java.util.HashMap`, assuming that the key and the value objects are identical to the map object[3] leads to the (useless) result that no parameter of `put` is read-only.

It is possible to strengthen the conditions for read-only parameters in order to prevent the unsoundness from Example 6. For example, the analysis can require that no node transitively reachable from a read-only parameter is the target of an inside edge (i.e., a new reference). Unfortunately, such additional conditions may prevent the analysis from detecting genuine read-only parameters in common cases. For example, consider the following setter method:

```
void setF(C a) { this.f = a; }
```

The parameter `a` is read-only, in spite of the new reference to the object that `a` points to.

As stronger conditions complicate the analysis, decrease its precision (as in the setter method example), and are not covered by the correctness proof from Chapter 6, we prefer the current unsound algorithm for read-only parameter detection.

[3]Perfectly possible from a type point of view, as the declared type of both the key and the value parameters is `Object`.

Chapter 6

Correctness Proof

This chapter formalizes and proves correct the properties of the points-to graphs that the pointer analysis computes. We introduced these properties, in an informal style, in Section 4.2. This chapter considers the pointer analysis from Chapter 4 with the modifications for the purity analysis from Section 5.2.[1] The analysis properties from this chapter imply the correctness of the stack allocation optimization (Section 5.1) and the correctness of the detection of pure methods (Section 5.2.1).

Proof Outline: We present our correctness proof in the context of SmallJava$^-$, a subset of the analyzed language SmallJava (see Section 3.2). SmallJava$^-$ has all the features of SmallJava, but does not support arrays. We define a *concrete semantics* that describes the precise execution of SmallJava$^-$ programs. The concrete semantics allows us to formalize the desired properties of the points-to graphs that the analysis computes.

To compensate for the big difference between the concrete semantics and the pointer analysis, we introduce an intermediate layer, the *abstract semantics*. Given a program trace, the abstract semantics computes a points-to graph for each "interesting" date[2] from the program execution. The abstract semantics uses almost the same transfer functions as the analysis. However, unlike the analysis, the abstract semantics "steps into" the called methods: when it encounters an analyzable CALL, the abstract semantics processes all instructions from the callee, one by one, instead of using a method summary. Additionally, for each interesting date, the abstract semantics explicitly constructs a modeling relation between nodes and objects.

First, we prove that the points-to graphs that the abstract semantics constructs satisfy properties that are even stronger than those from Section 4.2. Next, we prove that the pointer analysis conservatively approximates the abstract semantics: the points-to graphs that the analysis computes are bigger than the points-to graphs that the abstract semantics computes. Intuitively, as the properties from Section 4.2 are monotonic (i.e., they remain true for bigger points-to graphs), the combination of

[1] In particular, each points-to graph contains a set of mutated abstract fields.

[2] We explain later in this chapter what an "interesting date" is.

the two steps proves that the points-to graphs that the analysis computes satisfy the desired properties from Section 4.2.

Chapter Outline: First, Section 6.1 presents the concrete semantics of SmallJava$^-$. Section 6.2 introduces several auxiliary definitions on top of the operational semantics. Section 6.3 uses the concrete semantics and the auxiliary definitions to formalize the desired properties of the points-to graphs that the pointer analysis computes. Section 6.4 introduces the abstract semantics. Section 6.5 lists several properties of the points-to graphs that the abstract semantics computes. These properties are even stronger than those from Section 6.3. Finally, Section 6.6 proves that the pointer analysis is more conservative than the abstract semantics and that the analysis properties from Section 6.3 are valid.

6.1 Concrete Semantics of SmallJava$^-$

SmallJava$^-$ has all the features of SmallJava, but does not support arrays. In particular, SmallJava$^-$ has all the instructions of SmallJava, except NEW ARRAY, ARRAY LOAD, and ARRAY STORE. The concrete semantics describes the precise execution of SmallJava$^-$ programs. Our concrete semantics is an operational semantics, similar to the semantics from References [68, 8].

Program States: Figure 6-1 presents the sets and notations for the concrete semantics. A program state is a tuple $\Xi = \langle A, H, S, TY \rangle$:

- The *thread agenda* $A \in ThreadAgenda$ maintains the state of the different threads of execution from the program.
- The *heap* $H \in Heap$ maintains references between objects.
- The *static field state* S maintains the values of the static fields.
- The *type function* $TY \in OTypes$ assigns to each object its type (i.e., class). The concrete semantics uses TY to handle virtual method calls.

Each time the program executes a NEW instruction, it creates a new, fresh object $o \in Object$. There are two special objects: $o_{\texttt{null}}$ (that models the null pointers) and o_{main} (explained later). The heap H is a curried function: for a given object o_1 and a given field f, $o_2 = H(o_1)(f)$ is the object that the field f of o_1 points to. H is a partial function: $H(o_1)(f)$ is undefined for a (currently) nonexistent object o_1, or for a nonexistent field f. For convenience, we often use the notation $\langle o_1, f, o_2 \rangle \in H$ instead of $H(o_1)(f) = o_2$. For any o_1, f, there is at most one object o_2 such that $\langle o_1, f, o_2 \rangle \in H$. In our text, we write that $\langle o_1, f, o_2 \rangle \in H$ is a "heap reference."

The thread agenda A maps a thread identifier t to the stack that represents the local state of the corresponding thread. The identifier of a thread is the thread object itself. The identifier of the main thread is the dummy object $o_{main} \in Object = ThreadId$.

$$
\begin{array}{rcll}
\Xi & \in & \mathit{State} = \mathit{ThreadAgenda} \times \mathit{Heap} \times \mathit{Static} \times \mathit{OTypes} & \\
o & \in & \mathit{Object} = \{o_{\texttt{null}}, o_{\mathit{main}}, o_0, o_1, \ldots\} & \\
A & \in & \mathit{ThreadAgenda} = \mathit{ThreadId} \to \mathit{JavaStack} & \\
t & \in & \mathit{ThreadId} = \mathit{Object} & \\
K & \in & \mathit{JavaStack} = \texttt{list of}\ (\mathit{LocVar} \times \mathit{Label}) & \\
V & \in & \mathit{LocVar} = \mathit{Var} \to \mathit{Object}\ ; & V_{\text{all–null}} = \lambda v.\ o_{\texttt{null}} \\
lb & \in & \mathit{Label} = \mathit{Method} \times \mathit{Address} & \\
H & \in & \mathit{Heap} = \mathit{Object} \to \mathit{Field} \to \mathit{Object} & \\
S & \in & \mathit{Static} = \mathit{Class} \times \mathit{Field} \to \mathit{Object}\ ; & S_{\text{all–null}} = \lambda\langle C, f\rangle.\ o_{\texttt{null}} \\
TY & \in & \mathit{OTypes} = \mathit{Object} \to \mathit{Class} & \\
T & \in & \mathit{Trace} = \mathit{Date} \to \mathit{State} & \\
d & \in & \mathit{Date} = \mathbb{N} & \\
\mathit{getMeth} & : & \mathit{MethodName} \times \mathit{Class} \to \mathit{Method} &
\end{array}
$$

Figure 6-1: Sets and notations for the concrete semantics. Additionally, we continue to use the definitions from Section 3.2 (see Figure 3-1 on page 34).

The stack K of a thread t is a list of stack frames, one for each method from the current call chain in thread t. A stack frame contains the state of the local variables for the corresponding method and the current address inside that method. The state V of the local variables attaches to a local variable v the object $o = V(v)$ that v points to. At the beginning of a method, each local variable points to $o_{\texttt{null}}$, except the formal parameters that point to the actual argument objects. In the special local variable state $V_{\text{all–null}}$, all local variables point to $o_{\texttt{null}}$.

The state S of the static fields maps each static field (represented as a pair of the declaring class and the actual field) to the object it points to. In the special static field state $S_{\text{all–null}}$, each static field points to $o_{\texttt{null}}$.

Program Traces: A concrete execution trace T of a program is a series of states, indexed by *date*. The concrete semantics uses a discreet time model: a date is a natural number, $\mathit{Date} = \mathbb{N}$.

Any trace T starts with the initial state Ξ_0. The stack of the unique thread from Ξ_0 contains a single frame for the main method m_{main}. For simplicity, we assume that m_{main} does not have any parameter, i.e., all parameters are hard-coded into the program. Formally,

$$\Xi_0 = \langle\ \{\ o_{\mathit{main}} \mapsto [\langle V_{\text{all–null}}, \langle m_{\mathit{main}}, 0\rangle\rangle]\ \},\ \{\},\ S_{\text{all–null}},\ \{\}\ \rangle$$

Instruction $P(lb)$	Transition
COPY $v_1 = v_2$	$\langle A\,[t \mapsto \langle V, lb\rangle : K]\,, H, S, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle V\,[v_1 \mapsto V(v_2)], next(lb)\rangle : K]\,, H, S, TY\rangle$
NEW $v =$ `new` C	$\langle A\,[t \mapsto \langle V, lb\rangle : K]\,, H, S, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle V\,[v \mapsto o], next(lb)\rangle : K]\,, H_2, S, TY_2\rangle$ where o is a fresh object, $H_2 = H\left[o \mapsto \{f \mapsto o_{\texttt{null}}\}_{f \in fields(C)}\right]$ and $TY_2 = TY\,[o \mapsto C]$
NULLIFY $v =$ `null`	$\langle A\,[t \mapsto \langle V, lb\rangle : K]\,, H, S, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle V\,[v \mapsto o_{\texttt{null}}], next(lb)\rangle : K]\,, H, S, TY\rangle$
STORE $v_1.f = v_2$	$\langle A\,[t \mapsto \langle V, lb\rangle : K]\,, H, S, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle V, next(lb)\rangle : K]\,, H_2, S, TY\rangle$ where $H_2 = H\,[\,V(v_1) \mapsto H(\,V(v_1))\,[f \mapsto V(v_2)]]$
STATIC STORE $C.f = v$	$\langle A\,[t \mapsto \langle V, lb\rangle : K]\,, H, S, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle V, next(lb)\rangle : K]\,, H, S\,[\langle C, f\rangle \mapsto V(v)]\,, TY\rangle$
LOAD $v_2 = v_1.f$	$\langle A\,[t \mapsto \langle V, lb\rangle : K]\,, H, S, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle V\,[v_2 \mapsto H(\,V(v_1))(f)], next(lb)\rangle : K]\,, H, S, TY\rangle$
STATIC LOAD $v = C.f$	$\langle A\,[t \mapsto \langle V, lb\rangle : K]\,, H, S, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle V\,[v \mapsto S(C, f)], next(lb)\rangle : K]\,, H, S, TY\rangle$
IF `if` $(\ldots)$ `goto` a_t	$\langle A\,[t \mapsto \langle V, lb\rangle : K]\,, H, S, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle V, lb_2\rangle : K]\,, H, S, TY\rangle$ where $lb = \langle m, a\rangle$ and $lb_2 = \begin{cases} \langle m, a_t\rangle & \text{if the condition is true} \\ next(lb) & \text{otherwise} \end{cases}$
THREAD START `start` v	$\langle A\,[t \mapsto \langle V, lb\rangle : K]\,, H, S, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle V, next(lb)\rangle : K]\,[\,V(v) \mapsto K_{startee}]\,, H, S, TY\rangle$ where $startee = getMeth(\text{“}run\text{”}, TY\,(\,V(v)))$ $K_{startee} = [\langle V_{\text{all–null}}\,[p_0 \mapsto V(v)], \langle startee, 0\rangle\rangle]$
NOP `nop`	$\langle A\,[t \mapsto \langle V, lb\rangle : K]\,, H, S, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle V, next(lb)\rangle : K], H, S, TY\rangle$

Figure 6-2: Transition relation "⇒" for the concrete semantics - Part 1 of 2. $P(lb)$ is the instruction that the concrete semantics executes in the current transition. $next(lb)$ is the label immediately after label lb: $next(\langle m, a\rangle) = \langle m, a + 1\rangle$ (see Figure 3-1).

Instruction $P(lb)$	Transition
CALL $v_R = v_0.s(v_1, \ldots, v_j)$	$\langle A\,[t \mapsto \langle V, lb\rangle : K]\,, H, S, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle V_{callee}, \langle callee, 0\rangle\rangle : \langle V, lb\rangle : K]\,, H, S, TY\rangle$ where $callee = getMeth(s, TY(V(v_0)))$ $V_{callee} = V_{\text{all-null}}\,[p_i \mapsto V(v_i)]_{0\le i\le j}$ (A method of arity k has formal parameters $p_0, p_1, \ldots p_{k-1}$.)
RETURN `return` v	$\langle A\,[t \mapsto \langle V_{callee}, lb\rangle : \langle V, lb_2\rangle : K]\,, H, S, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle V[v_R \mapsto V_{callee}(v)], next(lb_2)\rangle : K]\,, H, S, TY\rangle$ where v_R is the variable that stores the result of the corresponding CALL (located at label lb_2) --- $\langle A\,[t \mapsto [\langle V, lb\rangle]]\,, H, S, TY\rangle \Rightarrow \langle A, H, S, TY\rangle$

Figure 6-3: Transition relation "⇒" for the concrete semantics - Part 2 of 2 (continued from Figure 6-2).

All local variables of the main method have `null` value. Similarly, all static fields have `null` value. The current label for the main thread is the first label inside the main method, i.e., $\langle m_{main}, 0\rangle$. As no object has been created yet, the initial heap and object type function are not defined for any object.

Concrete Transitions: Figures 6-2 and 6-3 present the transition relation "⇒" for the concrete semantics. In each step, the concrete semantics non-deterministically selects a thread t from the thread agenda and executes its current instruction. If the stack of thread t is $K = \langle V, lb\rangle : K_{tail}$, then the concrete semantics executes the instruction $P(lb)$ from label lb. Most of the instructions do not require any explanation. In spite of the hairy notation, the transition for a STORE instruction "$v_1.f = v_2$" simply updates the heap to make $H(o_1)(f) = o_2$ where $o_1 = V(v_1)$ and $o_2 = V(v_2)$. It does not change the value of H for other combinations of locations and fields.

Normally, the transition for a RETURN instruction pops the topmost stack frame and passes the returned value into the caller. If there is no caller stack frame (i.e., if K_{tail} is empty), then the program executes a RETURN from the "root" method of the thread t. In this case, the concrete semantics removes t from the thread agenda and ignores the returned value.

The transition for a CALL instruction invokes the method named s of the *receiver* object $o = V(v_0)$. The concrete semantics retrieves the class of o, $C = TY(o)$, and uses the auxiliary function $getMeth$ to obtain the appropriate method: $callee = getMeth(s, C)$. We provide only an informal definition of $getMeth$: if class C defines a method names s, then $getMeth(C, s)$ returns that method. Otherwise, $getMeth$

searches into C's superclass, next into the superclass of the superclass, etc., until it finds a method named s.

Discussion: The concrete semantics of SmallJava$^-$ performs several simplifications over the semantics of Java:

- The concrete semantics of SmallJava$^-$ does not handle the execution errors: e.g., it does not prevent the program from jumping to an invalid address, nor from writing a field of the object $o_{\texttt{null}}$. We consider only programs that avoid these situations with the help of (1) simple static checks (e.g., for detecting invalid jumps), and (2) explicit dynamic checks before each potentially "problematic" instruction (e.g., null-pointer checks before reading/writing an object field). Our analysis implementation uses an intermediate program representation that satisfies these conditions (see the "Exceptions" paragraph in Section 8.1.2 on page 119).

- The concrete semantics of SmallJava$^-$ uses the *sequential consistency* memory model [51]: the memory actions appear to execute one at a time in a single total order; the actions of each thread appear in this total order in the same order in which they appear in the program source code. Java uses a weaker consistency model. However, according to the most recent Java memory model [63] "correct programs are those that are data-race-free; such programs are guaranteed sequential consistency." We are not aware of any rigorous pointer analysis correctness proof that considers a weak consistency memory model; in fact, we are not aware of any such proof that explicitly considers threads.

- SmallJava$^-$ has no synchronization instructions; equivalently, the concrete semantics treats each such instruction as a NOP. The schedulings that the concrete semantics allows are a superset of the possible schedulings in the presence of synchronization instructions. Therefore, our results remain valid in the presence of synchronization instructions.

6.2 Auxiliary Definitions

When we presented the intended meaning of the analysis results in Section 4.2, we used several informally defined concepts: "method activation $A(m)$", "objects reachable from outside $A(m)$", etc. This section formalizes these concepts.

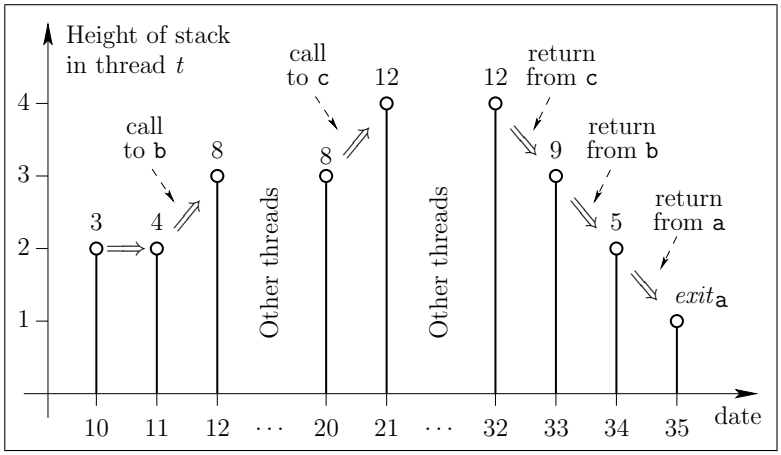
6.2.1 Method Activation and Interesting Dates

Consider a method m. During the analysis of method m, the analysis scope consists of the method m and the methods it transitively invokes using analyzable CALLs.

An *activation* $A(m)$ of a method m is an execution of the analysis scope for m. There can be multiple activations of m in a program execution. We identify a method

```
1  class C {
2    C a() {
3      nop;
4      v = this.b();
5      return v;
6    }
7    C b() {
8      v = this.c();
9      return v;
10   }
11   C c() {
12     return this;
13   }
14 }
```

a. Source code



b. Graphic representation of a possible execution

Figure 6-4: Sample program execution. An activation of method `a` starts at date 10, in thread t. We represent a program state by the height of the stack from thread t, and the current label in the top-most method in t: this is the label of the instruction that the thread t executes the next time t is scheduled. E.g., at date 11, the thread t has a stack of height 2, and the current label in its top-most method is 4.

activation by the date when its first instruction starts executing. Detecting the instructions from an activation $A(m)$ is non-trivial because of the possible interleavings with other threads. For each activation $A(m)$, the list $ID_{A(m)}$ of *interesting dates* is the list of the dates when $A(m)$ starts and finishes executing an instruction. We use the following notation:

$$ID_{A(m)} = [id_0, \ldots, id_{2i}, id_{2i+1}, \ldots]$$

The list $ID_{A(m)}$ can be infinite, for non-terminating activations. $A(m)$ executes an instruction from date id_{2i} to date id_{2i+1}. Usually, $id_{2i+1} = id_{2i} + 1$, except for the case when the executed instruction is an unanalyzable CALL; in this case, id_{2i+1} is the date when the matching RETURN terminates ($A(m)$ does not contain the instructions of the methods transitively invoked by an unanalyzable CALL). Between id_{2i+1} and $id_{2(i+1)}$, the program executes only instructions from other threads.

Example 7. Consider the program fragment from Figure 6-4.a; the code does not compute anything interesting, we use it only to illustrate method activations and interesting dates. Figure 6-4.b presents a possible program execution where method `a` starts executing at date 10, in thread t. Let $A(\texttt{a})$ be the activation of method `a` that starts at date 10. The activation $A(\texttt{a})$ invokes the methods `b` and `c`, terminates at date 35, and is interleaved with instructions from other threads.

First, assume that all CALLs from the example are analyzable. In this case, the

activation $A(\mathtt{a})$ that starts at date 10 has the following interesting dates:

$$ID_{A(\mathtt{a})} = [10, 11,\ 11, 12,\ 20, 21,\ 32, 33,\ 33, 34,\ 34, 35]$$

The activation $A(\mathtt{a})$ executes its first instruction between the dates 10 and 11. $A(\mathtt{a})$ executes its second instruction between the dates 11 and 12. The date 11 appears twice in the list of interesting dates: once as the termination date for the first instruction and once as the starting date for the second one. No instruction from other threads executes between the first two instructions of $A(\mathtt{a})$. The RETURN executed in the transition between the dates 32 and 33 matches the CALL executed by $A(\mathtt{a})$

Next, assume that the CALL from line 4 (executed at date 11) is unanalyzable. In this case, $A(\mathtt{a})$ does not contain the instructions executed between the CALL to `b` and the corresponding RETURN; accordingly,

$$ID_{A(\mathtt{a})} = [10, 11,\ 11, 34,\ 34, 35]$$

Notice that the successor of the second occurrence of 11 is 34, the date when the program returns from `b`: 34 is the earliest date strictly after 11 when the stack of the thread t has the same height (2) as at date 11. The RETURN instruction executed at date 34 terminates the activation $A(\mathtt{a})$: at date 35 the stack of thread t is shorter than at date 10 (the beginning of $A(\mathtt{a})$). The current label inside t at date 35 is the current label inside `a`'s caller; Figure 6-4.b uses the symbolic label $exit_{\mathtt{a}}$ for the end of the method `a`. △

Technically, we use the following definition:

Definition 6 (Interesting dates for $A(m)$). *Consider a concrete execution trace $\Xi_0 \Rightarrow \Xi_1 \Rightarrow \ldots$ and a method m. Let d_0 be a date such that the transition $\Xi_{d_0} \Rightarrow \Xi_{d_0+1}$ executes the first instruction of m (i.e., the instruction from label $\langle m, 0\rangle$), in a thread t. Then, the list $ID_{A(m)} = [id_0, \ldots, id_{2i}, id_{2i+1}, \ldots]$ of* interesting dates *for the activation $A(m)$ that starts at date d_0 is the list constructed by the following algorithm:*

1. *Set $id_0 = d_0$ and initialize a counter i: $i \leftarrow 0$.*
2. *Define id_{2i+1} by a case analysis on the instruction executed by thread t at date id_{2i}:*
   - *Unanalyzable CALL: id_{2i+1} is the earliest date $d > id_{2i}$ such that the stack for thread t has the same height at date d as at date id_{2i} (this is the date when the matching RETURN terminates). If such a date does not exist (i.e., if the callee does not terminate), then stop.*
   - *Otherwise: $id_{2i+1} = id_{2i} + 1$.*
3. *If the stack of thread t has smaller height at date id_{2i+1} than at date d_0, then stop. This case occurs when the instruction executed at date id_{2i} is the RETURN that terminates $A(m)$.*

Otherwise, let $id_{2(i+1)}$ be the earliest date $d \geq id_{2i+1}$ when the thread t starts executing an instruction. If no such date exists (e.g., because the thread t is never re-scheduled for execution), stop.

Repeat from step 2 with $i \leftarrow i + 1$.

6.2.2 Intra-procedural Dates for a Method Activation

Some of the interesting dates for $A(m)$ may correspond to the execution of instructions from the callees of m, not from m; e.g., in Example 7, the interesting date 20 from $ID_{A(\texttt{a})}$ corresponds to the execution of an instruction from `b`, not from `a`. The *intra-procedural* dates are those interesting dates that correspond to the execution of instructions from the execution of m that is the "root" of $A(m)$.

The intra-procedural dates are important to our proof because the analysis of the method m constructs points-to graphs for the program points *inside* m. These points-to graphs model the concrete program states at the intra-procedural dates.

Example 8. Consider the activation $A(\texttt{a})$ from Example 7. The list of intra-procedural dates for $A(\texttt{a})$ is $IP_{A(\texttt{a})} = [10, 11,\ 11, 34,\ 34, 35]$. △

The following definition identifies the intra-procedural dates by using the height of the stack of the thread t, the thread where the activation $A(m)$ takes place:

Definition 7 (Intra-procedural dates). *Consider a program trace, a method m, and a method activation $A(m)$ that takes place in thread t. The list $IP_{A(m)}$ of* intra-procedural dates *for $A(m)$*

$$IP_{A(m)} = [ip_0, \ldots, ip_{2i}, ip_{2i+1}, \ldots]$$

is the list obtained by preserving from $ID_{A(m)}$ only the dates when the height of t's stack is smaller or equal to the height of t's stack at date id_0 (the beginning of $A(m)$).

For each intra-procedural date ip_{2i}, either a CALL instruction from m starts at ip_{2i} and the corresponding RETURN terminates at ip_{2i+1} or the transition from ip_{2i} to ip_{2i+1} executes an instruction other than a CALL.[3] Between ip_{2i+1} and $ip_{2(i+1)}$, the program executes only instructions from other threads.

Definition 7 uses "smaller or equal" instead of "equal" to handle the case when $A(m)$ terminates. In this case, the last interesting date $d_{final} \in ID_{A(m)}$ corresponds to the end of the RETURN instruction that terminates $A(m)$. At date d_{final}, the height of t's stack is strictly smaller than at the beginning of $A(m)$ (e.g., see date 35 in Example 7). We want to include d_{final} in $IP_{A(m)}$: the points-to graph that the analysis constructs for the program point before $exit_m$ models the program state at date d_{final}.

[3]Proof sketch: Obviously, $id_0 \in IP_{A(m)}$. Let $id_{2i} \in ID_{A(m)}$ that appears in $IP_{A(m)}$. If the instruction executed at id_{2i}, is not a CALL, then id_{2i+1} appears in $IP_{A(m)}$ too (because non-CALL instructions do not increase the stack height). If the instruction executed at id_{2i} is a CALL, then the next intra-procedural date in $IP_{A(m)}$ is the date id_{2j+1} when the corresponding RETURN terminates (in between, the stack is strictly taller than at date id_0).

6.2.3 Escaped Objects

Intuitively, an object is reachable if (1) it is pointed to by a local variables or a static field, or (2) it is reachable along heap edges from another reachable objects. Formally,

Definition 8 (Reachable objects in a concrete state). *Given a concrete state $\Xi \in State$, $RObjs(\Xi)$, the set of reachable objects in state Ξ, is the least fixed point*[4] *of the following constraints:*

$$\frac{\Xi = \langle A\,[t \mapsto K_1@(\langle V\,[v \mapsto o], lb\rangle : K_2)]\,,\ H,\ S,\ TY\rangle}{o \in RObjs(\Xi)} \tag{6.1}$$

$$\frac{\Xi = \langle A,\ H,\ S\,[\langle C, f\rangle \mapsto o]\,,\ TY\rangle}{o \in RObjs(\Xi)} \tag{6.2}$$

$$\frac{\Xi = \langle A, H, TY\rangle \quad \langle o_1, f, o_2\rangle \in H \quad o_1 \in RObjs(\Xi)}{o_2 \in RObjs(\Xi)} \tag{6.3}$$

Definition 9. *Consider a trace T and an activation $A(m)$ in thread t. Let Ξ_d be the program state at date d. $outside_{A(m)}(\Xi_d)$ is a state that is almost identical to Ξ_d, but without the stack frames corresponding to methods from $A(m)$. More precisely, in $outside_{A(m)}(\Xi_d)$, the stack of the thread t contains only the stack frames below*[5] *the first stack frame of $A(m)$ and the stack frames above the first stack frame corresponding to a method that executes an unanalyzable CALL, if any.*

Example 9. Consider the code and the program execution from Example 7, with the graphic representation from Figure 6-4. Let $A(\texttt{a})$ be the activation of the method `a` that starts at date 10, and assume that the call to method `b` in line 4 is unanalyzable. In the state Ξ_{21} at date 21, the stack of the thread t has the form

$$[\ \langle V_4, 12\rangle,\ \langle V_3, 8\rangle,\ \boxed{\langle V_2, 4\rangle},\ \langle V_1, lb_1\rangle\ ]$$

The leftmost element is the stack top, and the rightmost element is the stack bottom. The four stack frames correspond to the following methods (in left-to-right order): `c`, `b`, `a`, and the caller of `a`. Before the start of $A(\texttt{a})$, the stack of thread t had only one stack frame $\langle V_1, lb_1\rangle$. The framed stack frame is the only one corresponding to methods from $A(\texttt{a})$. In the state $outside_{A(\texttt{a})}(\Xi_{21})$, the stack of the thread t is

$$[\ \langle V_4, 12\rangle,\ \langle V_3, 8\rangle,\ \langle V_1, lb_1\rangle\ ]$$

$outside_{A(\texttt{a})}(\Xi_{12})$ preserves the stack frame for the caller of `a`. As the call to method `b` in line 4 is unanalyzable, $outside_{A(\texttt{a})}(\Xi_{12})$ also preserves the stack frames for the methods `b` and `c`. △

Intuitively, $outside_{A(m)}(\Xi_d)$ contains the stack frames corresponding to the code "outside" the analysis scope for method m. The state $outside_{A(m)}(\Xi_d)$ is an artifact for our proof; it is not the result of a valid execution.

[4] *Over the usual ordering for set: set inclusion.*

[5] *"Below" and "above" are relative to our convention that a call stack grows up.*

Definition 10 (Escaped objects). *Consider a trace T, an activation $A(m)$ in thread t, and a date d. The object o* escapes from *$A(m)$ at date d iff $o \in RObjs(outside_{A(m)}(\Xi_d))$. The object o* is captured in *$A(m)$ iff o does not escape from $A(m)$.*

6.2.4 Additional Definitions

Consider an activation $A(m)$ of a method m. Let t be the thread where $A(m)$ takes place. For each date $d \in Date$, let $\langle V_d, lb_d \rangle$ be the top frame from the stack of the thread t. I.e., V_d is the state of the local variables for the top method from thread t, at date d. If d is an even-numbered interesting date $d = id_{2i} \in ID_{A(m)}$, then $P(lb_{id_{2i}})$ is the instruction that is executed in the transition $\Xi_{id_{2i}} \Rightarrow \Xi_{id_{2i}+1}$.

Definition 11 (Objects allocated by $A(m)$). *For each date $d \in Date$, $Allocs_d^{A(m)}$ denotes the set of all objects that $A(m)$ allocates before the date d; each object is paired up with the label of the NEW instruction that allocated it:*

$$Allocs_d^{A(m)} = \{\ \langle V_{id_{2i}+1}(v),\ lb_{id_{2i}} \rangle \ \mid\ id_{2i} \in ID_{A(m)},\ id_{2i} < d,\ P(lb_{id_{2i}}) = \text{“}v = \texttt{new } C\text{”}\ \}$$

Note: $V_{id_{2i}+1}(v)$ is the value of the local variable v immediately after the NEW instruction, i.e., $V_{id_{2i}+1}(v)$ is the newly allocated object.

$Allocs_{\text{all}}^{A(m)}$ denotes the set of all objects that $A(m)$ allocates, paired with the labels of their allocation sites:

$$Allocs_{\text{all}}^{A(m)} = \bigcup_{d \in Date} Allocs_d^{A(m)}$$

Definition 12 (Heap references created by $A(m)$). *For each date $d \in Date$, $I_d^{A(m)}$ denotes the set of all heap references that $A(m)$ creates before the date d:*

$$I_d^{A(m)} = \{\ \langle V_{id_{2i}}(v_1),\ f,\ V_{id_{2i}}(v_2) \rangle \ \mid\ id_{2i} \in ID_{A(m)},\ id_{2i} < d,\ P(lb_{id_{2i}}) = \text{“}v_1.f = v_2\text{”}\ \}$$

Intuitively, this definition finds each STORE instruction “$v_1.f = v_2$” that $A(m)$ executes before date d, identifies the objects pointed to by v_1 and v_2, and adds the newly created heap reference to $I_d^{A(m)}$.

Definition 13 (Locations mutated by $A(m)$). *For each date $d \in Date$, $W_d^{A(m)}$ denotes the set of all memory locations mutated by $A(m)$ before the date d. A memory location is a pair of an object and a field: $\langle o, f \rangle$ is the memory location that corresponds to the field f of the object o.*

$$W_d^{A(m)} = \{\ \langle V_{id_{2i}}(v_1),\ f \rangle \ \mid\ id_{2i} \in ID_{A(m)},\ id_{2i} < d,\ P(lb_{id_{2i}}) = \text{“}v_1.f = v_2\text{”}\ \}$$

6.3 Formal Properties of the Analysis Results

Section 4.2 informally presented the intended meaning of the points-to graphs that the analysis computes. Now, we have all the tools for a formal presentation.

Consider a program execution trace and let $A(m)$ be an activation of a method m, taking place in thread t. Consider an arbitrary intra-procedural date $d \in IP_{A(m)}$. Let lb_d be the label defined as follows:

- If the height of the stack for the thread t is smaller at date d than at date ip_0 (i.e., d is the termination date of the RETURN instruction that terminates $A(m)$), then $lb_d = exit_m$, the special label for the end of m.
- Otherwise, let lb_d be the current label inside the thread t. Due to the definition of the intra-procedural dates, lb_d is a label inside the method m.

Let $G = \langle L : [], I, O, E, R, W \rangle$ be the analysis points-to graph that the analysis computes for the program point before lb_d, i.e., $G = \circ A(lb_d)$. With these notations, our analysis guarantees that there exists a *modeling relation* between nodes and objects $\rho \subseteq CNode \times Object$ such that the following four properties hold:

Property 5. *If $lb_d \neq exit_m$, then the abstract state of local variables L conservatively models the state of the local variables of m. Let V_d be the state of the local variables of method m, at date d. Then:*

$$\forall v \in Var.\ \forall o \in Object.\\ (o = V_d(v)) \ \wedge \ (o \neq o_{\texttt{null}}) \ \rightarrow \ (\exists n \in CNode.\ n\ \rho_d\ o \ \wedge \ n \in L(v))$$

Note: if $lb_d = exit_m$, then d is the termination date for the final RETURN from $A(m)$. At that date, the stack frame for m (including the state of the local variables) no longer exists.

Property 6. *The set of inside edges I conservatively models all non-null heap references created by $A(m)$ between objects that are still reachable at date d:*

$$\forall o_1, o_2 \in RObjs(\Xi_d) \setminus \{o_{\texttt{null}}\}.\ \forall f \in Field.\\ \langle o_1, f, o_2 \rangle \in I_d^{A(m)} \ \rightarrow \ \exists n_1, n_2 \in CNode.\ (n_1\ \rho\ o_1) \ \wedge \ (n_2\ \rho_d\ o_2) \ \wedge \ \langle n_1, f, n_2 \rangle \in I$$

Property 7. *Any inside node $n^I_{lb,0}$ models only objects allocated by $A(m)$ by executing the NEW instruction from label lb:*

$$\forall lb \in Label.\ \forall o \in Object.\ \ n^I_{lb,0}\ \rho_d\ o \ \rightarrow \ \langle o, lb \rangle \in Allocs^{A(m)}_{\text{all}}$$

Property 8. *Let o be an object allocated by $A(m)$ by executing the NEW instruction from label lb. If the inside node $n^I_{lb,0}$ is captured, then (1) o is captured in $A(m)$, and (2) $n^I_{lb,0}$ is the only node that models o:*

$$\forall o \in Object.\ \forall lb \in Label.\\ \langle o, lb \rangle \in Allocs^{A(m)}_{\text{all}} \ \wedge \ \neg e(G)(n^I_{lb,0}) \ \rightarrow \\ \neg RObjs(outside_{A(m)}(\Xi_d)) \ \wedge \ (\forall n \in CNode.\ n\ \rho_d\ o \ \rightarrow \ n = n^I_{lb,0})$$

Property 8 establishes the correctness of the stack allocation optimization (Section 5.1). Let o be an object allocated by $A(m)$ by executing the NEW instruction from label lb. Let d_f be the termination date of the RETURN that terminates $A(m)$ and assume that $n^I_{lb,0}$ is captured in $G = \circ A(exit_m)$. As there are no stack frames of $A(m)$ at date d_f, $outside_{A(m)}(\Xi_{d_f}) = \Xi_{d_f}$. By Property 8, o is unreachable in Ξ_{d_f}. Hence, the lifetime of o is included in the lifetime of the stack frame for method m.

We add an extra property that proves the correctness of the detection of pure methods (Section 5.2.1).

Property 9. *The set W of mutated abstract fields conservatively models all locations that are (1) mutated by $A(m)$ before date d, and (2) allocated outside $A(m)$:*

$$\forall o \in Object.\ \forall f \in Field.$$
$$\langle o, f\rangle \in W_d^{A(m)} \ \wedge\ o \notin Allocs_{\text{all}}^{A(m)} \ \rightarrow\ \exists n \in CNode.\ (n\ \rho_d\ o)\ \wedge\ \langle n, f\rangle \in W$$

Property 9 uses an upper approximation of the set of prestate objects: the set of objects allocated outside $A(m)$. If Property 9 is valid and the set W from the points-to graph for the end of the method m is empty, then $A(m)$ does not mutate any prestate object.

6.4 Abstract Semantics

The abstract semantics works with a specific activation of a method m. Consider a concrete execution trace $\Xi_0 \Rightarrow \Xi_1 \Rightarrow \ldots$ and an activation $A(m)$ of m. The abstract semantics of $A(m)$ computes a points-to graph for each interesting date of $A(m)$. Each points-to graph models the execution of $A(m)$ up to the corresponding interesting date.

The abstract semantics uses almost the same transfer functions as the analysis. However, unlike the analysis, the abstract semantics "steps into" the called methods: when it encounters an analyzable CALL, the abstract semantics processes all instructions from the callee, one by one, instead of using a method summary. Additionally, for each interesting date, the abstract semantics explicitly constructs a modeling relation between nodes and objects.

The points-to graphs from the abstract semantics are a generalization of the analysis points-to graphs. The abstract semantics models the state of the local variables from m's transitive callees. Hence, the points-to graphs from the abstract semantics may have abstract stacks with more than one element.

Note on Node Contexts: Our correctness proof relies on the fact that the pointer analysis is more conservative than the abstract semantics: the pointer analysis constructs bigger points-to graphs than the abstract semantics. The node contexts are essential for this proof strategy. Consider an analyzable CALL that invokes the method *callee*. Without node contexts, when the abstract semantics steps inside

callee, it may reuse nodes and edges created by previous passes through *callee* (*callee* may be invoked by other CALLs from $A(m)$). As the pointer analysis analyzes *callee* in isolation (and next reuses the result for each call to *callee*) it may construct smaller points-to graphs (e.g., fewer edges) than the abstract semantics. In the presence of node contexts, when the abstract semantics steps inside a callee, the nodes it creates (e.g., the inside nodes associated with the NEW instructions) are distinct from existing nodes (e.g., the inside nodes associated with the same NEW instructions from previous invocations of *callee*). Each RETURN instruction decreases the node contexts, gradually "merging" nodes together. This temporary context sensitivity is essential for our proofs. An early attempt to prove the analysis correctness failed precisely due to the lack of context sensitivity.

This section has the following structure: First, Section 6.4.1 defines a conservative approximation of the set of escaped objects. Next, Section 6.4.2 uses this auxiliary definition and describes how the abstract semantics computes the points-to graphs for the interesting dates of $A(m)$.

6.4.1 Concrete Escape Predicates

The abstract semantics uses a conservative approximation of the set of escaped objects. More specifically, there is one *concrete escape predicate*[6] $e_d : Object \to \{\texttt{true}, \texttt{false}\}$ for each date d. As we prove below (Lemma 5), if an object o escapes from $A(m)$ at date d (according to Definition 10), then $e_d(o)$ is true. We use the concrete escape predicates because their definition resembles the Definition 2 for the (analysis) escape predicates on nodes: the concrete escape predicates identify the objects that escape directly (e.g., because the program stores a reference to them in a static field) and propagate the escape information along the heap references created by $A(m)$.

For each date $d \in Date$, let $\langle V_d, lb_d \rangle$ be the top frame from the stack of the thread t. I.e., V_d is the state of the local variables for the top method from thread t, at date d. If d is an even-numbered interesting date $d = id_{2i} \in ID_{A(m)}$, then $P(lb_{id_{2i}})$ is the instruction executed in the concrete semantics transition $\Xi_{id_{2i}} \Rightarrow \Xi_{id_{2i}+1}$.

Definition 14 (Concrete escape predicates). *The family of* concrete escape predicates $e_d : Object \to \{\texttt{true}, \texttt{false}\}, d \in Date$, *is the least fixed point of the following constraints (the ordering relation $\sqsubseteq$ is given in the first constraint):*

$$e_d \sqsubseteq e_{d+1}, \text{ i.e., } \forall o, e_d(o) \to e_{d+1}(o) \tag{6.4}$$

$$\frac{\forall lb.\ \langle o, lb \rangle \notin Allocs_{\text{all}}^{A(m)}, \text{ i.e., } o \text{ not allocated by } A(m)}{e_0(o)} \tag{6.5}$$

$$\frac{e_d(o_1) \quad \langle o_1, f, o_2 \rangle \in I_d^{A(m)}}{e_d(o_2)} \tag{6.6}$$

[6]The adjective "concrete" distinguishes these predicates from the analysis escape predicates.

$$\frac{P(lb_{id_{2i}}) = \text{“}\texttt{start}\ v\text{”}}{e_{id_{2i+1}}(V_{id_{2i}}(v))} \tag{6.7}$$

$$\frac{P(lb_{id_{2i}}) = \text{“}C.f = v\text{”}}{e_{id_{2i+1}}(V_{id_{2i}}(v))} \tag{6.8}$$

$$\frac{P(lb_{id_{2i}}) = \text{“}v_R = v_0.s(v_1, \ldots, v_k)\text{” } \textit{is unanalyzable}}{e_{id_{2i+1}}(V_{id_{2i}}(v_j)),\ \forall j \in \{0, \ldots, k\}} \tag{6.9}$$

$$\frac{\begin{array}{c}P(lb_{id_{2r}}) = \text{“}\texttt{return}\ v\text{” } \textit{is final RETURN of } A(m) \\ \textit{(i.e., thread } t \textit{ has the same stack height at } id_{2r} \textit{ and } id_0\textit{)}\end{array}}{e_{id_{2r+1}}(V_{id_{2r}}(v))} \tag{6.10}$$

Constraint 6.4 makes the escape predicates cumulative: once an object escapes, it escapes forever. Constraint 6.5 marks as escaped all objects created outside $A(m)$.[7] Constraint 6.6 propagates escapability across heap references. Although escapability propagates along all heap references, it suffices to consider only the heap references created by $A(m)$: if a program part outside $A(m)$ creates a reference, then it was able to access both ends of the newly created reference and so, those objects already escaped. The next constraints indicate how the instruction that $A(m)$ executes from id_{2i} to id_{2i+1} affects the escape predicate after the instruction. Several instructions escape objects outside $A(m)$: THREAD STARTs, STATIC STOREs, unanalyzable CALLs, and the final RETURN of $A(m)$. We discuss the case of the final RETURN; the other cases are similar. Suppose the final RETURN instruction from $A(m)$ (if it exists), executes from id_{2r} to $id_{2r+1} = id_{2r} + 1$ and has the form “`return` v.” After its execution, $o = V_{id_{2r}}(v)$ is reachable from the caller; accordingly, Constraint 6.10 sets $e_{id_{2r+1}}(o)$ to true.

The next lemma proves that the concrete escape predicates conservatively approximate Definition 10 for the escaped objects:

Lemma 5. *If the non-null object o escapes at date $d \in Date$, then $e_d(o)$ holds:*

$$\forall d \in Date, \forall o \in RObjs(outside_{A(m)}(\Xi_d)) \setminus \{o_{\texttt{null}}\}.\ e_d(o)$$

Proof sketch. Induction on the execution trace and case analysis on the instruction executed at each step. The proof is technical, but otherwise simple: it just follows the intuition from the definition of the escape predicates. We present the full proof in Appendix B.1. □

6.4.2 Abstract Execution of $A(m)$

Initialization: The abstract semantics of $A(m)$ starts with the same initial points-to graph as the pointer analysis of method m:

$$G_{id_0} = G^m_{\text{init}} = \langle\ (L_{\text{all–empty}}\left[p_i \mapsto \{n^P_{i,0}\}\right]_{0 \le i \le k-1}) : [],\ \emptyset,\ \emptyset,\ \emptyset,\ \emptyset\ \rangle$$

[7] To simplify the proofs, we consider that all objects created outside $A(m)$ escape from the very beginning of the program (i.e., we mark them as escaped right from e_0).

$[\![.]\!] : Label \rightarrow PTGraph \rightarrow PTGraph$

$P(lb)$	$G \mapsto [\![lb]\!](G)$
$v_R = v_0.s(v_1, \ldots, v_j)$	1. Unanalyzable CALL: already presented in Figure 4-3 2. **Analyzable CALL** $\langle L\!:\!J,\ I,\ O,\ E,\ R,\ W\rangle \mapsto$ $\langle L_{callee}\!:\!L\!:\!J,\ I,\ O,\ E,\ R,\ W\rangle$ where $L_{callee} = L_{\text{all-empty}}\,[p_i \mapsto L(v_i)]_{0\le i\le j}$
`return` v	1. Final RETURN from the activation $A(m)$: $\langle L\!:\![],\ I,\ O,\ E,\ R,\ W\rangle \mapsto\ \langle L\!:\![],\ I,\ O,\ E,\ L(v),\ W\rangle$ 2. **RETURN inside the activation** $A(m)$ $\langle L_1\!:\!L_2\!:\!J,\ I,\ O,\ E,\ R,\ W\rangle \mapsto$ $\alpha_{\lvert J\rvert}(gc(\langle L_2\,[v_R \mapsto L_1(v)]\!:\!J,\ I,\ O,\ E,\ R,\ W\rangle))$

Figure 6-5: Missing cases from the definition of the abstract semantics transfer functions. We use bold font for the new cases. Figure 4-3 on page 47 presents the other cases.

The initial modeling relation is

$$\rho_{id_0} = \{\langle n^P_{i,0},\, V_{id_0}(p_i)\rangle\}_{0\le i\le k-1}$$

In ρ_{id_0}, each parameter node $n^P_{i,0}$ models the object pointed to by the ith formal parameter of the method m.

Transitions: Between id_{2i+1} and $id_{2(i+1)}$, the program executes only instructions from outside $A(m)$. The abstract semantics of $A(m)$ does not model the actions of the other parts of the program. Hence, the abstract semantics simply propagates the points-to graph and the modeling relation from id_{2i+1} to $id_{2(i+1)}$: $G_{id_{2(i+1)}} = G_{id_{2i+1}}$ and $\rho_{id_{2(i+1)}} = \rho_{id_{2i+1}}$.

A transition from id_{2i} to id_{2i+1} executes an instruction from $A(m)$. Let $lb_{id_{2i}}$ be the label of this instruction. In this case,

$$G_{id_{2i+1}} = [\![lb_{id_{2i}}]\!](G_{id_{2i}})$$

where $[\![lb_{id_{2i}}]\!]$ is the abstract semantics transfer function attached to the label $lb_{id_{2i}}$. The modeling relation $\rho_{id_{2i+1}}$ is defined by the case rules from Figure 6-6 (explained later).

Figure 4-3 on page 47 already presented the transfer functions that the abstract semantics share with the pointer analysis. Figure 6-5 presents the two remaining abstract semantics transfer functions:

Analyzable CALLs: Unlike the pointer analysis, the abstract semantics "steps" into the callee and abstractly executes its instructions. To model the local variables from the callee, the abstract semantics uses a new stack frame, L_{callee}. Hence, unlike the pointer analysis, the abstract semantics can create points-to graphs with more than just one stack frame.

RETURN inside $A(m)$: A RETURN "inside $A(m)$" is any RETURN instruction executed by $A(m)$, except for the final RETURN that ends $A(m)$. The abstract semantics distinguishes between the two cases using with the depth of the abstract stack.

A final RETURN corresponds to an abstract stack with a single element. In this case, the abstract semantics and the pointer analysis use the same transfer function from Figure 4-3. We repeat it in Figure 6-5 for easy reference.

If the abstract stack has more than one element, the abstract semantics applies the processing for a RETURN inside $A(m)$:

1. Remove the abstract state L_1 of the callee local variables.
2. Set the caller variable v_R to point to the nodes that the callee returns.
3. Use gc to remove the nodes that are unreachable from local variables, parameter/load/global nodes and globally escaped nodes (see Figure 4-10 on page 57 for the exact definition of gc).
4. Use the node morphism $\alpha_{|J|}$ to "truncate" all node contexts to $|J|$.[8]

Figure 6-6 presents the case rules for computing the new modeling relation $\rho_{id_{2i+1}}$. The definition of $\rho_{id_{2i+1}}$ reflects the node model that our analysis uses (see Section 4.1). For example, the transfer function for a NEW instruction uses the inside node $n^I_{lb,c}$ to model the newly allocated object o.[9] Accordingly, $\rho_{id_{2i+1}} = \rho_{id_{2i}} \cup \{\langle n^I_{lb,c}, o\rangle\}$.

Consider a LOAD instruction "$v_2 = v_1.f$" that reads the heap reference $\langle o_1, f, o_2\rangle$. The corresponding transfer function may use a load node $n^L_{lb,c}$. Accordingly, if the object o_1 escapes outside $A(m)$, the abstract semantics records that $n^L_{lb,c}$ models the loaded object o_2.

For an unanalyzable CALL instruction, the abstract semantics records that the global node $n_{\texttt{GBL},c}$ models the returned object. The case of a STATIC LOAD instruction is similar.

For a RETURN inside $A(m)$, the abstract semantics "truncates" the contexts of all nodes from the modeling relation to $c-1$. The transfer function for a RETURN inside $A(m)$ (Figure 6-5) performs the same transformation for all nodes from the points-to graph. For other instructions, the modeling relation remains unchanged.

[8] I.e., $\alpha_{|J|}$ changes each node context bigger than $|J|$ to $|J|$; other node contexts are unaffected.
[9] The top of Figure 6-6 explains the notations lb and c.

Notation: The concrete state at date $d \in \{id_{2i}, id_{2i}+1\}$ has the form

$$\Xi_d = \langle A_d\,[t \mapsto \langle\, V_d, lb_d\rangle : K_d]\,,\ H_d,\ S_d,\ TY_d\rangle$$

Also, let $G_{id_{2i}} = \langle L_{id_{2i}} : J_{id_{2i}},\ I_{id_{2i}},\ O_{id_{2i}},\ E_{id_{2i}},\ R_{id_{2i}}\rangle$. For brevity, we denote $lb = lb_{id_{2i}}$, and $c = |J_{id_{2i}}|$. Hence, V_d is the state of the local variables for the top method from thread t, at date $d \in \{id_{2i}, id_{2i}+1\}$; $P(lb)$ is the instruction executed in the transition $\Xi_{id_{2i}} \Rightarrow \Xi_{id_{2i}+1}$. In all cases except that of an unanalyzable CALL, $id_{2i+1} = id_{2i} + 1$.

$P(lb)$	$\rho_{id_{2i+1}}$
NEW $v =$ `new` C	$\rho_{id_{2i}} \cup \{\langle n^I_{lb,c}, o\rangle\}$ where o is the object created in the concrete execution at date id_{2i}: $o = V_{id_{2i+1}}(v)$.
LOAD $v_2 = v_1.f$	Let $\langle o_1, f, o_2\rangle$ be the heap reference read in the concrete execution: $o_1 = V_{id_{2i}}(v_1)$, $o_2 = H_{id_{2i}}(o_1)(f)$. Then, $\rho_{id_{2i+1}} = \begin{cases} \rho_{id_{2i}} \cup \{\langle n^L_{lb,c}, o_2\rangle\} & \text{if } e_d(o_1) \\ \rho_{id_{2i}}(\text{unmodified}) & \text{otherwise} \end{cases}$
unanalyzable CALL $v_R = v_0.s(v_1, \ldots, v_j)$	$\rho_{id_{2i}} \cup \{\langle n_{\texttt{GBL},c}, o\rangle\}$ where o is the object returned from the unanalyzable CALL in the concrete execution: $o = V_{id_{2i+1}}(v_R)$.
STATIC LOAD $v = C.f$	$\rho_{ip_{2i}} \cup \{\langle n_{\texttt{GBL},c}, o\rangle\}$ where o is the object read in the concrete execution: $o = S_{id_{2i}}(C, f)$.
RETURN inside $A(m)$ `return` v	$\alpha_{c-1}(\rho_{id_{2i}}) = \{\ \langle \alpha_{c-1}(n), o\rangle \ \mid\ \langle n, o\rangle \in \rho_{id_{2i}}\ \}$
otherwise	$\rho_{id_{2i}}$ (unmodified)

Figure 6-6: Case rules for the construction of the modeling relation $\rho_{id_{2i+1}}$ for the interesting date id_{2i+1} of activation $A(m)$.

6.5 Abstract Semantics Invariants

This section states several properties of the abstract semantics. We prove these properties in Appendix B.2 on page 160.

Consider an interesting date $d \in ID_{A(m)}$. Let $G_d = \langle L_d : J_d, I_d, O_d, E_d, R_d, W_d \rangle$ and ρ_d be the points-to graph, respectively the modeling relation that the abstract semantics constructs for date d. With these notations, the following invariants are true:

Invariant 1. *Consider an object o already allocated by $A(m)$ by executing the NEW instruction from label lb. Then, there exists an inside node $n^I_{lb,c}$ such that either $n^I_{lb,c}$ models o, or $n^I_{lb,c}$ is one of the globally escaped nodes:*

$$\forall o \in Object.\ \forall lb \in Label.$$
$$\langle o, lb \rangle \in Allocs_d^{A(m)} \rightarrow \exists c \in \mathbb{N}.\ (n^I_{lb,c}\ \rho_d\ o)\ \vee\ (n^I_{lb,c} \in E_d)$$

Note: we could prove a stronger invariant, stating that each object allocated by $A(m)$ at allocation site lb is modeled by an inside node $n^I_{lb,c}$. The more complex invariant above allows us to later introduce an analysis technique that trades precision for speed, by replacing several nodes (including inside nodes) with a single $n_{\texttt{GBL},c}$ node, thus reducing the size of the manipulated points-to graphs.

Invariant 2. *Any inside node $n^I_{lb,c}$ models only the objects allocated by $A(m)$ by executing the instruction from label lb:*

$$\forall lb \in Label.\ \forall c \in \mathbb{N}.\ \ n^I_{lb,c}\ \rho_d\ o\ \rightarrow\ \langle o, lb \rangle \in Allocs_d^{A(m)}$$

Invariant 3. *Any captured object is modeled by at most one node:*

$$\forall o \in Object \setminus \{o_{\texttt{null}}\}.\ \ \neg e_d(o) \rightarrow |\ \{n\ |\ n\ \rho_d\ o\}\ | \leq 1$$

Let's examine the stack of the thread t, the thread where $A(m)$ takes place. Let h_d be the height of this stack at date d. At date id_0 (the beginning of $A(m)$), the top stack frame corresponds to the method m. The other $h_{id_0} - 1$ stack frames were created before the start of $A(m)$. At date d, the top $h_d - h_{id_0} + 1$ stack frames correspond to methods from $A(m)$.[10]

The next invariant states that the abstract stack from G_d models the state of the local variables from the frames created by $A(m)$:

Invariant 4. *The abstract stack $L_d : J_d$ conservatively models all stack frames created by $A(m)$:*

1. *$|L_d : J_d| = \max(1, h_d - h_{id_0} + 1)$, where h_d is the height of the concrete stack at date d. I.e., the abstract stack from G_d contains one frame for each frame of*

[10] No date $d \in ID_{A(m)}$ is in the middle of the execution of a method invoked by an unanalyzable CALL; hence, the top frames are created by $A(m)$.

$A(m)$. *We use the function* max *for technical reasons: if* d *is the termination date for the RETURN that terminates* $A(m)$, $A(m)$ *has no stack frames* ($h_d = h_{id_0} - 1$), *but the corresponding points-to graph has one element,* $L_{\text{all–empty}}$ *(see Figure 6-5).*

2. *Let* K_d *be the concrete stack of thread* t *at date* d. *Consider an arbitrary* k *such that* $0 \leq k \leq h_d - h_{id_0}$, *let* $\langle V_{d,k}, lb_{d,k} \rangle = K_d[k]$ *be the* k*-th frame of* K_d, *and let* $L_{d,k} = (L_d : J_d)[k]$ *be the* k*-th element of* $L_d : J_d$ *(e.g.,* $L_{d,0} = L_d$*). Consider an arbitrary variable* $v \in Var$ *and let* $o = V_{d,k}(v)$. *If* $o \neq o_{\texttt{null}}$, *then* $\exists n \in L_{d,k}(v)$ *such that* $n \ \rho_d \ o$.

   Intuitively, if in the k*-th concrete stack frame local variable* v *points to the object* $o \neq o_{\texttt{null}}$, *then in the* k*-th local variable state from the abstract stack,* v *points to (at least) one node* n *that models* o.

Invariant 5. *The set* I_d *of inside edges conservatively models all reachable heap edges created by* $A(m)$ *before the date* d, *with respect to the modeling relation* ρ_d:

$$\forall o_1, o_2 \in \mathit{RObjs}(\Xi_d) \setminus \{o_{\texttt{null}}\}.\ \forall f \in \mathit{Field}.$$
$$\langle o_1, f, o_2 \rangle \in I_d^{A(m)} \ \rightarrow \ \exists n_1, n_2.\ (n_1 \ \rho_d \ o_1) \ \wedge \ (n_2 \ \rho_d \ o_2) \ \wedge \ (\langle n_1, f, n_2 \rangle \in I_d)$$

Invariant 6. *Only escaped nodes may model the escaped objects:*

$$\forall o \in \mathit{Object}.\ \forall n \in \mathit{CNode}.\ \ e_d(o) \ \wedge \ (n \ \rho_d \ o) \ \rightarrow \ e(G_d)(n)$$

Invariant 7. *Each node that appears in the points-to graph* G_d *and/or the modeling relation* ρ_d *has context at most* $|J_d|$:

$$\forall n \in \mathit{Node},\ \forall c \in \mathit{Context},\ \forall o \in \mathit{Object}.$$
$$\langle n, c \rangle \in \mathit{nodes}(G_d) \ \vee \ \langle n, c \rangle \ \rho_d \ o \ \rightarrow \ c \leq |J_d|$$

Invariant 8. *The set* W_d *of mutated abstract fields conservatively models all locations that are (1) mutated by* $A(m)$ *before the date* d, *and (2) allocated outside* $A(m)$:

$$\forall o \in \mathit{Object}.\ \forall f \in \mathit{Field}.$$
$$\langle o, f \rangle \in W_d^{A(m)} \ \wedge \ o \notin \mathit{Allocs}_{\text{all}}^{A(m)} \ \rightarrow \ \exists n \in \mathit{CNode}.\ (n \ \rho_d \ o) \ \wedge \ \langle n, f \rangle \in W_d$$

6.6 Proof of the Analysis Properties

Section 6.3 formalized the intended properties of the points-to graphs that the analysis constructs. This section provides the skeleton of the proof of these properties. To preserve the readability of this chapter, we present the most technical parts of the proof in Appendix B.3 and Appendix B.4.

Our proof has two parts: First, Theorem 7 proves that if the pointer analysis

of a method m *conservatively approximates*[11] the abstract semantics of any possible activation of method m, then the analysis properties from Section 6.3 are valid. This proof uses the abstract semantics invariants from Section 6.5 (proved correct in Appendix B.2). Next, Theorem 8 proves that the pointer analysis conservatively approximates the abstract semantics.

Consider $\langle \circ A, A \circ \rangle$ a possible set of analysis results (i.e., $\langle \circ A, A \circ \rangle$ satisfy Constraints 4.1 on page 45). Consider $d \in IP_{A(m)}$ an arbitrary intra-procedural date for the activation $A(m)$. Let lb_d be the label defined as in Section 6.3:

- If the height of the stack for the thread t is smaller at date d than at date ip_0 (i.e., d is the termination date of the RETURN instruction that terminates $A(m)$), then $lb_d = exit_m$, the special label for the end of m.
- Otherwise, let lb_d be the current label inside the thread t. Due to the definition of the intra-procedural dates, lb_d is a label inside the method m.

Let $G = \circ A(lb_d)$ be the analysis points-to graph that the pointer analysis constructs for the program point before lb_d. Let G_d and ρ_d be the points-to graph, respectively the modeling relation that the abstract semantics constructs for date d.

Lemma 6. *$\forall d \in IP_{A(m)}$, G_d is an analysis points-to graph: $G_d \in PTGraph^a$. Additionally, ρ_d refers only to nodes with context 0: $\forall n, c, o.\ \langle n, c\rangle\ \rho_d\ o \rightarrow c = 0$.*

Proof. Let $L_d : J_d$ be the abstract stack from the points-to graph G_d. At each intra-procedural date $d \in IP_{A(m)}$, $h_d \leq h_{id_0}$ (see Definition 7). Therefore, by Invariant 4, $|L_d : J_d| = 1$, which implies $|J_d| = 0$, i.e. the abstract stack from G_d has a single element. By Invariant 7, all nodes that appear in G_d have context 0; hence, G_d is an analysis points-to graph. Also, by Invariant 7, all nodes that appear in ρ_d have context 0. □

The above lemma allows comparisons between G and G_d, using the partial order relation for analysis points-to graphs from Definition 4.[12] The next lemma proves that if the pointer analysis conservatively approximates the abstract semantics, i.e., $G \sqsupseteq G_d$, then Properties 5-9 are valid:

Theorem 7. *If $G \sqsupseteq G_d$, then G satisfies the Properties 5-9 from Section 6.3, for the modeling relation $\rho = \rho_d$.*

Proof. Let $G = \langle L : [], I, O, E, R, W\rangle$ and $G_d = \langle L_d : [], I_d, O_d, E_d, R_d, W_d\rangle$.

- Property 5 is a special case of Invariant 4 when $A(m)$ has only one stack frame.

[11] This term becomes obvious later in this section. Intuitively, the pointer analysis computes bigger points-to graphs.

[12] General points-to graphs cannot be compared, as they may have abstract stacks of different heights.

- Property 6 easily follows from Invariant 5 and from the observation that $I_d \subseteq I$ (due to $G \sqsupseteq G_d$).

- Property 7 is identical to Invariant 2.

- Property 8: consider an arbitrary object o allocated by $A(m)$ by executing the NEW instruction from label lb (i.e., $\langle o, lb\rangle \in Allocs_{\text{all}}^{A(m)}$), and assume that the inside node $n^I_{lb,0}$ is captured in G (i.e., $\neg e(G)(n^I_{lb,0})$). Property 8 requires that (1) o is captured in $A(m)$ (i.e., $\neg ROBjs(outside_{A(m)}(\Xi_d))$), and (2) $n^I_{lb,0}$ is the only node that models o in $\rho = \rho_d$.

  By Invariant 1, $\exists c$ such that $n^I_{lb,c}\ \rho\ o$ or $n^I_{lb,c} \in E_d$. As G_d and ρ_d contain only nodes with 0 context, $c = 0$. Furthermore, as $n^I_{lb,0}$ is captured in G, $n^I_{lb,0} \notin E \supseteq E_d$. Hence, $n^I_{lb,0}$ models o. By Invariant 3, $n^I_{lb,0}$ is the only node that models o in ρ.

  Assume for the sake of contradiction that o escapes from $A(m)$, i.e., $ROBjs(outside_{A(m)}(\Xi_d))$. By Lemma 5, $e_d(o)$. By Invariant 6, $n^I_{lb,0}$ escapes in G_d. As $G \sqsupseteq G_d$, $n^I_{lb,0}$ escapes in G too. Contradiction! Hence, o is captured inside $A(m)$.

- Property 9 is identical to Invariant 8.

□

Theorem 8. *The pointer analysis conservatively approximates the abstract semantics: with the notations from this section,* $\forall d \in IP_{A(m)}$, $\circ A(lb_d) \sqsupseteq G_d$.

Proof. We perform a proof by induction on the *call depth* of the activation $A(m)$. The call depth of an activation is the maximum number of nested analyzable CALLs in that activation: e.g., an activation that does not execute any analyzable CALL has call depth 0, an activation that execute an analyzable CALL, and, inside the callee, another analyzable CALL, has call depth at least 2.

Base Case: Assume that the call depth of $A(m)$ is 0, i.e., $A(m)$ does not execute any analyzable CALL. We prove by induction on the list of intra-procedural dates $IP_{A(m)} = [ip_0, \dots ip_{2i}, ip_{2i+1}, \dots]$ that $\forall j$, $\circ A(lb_{ip_j}) \sqsupseteq G_{ip_j}$.

- **Initial case $j = 0$:** The label lb_{ip_0} is the first label from method m. As $entry_m$ is a control-flow predecessor of the first instruction from m, by Constraints 4.1, $\circ A(lb_{ip_0}) \sqsupseteq A\circ(entry_m) \sqsupseteq G^m_{\text{init}}$. As $G_{ip_0} = G^m_{\text{init}}$ (see Section 6.4.2), we obtain $\circ A(lb_{ip_0}) \sqsupseteq G_{ip_0}$.

- **Induction step $j \to j{+}1$:** We assume $\circ A(lb_{ip_j}) \sqsupseteq G_{ip_j}$, and prove $\circ A(lb_{ip_{j+1}}) \sqsupseteq G_{ip_{j+1}}$. The case of an odd j, i.e., $j = 2i + 1$, is trivial: the program executes only instructions from other threads between ip_{2i+1} and $ip_{2(i+1)}$. Hence, $lb_{ip_{2(i+1)}} = lb_{ip_{2i+1}}$, and $G_{ip_{2(i+1)}} = G_{ip_{2i+1}}$ (see Section 6.4.2).

Consider an even $j = 2i$. Between ip_{2i} and ip_{2i+1}, $A(m)$ executes an instruction that is not an analyzable CALL (the call depth of $A(m)$ is 0). As $lb_{ip_{2i}}$ is a control-flow predecessor of $lb_{ip_{2i+1}}$, by Constraints 4.1,

$$\circ A(lb_{ip_{2i+1}}) \ \sqsupseteq \ A\circ(lb_{ip_{2i}}) \ \sqsupseteq \ [\![lb_{ip_{2i}}]\!]^a(\circ A(lb_{ip_{2i}}))$$

By the induction hypothesis, $\circ A(lb_{ip_{2i}}) \sqsupseteq G_{ip_{2i}}$. Lemma 14 in Appendix B.3 proves that $[\![lb_{ip_{2i}}]\!]^a$ is monotonic. Hence,

$$[\![lb_{ip_{2i}}]\!]^a(\circ A(lb_{ip_{2i}})) \ \sqsupseteq \ [\![lb_{ip_{2i}}]\!]^a(G_{ip_{2i}})$$

As the instruction from $lb_{ip_{2i}}$ is not an analyzable CALL,

$$[\![lb_{ip_{2i}}]\!]^a(G_{ip_{2i}}) \ = \ [\![lb_{ip_{2i}}]\!](G_{ip_{2i}})$$

As $[\![lb_{ip_{2i}}]\!](G_{ip_{2i}}) = G_{ip_{2i+1}}$, we obtain the desired relation $\circ A(lb_{ip_{2i+1}}) \sqsupseteq G_{ip_{2i+1}}$.

Induction Step: Assume that the lemma is valid for any activation with a call depth strictly smaller than the call depth of $A(m)$. As in the base case, we perform a proof by induction on the list of intra-procedural dates. The only difference is in the induction step, for the case when $j = 2i$ and the instruction from label $lb_{ip_{2i}}$ is an analyzable CALL. We focus on this case. Recall that the induction step assumes $\circ A(lb_{ip_{2i}}) \ \sqsupseteq \ G_{ip_{2i}}$ and proves $\circ A(lb_{ip_{2i+1}}) \ \sqsupseteq \ G_{ip_{2i+1}}$. First, as before, we prove that

$$\circ A(lb_{ip_{2i+1}}) \ \sqsupseteq \ [\![lb_{ip_{2i}}]\!]^a(\circ A(lb_{ip_{2i}})) \tag{6.11}$$

Let *callee* be the method invoked by the analyzable CALL at date ip_{2i}. By the definition of the analysis transfer function for an analyzable CALL (Equation 4.2),

$$[\![lb_{ip_{2i}}]\!]^a(\circ A(lb_{ip_{2i}})) \ \sqsupseteq \ \mathit{interproc}(\circ A(lb_{ip_{2i}}),\ \circ A(\mathit{exit}_{\mathit{callee}}),\ P(lb_{ip_{2i}})) \tag{6.12}$$

The CALL to *callee* starts an activation $A(\mathit{callee})$ for the method *callee*. This activation has its own intra-procedural dates. The last intra-procedural date for $A(\mathit{callee})$ is the date when the matching RETURN terminates, i.e., ip_{2i+1}. Let $G^{\mathit{callee}}_{ip_{2i+1}}$ be the points-to graph that the abstract semantics of $A(\mathit{callee})$ constructs for ip_{2i+1}. The call depth of $A(\mathit{callee})$ is strictly smaller than the call depth of $A(m)$. Hence, by the induction hypothesis, $\circ A(\mathit{exit}_{\mathit{callee}}) \ \sqsupseteq \ G^{\mathit{callee}}_{ip_{2i+1}}$. Lemma 16 in Appendix B.3 proves that *interproc* is monotonic in its first two arguments. Using equations 6.11 and 6.12 above, we obtain

$$\circ A(lb_{ip_{2i+1}}) \ \sqsupseteq \ \mathit{interproc}(G_{ip_{2i}},\ G^{\mathit{callee}}_{ip_{2i+1}},\ P(lb_{ip_{2i}}))$$

To complete the proof of Theorem 8, it is sufficient to prove that

$$\mathit{interproc}(G_{ip_{2i}},\ G^{\mathit{callee}}_{ip_{2i+1}},\ P(lb_{ip_{2i}})) \ \sqsupseteq \ G_{ip_{2i+1}} \tag{6.13}$$

Intuitively, Equation 6.13 states that the inter-procedural analysis is more conserva-

tive than the abstract semantics. The proof of Equation 6.13 is extremely technical. We present the proof in Appendix B.4. □

The combination of Theorem 7 and Theorem 8 proves the analysis properties.

Chapter 7

Analysis Optimizations

This chapter presents several optimizations that increase the analysis speed and/or precision. Section 7.1 presents an optimization that sacrifices precision in order to reduce the time spent while analyzing groups of mutually recursive methods. Section 7.2 describes the use of type information to avoid unfeasible edges and node mappings. Section 7.3 describes several optimizations that reduce the size of the method summaries that the inter-procedural analysis uses.

7.1 Knowing When to Stop Insisting

A general program analysis may converge too slowly to a result for a specific program part. In such cases, downgrading to a less precise (but faster) analysis for that program part allows the analysis to process the rest of the program in reasonable time.

Our analysis may spend significant time computing inter-procedural fixed-points for groups of mutually recursive methods. The analysis of a caller requires the points-to graphs for the end of the callees invoked through analyzable CALLs. Hence, the analysis requires a fixed-point computation for the strongly-connected components of the static call graph (see Step 3 of the algorithm from Section 4.5.2). Unfortunately, very large strongly-connected components are common in the static call graphs of real Java programs.[1]

To cope with such situations, an analysis implementation can impose a bound on the effort invested in the fixed-point computation for each strongly-connected component *scc* of the call graph. If the fixed-point for *scc* exceeds this bound, the implementation can perform the following steps:

1. Abort the fixed-point computation for *scc*.
2. Consider each *intra-scc* CALL unanalyzable. An intra-*scc* CALL is a CALL in the body of a method from *scc*, that may invoke a method from *scc*. This

[1]Sometimes, this situation is due to the imprecision of the call graph construction algorithm. Other times, there is real recursion in the analyzed program (e.g., consider a recursive-descent parser).

step eliminates the dependencies between the analyses of different methods from *scc*.[2]

3. Analyze each method from *scc* individually.

This optimization is sound because our analysis already handles unanalyzable CALLs correctly. This optimization may lose precision, but (1) it allows the analysis to obtain some results even for the methods from very large strongly-connected components, and (2) it allows the analysis to examine the rest of the program.

There are different possible bounds on the inter-procedural fixed-point effort (Step 3 of the algorithm from Section 4.5.2): bounds on the number of iterations until reaching a fixed-point, time bounds, bounds of the size of the involved method summaries, etc. Our implementation (Chapter 8) uses a bound on the number of iterations.

An implementation can apply this optimization idea even for non-recursive methods: if a CALL instruction invokes a callee whose summary is too big, then the implementation can consider that CALL unanalyzable.

7.2 Using Type Information

As Java is a type-safe language, the analysis can use type information to improve its speed and precision, e.g., by trimming inside edges that cannot model any heap reference. This optimization is possible because each node contains information about the possible types (Java classes) of the objects it models:

1. The inside node $n^I_{lb,c}$ for the NEW instruction "v = `new` C" from label lb represents objects of class C.

2. The parameter node $n^P_{i,c}$ represents objects whose class is a subclass of the declared type of the corresponding formal parameter of the currently analyzed method.

3. The load node $n^L_{lb,c}$ attached to the LOAD instruction "$v_2 = v_1.f$" from label lb represents objects whose class is a subclass of the declared type of the field f.

4. The global node $n_{\texttt{GBL},c}$ represents objects of any class.

The analysis can use the type information as follows:

- The analysis ignores *type-incorrect* edges. An edge $\langle n_1, f, n_2 \rangle$ is type-incorrect if (1) the field f does not appear in any of the possible classes for the node n_1 or (2) the declared type of the field f is not a supertype of any of the possible types for the node n_2.

[2]It is possible to eliminate only enough intra-*scc* dependencies to "break" the strongly-connected component. For simplicity, as it is unclear which dependencies are more important for the analysis speed and precision, we eliminate all intra-*scc* dependencies.

- The analysis ignores *type-incompatible* node mappings. A mapping $\langle n_1, n_2 \rangle$ is type-incompatible if there is no common possible type for the nodes n_1 and n_2.

  During the construction of the initial node map μ_0 in function *interproc* (Figure 4-6), this optimization may lead to situations where the parameter node that models the receiver of the callee (i.e., the `this` object) is not mapped to any node from the caller. This fact indicates that there is no receiver object in the concrete execution, i.e., the CALL instruction is unreachable in the program execution. Hence, the analysis can ignore it.

The use of type information increases the precision of the analysis. It involves some time overhead (e.g., to check the type-correctness of edges). In our experiments, the speedup due to the reduction in the size of the points-to graphs and node maps was generally bigger than the overhead of using type information, leading to an overall speedup (see Section 8.3). Currently, we do not have a formal proof that this optimization preserves the analysis correctness.

7.3 Simplifying the Method Summaries

This section presents optimizations that simplify the method summaries. These optimizations reduce the execution time for the inter-procedural analysis and the size of the points-to graphs for the program points after the CALLs.

For each method *callee*, the analysis computes a method summary starting from the points-to graph G_{callee} for the end of *callee*. More precisely, as presented in Figure 4-6 on page 53, the function *interproc* uses the summary

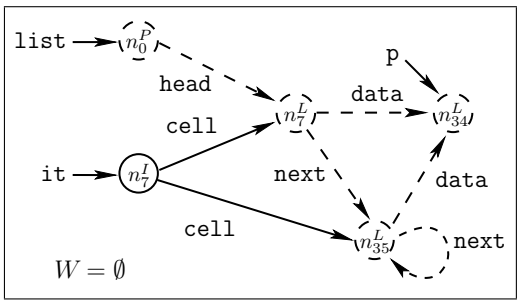
$$\langle T_{callee}, R_{callee} \rangle = summary(\tau(gc(\rho(G_{callee})))) \tag{7.1}$$

Points-to graph simplifications result in method summary simplifications. We formulate each optimization from this section as a points-to graph transformation $\xi : PTGraph \to PTGraph$.

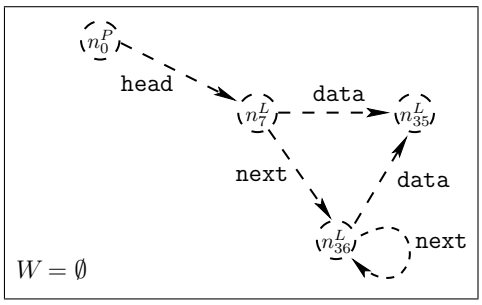
The analysis already uses one such optimization: the function $gc \circ \rho$ "garbage collects" the captured nodes from G_{callee}. Intuitively, these nodes represent only captured objects that are unreachable from the caller. The correctness proof from Chapter 6 takes this optimization into account.

Example 10. Figure 7-1.a presents the points-to graph G for the end of the method `Main.sumX` from Chapter 2. Figure 7-1.b presents the points-to graph $G' = gc(\rho(G))$. G' no longer contains the captured node n_7^I nor the edges with n_7^I as an endpoint. Also, G' does not contain the information about the local variables (these variables are irrelevant for the inter-procedural analysis). △

The rest of this section presents two additional optimizations. The first optimization (function *url* from Section 7.3.1) unifies redundant load nodes. The second optimization (function *uge* from Section 7.3.2) unifies globally escaped nodes. After



a. End-of-method points-to graph G.



b. Simplified points-to graph $gc(\rho(G))$.

Figure 7-1: Example of the captured node removal optimization on the points-to graph for the end of the method `Main.sumX` from the example from Chapter 2.

adding these optimizations, Equation 7.1 becomes:

$$\langle T_{callee}, R_{callee} \rangle = summary(\tau(uge(url(gc(\rho(G_{callee}))))))$$

Our correctness proof ignores these two optimizations. If desired, the analysis clients can disable these two optimizations. Section 8.3 presents the analysis slowdown if any of these two optimizations is disabled.

7.3.1 Unifying Redundant Load Nodes - Function *url*

Our analysis introduces one distinct load node for each LOAD instruction that reads a field of an escaped node (see Figure 4-5). This strategy may lead to a "proliferation" of load nodes and outside edges for methods that repeatedly read the same field of the same node. For such methods, the analysis generates several outside edges that start in the same node, have the same field label, but end in distinct load nodes. These multiple outside edges increase the inter-procedural analysis time. This section describes an optimization that unifies these outside edges by merging the load nodes they end in.

Example 11. Consider the method `ListItr.next()` from our example in Chapter 2 (see Figure 2-1 on page 22). For brevity, Chapter 2 uses an implementation of `ListItr.next()` that minimizes the number of generated outside edges.

Figure 7-2.a presents a more natural implementation of the class `ListItr`. Notice that the first two lines of the method `next` read the field `cell` of the `this` object. The Java compiler cannot store the result of the first read in a local variable and reuse it in the second line. This "optimization" may change the semantics of the program in the presence of multiple threads of execution. Figure 7-2.b presents the desugaring of the method `next`'s body in our program representation.

Figure 7-3.a presents the points-to graph for the end of `next()`. Notice the two outside edges that start in n_0^P: $\langle n_0^P, \texttt{cell}, n_3^L \rangle$ and $\langle n_0^P, \texttt{cell}, n_7^L \rangle$; they correspond to the LOAD instructions from lines 3 and 7. These outside edges generate two

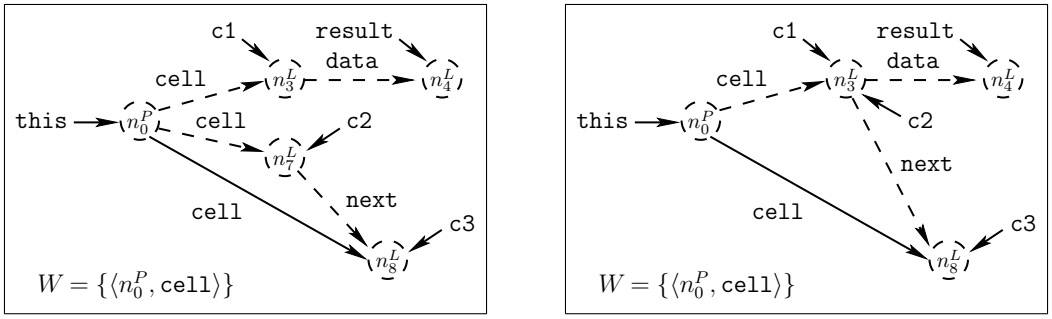
```
class ListItr implements Iterator {
    ListItr(Cell head) {
        this.cell = head;
    }
    Cell cell;
    public boolean hasNext() {
        return cell != null;
    }
    public Object next() {
        Object result = cell.data;
        cell = cell.next;
        return result;
    }
}
```

a. Java implementation.

```
 1  public Object next() {
 2      /* Object result = cell.data; */
 3      Cell c1 = this.cell;
 4      Object result = c1.data;
 5
 6      /* cell = cell.next; */
 7      Cell c2 = this.cell;
 8      Cell c3 = c2.next;
 9      this.cell = c3;
10
11      return result;
12  }
```

b. Bytecode-like desugaring of the Java source code of `ListItr.next()` in our program representation.

Figure 7-2: "Natural" implementation of the class `ListItr` from Figure 2-1 on page 22.



a. End-of-method points-to graph.

b. After merging n_7^L into n_3^L.

Figure 7-3: Load node merging simplification for the points-to graph for the end of the method `ListItr.next()`. We use the instruction labels from Figure 7-2.b.

atomic inter-procedural transformers: $\texttt{load}(n_0^P, \texttt{cell}, n_3^L)$ and $\texttt{load}(n_0^P, \texttt{cell}, n_7^L)$. It is easy to see that these two transformers produce the same node mappings for n_3^L and n_7^L. Additionally, our analysis does not keep track of the ordering between the outside edges: when using the summary for the method `ListItr.next()`, the inter-procedural analysis is not aware of the fact that the LOAD from line 3 is always executed before the LOAD from line 7. Therefore, the distinction between n_3^L and n_7^L is "blurred" during the inter-procedural analysis. Hence, intuitively, the analysis does not lose precision by merging n_7^L into n_3^L. Figure 7-3.b presents the simplified points-to graph. "Merging" refers to the operation of substituting n_7^L with n_3^L in every element of the points-to graph: the outside edge $\langle n_7^L, \texttt{next}, n_8^L\rangle$ becomes $\langle n_3^L, \texttt{next}, n_8^L\rangle$, the local variable `c2` points to n_3^L instead of n_7^L, etc. △

In general, this optimization unifies the load nodes that are the targets of outside edges with a common source node and a common field label. Consider a node n and

a field f and let $\langle n, f, n^L_{lb_1}\rangle$, ... $\langle n, f, n^L_{lb_l}\rangle$ be the f-labeled outside edges that start in n. This optimization unifies the load nodes $n^L_{lb_1}$, ... $n^L_{lb_k}$ by replacing each $n^L_{lb_i}$ with some representative $n^L_{lb_k}$. The current analysis implementation chooses lb_k to be the minimal label among lb_1, ... lb_l. This optimization is performed repeatedly: each unification of load nodes may enable other unifications. The function *url* returns the final result of this iterative simplification.

7.3.2 Unifying Globally Escaped Nodes - Function *uge*

There are two kinds of escaped nodes in a points-to graph. Some nodes escape only because they are reachable from the caller (i.e., from parameters and/or returned nodes). These nodes may become captured in the caller. Other nodes escape globally: they are reachable from unanalyzed methods, static fields and/or started threads. The analysis cannot "recapture" these nodes. Intuitively, the only important information about these nodes is the fact that they escape, and the analysis is unlikely to gain precision by distinguishing between them.

This section presents an optimization that reduces the size of the end-of-method points-to graphs by unifying each globally escaped node with $n_{\texttt{GBL},0}$ (a node that already models globally escaped objects). This optimization is important because the transfer functions for several instructions (STATIC LOADs, unanalyzable CALLs, etc.) cause nodes to escape globally. The optimization described in Section 7.1 causes more nodes to escape globally. The optimization from this section prevents the "cluttering" of the points-to graphs with useless information about globally escaped nodes.

Technically, consider a points-to graph $G = \langle J, I, O, E, R, W\rangle$. E is the set of *directly* globally escaped nodes: e.g., nodes that are passed as arguments to unanalyzable CALLs, stored in static fields, etc. We define the set $gen(G)$ of globally escaped nodes as

$$gen(G) = \{\ n \in nodes(G) \setminus CPNode \mid reachable(E \cup \{n_{\texttt{GBL},0}\}, I \cup O)(n)\ \}$$

I.e., the set $gen(G)$ contains any node that is transitively and reflexively reachable from directly escaped nodes and/or from $n_{\texttt{GBL},0}$, along inside and outside edges. For technical reasons, $gen(G)$ does not contain any parameter nodes (unifying these nodes with $n_{\texttt{GBL},0}$ would require additional changes in the inter-procedural analysis).

Let ρ_G be the function defined as $\rho_G(n) = n_{\texttt{GBL},0}$ if $n \in gen(G)$, and $\rho_G(n) = n$ otherwise. We extend ρ_G to process any node-based structure, by replacing each node n with $\rho_G(n)$; e.g., if I is a set of inside edges, then

$$\rho_G(I) = \{\ \langle \rho_G(n_1),\ f,\ \rho_G(n_2)\rangle \mid \langle n_1, f, n_2\rangle \in I\ \}$$

The optimization *uge* is defined as follows:

$$uge(G) = \langle\ \rho_G(J),\ \rho_G(I),\ \rho_G(O),\ \underline{\rho_G(E) \cup (gen(G) \cap CINode)},\ \rho_G(R),\ \rho_G(W)\ \rangle$$

Notice the underlined part of the definition above: the set of directly globally escaped nodes from $uge(G)$ contains all inside nodes from $gen(G)$ (in addition to the nodes from $\rho_G(E)$). Intuitively, the analysis "remembers" that the inside nodes from $gen(G)$ escape globally. This feature ensures that the analysis continues to satisfy Property 4 on page 44. Part 1 of Property 4 states that if an object allocated by the analysis scope at allocation site lb escapes, then the corresponding inside node $n^I_{lb,0}$ escapes. Consider a method m that invokes a method *callee*. Assume that *callee* allocates an object at allocation site lb and stores it into a static field. The corresponding inside node $n^I_{lb,0}$ escapes globally. If uge "forgets" that $n^I_{lb,0}$ escapes globally, then the node $n^I_{lb,0}$ could incorrectly appear as captured in the points-to graph for the end of m. In this case, the stack allocation optimization could incorrectly decide to inline *callee* and stack-allocate the object allocated at the allocation site lb.

As a minor technical point, after this optimization, some outside edges may end in the node $n_{\texttt{GBL},0}$, which contradicts our earlier definition that each outside edge ends in a load node (Figure 4-2). We solve this problem by assuming that $n_{\texttt{GBL},0}$ is a special load node.

Chapter 8

Experimental Validation

To evaluate our ideas, we implemented our analysis and the two analysis applications that we described in this thesis: the stack allocation optimization and the purity analysis. We used our implementation to analyze several benchmarks of significant size and complexity, including the entire SPECjvm98 industry-standard benchmark suite [76].

The rest of this chapter has the following structure: Section 8.1 describes our implementation. Section 8.2 describes our stack allocation experiments. Section 8.3 uses the stack allocation experiments to evaluate the impact of the analysis optimizations from Chapter 7. Section 8.4 describes our purity analysis experiments.

8.1 Analysis Implementation

Our analysis implementation uses the MIT `Flex` compiler infrastructure [2] and the `jpaul` library of program analysis utilities [31]. Our implementation, the `Flex` compiler infrastructure, and the `jpaul` library are all implemented in the Java 5 programming language (i.e., Java with generic types).

System Architecture: Our analysis implementation uses the `Flex` front-end to parse Java class files and to construct a Static Single Assignment (SSA) [26] program representation similar to the one we used in the presentation of the analysis (see Section 3.2). Also, `Flex` constructs a static call graph for the analyzed program; we present the call graph construction algorithm in Section 8.1.2. Our analysis proceeds as explained in Section 4.5: the analysis examines the SSA intermediate representation and the call graph, generates the relevant dataflow constraints, and next uses the generic constraint solver from the `jpaul` library.

Theoretically, we could have used a more established constraint solver like BANE [34], at the price of performing a massive rewriting of the analysis constraints in the BANE format. We preferred `jpaul` because of its flexibility: `jpaul` allows the definition of new kinds of constraints by simply subclassing a specific Java class. Additionally, solving a constraint system with `jpaul` requires a simple Java method invocation, with no need for expensive file-based communication with an external

solver.

To increase the speed of our prototype, we implemented all the optimizations from Chapter 7. We also implemented the two analysis applications described in this thesis: the stack allocation optimization and the purity analysis. Command line options allow the user to select the desired analysis client. In the case of the stack allocation optimization, our implementation uses the `Flex` back-end to generate native executable code.[1] The resulting executables use a runtime library that contains the BDW garbage collector [9] and the implementations of the native methods.

Our analysis implementation relies extensively on the `jpaul` library of program analysis utilities, an open-source Java library that contains our generic implementations of several important algorithms and data structures for program analysis. Our decision to develop `jpaul` as a stand-alone library was motivated by our observation that program analysis researchers often re-implement (sometimes in an inefficient way) a relatively small set of algorithms: graph algorithms, constraint solvers, algorithms for finite state automata, etc. Having these algorithms in a separate library used by different analyses allows (1) better testing, (2) better separation between the analysis specification and the constraint solver, and (3) efficient implementations of important algorithms. The `jpaul` library is available online from the SourceForge repository of open-source software [31]. Although `jpaul` is a very *niche* library, there were more than 150 downloads of `jpaul` in the first five months of 2006.

Implementation Size: The implementation of the pointer and purity analysis consists of 10K lines of Java code,[2] and the implementation of the stack allocation optimization (including the code for performing method inlining) consists of another 1.5K lines. The `jpaul` library itself consists of another 13K lines of Java code. We also performed countless bug fixes and additions to various parts of `Flex`. Our implementation is publicly available online [79].

Section 8.1.1 explains our efficient analysis implementation. Section 8.1.2 explains our handling of the complex features of the Java bytecode language: exceptions, native methods, dynamic class loading, etc. Section 8.1.3 discusses the limitations of the current prototype.

8.1.1 Efficient Analysis Implementation

Our implementation uses the algorithm from Section 4.5.2 as a top-level driver for the inter-procedural analysis. For each method, our implementation uses the efficient constraint solver from the `jpaul` library to solve the dataflow constraints. We describe

[1] For security reasons, the Java bytecode instruction set does not allow stack allocation. Therefore, the only way to test the impact of the stack allocation optimization is to generate native code.

[2] The vast majority of this code belongs to the pointer analysis. The purity analysis is implemented inside the pointer analysis; it is hard to count the lines of code that are specific only to the purity analysis.

first the constraint solver from the `jpaul` library, focusing on the optimizations that have a direct impact on the analysis execution speed. Next, we explain how the analysis generates constraints in a form that takes advantage of the optimizations from the constraint solver. Finally, we briefly discuss the data structures used by our implementation.

Description of the Constraint Solver from the `jpaul` Library

The constraint solver from the `jpaul` library is an improvement of the constraint solver from Section 4.5.1. The `jpaul` solver supports constraints of the general form

$$\langle \mathrm{v}_1^o, \mathrm{v}_2^o, \ldots, \mathrm{v}_r^o \rangle \sqsupseteq f(\mathrm{v}_1^i, \mathrm{v}_2^i, \ldots, \mathrm{v}_p^i)$$

where $\mathrm{v}_1^i, \mathrm{v}_2^i, \ldots, \mathrm{v}_p^i$ are the *in*-variables of the constraint, $\mathrm{v}_1^o, \mathrm{v}_2^o, \ldots, \mathrm{v}_r^o$ are the *out*-variables, and f is a function that can be expressed in Java.[3] The `jpaul` solver supports constraints with multiple out-variables, for those situations when it is faster and/or more natural to compute several results simultaneously.[4] Different flow variables may take values from different lattices, provided that all these lattices have finite depth.[5] Any constraint system has at least one solution: e.g., consider the solution where the value of each flow variable is the top value from the corresponding lattice.

The solver uses an advanced worklist algorithm to compute a solution ψ that assigns to each variable v a value $\psi(\mathrm{v})$ such that all constraints are satisfied. If all constraints are monotonic (i.e., the right-hand side functions are monotonic), then the solver computes the smallest solution. If at least one constraint is non-monotonic, then a smallest solution may not even exist; in this case, the solver computes one of the possible solutions.

The `jpaul` solver performs two classes of important optimizations. First, the solver simplifies the constraint system by unifying certain flow variables. Second, the solver uses a sophisticated iteration technique over the potentially unsatisfied constraints. We describe these optimizations below.

---

[3] In `jpaul`, each constraint is an instance object of a subclass of the special class `jpaul.Constraints.Constraint`. `jpaul` provides several basic kinds of constraints; e.g., bigger-than constraints of the form $\mathrm{v}_1 \sqsupseteq \mathrm{v}_2$. The user may add new kinds of constraints by subclassing `Constraint`. Each constraint declares its in-variables and out-variables. `jpaul` ensures that each constraint may read only its declared in-variables, and similarly, that a constraint may update only its declared out-variables. The only allowed update is by joining a value to the current value for a flow variable. This restriction ensures that the values of the flow variables never decrease and the constraint solver (see Section 4.5.1) terminates for finite lattices.

[4] One can always express a constraint with several out-variables as a set of constraints, each with a single out-variable. However, this may require duplication of the right-hand side function.

[5] In `jpaul`, each flow variable is an instance object of a subclass of the special class `jpaul.Constraints.Var`. The operators of the related lattice (e.g., $\sqcup$) are implemented as virtual methods of the `Var` class. Hence, the `jpaul` user can declare new variables that take values in new lattices.

Unification of Flow Variables: The unification of flow variables reduces the memory consumption of the solver and the number of iterations until reaching a fixed-point: intuitively, there are fewer values that "grow." In addition, this technique reduces the number of constraints: after unification, certain constraints may become identical; e.g., the constraints $v_1 \sqsupseteq v_3$ and $v_2 \sqsupseteq v_3$ become identical after the unification of v_1 and v_2. Similarly, after unification, certain constraints may become tautologically true and the solver can ignore them; e.g., consider the constraint $v_1 \sqsupseteq v_2$ after the unification of v_1 and v_2.

The unification of any two flow variables preserves the correctness of the resulting solution: it is easy to prove that any solution for the constraint system after the unification satisfies the original constraints too.[6] However, unifying arbitrary flow variables may reduce the precision of the resulting solution. A solution ψ' for the constraint system is more precise than another solution ψ iff for any flow variable v, $\psi'(v) \sqsubseteq \psi(v)$.[7] To avoid losing precision, the solver unifies variables only in special cases:

1. The solver unifies mutually-bigger flow variables. E.g., in the case of the constraints $v_1 \sqsupseteq v_2$, $v_2 \sqsupseteq v_3$, $v_3 \sqsupseteq v_1$, the solver unifies the flow variables v_1, v_2, and v_3. Technically, the solver identifies the constraints of the form $v_1 \sqsupseteq v_2$ and unifies the strongly-connected components of the corresponding "bigger" relation. Mutually-bigger variables have equal values in any solution. Therefore, this optimization does not affect the precision of the solution. In the research literature, this optimization is usually called "cycle elimination" [35].

2. The solver detects flow variables v_1 such that the only constraint with v_1 as an out-variable has the form $v_1 \sqsupseteq v_2$. For each such v_1, the solver unifies v_1 and v_2.

   If all constraints with v_1 as an in-variable are monotonic in v_1, then the unification of v_1 and v_2 does not lose any precision. For any solution ψ, there exists a more precise (i.e., smaller) solution $\psi' \sqsubseteq \psi$ such that $\psi'(v_1) = \psi'(v_2)$: just pick ψ' to be identical to ψ, except that $\psi'(v_1) = \psi(v_2) \sqsubseteq \psi(v_1)$. Obviously, ψ' satisfies all constraints that do not use v_1. Also, ψ' satisfies (with equality) the unique constraint $v_1 \sqsupseteq v_2$ that uses v_1 as an out-variable. Consider a constraint that uses v_1 as an in-variable: $\langle v_1^o, \ldots, v_r^o \rangle \sqsupseteq f(\ldots, v_1, \ldots)$. Because $\psi'(v_1) \sqsubseteq \psi(v_1)$, and f is monotonic in v_1, the right-hand side evaluates to a smaller value in ψ' than in ψ, and the constraint remains satisfied in ψ'. Therefore, ψ' is a solution to the constraint system.

   In the case of non-monotonic constraints, the unification of v_1 and v_2 may lose precision, while preserving correctness. Anyway, in that case, there is no guarantee that a smallest solution exists. In practice, we did not encounter any noticeable loss of precision due to this optimization.

[6] Intuitively, the reason is that `jpaul` does not allow disequality constraints of the form $v_1 \neq v_2$.

[7] This definition corresponds to the common convention in dataflow analysis (including in our analysis) that "smaller" means "more precise".

The solver performs this optimization repeatedly, until no longer possible. Each unification may enable new unifications. E.g., consider the constraints $v_1 \sqsupseteq v_2$, $v_2 \sqsupseteq v_3$, $v_1 \sqsupseteq v_3$. Initially, the solver unifies v_2 and v_3, and obtains the constraint $v_1 \sqsupseteq v_3$: the constraint $v_2 \sqsupseteq v_3$ became the tautology $v_3 \sqsupseteq v_3$, that the solver ignores. Next, the solver unifies v_1 and v_3, and obtains an empty constraint system.

Improved Iteration Technique: Similar to the constraint solver from Section 4.5.1, the `jpaul` solver uses a worklist to iterate over the potentially unsatisfied constraints.

The `jpaul` solver uses an advanced iteration technique called "Iterating Through Strong Components" [66, Chapter 6]. The solver iterates over strongly-connected components of mutually dependent constraints: a constraint C_1 depends on the constraint C_2 if one of C_1's in-variables is an out-variable for C_2. Technically, the solver computes the strongly-connected components (SCCs) of mutually dependent constraints.[8] The solver traverses the SCCs in topological order, starting with the SCCs of constraints that do not depend on constraints from other SCCs. For each SCC, the solver iterates over the constraints from that SCC until they are all satisfied, and next moves to the topologically-next SCCs. Notice that the processing of constraints from future SCCs does not invalidate the constraints from already-processed SCCs. Intuitively, this iteration technique splits the top-level fixed-point computation into several smaller fixed-point computations, one for each SCC, and performs these computations in a "smart" order that avoids unnecessary re-computations.

The `jpaul` solver implements other iteration order optimizations that do not have a noticeable impact for our analysis, but may be useful in a different context. We briefly mention one such optimization, that, to the best of our knowledge, is not described in other publications: the constraint generator (e.g., the pointer analysis) can assign to each constraint an estimation of its relative cost. E.g., constraints for analyzable CALLs are more expensive than simple inequality constraints of the form $v_1 \sqsupseteq v_2$. Inside each SCC of mutually dependent constraints, the `jpaul` solver iterates first over the cheap constraints. Only after all cheap constraints are satisfied does the analysis examine more expensive constraints.[9]

Generating Efficient Constraints

Our implementation generates constraints that are equivalent to the analysis constraints from Section 4.5.1. The generated constraints try to maximize the optimization opportunities inside the `jpaul` solver. In particular, our implementation tries to maximize the number of simple inequality constraints of the form $v_1 \sqsupseteq v_2$.

The constraints from Section 4.5.1 manipulate data at the granularity of points-to graphs. E.g., the constraints for the transfer functions have the form $v^a_{lb} \sqsupseteq [\![lb]\!]^a(v^b_{lb})$,

[8] The `jpaul` library contains a powerful graph package that allows the definition of directed graphs, constructs strongly-connected components, performs topological sorting of acyclic graphs, etc.

[9] Satisfying an expensive constraint may "break" some of the cheap constraints from the same SCC, requiring the solver to examine them again.

where the values of the flow variables v^a_{lb} and v^b_{lb} are points-to graphs. Most transfer functions change only a small part of the points-to graph. E.g., the transfer function for a STATIC STORE may add some nodes to the set of globally escaped nodes, but does not change the set of inside edges. As we explained earlier in this section, the `jpaul` constraint solver saves memory and time by unifying identical entities. To enable this optimization, our implementation generates constraints that manipulate data at lower, sub-points-to-graph granularity. Our implementation uses flow variables for each component of a points-to graph. E.g., in our STATIC STORE example, there is one variable for each set of inside edges, and one variable for each set of globally escaped nodes. The solver is now free to unify the variables for the sets of inside edges before and after the STATIC STORE, while keeping the variables for the sets of globally escaped nodes distinct.

Our implementation uses the optimizations described in Section 4.6.4: for each method, there is one method-wide set of outside edges and one method-wide set of returned nodes. As our implementation uses the SSA form, it represents the state of the local variables flow-insensitively. Still, there is one set of inside edges and one set of globally escaped nodes for each program point. To summarize, the dataflow constraints generated for the body of one method use the following flow variables:

- For each program point, one flow variable for the set of inside edges at that point and one flow variable for the set of globally escaped nodes at that point. More precisely, for each program label lb, the analysis uses the flow variable $\mathrm{v}^b_{I,lb}$ for the set of edges before the label lb and the flow variable $\mathrm{v}^b_{E,lb}$ for the set of globally escaped nodes before the label lb. The analysis uses the corresponding flow variables $\mathrm{v}^a_{I,lb}$ and $\mathrm{v}^a_{E,lb}$ for the program point after the label lb.
- For each local variable v, one flow variable v_v for the set of nodes v points to.
- One flow variable v_O for the method-wide set of outside edges.
- One flow variable v_R for the method-wide set of returned nodes.
- In the case of purity analysis, one flow variable v_W for the method-wide set of mutated prestate abstract fields.

Notice that the values of all variables are sets (of edges or nodes). Therefore, the ordering relation is set inclusion, i.e., in the generated constraints $\sqsupseteq$ is equivalent to $\supseteq$. Our implementation generates as many simple inclusion constraints $\mathrm{v}_1 \supseteq \mathrm{v}_2$ as possible.

Example 12. Assume that the instruction from label lb is a STATIC STORE "$C.f = v$." The analysis generates the following three constraints: $\mathrm{v}^a_{I,lb} \supseteq \mathrm{v}^b_{I,lb}$, $\mathrm{v}^a_{E,lb} \supseteq \mathrm{v}^b_{E,lb}$, and $\mathrm{v}^a_{E,lb} \supseteq \mathrm{v}_v$.

If $\mathrm{v}^a_{I,lb} \supseteq \mathrm{v}^b_{I,lb}$ is the only constraint[10] with $\mathrm{v}^a_{I,lb}$ as an out-variable, the solver is

[10]It indeed is: the information after a label depends only on the information before the label and on the transfer function for that label.

free to unify $v^a_{I,lb}$ and $v^b_{I,lb}$. This optimization would have been impossible if we kept our constraints at the granularity of points-to graphs.

The last two constraints ($v^a_{E,lb} \supseteq v^b_{E,lb}$ and $v^a_{E,lb} \supseteq v_v$) are equivalent to $v^a_{E,lb} \supseteq v^b_{E,lb} \cup v_v$, a constraint that is closer to the spirit of the transfer function for a STATIC LOAD (see Figure 4-3). However, using the two simpler constraints maximizes the number of inclusion constraints. Additionally, the resulting constraints are faster to execute. E.g., if there is a change in the value of v_v, the solver needs to re-evaluate only the constraint $v^a_{E,lb} \supseteq v_v$, in order to propagate elements from v_v to $v^a_{E,lb}$. Evaluating the more complex constraint $v^a_{E,lb} \supseteq v^b_{E,lb} \cup v_v$ requires propagating elements from $v^b_{E,lb}$ too; if the value of $v^b_{E,lb}$ did not change, this propagation is unnecessary. △

Data Structures

The `jpaul` solver allows the constraint generator (in our case, the pointer analysis) to specify the data structures for representing the values of the flow variables. The choice of the data structures significantly impacts the running time of our analysis.

After lowering the data granularity for the constraints, the values of the flow variables are either sets of edges or sets of nodes. A set of edges $S \subseteq \mathit{CNode} \times \mathit{Field} \times \mathit{CNode}$ is equivalent to the function $F_S : \mathit{CNode} \to \mathit{Field} \to \mathcal{P}(\mathit{CNode}) = \lambda n_1.\ \lambda f.\ \{n_2 \mid \langle n_1, f, n_2\rangle \in S\}$. Therefore, our implementation represents sets of edges as maps from starting nodes to maps from fields to sets of ending nodes.[11] This representation allows the implementation to quickly find the ending points of all edges that start in a node n and are labeled with a field f. The analysis uses such an operation in, e.g., the transfer functions for LOAD instructions (see definition of the function *process_load* in Figure 4-5 on page 49).

We experimented with several implementations of sets and maps, including the standard hash-based implementations from the Java standard library and the tree-based and copy-on-write implementations from the `jpaul` library. By default, our implementation uses the hash-based data structures, due to their superior speed.

Discussion: We do not claim that hash-based data structures are best for pointer analysis or for program analysis in general. An in-depth comparison of different data structures is beyond the scope of this thesis. Other researchers [59, 62] report that, for their analyses, generic hash-based implementations were slower than other data structures (e.g., functional data structures, Ordered Binary Decision Diagrams [12], etc.).

8.1.2 Handling of Complex Java Features

This section explains how our analysis implementation handles the complex features of the Java bytecode language.

[11] The corresponding Java 5 type signature is `Map<CNode,Map<Field,Set<CNode>>>`.

Exceptions: The `Flex` intermediate representation models the creation and the propagation of exceptions explicitly. Each instruction that might generate an exception is preceded by a test. If an exceptional situation is detected (e.g., a null pointer dereferencing), the `Flex` intermediate representation follows the Java convention of allocating and initializing an exception object, e.g., a `NullPointerException`, then propagating the exception to the appropriate catch block. This block is determined by a succession of "`instanceof`" tests. If no applicable catch block exists,[12] the exception is propagated into the caller of the current method by a THROW instruction "`throw` v." Unlike a `throw` instruction from Java, a THROW instruction from the `Flex` intermediate representation always terminates the execution of the current method.[13] A CALL instruction has the form "$\langle v_N, v_E \rangle = v_0.s(v_1, \ldots, v_j)$;" it has two successors: if the method returns normally, the control goes to the *normal* successor and v_N points to the returned object; if the method terminates due to an uncaught exception, the control goes to the *exceptional* successor and v_E points to the exception object.

Our analysis requires minimal changes for handling exceptions: in addition to the set of returned nodes, the analysis maintains a set of nodes that may be "thrown" out of the method.[14] We update the transfer function for an analyzable CALL to set the corresponding local variable v_E to point to the set of nodes thrown from the callee, projected through the inter-procedural map. The explicit treatment of exceptions introduces a large number of allocation sites in our program: any instruction that may throw an exception is an allocation site for an exception object. To avoid an explosive increase in the size of the analysis points-to graphs, we represent all exceptions allocated at these sites by the same node.

Call Graph Construction: The `Flex` compiler infrastructure requires a call graph not only for the pointer analysis, but also for detecting all methods whose code should be compiled in the resulting native executable. Computing a static call graph in the presence of dynamic dispatch (i.e., virtual calls) and dynamic class loading is non-trivial.

`Flex` uses the Rapid Type Analysis (RTA) [5] call graph construction algorithm, adapted to Java by the `Flex` developers (originally, RTA was designed for C++). To determine the targets of a virtual call "$v_0.s(v_1, \ldots, v_j)$," RTA considers all instantiated subclasses of the declared type of variable v_0. The set of instantiated classes and the set of executed methods are mutually dependent: executed methods may instantiate new classes and new classes may affect the set of methods that virtual calls invoke.

[12] This is the common case: most Java methods do not declare any catch block.

[13] There are no `finally` blocks in the Java bytecode. Java compilers translate "`finally`" blocks using (1) duplication or (2) calls to *subroutines* internal to the method (using the special bytecode instruction `jsr`, "jump to subroutine"). `Flex` inlines all `jsr` subroutine calls, duplicating the code of the subroutines. The number of duplicates is exponential in the number of nested `finally` blocks. In practice, `finally` blocks are rarely nested, and this duplication does not increase the code size significantly.

[14] Accordingly, for each method, the analysis introduces a flow variable v_T for the set of thrown nodes.

RTA uses a worklist algorithm that starts from the main method of the program and discovers all potentially invoked methods and all potentially instantiated classes. RTA examines the instructions from the body of each newly discovered method. For each virtual call, RTA computes the set of potential callees with respect to the currently known instantiated classes, potentially discovering new invoked methods. The examination of a "`new` C" instruction may reveal a new instantiated class C. In this case, RTA re-examines all virtual calls that perform dynamic dispatch on a variable whose declared type is a superclass of C.

The RTA algorithm produces a correct call graph only if it is able to discover all classes that the program may instantiate. This requirement is hard to satisfy if the analyzed program uses reflection to instantiate classes. E.g., the analyzed program may read a string representing the name of a class, use dynamic class loading to obtain a runtime representation of that class, and next use the `Class.newInstance` method to allocate objects of that class. The particular class instantiated through reflection is not obvious from the source code. Java programs may also use reflection to invoke Java methods that may instantiate Java classes (directly or through transitive callees). Additionally, the current RTA implementation from `Flex` cannot examine the non-Java source code of the native methods. However, the native methods may instantiate Java classes and invoke Java methods. To handle such situations, `Flex` requires the user to manually provide a conservative approximation of:

1. The set of classes that may be instantiated: (1) through reflection, or (2) by the native methods.

2. The set of Java methods that may be invoked: (1) through reflection, or (2) from the native methods.

In `Flex` terminology, the above classes and methods constitute the *root set*. Given the root set for the analyzed program, RTA can compute a conservative static call graph, even in the presence of reflection, native methods, and dynamic class loading. Currently, `Flex` does not provide any tool for helping the user to compute the root set. The development of such a tool (based on either static or dynamic analysis) is an interesting direction for future work.

Native Methods: Given a static call graph, our analysis has no difficulty handling native methods correctly: the analysis can simply consider all calls to native methods as unanalyzable. This is correct, but overly conservative. To increase the analysis precision, we manually provide the summaries (i.e., the end-of-method points-to graphs) for several commonly-used native methods: e.g., the methods `hashCode` and `equals` from `java.lang.Object`, the methods `open`, `available`, `read`, `readBytes`, `skip`, and `close` from `java.io.FileInputStream`, etc. In general, these points-to graphs are empty, stating that the corresponding native methods do not change the aliasing for their parameters.

Dynamic Class Loading and Reflection: The dynamic class loading and the reflection mechanism are implemented as calls to native methods. Our analysis considers these methods unanalyzable; e.g., in the analysis, a call to `Class.newInstance` returns a $n_{\texttt{GBL}}$ node. However, if the application uses dynamic loading / reflection, the user must supply a conservative root set, such that `Flex` can construct a conservative static call graph.

8.1.3 Limitations of the Prototype Implementation

The current implementation analyzes only whole programs; a whole program includes the application-specific code and all the invoked libraries. Unfortunately, the current implementation cannot analyze an isolated library. The idea of whole program compilation is deeply enshrined into many important components of the `Flex` compiler infrastructure, such as the call graph construction algorithm. Theoretically, it is possible to build a tool for analyzing isolated libraries: the tool could model virtual calls to common methods using common assumptions (e.g., a method that overrides `java.lang.Object.equals` does not mutate and does not create new aliasing to its arguments). The tool can classify the other virtual calls as unanalyzable. Non-virtual calls do not pose any problem because the analysis can identify the callee even in the absence of a complete program. This approach is not correct in general: e.g., a library client can define a class whose `equals` method mutates its arguments. However, this approach could provide useful information for applications that respect common programming rules for Java. Unfortunately, we did not have the time to develop such a tool, partially because we had to invest most of our implementation time into debugging the current `Flex` infrastructure. Developing a tool for analyzing separate libraries is an interesting direction for future work.

A second important limitation is that `Flex` supports only the (now outdated) GNU Classpath 0.08 implementation of the Java standard library: i.e., by default, `Flex` cannot analyze and compile applications that use more recent library classes. To analyze more recent benchmarks, we extended the GNU Classpath 0.08 library with several simple JDK 1.5 classes: `java.lang.StringBuilder`, `java.lang.Iterable`, `java.lang.Enum`, `java.util.Queue`, `java.util.AbstractQueue`, `java.util.PriorityQueue`. We also added new methods to several existing classes: `java.lang.String`, `java.lang.Integer`, and `java.lang.Collections`.

8.2 Stack Allocation Experiments

We used our prototype to analyze a significant set of benchmarks and to perform the stack allocation optimization on them. Figure 8-1 describes our benchmarks. We analyzed all programs from the SPECjvm98 [76] industry-standard benchmark suite. We also analyzed all programs from the Java Olden [15] benchmark suite, a set of pointer-intensive programs originally written for C and later ported to Java. Finally, we analyzed two programs popular with compiler researchers: `JLex` (a lexer

Program	Description
SPECjvm98 benchmark set [76]	
`check`	Simple program; tests JVM features
`compress`	File compression tool using the LZW compression algorithm
`jess`	Expert system shell
`db`	Database application
`javac`	JDK 1.0.2 Java compiler
`mpegaudio`	Audio file decompression tool
`mtrt`	Multi-threaded raytracer
`jack`	Java parser generator
Java Olden benchmark set [15, 14, 16]	
`BH`	Barnes-Hut n-body solver
`BiSort`	Bitonic Sort
`Em3d`	Simulation of electromagnetic wave propagation through 3D objects
`Health`	Simulation of a health-care system
`MST`	Bentley's algorithm for minimum spanning tree in a graph
`Perimeter`	Region perimeter computation in a quad-tree-represented image
`Power`	Optimizer of economic efficiency for power consumers
`TSP`	Randomized algorithm for the traveling salesman problem
`TreeAdd`	Recursive depth-first traversal of a tree to sum the node values
`Voronoi`	Computation of a Voronoi diagram for a random set of points
Miscellaneous	
`JLex`	Java lexer generator version 1.2.6 [32]
`JavaCUP`	Java parser generator version 0.10i [33]

Figure 8-1: Analyzed programs.

generator) and `JavaCUP` (a parser generator).

Figure 8-2 presents the size of the analyzed programs and the execution time of the analysis on each of them. For each program, our prototype analyzed all methods that are reachable from the main method, according to the static call graph (see Section 8.1.2). Our prototype analyzed methods both from the user code and from the library code; our prototype supports the GNU Classpath 0.08 [1] implementation of the Java standard library. The analyzed programs range in size from the small `Bisort` (4972 bytecode instructions, 298 methods, 89 classes) to the much larger `javac`, the Sun JDK 1.0.2 Java compiler (54236 bytecode instructions, 1960 methods,

Program	#Classes	#Methods	#Bytecode instructions	Analysis time (s)
`check`	134	560	10854	9.5
`compress`	157	644	13691	10.0
`jess`	312	1373	33057	23.7
`db`	160	771	16623	12.5
`javac`	332	1960	54236	56.4
`mpegaudio`	189	849	23940	17.7
`mtrt`	182	870	20299	14.9
`jack`	203	937	28841	32.8
`BH`	133	548	10011	9.5
`BiSort`	89	298	4972	4.3
`Em3d`	128	514	8957	8.2
`Health`	105	346	9253	6.2
`MST`	92	315	5059	4.5
`Perimeter`	96	320	8163	4.4
`Power`	127	513	9645	7.8
`TreeAdd`	88	288	7499	4.2
`TSP`	90	303	8162	4.4
`Voronoi`	131	561	10003	16.9
`JLex`	166	717	22946	14.2
`JavaCUP`	237	1323	36825	27.1

Figure 8-2: Analyzed program size and analysis time. The size measurements concern only the part of the program that is transitively reachable from the main method.

332 classes).[15]

We conducted our experiments on a 2.8 GHz Pentium 4 computer (no hyper-threading), with 512Kb cache and 1Gb RAM, running Debian Linux. We run our analysis using the Sun JDK 1.5.0, mixed mode.[16] Our prototype requires between 4.3 and 56.4 seconds to analyze each application. The analysis time from Figure 8-2 refers strictly to the time spent inside the pointer analysis: it does not include the execution time for the `Flex` front-end and the `Flex` back-end.

Figure 8-3 presents the execution time for the `Flex` front-end, the `Flex` back-end, and the entire compilation process. The front-end parses the analyzed program, constructs a preliminary program representation, builds the Static Single Assignment

[15]Other publications may report different sizes for the analyzed programs, due to the use of different call graph construction algorithms and/or different implementations of the Java standard library.

[16]Notice the difference between the Java library that the `Flex` infrastructure analyzes and compiles (GNU Classpath 0.08) and the Java library that the analysis implementation uses for its own execution (Sun JDK 1.5.0).

Program	Time for different compiler stages (s)					Total compilation time (s)
	Pointer analysis	`Flex` Front-end			`Flex` C Back-end	
		Total	RTA	SSA		
`check`	9.5	25.0	2.9	3.5	47.1	158.6
`compress`	10.0	24.7	2.9	4.9	69.3	194.6
`jess`	23.7	46.6	5.2	12.1	158.1	409.6
`db`	12.5	28.0	3.0	5.9	75.2	215.2
`javac`	56.4	192.3	43.2	18.8	304.2	897.6
`mpegaudio`	17.7	33.4	3.1	10.0	130.0	324.7
`mtrt`	14.9	33.2	3.1	9.3	116.9	285.7
`jack`	32.8	45.9	3.3	19.2	427.7	635.6
`BH`	9.5	22.6	2.6	3.6	33.6	164.6
`BiSort`	4.3	21.4	2.6	3.2	31.1	126.3
`Em3d`	8.2	21.8	2.6	3.3	30.2	140.6
`Health`	6.2	21.9	2.6	3.3	31.2	141.9
`MST`	4.5	21.8	2.6	3.3	29.6	139.8
`Perimeter`	4.4	21.9	2.6	3.3	40.2	146.6
`Power`	7.8	21.9	2.6	3.5	32.0	149.8
`TreeAdd`	4.2	21.4	2.6	3.2	29.1	146.2
`TSP`	4.4	21.7	2.6	3.3	37.7	157.6
`Voronoi`	16.9	23.4	2.8	3.6	71.2	176.1
`JLex`	14.2	30.6	3.1	7.9	102.1	244.3
`JavaCUP`	27.1	49.6	4.7	15.9	425.9	690.1

Figure 8-3: Execution time for different compiler stages. In this figure, RTA denotes the compiler stage that constructs the call graph using the Rapid Type Analysis [5]. SSA denotes the compiler stage that converts the intermediate program representation to the Static Single Assignment form by inserting a minimal number of ϕ nodes, according to an algorithm based on dominance frontiers [26]. The C back-end compiles the `Flex` intermediate representation into C code, that is later compiled using `gcc`. The total compilation time (last column) is the time spent from the start of the `Flex` compiler until the production of the native executable (including the time required by `gcc`.)

(SSA) [26] program representation, and constructs a call graph using the Rapid Type Analysis [5]. Our implementation uses a back-end that compiles the SSA program representation into C files. Our implementation compiles these C files to native code, using `gcc` (the GNU C compiler) version 3.3.5. When interpreting the numbers from Figure 8-3, one should consider the fact that many compiler stages from `Flex` (e.g., the back-end) are far less optimized than the pointer analysis.

Our prototype performed the stack allocation optimization for each analyzed application. To avoid increasing the stack too much, our prototype did not apply the stack allocation optimization to allocation sites inside loops (even if the corresponding inside nodes are captured). C. Scott Ananian modified the BDW garbage collector [9][17] to trace through the stack allocated objects, and to not deallocate them when they become unreachable. As we explained in Section 5.1, method inlining improves the stack allocation opportunities. We experimented with several maximal inlining depths, ranging from 0 (no method inlining) to 4. In each case, our implementation inlined only the call chains that create new stack allocation opportunities. Additionally, our implementation did not inline CALLs that are inside a loop, or have more than one callee. Unrestricted inlining increases the size of some methods beyond the capabilities of the Flex back-end. To avoid this problem, we do not perform inlinings that cause the size of the caller to exceed 2500 instructions (in our intermediate representation).[18]

Our prototype instrumented the optimized executables to count, at runtime, (1) the number of stack allocated objects, and (2) the total number of allocated objects (including the stack allocated ones). The instrumentation also measured the cumulated size of the allocated objects.

For each application, we executed the optimized executable and manually checked that it produces the same result as the non-optimized version.

Figure 8-4 presents our dynamic results for the stack allocation optimization. We present the percentage of stack allocated objects / memory for each maximal inlining depth. For twelve programs out of twenty, our analysis stack allocates more than 10% of all objects (with a maximum of 95% for `mtrt`). For six programs, our analysis stack allocates more than half of all objects.

To determine which objects are stack allocated, we instrumented the compiled executables to count how many objects from each class are allocated / stack allocated. We briefly discuss our findings:

- `db`: The database simulation `db` represents a database as a `Vector` of entries, each entry containing a `Vector` of items. Virtually every database operation creates at least one auxiliary `Enumeration` object to iterate over a `Vector`. These `Enumeration`s account for 91% of all objects and our analysis stack allocates all of them.

[17] The garbage collector that the generated executables use.

[18] Due to the lack of a clear criterion, the bound 2500 was chosen rather arbitrarily. We could have used a larger bound. However, a larger bound would have increased the compilation time.

Program	Allocated objects						Allocated memory					
	Total ($\times 10^3$)	% Stack allocated for inlining depth					Total (Mb)	% Stack allocated for inlining depth				
		0	1	2	3	4		0	1	2	3	4
`check`	2	18	18	**21**	21	21	0.1	17	17	**21**	21	21
`compress`	3	12	16	**25**	25	25	113.7	0	0	**13**	13	13
`jess`	7951	**0**	0	0	0	0	261.1	**0**	0	0	0	0
`db`	3211	0	**91**	91	91	91	74.7	2	**61**	61	61	61
`javac`	5835	8	**11**	11	11	11	166.6	5	**7**	7	7	7
`mpegaudio`	5	0	0	**1**	1	1	3.3	**0**	0	0	0	0
`mtrt`	6391	87	92	**95**	95	95	132.3	84	89	**94**	94	94
`jack`	7440	10	30	**31**	31	31	189.9	8	28	**30**	30	30
`BH`	15480	47	**93**	93	93	93	355.2	24	**93**	93	93	93
`BiSort`	132	**0**	0	0	0	0	2.5	**0**	0	0	0	0
`Em3d`	41	**0**	0	0	0	0	16.1	**0**	0	0	0	0
`Health`	1198	14	**57**	57	57	57	21.1	22	**59**	59	59	59
`MST`	8394	**0**	0	0	0	0	132.1	**0**	0	0	0	0
`Perimeter`	454	**0**	0	0	0	0	13.9	**0**	0	0	0	0
`Power`	784	**97**	97	97	97	97	21.0	**97**	97	97	97	97
`TreeAdd`	2098	**0**	0	0	0	0	40.0	**0**	0	0	0	0
`TSP`	197	**66**	66	66	66	66	5.9	**53**	53	53	53	53
`Voronoi`	1433	0	**1**	1	1	1	34.0	0	**1**	1	1	1
`JLex`	16	9	17	22	**28**	28	0.8	17	21	26	**35**	33
`JavaCUP`	960	15	19	**34**	34	34	26.6	11	14	**37**	37	37

Figure 8-4: Results for the stack allocation optimization. For each analyzed application, we present the total number of allocated objects and the percentage of objects that are stack allocated for different inlining depths. We also present statistics about the total size of the allocated objects. In each case, we use **bold** font to indicate the smallest inlining depth that achieves the best result. E.g., for `mtrt`, our analysis stack allocates 95% of the objects, inlining call chains of length at most 2; increasing the inlining depth to 3 or 4 does not improve the percentage of stack allocated objects.

Program	Execution time for the unoptimized version (s)	% Speedup for the stack allocation optimization (with inlining depth 3)		
		Total	Inlining alone	Delta
`check`	0.0	–	–	–
`compress`	6.5	9	3	6
`jess`	17.4	1	0	1
`db`	31.7	4	-1	5
`javac`	15.3	10	3	7
`mpegaudio`	4.4	0	0	0
`mtrt`	6.0	28	1	27
`jack`	9.9	5	-1	6
`BH`	10.2	30	1	29
`BiSort`	0.7	–	–	–
`Em3d`	1.8	0	1	-1
`Health`	2.5	30	4	26
`MST`	8.2	4	-1	5
`Perimeter`	0.4	–	–	–
`Power`	13.9	2	0	2
`TreeAdd`	0.8	1	1	0
`TSP`	6.8	2	-1	3
`Voronoi`	1.7	0	-1	1
`JLex`	0.2	–	–	–
`JavaCUP`	1.2	12	1	11

Figure 8-5: Execution time speedup due to the stack allocation optimization. For each program, we compute the speedup using the formula $100 \times (t_u - t_o)/t_u$, where t_u is the execution time for the unoptimized version and t_o is the execution time for the optimized version. E.g., the optimized version of `mtrt` runs 28% faster than the unoptimized version. A negative speedup corresponds to a slowdown. For each program, we present the total speedup T, the speedup I due to the method inlining alone, and the difference $T - I$. We do not report speedups for the programs whose execution time is less than a second. We run each SPECjvm98 program with its default (i.e., size 100) workload. We run each Java Olden program with its default workload. Finally, we run `JLex` on the example `sample.lex` input file (from the official `JLex` distribution [32]), and we run `JavaCUP` on the specification of a parser for Java 1.4 (the file `Parser.cup` from the `Flex` compiler infrastructure [2]).

- `jack`: Most stack allocated objects are `Enumerator`s.
- `mtrt`: Most stack allocated objects are vectors of three elements that store intermediate results of 3D graphic computations.
- `BH`: Almost all stack allocated objects are objects that represent 3D points.
- `Health`: Most stack allocated objects are list iterators.
- `TSP`: Almost all stack allocated objects in `TSP` (66% of all objects!) are instances of the class `java.util.Random`. At a closer examination, we discovered a design bug in `TSP`, most likely introduced while this application was ported from C to Java: every time a random number is required, the program allocates a new random number generator, instead of reusing a program-wide generator. This error results in needless object allocation and lack of real randomness.
- For the rest of the programs, our analysis stack allocates a mix of `String`s, `StringBuffer`s, `Enumerator`s, etc.

For most of the programs, the percentage of stack allocated objects does not increase if we augment the inlining depth beyond 1. The exceptions are `check`, `compress`, `mtrt`, `JLex`, and `JavaCUP`. In particular, `JLex` achieves the maximum percentage of stack allocated objects (28%) only for the inlining depth 3.

The careful reader may notice an anomaly for `JLex`: increasing the inlining depth from 3 to 4 causes the percentage of stack-allocated memory to decrease from 35% to 33%! The explanation is our policy of not performing inlinings into callers that have already reached the limit of 2500 instructions, possibly due to previous inlinings. Some of the inlining chains of depth 4 prevent the inlining of some of the call chains of depth at most 3 (our current implementation has no estimate of which inlining chain is more profitable at run-time).

Figure 8-5 presents the execution speedup due to the stack allocation optimization, for the inlining depth 3. One part of the speedup is due to the reduction in the garbage collection time: the space occupied by stack allocated objects is reclaimed without garbage collection overhead. The rest of the speedup is due to the method inlining itself: inlining may improve code locality and enable more optimizations in later stages of the compiler. Figure 8-5 presents the total speedup, the speedup due to the method inlining alone, and the difference of the two. To measure the speedup due to the method inlining alone, we used our implementation to perform the same inlining as for the stack allocation, but without actually modifying any allocation site.

For five benchmarks — `javac`, `mtrt`, `BH`, `Health`, and `JavaCUP` — the total speedup is at least 10%. In spite of the huge percentage of stack allocated objects, 91%, `db` has a speedup of only 4%; a possible explanation is that the garbage collection overhead is only a small fraction of `db`'s execution time. The speedup due to the method inlining alone is only a small part of the total speedup.

8.3 Impact of the Analysis Optimizations

Our analysis prototype implements all optimizations from Chapter 7. These optimizations increase the analysis speed, e.g., by using type information to eliminate unfeasible inside/outside edges (see Section 7.2). To evaluate the impact of each optimization, we measured the slowdown caused by eliminating each specific optimization (while keeping the others). As there are interactions between different optimizations, our methodology is imperfect. Unfortunately, due to the exponential number of combinations of optimizations, we cannot evaluate all combinations.

Figure 8-6 presents the impact of each individual analysis optimization, as measured when performing the analysis of all methods from the twenty benchmarks from Figure 8-1.[19] By far, the most effective optimization is the iteration limit for the computation of inter-procedural fixed-points (Section 7.1). For each strongly-connected component (SCC) of mutually-recursive methods, our prototype performs at most $k \times m$ iterations, where m is the number of methods from the SCC, and k is an empirically-chosen factor that exponentially decreases from 8 (for small SCCs with $m \leq 5$) to 0, i.e., no iteration, for large SCCs with $m > 30$. This optimization is essential when processing large benchmarks: e.g., without this optimization, the analysis of `javac`, that normally takes less than a minute, does not terminate even after 24 hours.

The other optimizations have a smaller, but still important impact: e.g., the analysis of `javac` is almost 2 times slower without the use of type information (Section 7.2), and 2.69 times slower without the merging of globally escaped nodes (Section 7.3.2).

We also measured the number of stack allocated objects in the absence of each optimization. These numbers are practically unchanged from the case when all optimizations are activated.[20] In particular, computing the inter-procedural fixed-points with no iteration limit does seem to impact the number of stack allocated objects. A natural question is whether we could obtain the same ratio of stack allocated objects by not computing any inter-procedural fixed-point, i.e., by considering all CALLs between mutually-recursive methods as unanalyzable. The experimental results from Figure 8-7 show that not computing any inter-procedural fixed-point speeds up the analysis, but reduces the number of stack allocated objects. Hence, it is useful to try to solve the inter-procedural fixed-points, as long as this operation does not prevent the analysis from terminating on large programs.

8.4 Purity Analysis Experiments

We evaluated our purity analysis using two sets of experiments. First, we performed a set of qualitative experiments: we used our prototype to infer the purity of several complex consistency predicates for data structures. Second, we performed a set of quantitative experiments: we used our prototype to detect pure methods and read-

[19] The same kind of analysis we perform in Section 8.2.

[20] We do not present these numbers because the very few differences are less than 1%.

Acronyms for the analysis optimizations:

- `il` - iteration limit for inter-procedural fixed points (Section 7.1)
- `ti` - use of type information to avoid unfeasible edges and node mappings (Section 7.2)
- `rc` - remove captured nodes at the end of a method (see transfer function for RETURN in Figure 4-3)
- `url` - unify redundant load nodes (Section 7.3.1)
- `uge` - unify globally escaped nodes (Section 7.3.2)

Program	Slowdown in the absence of one optimization				
	`il`	`ti`	`rc`	`ml`	`mge`
`check`	27	0.20	0.16	0.01	0.07
`compress`	21	0.25	0.12	0	0.09
`jess`	73	0.30	**0.21**	0.01	0.20
`db`	18	0.27	**0.21**	0.04	0.10
`javac`	**n/t**	**0.94**	0.18	0.58	**1.69**
`mpegaudio`	29	0.13	0.09	0	0.04
`mtrt`	15	0.18	0.16	0.3	0.13
`jack`	8	0.29	0.16	0.04	0.66
`BH`	30	0.28	0.11	0.02	0.02
`BiSort`	10	-0.03	0.16	0.01	-0.01
`Em3d`	29	0.27	0.14	0.02	0.03
`Health`	7	0.14	0.14	0.01	0
`MST`	9	-0.04	0.16	0	0
`Perimeter`	9	-0.05	0.16	0	0
`Power`	29	0.20	0.09	0.01	0.07
`TreeAdd`	9	-0.07	0.16	-0.01	-0.01
`TSP`	11	-0.04	0.18	**1.73**	0.01
`Voronoi`	25	0.17	0.12	0.04	0.02
`JLex`	14	0.43	0.08	0.03	0.02
`JavaCUP`	146	0.54	0.10	0.09	0.08

Figure 8-6: Impact of the analysis optimizations. For each optimization, we compute the analysis slowdown in the absence of that optimization. We use the formula $(t_w - t_o)/t_o$, where t_o is the execution time of the analysis with all optimizations turned on and t_w is the execution time of the analysis with all optimizations except a specific one. For each optimization, we use **bold** font to indicate the maximum slowdown. E.g., without `ti`, the analysis runs with up to 94% slower, i.e., almost 2 times slower. We use "n/t" to indicate that the analysis of `javac` does not terminate even after 24 hours. A negative slowdown indicates a speedup.

Program	Speedup (%)	% Stack allocated objects		
		Normal version	No inter-proc. fixed-points	Delta
`check`	31	21	5	16
`compress`	26	25	20	5
`jess`	20	0	0	0
`db`	24	91	91	0
`javac`	25	11	9	2
`mpegaudio`	18	1	0	1
`mtrt`	21	95	82	13
`jack`	30	31	23	8
`BH`	37	93	93	0
`BiSort`	32	0	0	0
`Em3d`	31	0	0	0
`Health`	43	57	57	0
`MST`	33	0	0	0
`Perimeter`	31	0	0	0
`Power`	28	97	97	0
`TreeAdd`	33	0	0	0
`TSP`	31	66	66	0
`Voronoi`	65	1	1	0
`JLex`	21	28	25	3
`JavaCUP`	10	34	24	10

Figure 8-7: Impact of not computing inter-procedural fixed-points, by considering all CALLs between mutually-recursive methods unanalyzable. Second column shows the speedup over the normal version of the analysis, the version that "abandons" the fixed-point computation for a group of mutually-recursive methods only after a certain number of iterations. The last column shows the percentage of objects that are no longer stack allocated if the analysis does not compute inter-procedural fixed-points. E.g., without computing inter-procedural fixed-points, the analysis runs 21% faster on `mtrt`, but stack allocates only 82% of the objects, instead of 95%. We run each program with the same workload as for the data from Figure 8-5 on page 128.

Name	Description
Data structures from the Java Collections Framework:	
`LinkedList`	Implementation of a doubly-linked list.
`TreeMap`	Implementation of a map with ordered keys, using a red-black tree.
`HashSet`	Implementation of a set using a hash table.
Miscellaneous:	
`BinarySearchTree`	Implementation of a set of comparable elements, using a binary search tree.
`DisjSet`	Array-based implementation of the union-find data structure, using path compression to improve efficiency of find operations.
`HeapArray`	Array-based implementation of a heap (a.k.a. priority queue).
`BinomialHeap`	Advanced implementation of a heap [24, Chapter 19].
`FibonacciHeap`	Advanced implementation of a heap [24, Chapter 20].

Figure 8-8: Korat benchmarks.

only parameters in the twenty benchmarks used in the stack allocation experiments (see Figure 8-1). The two parts of this section explain our two sets of purity analysis experiments.

8.4.1 Checking Purity of Data Structure Consistency Predicates

We ran our analysis on several benchmarks borrowed from the Korat project [10, 64]. Korat is a tool that generates non-isomorphic test cases up to a finite bound. Korat's input consists of (1) a type declaration of a data structure, (2) a finitization (e.g., at most 10 objects of type `A` and 5 objects of type `B`), and (3) `repOk`, a pure boolean predicate written in Java that checks the consistency of the internal representation of the data structure. Given these inputs, Korat generates all non-isomorphic data structures that satisfy the `repOk` predicate. Korat does so efficiently, by monitoring the execution of the `repOk` predicate and back-tracking only over those parts of the data structure that `repOk` actually reads.

Korat relies on the purity of the `repOk` predicates but cannot statically check this property. Writing `repOk`-like predicates is considered good software engineering practice [61]: during the development of the data structure, programmers can write assertions that use `repOk` to check the data structure consistency. Programmers do not want assertions to change the semantics of the program, other than aborting the program when it violates an assertion. The use of `repOk` in assertions provides additional motivation for checking the purity of `repOk` methods.

Figure 8-8 presents the Korat benchmarks that we analyze. These benchmarks consist of implementations of data structures, ranging from simple ones like linked lists to very advanced ones like Fibonacci heaps. The first three data structures are taken from the standard Java library. The only change the Korat developers performed was to add the corresponding `repOk` methods; we did not perform any additional change to any of the Korat benchmarks. The `repOk` methods for these benchmarks use complex auxiliary data structures: sets, linked lists, wrapper objects, etc. Checking the purity of these methods is beyond the reach of simple purity checkers that prohibit pure methods to call impure methods, or to do any heap mutation.

The obvious difficulty with analyzing the Korat benchmarks is that our prototype is a whole-program analyzer that operates under a closed world assumption. We addressed this problem as follows:

- For each Korat benchmark, we provided a small "driver" program that instantiates the corresponding data structure and invokes `repOk` on it.
- The data structure implementation invokes overriders of the following methods: `java.lang.Object.equals`, `java.lang.Object.hashCode`, `java.util.Comparable.compareTo`, and `java.lang.Object.toString`. We call these methods, and all methods that override them, *special* methods.

  It is impossible to analyze the data structures for all (infinitely many) possible special methods. Instead, we specified to the analysis the intended behavior of the special methods: we specified to the analysis that all special methods are pure. This assumption corresponds to the common intuition about the special methods: e.g., programmers do not expect `equals` to change the objects it compares.

  With this assumption, calls to special methods do not change the externally visible heap aliasing: creating a new reference from an existing object would violate the purity requirement.[21] Special methods can create new references from captured objects, but these references are not visible from outside the special methods. Therefore, the analysis can treat calls to the special methods as simple instructions that produce some values, i.e., the analysis can ignore the calls to the special methods. Additional processing is required to model the result of the `toString` special methods: we instructed the implementation to treat each call to `toString` as an allocation site for the returned `String` object and for the underlying array of characters.

We used our prototype to analyze the `repOk` methods for all the Korat benchmarks. The analysis was able to verify that all `repOk` methods mutate only new

[21] Normally, a method can also create externally visible aliasing if it (1) allocates a new object, (2) sets one of its fields to point to an existing object, and (3) returns the new object. However, the `equals`, `hashCode`, and `compareTo` methods return primitive values (integers and booleans); technically, a `toString` method does return an object, but Java strings are immutable objects, equivalent to simple primitive values.

Program	Purity analysis time (s)	Analyzed methods		Object parameters		
		Total	% pure	Total	% read-only	% trivially read-only
`check`	7.6	560	50	726	68	14
`compress`	8.3	640	51	875	66	16
`jess`	19.8	1373	50	2038	60	9
`db`	10.4	771	47	1042	61	15
`javac`	44.7	1960	37	3685	48	7
`mpegaudio`	16.0	849	45	1254	58	11
`mtrt`	14.0	870	48	1241	60	12
`jack`	26.3	937	39	1236	54	13
`BH`	7.2	548	51	738	65	12
`BiSort`	3.8	298	54	403	68	10
`Em3d`	6.6	514	51	679	65	13
`Health`	4.7	346	52	468	65	9
`MST`	3.9	315	54	430	69	10
`Perimeter`	3.9	320	57	454	71	9
`Power`	7.0	513	50	686	64	12
`TreeAdd`	3.7	288	55	394	69	10
`TSP`	4.0	303	53	412	67	9
`Voronoi`	7.3	561	53	754	66	12
`JLex`	15.4	717	43	945	60	12
`JavaCUP`	25.7	1323	40	1842	62	14

Figure 8-9: Purity analysis results for the programs from Figure 8-1.

objects, and are therefore pure. On a Pentium 4 @ 2.8Ghz with 1Gb RAM, our analysis took between 2 and 10 seconds for each benchmark.

Of course, our results are valid only if our assumptions about the purity of the special methods are true. Our tool tries to verify these assumptions for all the special methods that the analysis encountered. Unfortunately, some of these methods use caches for performance reasons, and are not pure. For example, several classes (e.g., `java.lang.String`) cache their hashcode. Other classes cache more complex data, e.g., `java.util.AbstractMap` caches its set of keys and entries (these caches are nullified each time a map update is performed). We manually examined the impure special methods and checked that the mutation they perform corresponds to caching.

8.4.2 Quantitative Experiments for the Purity Analysis

To evaluate the speed of our purity analysis, we executed our prototype on the twenty benchmarks from Figure 8-1. In each case, we analyzed all methods that are transi-

tively reachable from the main method. As in the case of the Korat benchmarks, our prototype assumes that all special methods are pure.

Figure 8-9 presents our experimental measurements. The time required to run the purity analysis is slightly smaller than the time required to run the plain pointer analysis that does not collect any mutation information (see Figure 8-2). The explanation is that the assumptions about the special methods allow the purity analysis to skip the (rather expensive) inter-procedural analysis for CALLs to such methods.

Our analysis identifies a significant percentage of the analyzed methods as pure: from a minimum of 37% for `javac` to a maximum of 57% for `Perimeter`.

The last three columns from Figure 8-9 present the total number of object parameters,[22] the percentage of object parameters that our analysis identifies as read-only, and finally, the percentage of object parameters with immutable types like `String`, `Integer`, etc. The object parameters with immutable types are trivially read-only. Therefore, the power of the analysis is reflected by the discovery of read-only parameters that do not have immutable types. As the data from Figure 8-9 indicates, our analysis identifies between 48% and 71% of the object parameters as read-only. Moreover, the percentage of read-only parameters is far bigger than the percentage of trivially read-only parameters with immutable types.

[22] I.e., we do not count the parameters with primitive types like `int`, `float`, etc.

Chapter 9

Related Work

Pointer analysis is a very active research area: in 2001, Hind [42] surveyed more than seventy-five research papers on pointer analysis. There are several overlapping categories of pointer analyses: A *points-to* analysis identifies the memory locations pointed to by a specific pointer variable / field. An *alias* analysis identifies pairs of pointer expressions that are aliased, i.e., point to the same memory location. An *escape* analysis detects the memory locations that escape a given scope (a method, a thread, etc). A *shape* analysis detects the shape of the program data-structures.[1] Because shape analyses [38, 72] have very different objectives than our analysis, we do not discuss them in this chapter. Our analysis is a combined points-to and escape analysis. The following features make our pointer analysis different from most other pointer analyses:

Rigorous Theoretical Foundation: Our analysis comes with a detailed presentation and correctness proof. Our analysis is one of the very few pointer analyses with a correctness proof. While some pointer analyses [3, 77] are simple enough that their correctness is intuitive, the correctness of other analyses (especially compositional ones) is far from trivial. We are aware of only one other compositional pointer analysis with a rigorous correctness proof: the escape analysis of Blanchet [8].

Solid Evaluation: We implemented our analysis and used it to perform practical applications, such as the stack allocation optimization (Section 5.1) and the purity analysis (Section 5.2). As Hind points out [42], direct measurements of the analysis results (e.g., the size of the points-to sets) can be irrelevant for the analysis effectiveness for practical uses.

We performed experiments on a large set of benchmarks, including the entire SPECjvm98 [76] benchmark suite.

We performed experiments that can reveal errors in the analysis design and implementation: we used the stack allocation optimization to produce optimized

[1] E.g., a shape analysis detects information of the form "variable v points to an acyclic linked list."

executables that, as we manually checked, produce the same result as the unoptimized ones. In contrast, some other analyses [60, 71] are evaluated by identifying allocation sites that allocate method-local objects, and next counting the objects allocated at these sites during the program execution. We call this second methodology *simulated stack allocation*. Simulated stack allocation does not reveal errors in the analysis design and implementation. In our case, passing from simulated to real stack allocation required non-trivial implementation effort and revealed bugs in the analysis implementation.

Our analysis is one of the very few pointer analyses with a publicly available implementation [79]. Our implementation has been used with positive results by other researchers [27].

Technical Differences: Our analysis is able to obtain correct and useful information by analyzing only parts of a whole program.[2] Our analysis analyzes methods without knowing their calling context; our analysis can also analyze methods that invoke unanalyzable methods (e.g., native methods). Many other pointer analyses [3, 77, 60, 71, 84, 57] are whole-program analyses. To be correct, these analyses need to simulate the behavior of all native methods. Also, the execution time of a whole-program analysis depends on the size of the entire program. In contrast, our analysis can reduce the size of the analysis scope in order to trade precision for speed and terminate in reasonable time.

Additionally, our analysis is compositional: it computes method summaries that are instantiated for each relevant call site. This features achieves some context sensitivity without re-analyzing a method for each calling context.

The rest of this chapter expands the comparison between our analysis and other pointer analyses. We describe the applications of each discussed analysis that analyzes Java programs. Due to the lack of publicly-available implementations and the diversity of evaluation conditions — different benchmarks, different versions of the Java standard library,[3] different compiler infrastructures, etc. — it is impossible to perform a meaningful numeric comparison between the results for each analysis application.[4] Therefore, we compare only the magnitude of the numeric results.

Section 9.1 describes techniques for modeling the heap. Section 9.2 presents general information about context-sensitivity. We use this information while discussing related analyses in the next two sections. Section 9.3 describes two popular pointer

[2]The current implementation examines the whole program in order to construct a static call graph using the Rapid Type Analysis algorithm [5]. However, the pointer analysis (much more expensive than the call graph construction algorithm) does not need to examine the entire program.

[3]An analyzed program consists of the user code *and* the transitively invoked library methods.

[4]Due to the different evaluation conditions, even the number of allocated objects can vary by more than one order of magnitude. E.g., the analysis of Blanchet [8] stack allocates 19% of the 169×10^3 objects allocated by `javac`. The analysis of Rountev *et al.* [71] identifies as stack allocatable 21.2% of the 3738×10^3 objects allocated by `javac` (without actually stack allocating them). Finally, our analysis stack allocates 11% of the 5835×10^3 objects allocated by `javac`.

analyses by Andersen [3] and Steensgaard [77], and several of their variations. Section 9.4 comments on other compositional analyses. Section 9.5 discusses other purity and side-effect analyses.

9.1 Techniques for Modeling the Heap

Many analyses, including ours, model the heap as a graph of bounded size [18, 86, 21, 85, 60, 71, 84, 57]. Usually, these analyses express escapability in terms of graph reachability.

As the number of objects created by the program execution might be unbounded, it is necessary to *summarize* many objects into a single node. The *k-limiting* model [46] uses distinct nodes for the objects that are reachable from program variables along paths of length at most k, and one summary node for all other objects. To prevent an exponential blowup of the number of nodes, k is usually chosen to be very small.

Like many other analyses [21, 85, 60, 71, 84, 57], our analysis uses the alternative *object allocation site* model [18]: all objects allocated by a program statement are modeled by the same node. This model assumes that objects allocated by the same program statement are likely to be manipulated in a similar way. Being directly related to the structure of the program, the results of an analysis that uses this model are conceptually easy to understand and use. For compositionality reasons, we extended the object allocation site model with parameter and load nodes.

Other analyses [52, 20] use pairs of aliased heap paths instead of points-to graphs. Although alias pairs can be more expensive to represent than graphs, the compact representation of the alias pairs of Hind *et al.* [43] retains the same memory consumption as the points-to graphs. Deutsch [28] goes beyond k-limiting by representing the heap paths as regular expressions. Deutsch's model improves the accuracy of the analysis of recursive data structures that are accessed in a regular way. Unfortunately, the prototype implementation of the analysis of Deutsch has been used only to compute alias pairs for C programs of less than 50 lines [28, Section 5].

Several fast escape analyses avoid constructing points-to graphs by using very conservative approximations of escapability. One such approximation is the use of integer levels to represent the escaped objects that are transitively reachable from a variable [39, 29, 7]. Intuitively, the escaped objects are first approximated by their types, and next the types are approximated by integer levels in a properly chosen type hierarchy. For example, Blanchet [7, 8] presents a flow-insensitive escape analysis for Java programs. The level of a type/class C is chosen to be at least as big as (1) the level of each subclass of C and (2) the level of the type of each field of C. For each variable v, the analysis computes an integer that is at least as big as the level of any escaped object that v transitively points to. Consider an allocation site "v = `new` C". If the integer the analysis computes for v is strictly smaller than the level of C, then the objects created at this site are captured and can be stack allocated. As it uses integer levels (instead of graphs), Blanchet's analysis is conceptually less precise

but much faster than ours. Blanchet's analysis has a rigorous correctness proof and achieves stack allocation results comparable to ours. Blanchet's analysis is effective for stack allocation, but it is unclear whether it can be used for other applications. In contrast, our analysis can also detect pure methods.

Gay and Steensgaard [36] present a whole-program escape analysis for stack allocating captured objects in Java programs. Their analysis considers that an object escapes as soon as it is stored in an object field. As most stack allocatable objects in Java programs are manipulated only through local variables (e.g., iterators and string buffers), their analysis [36] obtains good stack allocation results for practical benchmarks. Our analysis can be simplified to use similar ideas and treat heap references very conservatively. In the simplified analysis, any field store would globally escape nodes, and each field load would return $n_{\texttt{GBL},0}$.[5] Gheorghioiu, Sălcianu and Rinard [37] present such an analysis. They use their analysis to detect allocation sites with the property that at most one object allocated there is live at any point during the program execution. Such allocation sites can be transformed to reuse statically preallocated memory space.

9.2 Techniques for Context Sensitivity

A *context-insensitive* analysis does not distinguish between different invocations of the same method. A *context-sensitive* analysis computes specialized results for different *contexts*. A context is a static abstraction of a set of dynamic invocations of a method. In our analysis, a context is an abstraction of the objects transitively pointed to by the method parameters. In other analyses [58, 84, 87], a context is a static call site from which a method is called. An obvious extension is to use contexts consisting of longer *call-chains*: e.g., the last k call sites on the execution path that invokes a method, where k is a small constant.

Usually, context-insensitive analyses [3, 77] model each call instruction using a control flow edge from the call instruction to the beginning of the callee and another control flow edge from the end of the callee back into the caller. Such context-insensitive analyses lose precision by propagating information along *unfeasible paths*. An unfeasible path is a control flow path that cannot be executed at runtime. E.g., a path that starts from a caller, enters a callee, and next returns into a different caller is unfeasible.

Some analyses [65, 58, 84, 87] achieve context-sensitivity by "stamping" each analysis fact with a context. The analysis of a callee does not "mix" analysis facts with different contexts, and the analysis propagates into the caller only the analysis facts stamped with the appropriate context.

Our analysis, like several others [86, 7, 19, 21, 85], achieves context-sensitivity using a *compositional* technique: for each method, our analysis computes a method

[5] This is equivalent to considering each field load/store as a call to an equivalent, but unanalyzable method. Hence, the correctness proof automatically extends to the simplified analysis.

summary that it later instantiates for each relevant context.[6]

9.3 Andersen- and Steensgaard-style Analyses

Andersen [3] and Steensgaard [77] propose two context-insensitive and flow-insensitive analyses for C programs. The two analyses differ in the treatment of assignment instructions. For each assignment $v_1 = v_2$, Andersen's analysis constrains the set of objects pointed to by v_2 to be a subset of the set of objects pointed to by v_1. Steensgaard's analysis constrains the two sets to be equal. Both analyses are whole-program analyses. Conceptually, they regard the entire program as a big procedure. Each call "$v_R = m(v_1, v_2, \ldots v_k)$" is treated as a series of assignments from the arguments to the formal parameters plus an assignment from the returned value to the variable v_R. These analyses generate set inclusion constraints and solve them using set-constraint solvers like BANE [34] and Banshee [47]. The complexity of Andersen's analysis is cubic in the size of the analyzed program. The complexity of Steensgaard's analysis is almost linear in the size of the analyzed program.[7] Heintze and Tardieu [41] describe an ultra-fast implementation of Andersen's analysis that is able to process a million lines of C code in one second. They use their implementation to detect data-dependencies.

Rountev *et al.* [71] and Liang *et al.* [60] are among the first to adapt the analyses of Andersen and Steensgaard to Java, by adding support for dynamic method dispatch and increasing the field-sensitivity.[8] These analyses are generally faster than ours, but not always: e.g., Rountev's analysis analyzes `jack` in 5871 seconds on a 60Mhz Sun Ultra 60 with 512Mb of RAM. The analysis of Liang *et al.* [60] increases its speed using user-supplied models for the methods of the collection data structures. Both analyses are used for simulated stack allocation. In general, the stack allocation results are comparable to those of our analysis. However, our analysis achieves significantly better results in some cases: e.g., for `db`, our analysis stack allocates 91% of all objects, while Rountev's analysis identifies only 0.01% objects as method-local.

Hirzel *et al.* [45] implement Andersen's analysis inside a Java Virtual Machine (JVM). Running inside a JVM, their analysis can handle dynamic class loading by incrementally re-analyzing the program each time a new class is loaded. Hirzel *et al.* [45] do not report experimental results about any analysis application.

Milanova *et al.* [65] notice that context-insensitivity compromises the precision of pointer analysis for Java programs. To address this problem, Milanova *et al.* extend Rountev's analysis [71] with *object-sensitivity*, a form of context-sensitivity.

[6] Sharir and Pnueli [73] call this technique *functional*.

[7] Steensgaard's analysis is *almost* linear because it uses a union-find data structure [24, Chapter 21]. Hence, its complexity also depends on the (practically constant) inverse of the extremely fast-growing Ackermann function.

[8] Andersen and Steensgaard-style analyses for C programs are either *field-based* or *field-independent*. A field-based analysis ignores the source o_1 of a heap reference $\langle o_1, f, o_2 \rangle$. A field-independent analysis ignores the field f. A *field-sensitive* analysis models all three elements of the heap reference: o_1, f, and o_2.

Milanova's analysis re-analyzes each method separately for each possible abstraction of the receiver object.[9] Milanova *et al.* present several analysis uses: detection of locations potentially modified by each statement, call graph construction, and virtual call resolution.[10] Milanova's analysis is faster, but of the same order of magnitude as ours.

Recently, three independent research groups — Lhoták and Hendren [58, 57], Whaley and Lam [84], and Zhu and Calman [87] — present Andersen-style analyses that use Ordered Binary Decision Diagrams (BDDs) [12] to represent large points-to sets efficiently. The high speed and the low memory consumption of BDDs allow these analyses to increase their context sensitivity by using very long call chains. Regarding the analysis applications, Lhoták and Hendren [58, 57] perform call graph construction, virtual call resolution, and cast safety analysis (an analysis that detects type casts that may fail at runtime). Whaley and Lam [84] present experimental results for the static number of synchronization operations that are unnecessary because they synchronize only on thread-local objects. Whaley and Lam also present results for an analysis application that refines the declared types of local variables.[11] Zhu and Calman [87] present only internal analysis measurements: analysis execution time, size of points-to sets, etc. These three analyses are very fast. However, more work is required in order to evaluate their practical applications. In particular, we would like to see an analysis application (like stack allocation) that could fail in case of errors in the analysis design and implementation.

All Andersen and Steensgaard-style analyses presented so far are whole-program analyses. In contrast, our analysis can obtain useful results by analyzing only parts of a program. Several researchers propose versions of Andersen's analysis that are not whole-program. Rountev [69] applies Andersen's analysis to *software fragments* (arbitrary collections of C procedures) by using worst-case assumptions about the unknown callers and callees (e.g., each callee creates a heap reference between each two objects it may access). Unfortunately, Rountev [69] does not present results about any analysis application.

Heintze and Tardieu [40] present a demand-driven version of Andersen's analysis for C programs. Sridharan *et al.* [75] present a similar demand-driven analysis for Java programs. Sridharan's analysis answers queries about the objects pointed to by a variable. For each query, the analysis evaluates only the analysis constraints that are relevant for that particular query. The analysis aborts each query after a certain time threshold. To gain context sensitivity, the analysis inlines all setter[12] and getter[13] methods. The analysis seems promising for environments with extreme resource constraints (e.g., Java Virtual Machines). So far, Sridharan's analysis has

[9] The receiver object is the object pointed to by the implicit `this` argument of the method.

[10] Virtual call resolution detects virtual calls with at most one possible callee. The compiler can replace each such virtual call with a non-virtual call, reducing the dynamic dispatch overhead.

[11] E.g., if a local variable v has type `java.lang.Object`, and the analysis discovers that v may point only to objects of class C_1 and C_2, that are both subclasses of the class C, then the analysis refines the type of v from `java.lang.Object` to C.

[12] A setter is a small method that sets a field of its receiver object to a certain value.

[13] A getter is a small method that returns the value of a field of its receiver object.

been applied for virtual call resolution and for alias pair detection.

9.4 Compositional Analyses

As we discussed in Section 9.2, several analyses, including ours, achieve context sensitivity by computing method summaries and instantiating them for each relevant context. This section discusses several such analyses.

The pointer analysis of Wilson and Lam [86] analyzes C programs in a top-down fashion: their analysis starts from the main procedure and re-analyzes each procedure for each new calling context. For each method, the analysis caches the results for already examined contexts. At the end of the analysis, the cache for a method constitutes a method summary that covers only the contexts that appear in the program execution.[14]

Alternatively, our analysis analyzes the program in a bottom-up fashion, from the leaves of the call graph toward the main procedure. For each method, the analysis computes a summary that is later instantiated for the different call sites that might call that method. The analysis of Blanchet [7] (already discussed in Section 9.1) uses a similar bottom-up approach.

Chatterjee *et al.* [19] present a pointer analysis for C++ programs that combines the top-down and the bottom-up approaches: in an initial bottom-up pass, their analysis infers the calling context conditions that are *relevant* for the points-to relation, and computes one method summary for each instantiation of the relevant conditions. Next, in a top-down pass, the analysis propagates information from the callers to the callees. In contrast, our analysis extracts a single summary for each method. The disadvantage of our approach is that it may produce less precise results than an approach that maintains multiple method summaries. The advantage is that it leads to a simpler algorithm and smaller analysis results.

Compositional analyses are very complex and their correctness is non-trivial. Unfortunately, we are unaware of correctness proofs for the analysis of Wilson and Lam [86] and the analysis of Chatterjee *et al.* [19].

The analysis of Choi *et al.* [21] and the analysis of Whaley and Rinard [85] are similar to our analysis. These analyses analyze the program in a bottom-up fashion and use the object allocation site model extended for compositionality reasons. Both analyses [21, 85] perform synchronization removal and stack allocation, obtaining results comparable to ours. Our work improves on these two analyses in several ways:

1. Our analysis comes with a rigorous correctness proof. Choi *et al.* [22] present a correctness proof for their analysis. Their proof relies on an informal semantics of Java (instead of a precise semantics, as the one from Section 6.1) and we are skeptical about its correctness.

[14] Wilson and Lam call these method summaries *partial transfer functions*.

2. The presentation of our analysis is more detailed, making the analysis easier to understand, implement, and reproduce. Our analysis also has a publicly-available implementation [79].

3. The analysis of Choi *et al.* [22] does not distinguish between inside and outside edges: all edges potentially model reads from the calling context. This fact may conceptually lead to a lack of precision.

4. Although our analysis was initially based on the Whaley and Rinard analysis [85] (referred to below as the WR analysis), the correctness proof required important modifications in the design of the inter-procedural analysis.

   For example, the inter-procedural stage of the WR analysis has two distinct parts: (1) the construction of a node map by matching outside edges from the callee against inside edges from the points-to graph right before the call, followed by (2) the projection of the callee edges into the points-to graphs after the call.

   During the correctness proof from our SM thesis [78], we discovered the need to match outside edges from the callee against inside edges from both the caller *and the callee*. This change is necessary for correctly handling all possible parameter aliasing situations.

   The inter-procedural analysis from this dissertation achieves correctness by conceptually merging the steps (1) and (2) from WR. Each inter-procedural transformer can both extend the map and project edges into the caller. Future applications of inter-procedural transformers can use these newly added edges.

   We found it difficult to understand all of the details of the analysis of Choi *et al.* [22]. However, we note that their analysis, like the WR analysis, also has distinct phases for the node map construction and the projection of the callee edges into the points-to graphs after the call. Thus, it suffers from the same problem as the WR analysis.

Vivien and Rinard [83] modified an early implementation of our analysis to obtain an incremental analysis. The analysis starts by analyzing only a small part of the program "around" the "interesting" instructions. The set of interesting instructions depends on the specific optimization. For stack allocation, these are the instructions that allocate most of the objects, as indicated by profile data. The analysis uses a performance/cost strategy to extend the scope of the analysis to capture more nodes, until either the analysis budget was exhausted or the fate of all the interesting nodes has been decided.

9.5 Purity and Side-Effect Analyses

The Java Modeling Language (JML) [54, 55, 13] is a behavioral specification language for Java that allows users to write annotations like, for example, method preconditions, method postconditions and class invariants. In addition, JML allows users to

specify pure methods. JML uses the same method purity definition as our analysis (see Definition 5 on page 73). JML annotations can invoke Java methods, provided those methods are pure. Currently, JML tools either do not check the purity annotations, or check them using very conservative analyses: e.g., a method is pure iff (1) it does not do any I/O, (2) it does not write any heap field, and (3) it does not invoke any impure method [53]. Our purity analysis does not require purity annotations and is more precise, due to the underlying pointer analysis: our purity analysis identifies (and ignores) mutations that occur only on new objects.

Other researchers — e.g., Choi *et al.* [20], Milanova *et al.* [65], and Rountev [70] — have already considered the use of pointer analysis for inferring side-effects. Unlike these previous analyses, our analysis uses a separate abstraction (the inside nodes) for the objects allocated by the current invocation of the analyzed method. Therefore, our analysis focuses on prestate mutation and supports pure methods that mutate newly allocated objects. Previous analyses can at most allow pure methods to mutate newly allocated objects that are also captured. Rountev [70] presents evidence that almost all pure methods can be detected using a simple Andersen-style pointer analysis. However, Rountev's method purity definition is more rigid than ours; e.g., Rountev's definition does not allow pure methods to construct and return new objects. We are not aware of any publicly-available implementation of these previous analyses.

A different research direction consists of requiring the programmer to write annotations about the side-effects, as part of an extended type system. We briefly mention a few such approaches: Javari (short for "Java with Reference Immutability") [6, 82] is an extension to Java that allows the programmer to specify `readonly` parameters and fields. The Javari type system ensures that the program does not perform mutation on objects it accesses only through `readonly` references. The program may still mutate objects pointed to by a `readonly` reference, if those objects are accessible through non-`readonly` references. Data groups [56], region types [80, 25] and/or ownership types [23, 11] can be used to specify effects at a larger-than-object granularity: groups of objects, regions, ownership boundaries, etc. There are multiple advantages of these annotation-based approaches: (1) annotations are well suited for modular checking, (2) annotations are useful for program documentation, and (3) checking user-provided annotations is generally faster than an analysis like ours, that analyzes programs without any annotations. The disadvantage is that these approaches require additional programmer effort for writing the annotations (although tools for inferring some of the annotations are possible and exist, in the case of Javari [81]).

Chapter 10

Conclusion

This chapter summarizes the dissertation, enumerates possible directions for future work, and presents a few thoughts on the future of pointer analysis.

10.1 Summary

This dissertation presented a pointer analysis for Java programs, together with two practical analysis applications: a program optimization (stack-allocation of local objects) and a program understanding application (detection of pure methods).

The work described in this dissertation is different from most published pointer analysis research due to its depth. This dissertation described the design of our analysis, formalized the meaning of the analysis results, explained practical analysis applications, and provided a rigorous correctness proof. Moreover, this dissertation described our analysis implementation and reported experimental results for a large set of benchmarks including all programs from the SPECjvm98 benchmark suite [76]. Our analysis implementation [79] is publicly-available and has already been used by other researchers [27].

For each program point, our pointer analysis is able to construct a points-to graph that describes how local variables and object fields point to objects. Each points-to graph also contains escape information that identifies the objects that are reachable from outside the analysis scope. Our pointer analysis has two interesting features:

- Our analysis extracts correct information by analyzing only parts of a whole program. First, our analysis analyzes a method without knowing its callers. Second, our analysis correctly handles calls to unanalyzable methods (e.g., native methods). Hence, our analysis can trade precision for speed without sacrificing correctness: if the analysis of a call to a specific callee requires too much time, the analysis can treat that call as unanalyzable.

- Our analysis is compositional. Our analysis analyzes each method without knowing its calling context. The analysis uses the points-to graph for the end

of the method to construct a method summary. The inter-procedural analysis instantiates this method summary for the calling context at each relevant method invocation. Hence, the analysis achieves some context sensitivity without re-analyzing a method for each new calling context.

This dissertation described the use of the analysis results to perform a program optimization that stack-allocates local objects. This optimization reduces the garbage-collection overhead. For the SPECjvm98 benchmarks suite, our analysis implementation stack-allocates up to 95% of the dynamically allocated objects (32% on average).

This dissertation also explained how to extend the analysis to detect pure methods. Our analysis supports a flexible definition of method purity: a method is pure if it does not mutate any object that exists in the program state before the start of the method. Therefore, our analysis allows pure methods to allocate and mutate temporary objects (e.g., iterators) and/or construct complex object structures and return them as a result. However, our analysis does not allow pure methods to use caches; identifying and reasoning about caching mechanisms is an interesting (and difficult!) direction for future work. In practice, our implementation detects the purity of several complex methods that are beyond the reach of previously implemented purity analyses.

10.2 Future Work

This dissertation already mentioned several possible directions for future work. We briefly enumerate these directions here:

- Adding support for strong updates of node fields (Section 4.6.2).
- Increasing the analysis precision for static fields (Section 4.6.3).
- Experimenting with a fully flow-insensitive version of the analysis (Section 4.6.4).

Experimenting with analysis ideas requires an impressive infrastructure: intermediate representation builders, code generators, constraint solvers, etc. Unfortunately, good analysis infrastructure is hard to find. To address this problem, we released `jpaul` [31], an open-source library of data-structures and algorithms that are useful for program analysis. We plan to continue developing `jpaul`, especially the constraint solver.

Currently, the method summaries do not contain any information about the ordering between the atomic inter-procedural transformers. Therefore, the application of a method summary requires a fixed-point even if the code of the corresponding method does not contain any loop. An interesting direction for future work is to extend the points-to graphs with ordering information. This extension may increase the analysis precision.

10.3 Thoughts on the Future of Pointer Analysis

Designing and implementing our pointer analysis required a substantial effort. The effort to prove the correctness of our pointer analysis was enormous, in large part

because of the complexity of the inter-procedural part of the analysis. Due to these difficulties and due to the correctness concerns associated with successfully developing complex analyses, we believe that the field will be dominated by a few analyses that have rigorous, well-established foundations. Once these analyses become widely implemented and available, we believe that the research focus will shift toward two classes of pointer analyses:

- Provably-sound analyses that require user-provided specification at some abstraction boundary (methods, modules, etc). Conceptually, the specification should allow precise, fast, and modular analyses. Hob [50, 49] and Jahob [48] are two recent analysis systems that build on these ideas. Developing a usable specification language that handles the commonly used features of Java is still an open problem.

- Unsound analyses that handle common situations and produce practically useful results for bug-finding, profiling, and program understanding tools.

We hope that future pointer analyses will use some of the ideas from this dissertation. We identify two such ideas: First, the use of placeholders to abstract over the unknown calling context. Second, the combination of points-to and escape information that allows our analysis to analyze parts of a whole program.

Appendix A

Optimized Algorithm for the Inter-procedural Analysis

This chapter presents an optimized algorithm for the critical part of the inter-procedural analysis, the computation of $\langle I_a, O_a, E_a, \mu_a \rangle = T_{callee}(I, O, E, \mu_0)$, where $T_{callee} = mct(G_2) = \texttt{star}(\bigsqcup \mathcal{AT}(G_2))$ and $G_2 = \tau(gc(\rho(G_{callee})))$ (see Section 4.4). The optimized algorithm has asymptotic time complexity $\mathcal{O}(N^5)$, where N is the size of the analysis scope: still big, but a clear improvement over the $\mathcal{O}(N^9)$ naive algorithm from Section 4.4.

Idea: The optimized algorithm keeps track of the changes (e.g., new node mappings) in order to re-execute only those atomic transformers from $\mathcal{AT}(G_2)$ that may produce new information. For example, the optimized algorithm does not re-execute an atomic transformer $\texttt{gesc}(n)$ unless there are new mappings for the node n.

Figures A-1 and A-2 present the pseudocode for the optimized algorithm. The algorithm computes its result in the tuple of variables $\langle I_a, O_a, E_a, \mu_a \rangle$: the algorithm initializes this tuple to $\langle I, O, E, \mu_0 \rangle$ and increases[1] its value during successive iterations. Our algorithm uses a worklist W_{change}; this worklist contains notifications of the changes that are relevant for the atomic transformers. W_{change} contains three kinds of notifications:

- $\langle \texttt{newesc}, n \rangle$: the predicate $e_2(E_a, I_a)(n)$ became true (due to new elements in E_a and/or I_a).
- $\langle \texttt{newiedge}, n_1, f, n_2 \rangle$: I_a contains the new inside edge $\langle n_1, f, n_2 \rangle$.
- $\langle \texttt{newmap}, n_1, n_2 \rangle$: μ_a contains the new node mapping $\langle n_1, n_2 \rangle$.

The inter-procedural analysis manipulates only the nodes from $G_2 = \tau(gc(\rho(G_{callee})))$ (where G_{callee} is the points-to graph from the end of the callee) and

[1] We use the common element-wise ordering between tuples and the subset ordering between sets: $\langle I_1, O_1, E_1, \mu_1 \rangle \sqsubseteq \langle I_2, O_2, E_2, \mu_2 \rangle$ iff $I_1 \subseteq I_2$, $O_1 \subseteq O_2$, $E_1 \subseteq E_2$, and $\mu_1 \subseteq \mu_2$.

In the algorithm below, let $\mathcal{N}$ be the set of nodes that appear in $G_2 = \tau(gc(\rho(G_{callee})))$ or in G (the points-to graph before the CALL).

GLOBAL DATA
I_a, O_a, E_a, μ_a - results (under construction)
Esc - set of nodes n such that $e_2(E_a, I_a)(n)$ is true
W_{change} - worklist of change notifications

INITIALIZATION
$I_a \leftarrow I$; $O_a \leftarrow O$; $E_a \leftarrow E$; $\mu_a \leftarrow \mu$;
$W_{\text{change}} \leftarrow empty$;
$Esc \leftarrow \emptyset$;
$\forall n \in \mathcal{N} \cap (CPNode \cup CLNode \cup \mathcal{G} \cup E_a)$:
    $findNewEscNodes(n)$;
$\forall n_1 \in \mathcal{N}$, $\forall n_2 \in \mu_a(n_1)$:
    add $\langle \texttt{newmap}, n_1, n_2 \rangle$ to W_{change};
$\forall n \in Esc$:
    add $\langle \texttt{newesc}, n \rangle$ to W_{change};

ITERATIONS
while W_{change} is not empty
    extract change notification cn from W_{change};
    case analysis on cn:
    Case 1: $cn = \langle \texttt{newesc}, n \rangle$
        $\forall n_1 \in \mu_a^{-1}(n)$, $\forall f, n_2$ such that $\texttt{load}(n_1, f, n_2) \in \mathcal{AT}(G_2)$:
            $O_a \leftarrow O_a \cup \{\langle n, f, n_2 \rangle\}$
            if $\langle n_2, n_2 \rangle \notin \mu_a$ then
                $\mu_a \leftarrow \mu_a \cup \{\langle n_2, n_2 \rangle\}$;
                add $\langle \texttt{newmap}, n_2, n_2 \rangle$ to W_{change};
    Case 2: $cn = \langle \texttt{newiedge}, n_1, f, n_2 \rangle$
        $\forall n_3 \in \mu_a^{-1}(n_1)$, $\forall n_4$ such that $\texttt{load}(n_3, f, n_4) \in \mathcal{AT}(G_2)$:
            if $\langle n_4, n_2 \rangle \notin \mu_a$ then
                $\mu_a \leftarrow \mu_a \cup \{\langle n_4, n_2 \rangle\}$;
                add $\langle \texttt{newmap}, n_4, n_2 \rangle$ to W_{change};

Figure A-1: Optimized algorithm for computing $\langle I_a, O_a, E_a, \mu_a \rangle = T_{callee}(I, O, E, \mu_0)$, where $T_{callee} = mct(G_2)$ and $G_2 = \tau(gc(\rho(G_{callee})))$ - Part 1 of 2. This algorithm is the critical step from the inter-procedural analysis (see Section 4.4).

```
30      Case 3: cn = ⟨newmap, n1, n2⟩
31          ∀f, n3 such that load(n1, f, n3) ∈ AT(G2),
32              ∀n4 such that ⟨n2, f, n4⟩ ∈ Ia:
33                  if ⟨n3, n4⟩ ∉ μa then
34                      μa ← μa ∪ {⟨n3, n4⟩};
35                      add ⟨newmap, n3, n4⟩ to Wchange;
36                  if n2 ∈ Esc then
37                      Oa ← Oa ∪ {⟨n2, f, n3⟩}
38                      if ⟨n3, n3⟩ ∉ μa then
39                          μa ← μa ∪ {⟨n3, n3⟩};
40                          add ⟨newmap, n3, n3⟩ to Wchange;
41          if gesc(n1) ∈ AT(G2) then
42              Ea ← Ea ∪ {n2}
43              if n2 ∉ Esc then
44                  findNewEscNodes(n2);
45          ∀f, n3 such that store(n1, f, n3) ∈ AT(G2),  ∀n4 ∈ μa(n3):
46              if ⟨n2, f, n4⟩ ∉ Ia then
47                  Ia ← Ia ∪ {⟨n2, f, n4⟩};
48                  add ⟨newiedge, n2, f, n4⟩ to Wchange;
49                  if n2 ∈ Esc  ∧  n4 ∉ Esc then
50                      findNewEscNodes(n4);
51          ∀f, n3 such that store(n3, f, n1) ∈ AT(G2),  ∀n4 ∈ μa(n3):
52              if ⟨n4, f, n2⟩ ∉ Ia then
53                  Ia ← Ia ∪ {⟨n4, f, n2⟩};
54                  add ⟨newiedge, n4, f, n2⟩ to Wchange;
55                  if n4 ∈ Esc  ∧  n2 ∉ Esc then
56                      findNewEscNodes(n2);

57  procedure findNewEscNodes(ns)
58      Wesc ← empty;
59      Esc ← Esc ∪ {ns};
60      add ns to Wesc;
61      add ⟨newesc, ns⟩ to Wchange;
62      while Wesc is not empty
63          extract node n from Wesc;
64          ∀f, n2 such that ⟨n, f, n2⟩ ∈ Ia:
65              if n2 ∉ P then
66                  Esc ← Esc ∪ {n2};
67                  add n2 to Wesc;
68                  add ⟨newesc, n2⟩ to Wchange;
```

Figure A-2: Optimized algorithm for computing $\langle I_a, O_a, E_a, \mu_a \rangle = T_{callee}(I, O, E, \mu_0)$, where $T_{callee} = mct(G_2)$ and $G_2 = \tau(gc(\rho(G_{callee})))$ - Part 2 of 2.

the nodes from G (the points-to graph before the CALL); the inter-procedural analysis does not create new nodes. Therefore, the algorithm considers only the nodes from the set $\mathcal{N} = nodes(G_2) \cup nodes(G)$. The set $\mathcal{N}$ has $\mathcal{O}(N)$ elements.

The algorithm initializes W_{change} to contain one $\langle \texttt{newmap}, n_1, n_2 \rangle$ notification for each mapping $\langle n_1, n_2 \rangle$ from μ_0, and one $\langle \texttt{newesc}, n \rangle$ notification for each node n such that $e_2(E, I)(n)$ is true.

In each iteration, the algorithm extracts a change notification cn from W_{change} and re-executes the atomic transformers that may increase one or more elements of the tuple $\langle I_a, O_a, E_a, \mu_a \rangle$.

E.g., assume $cn = \langle \texttt{newmap}, n_1, n_2 \rangle$ (see Case 3 of the main loop, in Figure A-2). If the atomic transformer $\texttt{gesc}(n_1)$ exists, the algorithm executes it and adds the node n_2 to E_a (see lines 41-44 in the algorithm). The algorithm does not waste time executing transformers of the form $\texttt{gesc}(n')$, where $n' \neq n_1$.

As a result of the new mapping for n_1, the algorithm needs to re-execute the atomic transformers of the form $\texttt{store}(n_1, f, n_3)$. Normally, such a transformer adds to I_a the edges $\mu_a(n_1) \times \{f\} \times \mu_a(n_3)$. The algorithm considers only the new mapping for n_1, i.e., the algorithm adds only the edges $\{n_2\} \times \{f\} \times \mu_a(n_3)$ (intuitively, the other mappings for n_1 have already been processed, or are pending in the worklist and will be processed later). For each new inside edge $\langle n_2, f, n_4 \rangle$, the algorithm adds a notification $\langle \texttt{newiedge}, n_2, f, n_4 \rangle$ to the worklist W_{change}. Any transformer of the form $\texttt{store}(n_3, f, n_1)$ requires a similar processing.

A notification of the form $\langle \texttt{newiedge}, n_1, f, n_2 \rangle$ triggers the re-execution of all transformers $\texttt{load}(n_3, f, n_4)$ with $\langle n_3, n_1 \rangle \in \mu_a$ (equivalently, $n_3 \in \mu_a^{-1}(n_1)$); such a `load` transformer creates a map from n_4 to n_2 (see lines 25-29). If this mapping is new for μ_a, the algorithm adds a notification $\langle \texttt{newmap}, n_4, n_2 \rangle$ to the worklist W_{change}.

To avoid recomputing the predicate $e_2(E_a, I_a)$ every time a `load` transformer uses it, the algorithm *incrementally* computes the set *Esc* of nodes n such that $e_2(E_a, I_a)(n)$ is true:

1. If a `gesc` transformer adds a node $n \notin Esc$ to E_a, n becomes escaped. Accordingly, the algorithm uses the procedure *findNewEscNodes*(n) to add to *Esc* any previously unescaped node that is transitively reachable from n.

2. When a `store` transformer adds an edge $\langle n_1, f, n_2 \rangle$ to I_a, if $n_1 \in Esc$ and $n_2 \notin Esc$, then n_2 becomes escaped. As in the previous case, the algorithm uses the procedure *findNewEscNodes*(n_2) to add to *Esc* any previously unescaped node that is transitively reachable from n.

In both cases, every time the algorithm adds a new node n to the set *Esc*, the algorithm adds the notification $\langle \texttt{newesc}, n \rangle$ to W_{change}. Notifications of the form $\langle \texttt{newesc}, n \rangle$ are important, because they cause each load transformer $\texttt{load}(n_1, f, n_2)$ such that $n \in \mu_a(n_1)$ to generate the potentially new mapping from n_2 to n_2.

Notes: each test $\texttt{store}(n_1, f, n_2) \in \mathcal{AT}(G_2)$ is equivalent to the test $\langle n_1, f, n_2 \rangle \in I_2$, where I_2 is the set of inside edges from the points-to graph G_2. Similar considerations apply to the tests $\texttt{load}(n_1, f, n_2) \in \mathcal{AT}(G_2)$, and $\texttt{gesc}(n) \in \mathcal{AT}(G_2)$. Therefore, an implementation does not need to explicitly construct the set $\mathcal{AT}(G_2)$.

Asymptotic Complexity

We assume that all sets require $\mathcal{O}(1)$ time for element addition and membership testing. We assume that a worklist requires $\mathcal{O}(1)$ time for element addition and extraction.

The algorithm represents the node map μ_a (in fact a relation) as a map from keys to sets of values; it also represents the reverse relation μ_a^{-1} in a similar way. Therefore, the algorithm can add a mapping and check the existence of a mapping in $\mathcal{O}(1)$ time. The algorithm can also iterate over the elements from $\mu_a(n)$ and $\mu_a^{-1}(n)$ in $\mathcal{O}(1)$ time for each element.

The algorithm represents the sets of inside/outside edges as maps from start nodes to maps from fields to sets of target nodes.[2] Therefore, the algorithm can add an edge and check for edge membership in $\mathcal{O}(1)$ time. The algorithm can also iterate over all edges that start in a node n_1 (e.g., in line 45) in $\mathcal{O}(1)$ time for each such edge. Similarly, the algorithm can iterate over all f-labeled edges that start in a node n_2 (e.g., in line 32) in $\mathcal{O}(1)$ time for each such edge. Additionally, the algorithm indexes the callee inside edges by their target node. Therefore, the algorithm iterates over the edges that end in a node n_1 (e.g., in line 51) in $\mathcal{O}(1)$ time for each such edge. This indexing (if it does not exist already) requires $\mathcal{O}(N^3)$ time at the beginning of the algorithm, without affecting the overall $\mathcal{O}(N^5)$ complexity.

The procedure *findNewEscNodes* explores the edges exiting a node n only when n is extracted from the worklist W_{esc}, and each node is added to W_{esc} at most once, immediately after it is added to *Esc*. Hence, all invocations of *findNewEscNodes* explore the edges from I_a at most once. Therefore, the cumulated time for all invocations of *findNewEscNodes* is $\mathcal{O}(N^3)$.

The initialization step requires $\mathcal{O}(N^3)$ time, due to the assignments from line 6: the size of I_a and O_a is $\mathcal{O}(N^3)$.

The algorithm adds each notification $\langle \texttt{newesc}, n \rangle$ to W_{change} at most once, when the algorithm adds the new node n to *Esc* (see pseudocode of *findNewEscNodes*). Hence, Case 1 (lines 19-24) occurs $\mathcal{O}(N)$ times; each time there are $\mathcal{O}(N^3)$ choices for n_1, f, and n_2 and the processing for each choice requires $\mathcal{O}(1)$ time. Therefore, all executions of Case 1 require $\mathcal{O}(N^4)$ time.

The algorithm adds each notification $\langle \texttt{newiedge}, n_1, f, n_2 \rangle$ to W_{change} at most once, when the algorithm adds the new edge $\langle n_1, f, n_2 \rangle$ to I_a (see lines 48 and 54). Hence, Case 2 (lines 25-29) occurs $\mathcal{O}(N^3)$ times; each time, there are $\mathcal{O}(N^2)$ choices for n_3 and n_4 and the processing for each choice requires $\mathcal{O}(1)$ time. Therefore, all executions of Case 2 require $\mathcal{O}(N^5)$ time.

[2] An equivalent Generic Java / C++ signature is `Map<CNode,Map<Field,Set<CNode>>>`.

The algorithm adds each notification $\langle \texttt{newmap}, n_1, n_2 \rangle$ to W_{change} at most once, when the algorithm adds the new mapping $\langle n_1, n_2 \rangle$ to μ_a. Hence, Case 3 occurs $\mathcal{O}(N^2)$ times. For the code from lines 31-40, there are $\mathcal{O}(N^3)$ choices for f, n_3, n_4, and $\mathcal{O}(1)$ processing time for each choice; we can apply similar reasoning for the blocks of code from lines 45-50 and 51-56. The code from lines 41-44 requires $\mathcal{O}(1)$ time. Therefore, all executions of Case 3 require $\mathcal{O}(N^2 \times (N^3 + 1 + N^3 + N^3)) = \mathcal{O}(N^5)$ time.

The complexity of the entire algorithm is $\mathcal{O}(N^3 + N^3 + N^4 + N^5 + N^5) = \mathcal{O}(N^5)$.

Correctness

Let $X_i = \langle I_a^i, O_a^i, E_a^i, \mu_a^i \rangle$ be the value of $\langle I_a, O_a, E_a, \mu_a \rangle$ at the beginning of the ith iteration of the algorithm. As we proved in the complexity analysis above, the algorithm adds $\mathcal{O}(N^3)$ notifications to the worklist W_{change}. Hence, the algorithm terminates after $k = \mathcal{O}(N^3)$ iterations, with the result $X_k = \langle I_a^k, O_a^k, E_a^k, \mu_a^k \rangle$. We need to show that this result equals $T_{\mathit{callee}}(I, O, E, \mu_0)$, where $T_{\mathit{callee}} = \texttt{star}(\bigsqcup \mathcal{AT}(G_2))$.

We prove the desired equality by proving two inequalities. For brevity, we write $\mathcal{A} = \mathcal{AT}(G_2)$ and $F = \bigsqcup \mathcal{A}$. With these notations $T_{\mathit{callee}} = \texttt{star}(F)$.

Lemma 9. $X_k = \langle I_a^k, O_a^k, E_a^k, \mu_a^k \rangle \sqsupseteq T_{\mathit{callee}}(I, O, E, \mu_0)$

Proof. As we proved in Section 4.4, each atomic transformer is extensive and monotonic. Therefore, F is extensive and monotonic too. By Part 2 of Lemma 17 on page 169, $T_{\mathit{callee}}(I, O, E, \mu_0)$ is the smallest fixed point of F that is bigger than $\langle I, O, E, \mu_0 \rangle$. As we prove below, X_k is a fixed-point for each atomic transformer $T \in \mathcal{A}$. This implies that X_k is a fixed point for F too. As X_k is also bigger than $X_0 = \langle I, O, E, \mu_0 \rangle$,[3] we obtain the desired result of the lemma.

To prove that X_k is a fixed point for each atomic transformer T, we do a case analysis on T:

$T = \texttt{gesc}(n)$**:** $\texttt{gesc}(n)$ adds to E_a^k all nodes from $\mu_a^k(n)$. We prove that all these nodes are already in E_a^k, and hence X_k is a fixed point of $\texttt{gesc}(n)$. Let n' be a node from E_a^k. For each mapping $\langle n_1, n_2 \rangle$, our algorithm added (once) the notification $\langle \texttt{newmap}, n_1, n_2 \rangle$ to the worklist W_{change}.[4] As W_{change} is empty at the end of the algorithm, the algorithm processed the notification $\langle \texttt{newmap}, n, n' \rangle$ in some iteration $i < k$. Hence, the code for Case 1 already added n' to $E_a^{i+1} \subseteq E_a^k$.

$T = \texttt{store}(n_1, f, n_2)$**:** this transformer adds the inside edges $\langle n_3, f, n_4 \rangle$ for each $n_3 \in \mu_a^k(n_1)$ and each $n_4 \in \mu_a^k(n_2)$. For each such edge, let $i < k$ be the iteration that processed the last of the notifications $\langle \texttt{newmap}, n_1, n_3 \rangle$ and $\langle \texttt{newmap}, n_2, n_4 \rangle$.[5] This iteration already added $\langle n_3, f, n_4 \rangle$ to $I_a^{i+1} \subseteq I_a^k$ (in line 47 if $\langle \texttt{newmap}, n_1, n_3 \rangle$ is processed last, in line 53 otherwise).

[3] Our algorithm is cumulative (it never removes elements); hence, $X_0 \sqsubseteq X_1 \sqsubseteq \ldots \sqsubseteq X_k$.

[4] In the initialization step, if $\langle n_1, n_3 \rangle \in \mu_0$, or in the iteration that added $\langle n_1, n_3 \rangle$ to μ_a.

[5] "Last" according to the chronologic order of the processing time.

$T = \texttt{load}(n_1, f, n_2)$**:** For any nodes n_3 and n_4 such that $n_3 \in \mu_a^k(n_1)$ and $\langle n_3, f, n_4\rangle \in I_a^k$, this transformer adds the mapping $\langle n_2, n_4\rangle$ to μ_a^k. We prove that $\langle n_2, n_4\rangle$ already exists into μ_a^k.

Assume that $\langle n_3, f, n_4\rangle \notin I_a^0$. Hence, the algorithm added this edge to I_a at line 47 or 53. In each case, the algorithm added the notification $\langle \texttt{newiedge}, n_3, f, n_4\rangle$ to W_{change}. As $n_3 \in \mu_a^k(n_1)$, the algorithm added the notification $\langle \texttt{newmap}, n_1, n_3\rangle$ to W_{change}. Let $i < k$ be the iteration that processed the last of these two notifications. This iteration added $\langle n_2, n_4\rangle$ to μ_a^{i+1} either in line 28 (if `newiedge` is processed last) or in line 34 (if `newmap` is processed last). Hence, $\langle n_2, n_4\rangle \in \mu_a^{i+1} \subseteq \mu_a^k$.

The case $\langle n_3, f, n_4\rangle \in I_a^0$ is even easier: we simply pick $i < k$ to be the iteration that processes the element $\langle \texttt{newmap}, n_1, n_3\rangle$. Line 34 ensures that $\langle n_2, n_4\rangle \in \mu_a^{i+1} \subseteq \mu_a^k$.

If there exists a node $n_3 \in \mu_a^k(n_1)$ such that $e_2(E_a^k, I_a^k)(n_3)$, the `load` transformer adds the mapping $\langle n_2, n_2\rangle$ and the outside edge $\langle n_3, f, n_2\rangle$. We prove that μ_a^k and O_a^k already contain these elements. For the mapping $\langle n_1, n_3\rangle$, the algorithm added the notification $\langle \texttt{newmap}, n_1, n_3\rangle$ to W_{change}. Similarly, when $e_2(E_a^k, I_a^k)(n_3)$ became true, the algorithm added $\langle \texttt{newesc}, n_3\rangle$ to W_{change}. Let $i < k$ be the iteration that processed the last of these two notifications. The code from lines 19-24 (if `newesc` is processed last) or the code from lines 36-40 (if `newmap` is processed last) already added the mapping $\langle n_2, n_2\rangle$ to $\mu_a^{i+1} \subseteq \mu_a^k$ and the outside edge $\langle n_3, f, n_2\rangle$ to $O_a^{i+1} \subseteq O_a^k$.

In all cases, T is a fixed-point for X_k; this completes the proof of our lemma. □

Lemma 10. $X_k = \langle I_a^k, O_a^k, E_a^k, \mu_a^k\rangle \sqsubseteq T_{callee}(I, O, E, \mu_0)$

Proof. Let $X = T_{callee}(I, O, E, \mu_0)$. As we mentioned in the proof of Lemma 9, X is the smallest fixed point of F that is bigger than $\langle I, O, E, \mu_0\rangle$: $F(X) = X$ and $X \sqsupseteq \langle I, O, E, \mu_0\rangle$.

We prove by induction on i that $\forall i \geq 0.\ X_i \sqsubseteq X$. The relation is true for the base case $i = 0$, because $X_0 = \langle I, O, E, \mu_0\rangle \sqsubseteq T_{callee}(I, O, E, \mu_0) = X$ (T_{callee} is extensive, because each atomic transformer is extensive). For the induction step, we assume $X_i \sqsubseteq X$ and prove that $X_{i+1} \sqsubseteq X$. As we prove below, there exists a finite l such that $X_{i+1} \sqsubseteq F^l(X_i)$, where F^l denotes the composition of F with itself l times. As F is monotonic, $X_i \sqsubseteq X$ (by the induction hypothesis), and X is a fixed point for F, we complete the proof by induction as follows: $X_{i+1} \sqsubseteq F^l(X_i) \sqsubseteq F^l(X) = X$.

We still need to prove that $\forall i,\ \exists l$ such that $X_{i+1} \sqsubseteq F^l(X_i)$. Assume that in iteration i, the algorithm processes a $\langle \texttt{newesc}, n\rangle$ notification (i.e., Case 1). This notification indicates that $e_2(E_a, I_a)(n)$ became true. The key observation is that lines 21-24 execute "parts" of the transformer $\texttt{load}(n_1, f, n_2)$: those lines add only the outside edge $\langle n, f, n_2\rangle$ and the mapping $\langle n_2, n_2\rangle$.

If $\langle I_1, O_1, E_1, \mu_1\rangle$ are the values of the variables $\langle I_a, O_a, E_a, \mu_a\rangle$ right before line 21, and $\langle I_2, O_2, E_2, \mu_2\rangle$ are the corresponding values right after line 24, then $\langle I_2, O_2, E_2, \mu_2\rangle \sqsubseteq \texttt{load}(n_1, f, n_2)(I_1, O_1, E_1, \mu_1)$. Also, $\texttt{load}(n_1, f, n_2) \sqsubseteq F$ (because

$F = \bigsqcup \mathcal{A}$ and $\texttt{load}(n_1, f, n_2) \in \mathcal{A}$). Hence, $\langle I_2, O_2, E_2, \mu_2 \rangle \sqsubseteq F(I_1, O_1, E_1, \mu_1)$. As we mentioned in the proof of Lemma 9, F is monotonic. Therefore, if l is the number of executions of the lines 21-24 in the current iteration, then $X_{i+1} \sqsubseteq F^l(X_i)$.

The other cases are similar: the algorithm applies a composition of monotonic parts of a finite number of atomic transformers. For brevity, we omit the details. $\square$

Appendix B

Technical Parts of the Correctness Proof from Chapter 6

B.1 Proof of Lemma 5 on page 95

Lemma 5 states that the concrete escape predicates defined in Section 6.4.1 conservatively approximate the set of escaped objects. More specifically, if a non-null object o escapes at date $d \in Date$, then $e_d(o)$ holds:

$$\forall d \in Date, \underline{\forall o \in RObjs(outside_{A(m)}(\Xi_d)) \setminus \{o_{\texttt{null}}\}.\ e_d(o)}$$

We prove Lemma 5 by induction on the program execution trace. The base case is trivial: no object is reachable in the initial concrete state Ξ_0. For the induction step, we assume that the underlined property above is true at each date $d_2 \leq d$ and prove it at date $d+1$. We do a case analysis on the type of the instruction executed in the transition $\Xi_d \Rightarrow \Xi_{d+1}$.

For each object o that already escapes at date d, by the induction hypothesis, $e_d(o)$ holds, and, by Constraint 6.4, $e_{d+1}(o)$ holds too. Hence, it is sufficient to examine only the instructions that can cause objects to escape from $A(m)$.

Each instruction executed outside $A(m)$ accesses only objects that already escape from $A(m)$. The only exception is a NEW instruction executed outside $A(m)$ (e.g., in a different thread, or in a method invoked using an unanalyzable CALL). Such a NEW instruction creates a new object o that is reachable from outside $A(m)$. Constraint 6.5 ensures that $e_{d+1}(o)$ is true.

Consider an instruction executed by $A(m)$. There are two classes of instructions that can escape objects. STORE instructions can create new heap paths. Other instructions (STATIC STORE, unanalyzable CALL, and final RETURN) can store object references into static fields or into local variables from stack frames outside $A(m)$.

Consider a STORE instruction "$v_1.f = v_2$", and assume that at date d, v_1 points to the object o_1 and v_2 points to the object o_2. Consider a previously captured object o that escapes at date $d+1$. We need to prove that $e_{d+1}(o)$ holds. It is sufficient to consider the case $\neg e_d(o)$ (otherwise, by Constraint 6.4, $e_{d+1}(o)$). As o escapes at date

$d+1$, there exists a path of heap references that reaches o from an object o_3 pointed to by a static field or by a local variable outside $A(m)$. As o does not escape at date d, this path includes the new heap reference $\langle o_1, f, o_2\rangle$. As $\langle o_1, f, o_2\rangle$ is the only new heap reference, the path from o_3 to o_1 and the path from o_2 to o both exist at date d. Hence, o_1 escapes at date d, and, by the induction hypothesis, $e_d(o_1)$. Let $\langle o_4, f, o\rangle$ be the last heap reference on the path from o_2 to o. This heap reference was created by $A(m)$: otherwise, when the corresponding STORE instruction was executed at some date $d_3 < d$, o was pointed to by a local variable outside $A(m)$, and by the induction hypothesis, $e_{d_3}(o)$, which contradicts $\neg e_d(o)$. Therefore $\langle o_4, f, o\rangle \in I_d^{A(m)}$, and $\neg e_d(o_4)$. By repeating this reasoning, we prove that the path from o_2 to o exists in $I_d^{A(m)} \subseteq I_{d+1}^{A(m)}$. As $\langle o_1, f, o_2\rangle \in I_{d+1}^{A(m)}$ too, there exists a path of heap references from $I_{d+1}^{A(m)}$ that reaches o from o_1. As $e_d(o_1)$, by Constraint 6.4, $e_{d+1}(o_1)$, and, by Constraint 6.6, $e_{d+1}(o)$.

Consider a STATIC STORE instruction "$C.f = v$", and assume that v points to an object o_1 at date d. Let o be an object that is captured inside $A(m)$ at date d and escapes at date $d+1$. We need to prove that $e_{d+1}(o)$ holds. Consider the interesting case $\neg e_d(o)$. Notice that all heap references that exist at date $d+1$ exist at date d too. Hence, as o is captured at date d and escapes at date $d+1$, there exists a path of heap references from o_1 to o. As in the case of a STORE instruction, we prove the existence of a path from o_1 to o, along heap references from $I_d^{A(m)} \subseteq I_{d+1}^{A(m)}$. By Constraint 6.8, $e_{d+1}(o_1)$, and, by Constraint 6.6, $e_{d+1}(o)$. The cases of an unanalyzable CALL or a final RETURN from $A(m)$ are similar.

B.2 Proof of the Abstract Semantics Invariants from Section 6.5

For each interesting date $d \in ID_{A(m)}$, $\exists j$ such that $d = id_j$. We prove Invariants 1-8 together, by induction on j. For each date $d \in ID_{A(m)}$, let $\langle V_d, lb_d\rangle$ be the top stack frame from thread t. I.e., V_d is the state of the local variables from the top method from thread t,[1] and lb_d is the current label in that method. Additionally, H_d denotes the concrete heap at date d. As in Section 6.5, $G_d = \langle L_d : J_d, I_d, O_d, E_d, R_d, W_d\rangle$.

Initial State

In the initial state, $d = id_0$. All invariants are true:

Invariant 1: $A(m)$ has not created any objects yet.

Invariant 2: The initial modeling relation ρ_{id_0} involves only parameter nodes.

Invariant 3: The only modeled objects are the arguments, that were created before $A(m)$; hence, they escape due to Constraint 6.5 on page 94.

[1] "Top" is relative to our convention that the execution stack grows from callers to callees.

Invariant 4: The concrete stack has a single relevant frame, $\langle V_{id_0}, lb_{id_0} \rangle$, and the only potentially non-null variables are the parameters. In ρ_{id_0}, each parameter node $n^P_{l,0}$ models the object $V_{id_0}(p_l)$ pointed to by the corresponding parameter p_l, $\forall l \in \{0, 1, \ldots k-1\}$.

Invariant 5: $A(m)$ has not created any heap references yet.

Invariant 6: The only nodes that model an object in ρ_{id_0} are the parameter nodes, and they all escape by Definition 2 on page 41.

Invariant 7: The only nodes that appear in G_{id_0} and ρ_{id_0} are parameter nodes with context $0 \leq |J_{id_0}|$.

Invariant 8: $A(m)$ has not mutated any location yet.

Induction Step

We assume the invariants hold for $d = id_j$, and prove them correct for $d = id_{j+1}$ too. The case $j = 2i + 1$ is very easy: nothing relevant changes between $id_j = id_{2i+1}$ and $id_{j+1} = id_{2(i+1)}$: the points-to graphs, the modeling relations, and even the concrete escape predicates are identical.[2] Therefore the invariants remain valid for id_{j+1}.

The rest of this proof focuses on the more interesting case $j = 2i$, when $G_{id_{2i+1}} = [\![lb_{id_{2i+1}}]\!](G_{id_{2i}})$ (see Figure 4-3 and Figure 6-5 for the definition of $[\![.]\!]$) and $\rho_{id_{2i+1}}$ is defined by the rules from Figure 6-6.

For each invariant, we perform case analysis on the type of the instruction $P(lb_{id_{2i}})$. For an unanalyzable CALL, this is the first instruction from the chain of transitions $\Xi_{id_{2i}} \Rightarrow^* \Xi_{id_{2i+1}}$. In all other cases, $P(lb_{id_{2i}})$ is the instruction from the transition $\Xi_{id_{2i}} \Rightarrow \Xi_{id_{2i+1}}$. By a simple inspection of Figure 6-6, notice that, except for the case of a RETURN inside $A(m)$, $\rho_{id_{2i}} \subseteq \rho_{id_{2i+1}}$. Therefore, it suffices to consider only the elements that change in the concrete/abstract semantics. For brevity, we denote $lb = lb_{id_{2i}}$ and $c = |J_{id_{2i}}|$.

Invariant 1: It suffices to examine the instructions that allocate new objects (adding new objects to the set $Allocs_d^{A(m)}$), change the modeling relation ρ_d, or change the set of globally escaped nodes E_d. A NEW from label lb creates a new object o, but it also adds a modeling relation between $n^I_{lb,c}$ and o. A RETURN inside $A(m)$ may change the modeling relation and the points-to graph by adjusting the contexts of some nodes; still, inside nodes are transformed into inside nodes, which preserves the invariant. Other instructions are irrelevant, as they may at most add new pairs to the modeling relation or add new globally escaped nodes.

[2] To see why $e_{id_{2(i+1)}} = e_{id_{2i+1}}$, notice that the only applicable constraint from Definition 14 is 6.4: $e_{id_{2i+1}} \sqsubseteq e_{id_{2(i+1)}}$. As Definition 14 uses a least fixed point, this constraint implies equality.

Invariant 2: It suffices to examine the cases of NEW and RETURN inside $A(m)$: the other instructions do not change the set of objects that the inside nodes model. A NEW only adds a modeling relation between the inside node $n^I_{lb,c}$ and the object allocated at lb. A RETURN inside $A(m)$ may only change the contexts of the nodes from the modeling relation. In both cases, the invariant remains valid.

Invariant 3: It suffices to examine the instructions that change the modeling relation. Some other instructions may escape previously-captured objects, which does not affect Invariant 3. No instruction can cause an escaped object to become captured: due to Constraint 6.4 on page 94, once an object escapes, it escapes forever.

NEW creates a new object o and models it with the inside node $n^I_{lb,c}$. As o is fresh, no other nodes model o, and Invariant 3 is satisfied. A RETURN inside $A(m)$ affects the contexts of the nodes from the modeling relation, but, for any object o, the cardinality of the set $\{n \mid n\ \rho_d\ o\}$ does not increase.[3] The remaining instructions preserve Invariant 3 because they extend the modeling relation only by putting a node to model an escaped object: LOAD uses a load node to model the object it reads, but only if that object escapes (see Figure 6-6 on page 98). Similarly, STATIC LOAD puts $n_{\texttt{GBL},c}$ to model the object o that it reads from a static field; by Lemma 5, $e_{id_{2i}}(o)$, and so, by Constraint 6.4, $e_{id_{2i+1}}(o)$. Finally, an unanalyzable CALL puts $n_{\texttt{GBL},c}$ to model the returned object o: as o was reachable from outside $A(m)$ right before being returned, it escapes by Lemma 5, i.e., $e_{id_{2i}}(o)$; by Constraint 6.4, $e_{id_{2i+1}}(o)$.

Invariant 4: STORE, STATIC STORE, IF, and THREAD START do not change the state of local variables nor the modeling relation. NULLIFY may set a local variable to point to $o_{\texttt{null}}$, but this is irrelevant, because Invariant 4 refers only to non-null variables. In all these simple cases, Invariant 4 remains true.

For a STATIC LOAD instruction "$v = C.f$", the concrete semantics sets the local variable v (from the topmost stack frame) to point to the loaded object o, while the abstract semantics sets the local variable v (also from the topmost stack frame) to point to $n_{\texttt{GBL},c}$. As $n_{\texttt{GBL},c}\ \rho_{id_{2i+1}}\ o$ (see Figure 6-6), Invariant 4 remains true. The case of an unanalyzable CALL is similar.

For a NEW instruction "$v = \texttt{new}\ C$", the concrete semantics sets v to point to the new object o. Accordingly, the abstract semantics sets v to point to the corresponding inside node $n^I_{lb,c}$. As $n^I_{lb,c}\ \rho_{id_{2i+1}}\ o$ (see Figure 6-6), Invariant 4 remains true.

For a COPY instruction, the concrete and the abstract semantics operate similarly: the concrete semantics sets v_2 to point to the object that v_1 points to, while the abstract semantics sets v_2 to point to all nodes pointed to by v_1. Therefore, the validity of Invariant 4 propagates from date id_{2i} to date id_{2i+1}. The case of an analyzable CALL is similar: a CALL copies the actual arguments into the formal parameters. A RETURN inside $A(m)$ operates in three steps (see Figure 6-5). Each of them preserves Invariant 4. The first step is similar to a COPY instruction: it

[3] In fact, the cardinality of this set may decrease because nodes with previously different contexts may become identical.

copies the returned object into a variable of the caller. Next, unreachable nodes are garbage collected. As the nodes pointed to by local variables are trivially reachable, they are unaffected by this step. Finally, a RETURN inside $A(m)$ may change the contexts of the nodes; as this changes are done consistently in the points-to graph and the modeling relation, Invariant 4 remains true.

Before advancing to the next case, consider the following auxiliary lemma:

Lemma 11. *Let $\langle o_1, f, o_2\rangle \in H_{id_{2i}}$ be a heap reference at date id_{2i}. If $\neg e_{id_{2i}}(o_1)$ or $\neg e_{id_{2i}}(o_2)$, then $\langle o_1, f, o_2\rangle \in I^{A(m)}_{id_{2i}}$.*

Proof. Assume $\neg e_{id_{2i}}(o_1)$ (the case $\neg e_{id_{2i}}(o_2)$ is similar). For the sake of contradiction, assume that the heap reference $\langle o_1, f, o_2\rangle \notin I^{A(m)}_{id_{2i}}$. Therefore, $\langle o_1, f, o_2\rangle$ was created by a STORE instruction outside $A(m)$, at some date $d_2 < id_{2i}$. Since o_1 was reachable from outside $A(m)$ at date d_2, by Lemma 5, $e_{d_2}(o_1)$ is true, which implies $e_{id_{2i}}(o_1)$ (concrete escape predicates are cumulative, by Constraint 6.4 from Definition 14). Contradiction! □

The case of a LOAD instruction "$v_2 = v_1.f$" is more interesting. Assume that in the concrete semantics v_1 points to o_1 and that the heap contains the heap reference $\langle o_1, f, o_2\rangle$; hence, after this instruction, in the concrete semantics, v_2 points to o_2. By Invariant 4 at date id_{2i}, $\exists n_1 \in L_{id_{2i}}(v_1)$ such that $n_1\ \rho_{id_{2i}}\ o_1$. There are two cases:

1. If $e_{id_{2i}}(o_1)$, by Invariant 6 at date id_{2i}, $e(G_{id_{2i}})(n_1)$ is true. As $R_{id_{2i}} = \emptyset$ (the set of returned nodes is empty, except after the final RETURN, see Lemma 28 on page 177), $e_2(E_{id_{2i}}, I_{id_{2i}})(n_1)$. Hence, the abstract semantics will set v_2 to point to the load node $n^L_{lb,c}$ (maybe among other nodes; see Figure 4-5 on page 49): $n^L_{lb,c} \in L_{id_{2i+1}}(v_2)$. As $n^L_{lb,c}\ \rho_{id_{2i+1}}\ o$ (see Figure 6-6 on page 98), Invariant 4 is preserved.

2. If $\neg e_{id_{2i}}(o_1)$, by Invariant 3, n_1 is the only node that models o_1. By Lemma 11, $\langle o_1, f, o_2\rangle \in I^{A(m)}_{id_{2i}}$. Therefore, by Invariant 5 at moment id_{2i}, there exists n_2 such that $n_2\ \rho_{id_{2i}}\ o_2$ and $\langle n_1, f, n_2\rangle \in I_{id_{2i}}$. Hence, the abstract semantics will ensure $n_2 \in L_{id_{2i+1}}(v_2)$. As $n_2\ \rho_{id_{2i}}\ o_2$ and $\rho_{id_{2i}} \subseteq \rho_{id_{2i+1}}$, Invariant 4 is preserved. Notice the importance of the fact that each captured object (e.g., o_1) is modeled by exactly one node: otherwise, $\langle o_1, f, o_2\rangle$ may have been modeled by an inside edge that does not start in n_1 and we would have been unable to prove that $n_2 \in L_{id_{2i+1}}(v_2)$.

Invariant 5: A NEW instruction may create several heap references in order to initialize the fields of the newly created object to null. However, this fact is irrelevant for Invariant 5, that deals only with non-null references. A STORE instruction "$v_1.f = v_2$" creates a heap reference $\langle o_1, f, o_2\rangle \in I^{A(m)}_{id_{2i+1}}$, where o_1 is the object pointed to by v_1 and o_2 is the object pointed to by v_2: $o_1 = V_{id_{2i}}(v_1)$ and $o_2 = V_{id_{2i}}(v_2)$. By Invariant 4, there exists $n_1 \in L_{id_{2i}}(v_1)$ and $n_2 \in L_{id_{2i}}(v_2)$ such that $n_1\ \rho_{id_{2i}}\ o_1$ and $n_2\ \rho_{id_{2i}}\ o_2$. As the abstract semantics for STORE ensures $\langle n_1, f, n_2\rangle \in I_{id_{2i+1}}$, Invariant 5 holds at moment id_{2i+1}.

The case of a RETURN inside $A(m)$ is more complex. Still, the α transformation from the abstract semantics (see Figure 6-5) does not affect Invariant 5, because it is applied uniformly to the nodes from the inside edges and to the nodes from the modeling relation. Therefore, it suffices to do the proof for this case in the absence of α. With this assumption, if

$$\begin{aligned}
\Xi_{id_{2i}} &= \langle A_{id_{2i}}\,[t \mapsto \langle V_{id_{2i},1}, lb_1\rangle : \langle V_{id_{2i},2}, lb_2\rangle : K_{id_{2i}}]\,, H_{id_{2i}}, S_{id_{2i}}, TY_{id_{2i}}\rangle \\
G_{id_{2i}} &= \langle L_{id_{2i},1} : L_{id_{2i},2} : J_{id_{2i}},\ I_{id_{2i}},\ O_{id_{2i}},\ E_{id_{2i}},\ R_{id_{2i}},\ W_{id_{2i}}\rangle
\end{aligned}$$

then

$$\begin{aligned}
\Xi_{id_{2i+1}} &= \langle A_{id_{2i}}\,[t \mapsto \langle V_{id_{2i},2}\,[v_R \mapsto V_{id_{2i},1}(v)], next(lb_2)\rangle : K_{id_{2i}}]\,, \\
&\qquad\qquad H_{id_{2i}}, S_{id_{2i}}, TY_{id_{2i}}\rangle \\
G_a &= \langle L_{id_{2i},2}\,[v_R \mapsto L_{id_{2i},1}(v)] : J_{id_{2i}},\ I_{id_{2i}},\ O_{id_{2i}},\ E_{id_{2i}},\ R_{id_{2i}},\ W_{id_{2i}}\rangle \\
G_{id_{2i+1}} &= gc(G_a) \\
\rho_{id_{2i+1}} &= \rho_{id_{2i}}
\end{aligned}$$

The only potential problem is that the garbage collection may eliminate an inside edge that models a heap reference between two reachable objects. We prove that nodes that model reachable objects are not garbage collected. Therefore, all important inside edges "survive" the garbage collection and a RETURN inside $A(m)$ preserves Invariant 5. We first prove the following auxiliary lemma:

Lemma 12. *Consider a path of heap references from $H_{id_{2i}}$: $\langle o_0, f_0, o_1\rangle$, $\langle o_1, f_1, o_2\rangle$, ... $\langle o_{l-1}, f_{l-1}, o_l\rangle \in H_{id_{2i}}$. Assume that (1) $n_l\ \rho_{id_{2i}}\ o_l$, and (2) n_l is captured in $G_{id_{2i}}$: $\neg e(G_{id_{2i}})(n_l)$. Then, there exists a path of inside edges from $I_{id_{2i}}$, $\langle n_0, f_0, n_1\rangle$, $\langle n_1, f_1, n_2\rangle$, ... $\langle n_{l-1}, f_{l-1}, n_l\rangle \in I_{id_{2i}}$, such that $\neg e(G_{id_{2i}})(n_0)$ and $n_0\ \rho_{id_{2i}}\ o_0$.*

Proof. As n_l is captured, by Invariant 6 at date id_{2i}, object o_l is captured, i.e. $\neg e_{id_{2i}}(o_l)$. Therefore, (1) by Invariant 3 at date id_{2i}, n_l is the only node that models o_l, and (2) by Lemma 11, $\langle o_{l-1}, f_{l-1}, o_l\rangle \in I^{A(m)}_{id_{2i}}$. Hence, by Invariant 5 at date id_{2i}, there exists n_{l-1} such that $\langle n_{l-1}, f_{l-1}, n_l\rangle \in I_{id_{2i}}$, $n_{l-1}\ \rho_{id_{2i}}\ o_{l-1}$, and n_{l-1} is captured in $G_{id_{2i}}$ (otherwise, n_l would escape). To prove the lemma, it suffices to use mathematical induction to repeat this reasoning $l-1$ times. □

Consider an arbitrary node n and an arbitrary object o such that o is reachable in $\Xi_{id_{2i+1}}$ and n models o in $\rho_{id_{2i+1}} = \rho_{id_{2i}}$ (i.e., $n\ \rho_{id_{2i}}\ o$). We prove that n is reachable in G_a. Assume for the sake of contradiction that n is unreachable. Therefore, n is captured in G_a. As G_a and $G_{id_{2i}}$ are identical except for the state of local variables, n is captured in $G_{id_{2i}}$ too. As o is reachable in $\Xi_{id_{2i+1}}$, there exists a path in $H_{id_{2i+1}} = H_{id_{2i}}$, that reaches o from an object o_0 that is pointed to (in $\Xi_{id_{2i+1}}$) by a local variable or by a static field. By Lemma 12 above, there exists a path in $I_{id_{2i}}$ that reaches n from a node n_0 such that n_0 is captured in $G_{id_{2i}}$ (and hence in G_a too) and $n_0\ \rho_{id_{2i}}\ o_0$. By Invariant 6 at date id_{2i}, $\neg e_{id_{2i}}(o_0)$ and, by Invariant 3 at date id_{2i}, n_0 is the only node that models o in $\rho_{id_{2i}}$.

$\neg e_{id_{2i}}(o_0)$ implies $\neg e_{id_{2i+1}}(o_0)$ (a RETURN inside $A(m)$ does not escape objects). Hence, by Lemma 5, o_0 cannot be pointed to in $\Xi_{id_{2i+1}}$ by a static field or by a local variable from a stack frame outside $A(m)$. It remains that o_0 is pointed to by a local variable v from one of $A(m)$'s stack frames in $\Xi_{id_{2i+1}}$. By Invariant 4 at date id_{2i+1} (already proved above), in $G_{id_{2i+1}}$, v points to n_0, the only node that models o. Hence, v points to n_0 in G_a too.[4] Hence, in G_a, n is reachable along a path of inside edges that starts in n_0, a node pointed to by a local variable. Therefore, n is not garbage-collected. This completes the proof of Invariant 5.

Invariant 6: Recall that the escape predicate $e(G)(n)$ (Definition 2 on page 41) checks whether in the points-to graph $G = \langle L{:}J, I, O, E, R, W\rangle$, the node n is reachable along an *escaping path*: a path of inside edges, that starts in one of the *escapability sources* (the nodes from $CPNode \cup CLNode \cup \mathcal{G} \cup E \cup R$). Similarly, the concrete escape predicate $e_d(o)$ (Definition 14 on page 94) checks whether the object o is reachable along edges from $I_d^{A(m)}$ (heap references created by $A(m)$), from one of the objects directly escaped by one of the constraints 6.5, 6.7, 6.8, 6.9, and 6.10.

As usual, we perform a case analysis on the instruction executed at date id_{2i}. The following fact allows us to study only the changes to the modeling relation, the concrete escape predicates, and the points-to graphs:

Fact 13. *If the instruction executed at date id_{2i} is not a RETURN inside $A(m)$, $G_{id_{2i+1}}$ contains all escapability sources and inside edges from $G_{id_{2i}}$. Therefore, a node that escapes in $G_{id_{2i}}$ continues to escape in $G_{id_{2i+1}}$: $\forall n.\ e(G_{id_{2i}})(n) \rightarrow e(G_{id_{2i+1}})(n)$*

COPY, NULLIFY, IF, and analyzable CALL instructions do not escape objects and do not add new heap references. Therefore, $e_{id_{2i+1}}(o)$ iff $e_{id_{2i}}(o)$. In the abstract semantics, they preserve all escapability sources and inside edges. As a result, all nodes that escaped at date id_{2i} still escape at date id_{2i+1}. As these instructions do not modify the modeling relation, they preserve Invariant 6.

For a RETURN inside $A(m)$, $e_{id_{2i}} = e_{id_{2i+1}}$ too. For such a statement, the abstract semantics proceeds in three steps (see Figure 6-5 on page 96). Each step preserves Invariant 6: (1) The caller's variable v_R[5] is set to point to the nodes returned from the caller; this step is similar to a COPY instruction. (2) Unreachable nodes are garbage collected; still, none of the nodes appearing on an escaping path is unreachable (all unreachable nodes are captured; see definition of gc in Figure 4-10). (3) The same α_{c-1} transformation is applied to all nodes from the points-to graph and the modeling relation. Step (3) preserves the invariant: pick o and n such that $e_{id_{2i}}(o)$ and n models o after the α_{c-1} transformation. Therefore, $n = \alpha_{c-1}(n_2)$, and n_2 models o in $\rho_{id_{2i}}$. By Invariant 6 at date id_{2i}, n_2 escapes in $G_{id_{2i}}$. Finally, an escaping path in a points-to graph is projected by α_{c-1} into an escaping path into the projected graph; hence, n escapes in $G_{id_{2i+1}}$.

A NEW instruction updates the modeling relation to model the newly created object o. As o does not escape anywhere yet, Invariant 6 remains true. A LOAD or

[4] $G_{id_{2i+1}}$ is constructed by eliminating some nodes from G_a.

[5] This variable is specified by the matching CALL: "$v_R = v_0.s(v_1, \ldots, v_j)$"

a STATIC LOAD instruction may introduce a load node $n^L_{lb,c}$ to model the loaded object. As $e(G_{id_{2i+1}})(n^L_{lb,c})$ (as any other load node, see Def. 2 on page 41), this instruction preserves Invariant 6.

The final RETURN from $A(m)$, THREAD START, STATIC STORE, and unanalyzable CALL instructions escape objects, due to one of the constraints 6.10, 6.7, 6.8, and 6.9 from Definition 14. This escape information propagates along the heap references by Constraint 6.6. We treat only the case of an unanalyzable CALL; the other cases are similar.

We pick arbitrary o and n such that $e_{id_{2i+1}}(o)$ and $n\ \rho_{id_{2i+1}}\ o$ and prove that $e(G_{id_{2i+1}})(n)$ holds. The modeling relation $\rho_{id_{2i+1}}$ is identical to $\rho_{id_{2i}}$, except that $n_{\texttt{GBL},c}$ models the object returned from the unanalyzable CALL. If $n = n_{\texttt{GBL},c}$, then n escapes trivially in $G_{id_{2i+1}}$. Otherwise, n models o even at date id_{2i}. If $e_{id_{2i}}(o)$, Invariant 6 at date id_{2i} and Fact 13 imply $e(G_{id_{2i+1}})(n)$. If $\neg e_{id_{2i}}(o)$, o escapes at date id_{2i+1} only because it is reachable from an object that directly escapes into the unanalyzable CALL by Constraint 6.9 on page 95, by a path of edges from $I^{A(m)}_{id_{2i+1}} = I^{A(m)}_{id_{2i}}$.

Hence, there is a (possibly empty) path, $\langle o_0, f_0, o_1\rangle$, $\langle o_1, f_1, o_2\rangle$, $\ldots$ $\langle o_{p-1}, f_0, o_p\rangle \in I^{A(m)}_{id_{2i}}$, that reaches o from an object $o_0 = V_{id_{2i}}(v)$, where v is an argument of the unanalyzable CALL. As $\neg e_{id_{2i}}(o)$ and escapability propagates along the edges from $I^{A(m)}_{id_{2i}}$, $\forall j$, $\neg e_{id_{2i}}(o_j)$. By Invariant 3 at date id_{2i}, each object o_j is modeled by at most one node. Hence, by Invariant 5 at date id_{2i}, there exists a corresponding path along the inside edges from $G_{id_{2i}}$: $\langle n_0, f_0, n_1\rangle$, $\langle n_1, f_1, n_2\rangle$, $\ldots$ $\langle n_{p-1}, f_1, n_p\rangle \in I_{id_{2i}} \subseteq I_{id_{2i+1}}$, where $n_p = n$.

By Invariant 4 at date id_{2i}, in $G_{id_{2i}}$, v points to n_0, the unique node that models o_0, i.e., $n_0 \in L_{id_{2i}}(v)$, which implies $n_0 \in E_{id_{2i+1}}$ (by the definition of the abstract transfer function for an unanalyzable CALL, from Figure 4-3 on page 47). Hence, n is reachable from a node from $E_{id_{2i+1}}$, along a path of inside edges from $I_{id_{2i+1}}$. Therefore, $e(G_{id_{2i+1}})(n)$ holds in this case too.

The only remaining case is that of a STORE instruction "$v_1.f = v_2$." This instruction does not create new objects and does not change the modeling relation. However, it creates a new heap reference, which can generate new paths in $I^{A(m)}_{id_{2i+1}}$, causing more objects to escape.

As before, we pick arbitrary o and n such that $e_{id_{2i+1}}(o)$ and $n\ \rho_{id_{2i+1}}\ o$, and prove that $e(G_{id_{2i+1}})(n)$ holds. As STORE does not change the modeling relation, $n\ \rho_{id_{2i}}\ o$ too. If $e_{id_{2i}}(o)$, Invariant 6 at date id_{2i} and Fact 13 imply $e(G_{id_{2i+1}})(n)$. If $\neg e_{id_{2i}}(o)$, then o escapes at date id_{2i+1} only because the heap reference introduced by the STORE instruction made o reachable from an object that already escaped at date id_{2i}.

Consider one of the *shortest* paths in $I^{A(m)}_{id_{2i+1}}$ that reaches o from an object o_0 such that $e_{id_{2i}}(o_0)$. Let the objects from this path be $o_0, o_1, \ldots, o_p = o$. Due to the way we selected this path, o_0 is the only object from this path that escapes at date id_{2i}. By applying the same technique as in the case of an unanalyzable CALL above, we

construct a corresponding path in $I_{id_{2i+1}}$ from the unique node which models o_1, say n_1, to the unique node which models o, the node n.

As o_1 does not escape at date id_{2i}, the heap reference between o_0 and o_1 is not present in the set $I^{A(m)}_{id_{2i}}$. But it appears in the set $I^{A(m)}_{id_{2i+1}}$, which means that the STORE instruction created it, i.e., $V_{id_{2i}}(v_1) = o_0$, and $V_{id_{2i}}(v_2) = o_1$. By Invariant 4 at date id_{2i}, $L_{id_{2i}}(v_1)$ contains one of the nodes that models o_0, call it n_0, and $L_{id_{2i}}(v_2)$ contains the unique node n_1 that models o_1. Hence, at date id_{2i+1}, there exists a path of inside edges from n_0 to n. Invariant 6 at date id_{2i} and Fact 13 imply $e(G_{id_{2i+1}})(n_0)$; hence, $e(G_{id_{2i+1}})(n)$ in this case too.

In all cases Invariant 6 is valid at date id_{2i+1}.

Invariant 7: NEW and LOAD instructions may introduce new nodes, but the context of this nodes is $|J_{id_{2i}}| = |J_{id_{2i+1}}|$, preserving the invariant. A RETURN inside $A(m)$ decreases the height of the stack: $|J_{id_{2i+1}}| = |J_{id_{2i}}| - 1$. However, the node morphism $\alpha_{|J_{id_{2i}}|-1}$ "truncates" all node contexts to $|J_{id_{2i}}| - 1$, preserving the invariant. The other instruction are irrelevant, as they can at most increase the stack height.

Invariant 8: Consider the case of a STORE instruction "$v_1.f = v_2$" and assume that in the concrete state, v_1 points to the object $o \notin Allocs^{A(m)}_{\text{all}}$ (the case when o is allocated by $A(m)$ is irrelevant for Invariant 8). By Invariant 4, there exists a node $n \in L_{id_{2i}}(v)$ such that $n\ \rho_{id_{2i}}\ o$. Notice that n cannot be an inside node: by Invariant 2 at date id_{2i}, in $\rho_{id_{2i}}$, an inside node models only object allocated by $A(m)$. Hence, by the updated definition of the transfer function for a STORE instruction (see Section 5.2), $\langle n, f\rangle \in W_{id_{2i+1}}$. Hence, in this case, Invariant 8 is valid at date id_{2i+1}.

In the case of a RETURN inside $A(m)$, notice first that the garbage collection function *gc* is irrelevant because it removes only inside nodes (other nodes are trivially reachable according to the definition from Figure 4-10), and $W_{id_{2i}}$ does not use any inside node. Also notice that the abstract semantics applies the morphism α_{c-1} (where $c = |J_{id_{2i}}|$) uniformly to the nodes from the modeling relation $\rho_{id_{2i}}$ and the nodes from $W_{id_{2i}}$. Hence, Invariant 8 is valid at date id_{2i+1} in this case too.

The cases of the other instructions are trivial, as they do not update the set of mutated locations and the set of mutated abstract fields.

This completes the inductive proof of Invariants 1-8.

B.3 Monotonicity Lemmas

This section proves the monotonicity of the analysis transfer functions (Lemma 14) and the monotonicity of the function *interproc* (Lemma 16). We used these results in the proof of Theorem 8 on page 102.

Lemma 14. *If the instruction from label lb is not an analyzable CALL, then the analysis transfer function* $[\![lb]\!]^a$ *is monotonic.*

Proof. Case analysis on the instruction from label lb, followed by an inspection of the definitions from Figure 4-3. The case of a LOAD instruction is the only non-trivial case.

The analysis transfer function for a LOAD instruction uses the auxiliary function *process_load* (Figure 4-5) that, itself, uses the predicate e_2. As e_2 is defined as a reachability predicate (Definition 3), e_2 is monotonic too (i.e., there are more escaped nodes in a bigger analysis points-to graph). Hence, $[\![lb]\!]$ is monotonic in the case of a LOAD instruction too. □

Before proving the monotonicity of *interproc*, we prove an auxiliary result, the monotonicity of $gc \circ \rho$:

Lemma 15. *Consider* $G_1, G_2 \in PTGraph^a$*, such that* $G_1 \sqsubseteq G_2$*. Then,* $gc(\rho(G_1))$*,* $gc(\rho(G_2)) \in PTGraph^a$ *and* $gc(\rho(G_1)) \sqsubseteq gc(\rho(G_2))$*.*

Proof. The function $gc \circ \rho$ does not add new nodes, and does not change the stack height. Therefore, $gc(\rho(G_1))$ and $gc(\rho(G_2))$ have only 0-context nodes and their abstract stacks have a single element. Hence, $gc(\rho(G_1))$, $gc(\rho(G_2)) \in PTGraph^a$ and we can compare them.

Intuitively, for each analysis points-to graph $G \in PTGraph^a$, $gc(\rho(G))$ consists of those parts of G that escape (see Figure 4-10 on page 57). As $G_1 \sqsubseteq G_2$, any path that exists in G_1 exists in G_2 too. Hence, any node that escapes in G_1 escapes in G_2 too. This observation proves that $gc(\rho(G_1)) \sqsubseteq gc(\rho(G_2))$. □

Lemma 16. *The function interproc (Figure 4-6 on page 53) is monotonic in its first two arguments.*

Proof. Let lb be the label of an analyzable CALL instruction.

First, we consider $G_1, G_2, G_{callee} \in PTGraph^a$ such that $G_1 \sqsubseteq G_2$, and prove that

$$interproc(G_1,\ G_{callee},\ P(lb)) \sqsubseteq interproc(G_2,\ G_{callee},\ P(lb))$$

By an examination of the *interproc* definition from Figure 4-6, we notice that it is sufficient to prove that $T_{callee} = mct(\tau(gc(\rho(G_{callee}))))$ is monotonic. By Lemma 3 on page 59, each atomic inter-procedural transformer $T \in \mathcal{AT}(\tau(gc(\rho(G_{callee}))))$ is monotonic. Hence $F = \bigsqcup \mathcal{AT}(\tau(gc(\rho(G_{callee}))))$ is monotonic too, and so is each $F^i, i \geq 0$. Therefore, $T_{callee} = \bigsqcup_{i \geq 0} F^i$ is monotonic.

Second, we consider $G, G_{callee,1}, G_{callee,2} \in PTGraph^a$ such that $G_{callee,1} \sqsubseteq G_{callee,2}$, and prove that

$$interproc(G,\ G_{callee,1},\ P(lb)) \sqsubseteq interproc(G,\ G_{callee,2},\ P(lb))$$

It is sufficient to prove that $T_{callee,1} \sqsubseteq T_{callee,2}$, where $T_{callee,k} = mct(\tau(gc(\rho(G_{callee,k}))))$, $\forall k \in \{1,2\}$. $\forall k \in \{1,2\}$, $T_{callee,k} = \texttt{star}(F_k) = \bigsqcup_{i \geq 0} F_k^i$, where $F_k = \bigsqcup \mathcal{AT}(\tau(gc(\rho(G_{callee,k}))))$. By Lemma 15, $gc(\rho(G_{callee,1})) \sqsubseteq$

$gc(\rho(G_{callee,2}))$. Hence, $\mathcal{AT}(\tau(gc(\rho(G_{callee,1})))) \subseteq \mathcal{AT}(\tau(gc(\rho(G_{callee,2}))))$, which implies $F_1 \sqsubseteq F_2$. Therefore, $\forall i \geq 0$, $F_1^i \sqsubseteq F_2^i$, and, ultimately, $T_{callee,1} \sqsubseteq T_{callee,2}$. $\square$

B.4 Proof of Equation 6.13 on page 103

We first prove several auxiliary results in Section B.4.1. We use these results extensively during the proof of Equation 6.13 in Section B.4.2.

B.4.1 Auxiliary Results

Lemma 17. *Consider a semi-join lattice L of finite depth (i.e., L does not have any infinite strictly increasing chains), and $F : L \to L$ an extensive function over L. Let $\texttt{star}(F) = \bigsqcup_{i\geq 0} F^i$. Then,*

1. *$\texttt{star}(F)$ is well-defined, i.e., $\forall x \in L$, $\texttt{star}(F)(x) = \bigsqcup_{i\geq 0} F^i(x)$ exists. Moreover, $\forall x \in L$, $\exists k_x \in \mathbb{N}$ such that $\texttt{star}(F)(x) = F^k(x)$.*
2. *$\forall x \in L$, $\texttt{star}(F)(x)$ is a fixed-point of F. Moreover, if F is monotonic, $\texttt{star}(F)(x)$ is the smallest fixed point of F that is bigger than x.*
3. *$\texttt{star}(F) \sqcup \texttt{star}(F) = \texttt{star}(F)$;*
4. *$\texttt{star}(F) \circ \texttt{star}(F) = \texttt{star}(F)$;*
5. *$\texttt{star}(\texttt{star}(F)) = \texttt{star}(F)$.*

Proof. 1: As F is extensive, $\forall x \in L$, $x \sqsubseteq F(x) \sqsubseteq F^2(x) \sqsubseteq \ldots \sqsubseteq F^i(x) \sqsubseteq \ldots$. As L has finite depth, this chain stabilizes after a finite number of steps, i.e., $\exists k_x$ such that $F^j(x) = F^{k_x}(x), \forall j \geq k_x$ (k_x, the number of steps until stabilization, depends on x). Therefore, $\texttt{star}(F)(x) = \bigsqcup_{i\geq 0} F^i(x) = F^{k_x}(x)$ and $\texttt{star}(F)$ is defined for any $x \in L$.

2: Consider an arbitrary $x \in L$; as we have just proved, $\exists k_x \in \mathbb{N}$ such that $\texttt{star}(F)(x) = F^{k_x}(x)$ and $F^j(x) = F^{k_x}(x), \forall j \geq k_x$. Hence, $F(\texttt{star}(F)(x)) = F^{k_x+1}(x) = F^{k_x}(x) = \texttt{star}(F)(x)$, i.e., $\texttt{star}(F)(x)$ is a fixed-point of F.

Assume F is monotonic. Obviously, $\texttt{star}(F)(x) = \bigsqcup_{i\geq 0} F^i(x) \sqsupseteq F^0(x) = x$. Let $y \sqsupseteq x$ be another fixed-point of F. As F is monotonic, $y = F(y) \sqsupseteq F(x)$. We can prove by induction on i that $y \sqsupseteq F^i(x)$; hence, $y \sqsupseteq \bigsqcup_{i\geq 0} F^i(x) = \texttt{star}(F)(x)$. Therefore, $\texttt{star}(F)(x)$ is the smallest fixed-point of F that is bigger than x.

3: $\texttt{star}(F) \sqcup \texttt{star}(F) = (\bigsqcup_{i\geq 0} F^i) \sqcup (\bigsqcup_{i\geq 0} F^i) = \bigsqcup_{i\geq 0}(F^i \sqcup F^i) = \bigsqcup_{i\geq 0} F^i = \texttt{star}(F)$.

4: Consider an arbitrary $x \in L$. As we have just proved, $\texttt{star}(F)(x)$ is a fixed-point of F. Hence, $(\texttt{star}(F) \circ \texttt{star}(F))(x) = \bigsqcup_{i\geq 0} F^i(\texttt{star}(F)(x)) = \bigsqcup_{i\geq 0} \texttt{star}(F)(x) = \texttt{star}(F)(x)$. As we picked x arbitrarily, $\texttt{star}(F) \circ \texttt{star}(F) = \texttt{star}(F)$

5: $\texttt{star}(\texttt{star}(F)) = \texttt{id} \sqcup \bigsqcup_{i\geq 1} (\texttt{star}(F))^i = \texttt{id} \sqcup \texttt{star}(F) = \texttt{star}(F)$, since, by Part 4 above, $(\texttt{star}(F))^i = \texttt{star}(F)$, $\forall i \geq 1$. $\square$

Lemma 18. *For each $G \in PTGraph$, each $T \in Trans(G)$ is extensive and monotonic.*

Proof. Structural induction on T. Lemma 3 on page 59 already proved that each atomic transformer is extensive and monotonic. For the compound transformers, the only non-trivial complex case is for $T = \texttt{star}(T_1) = \bigsqcup_{i\geq 0} T_1^i$. By the induction hypothesis, T_1 is extensive and monotonic. Therefore, we can prove that $\forall i \geq 0$, T_1^i is extensive and monotonic. Hence, T is extensive and monotonic itself. □

Lemma 19. *For each $G \in PTGraph$, $mct(G)$ is the biggest transformer from $Trans(G)$: $\forall T \in Trans(G)$, $T \sqsubseteq mct(G)$.*

Proof. Structural induction on T, using the definition $mct(G) = \texttt{star}(F)$, where $F = \bigsqcup \mathcal{AT}(G)$ (extensive according to Lemma 18). The case $T = \texttt{id}$ is trivial, as $mct(G)$ is extensive (Lemma 18). The case of atomic transformers $T \in \mathcal{AT}(G)$ is also simple: $T \sqsubseteq \bigsqcup \mathcal{AT}(G) = F \sqsubseteq \bigsqcup_{i\geq 0} F^i = mct(G)$. For the compound transformers, we use Lemma 17. E.g., if $T = T_1 \sqcup T_2$, then by the structural induction hypothesis and Part 3 of Lemma 17, $T = T_1 \sqcup T_2 \sqsubseteq \texttt{star}(F) \sqcup \texttt{star}(F) = \texttt{star}(F) = mct(G)$. □

Lemma 20. *Consider $\mu_0 \in Map$, $I_0 \in IEdges$, $O_0 \in OEdges$, $E_0 \in \mathcal{P}(CNode)$, $G = \langle J, I, O, E, R, W\rangle \in PTGraph$, and $T \in Trans(G)$. Let*

$$\langle I', O', E', \mu'\rangle = T(I_0, O_0, E_0, \mu_0)$$

Then, each new edge from I' is the μ'-projection of some edge from I:

$$\forall \langle n_1, f, n_2\rangle \in I' \setminus I_0.\ \ \exists \langle n_3, f, n_4\rangle \in I.\ \ n_1 \in \mu'(n_3)\ \wedge\ n_2 \in \mu'(n_4)$$

Proof. Structural induction on T:

$T = \texttt{id}$ or $T = \texttt{load}(n_1, f, n_2)$**:** Trivial, as $I' \setminus I_0 = \emptyset$.

$T = \texttt{store}(n_1, f, n_2)$, where $\langle n_1, f, n_2\rangle \in I$**:** Notice that $\mu' = \mu_0$ and that $I' \setminus I_0 \subseteq \mu_0(n_1) \times \{f\} \times \mu_0(n_2)$.

$T = T_1 \sqcup T_2$: Let $\langle I_i', O_i', E_i', \mu_i'\rangle = T_i(I_0, O_0, E_0, \mu_0)$, where $i \in \{1, 2\}$. Obviously, $I' \setminus I_0 = (I_1' \setminus I_0) \cup (I_2' \setminus I_0)$. Consider $\langle n_1, f, n_2\rangle \in I_1' \setminus I_0$. By the induction hypothesis for T_1, there exists $\langle n_3, f, n_4\rangle \in I$ such that $n_1 \in \mu_1'(n_3) \subseteq \mu'(n_2)$, and $n_2 \in \mu_1(n_4) \subseteq \mu(n_4)$. The case $\langle n_1, f, n_2\rangle \in I_2' \setminus I_0$ is similar.

$T = T_1 \circ T_2$: Let $\langle I_m, O_m, E_m, \mu_m\rangle = T_2(I_0, O_0, E_0, \mu_0)$. As all transformers are extensive (Lemma 18), $\mu_0 \subseteq \mu_m \subseteq \mu'$ and $I_0 \subseteq I_m \subseteq I'$. Hence, $I' \setminus I_0 = (I' \setminus I_m) \cup (I_m \setminus I_0)$. Consider $\langle n_1, f, n_2\rangle \in I_m \setminus I_0$. By the induction hypothesis for T_2, there exists $\langle n_3, f, n_4\rangle \in I$ such that $n_1 \in \mu_m(n_3) \subseteq \mu'(n_3)$, and $n_2 \in \mu_m(n_4) \subseteq \mu'(n_4)$. The case $\langle n_1, f, n_2\rangle \in I' \setminus I_m$ is a direct application of the induction hypothesis for T_1.

$T = \texttt{star}(T_1)$**:** Let $\langle I_i, O_i, E_i, \mu_i\rangle = T_1^i(I_0, O_0, E_0, \mu_0), \forall i \geq 0$. By the definition of $\texttt{star}(T_1)$, $\langle I', O', E', \mu'\rangle = \bigsqcup_{i\geq 0}\langle I_i, O_i, E_i, \mu_i\rangle$. Consider $\langle n_1, f, n_2\rangle \in I' \setminus I_0$. As

$\langle n_1, f, n_2\rangle \in I' = \bigcup_{i\geq 0} I_i$, $\exists k \geq 0$ such that $\langle n_1, f, n_2\rangle \in I_k \setminus I_0$. As $T_1^k = T_1 \circ T_1 \circ \ldots \circ T_1$ (k times), we can use the same technique as in the case of $T = T_1 \circ T_2$ above.

□

Definition 15 (NICE **property).** *Given $G = \langle J, I, O, E, R, W\rangle \in PTGraph$, $I_0 \in IEdges$, $E_0 \in \mathcal{P}(CNode)$, and $\mu_0 \in Map$, property* NICE(G, I_0, E_0, μ_0) *holds iff*

1. *Only parameter and load nodes may have non-reflexive mappings in μ_0:*
   $\forall n_1, n_2.\ \ n_2 \in \mu_0(n_1)\ \wedge\ n_1 \neq n_2\ \rightarrow\ n_1 \in CPNode \cup CLNode$
2. *The only nodes from G that may appear as targets of non-reflexive mappings in μ_0 are those nodes that escape according to $\langle E, I\rangle$:*
   $\forall n_1, n_2.\ \ n_2 \in nodes(G)\ \wedge\ n_2 \in \mu_0(n_1)\ \wedge\ n_1 \neq n_2\ \rightarrow\ e_2(E, I)(n_2)$
3. *Any edge from I_0 that uses at least one node from G is the μ_0-projection of an edge from I:*
   $\forall \langle n_1, f, n_2\rangle \in I_0.\ \{n_1, n_2\} \cap nodes(G) \neq \emptyset\ \rightarrow$
   $\exists n_3, n_4.\ \langle n_3, f, n_4\rangle \in I\ \wedge\ n_1 \in \mu_0(n_3)\ \wedge\ n_2 \in \mu_0(n_4)$
4. *Any node from E_0 that appears in G is the μ_0-projection of a node from E:*
   $\forall n \in E_0 \cap nodes(G).\ \ \exists n_2 \in E.\ n \in \mu_0(n_2)$

Lemma 21. *Consider $G = \langle J, I, O, E, R, W\rangle \in PTGraph$, $I_0 \in IEdges$, $E_0 \in \mathcal{P}(CNode)$, $\mu_0 \in Map$, such that* NICE(G, I_0, E_0, μ_0) *holds. Let n_1 be a node from G such that $\neg e_2(E, I)(n_1)$. If n_1 is reachable from a node n_2 along edges from I_0, then n_2 is a node from G, $\neg e_2(E, I)(n_2)$, and n_1 is reachable from n_2 along edges from I:*

$$\forall n_1, n_2.\ \ n_1 \in nodes(G)\ \wedge\ \neg e_2(E, I)(n_1)\ \wedge\ reachable(\{n_2\}, I_0)(n_1)\ \rightarrow$$
$$n_2 \in nodes(G)\ \wedge\ \neg e_2(E, I)(n_2)\ \wedge\ reachable(\{n_2\}, I)(n_1)$$

Proof. Induction on the distance between n_2 and n_1. We prove that $\forall k \geq 0$, $\forall n_1, n_2$, if $n_1 \in nodes(G)$, $\neg e_2(E, I)(n_1)$, and n_1 is reachable from n_2 along a path of k edges from I_0, then n_1 is reachable from n_2 along a path of k edges from I and $n_2 \in nodes(G)$.

The base case $k = 0$, i.e., $n_2 = n_1$, is trivial. For the induction step, assume n_1 is reachable from n_2, along a path of $k + 1$ edges from I_0. Therefore, there exists a node n and a field f such that $\langle n, f, n_1\rangle \in I_0$ and n is reachable from n_2, using k edges from I_0. As $n_1 \in nodes(G)$ and NICE(G, I_0, E_0, μ_0) holds, there exists n_3, n_4 such that $\langle n_3, f, n_4\rangle \in I, n \in \mu_0(n_3), n_1 \in \mu_0(n_4)$. As $\neg e_2(E, I)(n_1)$, by Condition 2 of Definition 15, $n_4 = n_1$. Hence, $\neg e_2(E, I)(n_4)$, which implies $\neg e_2(E, I)(n_3)$. Therefore, n_3 is not a load or a parameter node. By Condition 1 of Definition 15, $n_3 = n$.

Hence, $n \in nodes(G)$, $\neg e(G)(n)$, and n is reachable from n_2, using k edges from I_0. By the induction hypothesis, $n_2 \in nodes(G)$, $\neg e_2(E, I)(n_2)$, and there exists a path of k edges from I from n_2 to n. We continue this path with the edge $\langle n_3, f, n_4\rangle = \langle n, f, n_1\rangle$, and prove that n_1 is reachable from n_2, along a path of $k + 1$ edges from I. □

Lemma 22. *Consider* $G = \langle J, I, O, E, R, W\rangle \in PTGraph$, $I_0 \in IEdges$, $E_0 \in \mathcal{P}(CNode)$, $\mu_0 \in Map$, *such that* $\text{NICE}(G, I_0, E_0, \mu_0)$ *holds. Consider a node* $n \in nodes(G)$ *such that* $e_2(E_0, I_0)(n)$. *Then* $e_2(E, I)(n)$ *holds.*

Proof. Assume for the sake of contradiction that $\neg e_2(E, I)(n)$. As $e_2(E_0, I_0)(n)$, there exists a path in I_0 that reaches n from a node $n_2 \in CPNode \cup CLNode \cup \mathcal{G} \cup E_0$. By Lemma 21, n is also reachable from n_2 along edges from I, $n_2 \in nodes(G)$, and $\neg e_2(E, I)(n_2)$. As $\neg e_2(E, I)(n_2)$, the only possibility is $n_2 \in E_0$. As $\text{NICE}(G, I_0, E_0, \mu_0)$ holds, by Condition 4 of Definition 15, $\exists n_3 \in E$ such that $n_2 \in \mu_0(n_3)$. If $n_2 = n_3$, then $e_2(E, I)(n_2)$. Otherwise, as $\text{NICE}(G, I_0, E_0, \mu_0)$ holds, by Condition 2 of Definition 15, $e_2(E, I)(n_2)$ again. In both cases, $e_2(E, I)(n_2)$. Contradiction! □

Definition 16. *A points-to graph* $G = \langle J, I, O, E, R, W\rangle$ has well-formed outside edges *iff any outside edge from* O *starts in a node that escapes according to* $\langle E, I\rangle$, *i.e.,*

$$\forall n_1, f, n_2.\ \langle n_1, f, n_2\rangle \in O \ \rightarrow\ e_2(E, I)(n_1)$$

Lemma 23 (Preservation of the NICE **property).** *Consider* $G \in PTGraph$, $\mu_1 \in Map$, $I_1 \in IEdges$, $O_1 \in OEdges$, $E_1 \in \mathcal{P}(CNode)$, *and* $T \in Trans(G)$, *such that* G *has well-formed outside edges and* $\text{NICE}(G, I_1, E_1, \mu_1)$ *holds. If* $\langle I_2, O_2, E_2, \mu_2\rangle = T(I_1, O_1, E_1, \mu_1)$, *then* $\text{NICE}(G, I_2, E_2, \mu_2)$ *holds.*

Proof. Structural induction on T.

$T = \texttt{id}$**:** Trivial.

$T = T_1 \circ T_2$**:** We apply the induction hypothesis twice, for T_2 and T_1.

$T = T_1 \sqcup T_2$**:** We apply the induction hypothesis for T_1 and T_2 and notice that $\text{NICE}(G, I'_{2,1}, E'_{2,1}, \mu_{2,1})$ and $\text{NICE}(G, I'_{2,2}, E'_{2,2}, \mu_{2,2})$ imply $\text{NICE}(G, I'_{2,1} \sqcup I'_{2,2}, E'_{2,1} \sqcup E'_{2,2}, \mu_{2,1} \sqcup \mu_{2,2})$.

$T = \texttt{star}(T_1)$**:** We first use the same technique as for the case $\circ$ above to prove that $\forall i \geq 0$, T_1^i satisfies the lemma, and next use the same technique as for the $\sqcup$ case.

$T = \texttt{store}(n_1, f, n_2)$**:** This transformer introduces new inside edges that are μ_1-projections of the edge $\langle n_1, f, n_2\rangle \in I$; also, $\mu_1 \subseteq \mu_2$.

$T = \texttt{gesc}(n)$**:** This transformer extends E_1 with the μ_1-projections of the node $n \in E$; also, $\mu_1 \subseteq \mu_2$.

$T = \texttt{load}(n, f, n^L)$**:** This transformer may create new non-reflexive mappings if $\exists\langle n_1, f, n_2\rangle \in I_1$, such that $n_1 \in \mu_1(n)$. In this case, μ_2 contains (in addition to all non-reflexive mappings from μ_1), a mapping from the load node n^L to the node n_2. We assume $n_2 \in nodes(G)$ and $n_2 \neq n^L$, and prove $e_2(E, I)(n_2)$.

Suppose for the sake of contradiction that $\neg e_2(E, I)(n_2)$. As n_2 is a node from G and $\text{NICE}(G, I_1, E_1, \mu_1)$ holds, by Condition 3 from Definition 15, there exists an inside edge $\langle n_3, f, n_4\rangle \in I$, such that $n_1 \in \mu_1(n_3)$ and $n_2 \in \mu_1(n_4)$.

As $\text{NICE}(G, I_1, E_1, \mu_1)$ holds, $n_2 \in \mu_1(n_4)$, $n_2 \in \mathit{nodes}(G)$, and $\neg e_2(E, I)(n_2)$, by Condition 2 from Definition 15, $n_2 = n_4$. Hence, $\neg e_2(E, I)(n_4)$ and $\neg e_2(E, I)(n_3)$ (otherwise, escapability of n_3 would propagate to n_4). Therefore, n_3 is not a load or a parameter node. By Condition 1 of $\text{NICE}(G, I_1, E_1, \mu_1)$, $n_1 = n_3$.

To sum up, $n_3 \in \mu_1(n)$, $n_3 \in \mathit{nodes}(G)$, and $\neg e_2(E, I)(n_3)$. By Condition 2 of $\text{NICE}(G, I_1, E_1, \mu_1)$, $n = n_3$. As $T = \texttt{load}(n, f, n^L) \in \mathit{Trans}(G)$, $\langle n, f, n^L\rangle \in O$. Hence, $n_3 = n$ is the source of an outside edge in the well-formed points-to graph G and so, by Definition 16, $e_2(E, I)(n_3)$. Contradiction! Hence, $e(G)(n_2)$.

□

Lemma 24. *Consider a node morphism $\beta : \mathit{CNode} \to \mathit{CNode}$ such that*

1. $\forall n \in \mathit{CPNode} \cup \mathit{CLNode} \cup \mathcal{G}.\ \beta(n) \in \mathit{CPNode} \cup \mathit{CLNode} \cup \mathcal{G}$;
2. $\forall n \in \mathit{CLNode}.\ \beta(n) \in \mathit{CLNode}$.

When applied to a mathematical entity that uses nodes (e.g., a points-to graph, an inter-procedural transformer, etc.), β propagates inside it and changes each node n to $\beta(n)$. E.g.,

$$\beta(I) = \{\langle \beta(n_1), f, \beta(n_2)\rangle \mid \langle n_1, f, n_2\rangle \in I\}$$
$$\beta(T_1 \circ T_2) = \beta(T_1) \circ \beta(T_2)$$
$$\beta(\texttt{store}(n_1, f, n_2)) = \texttt{store}(\beta(n_1), f, \beta(n_2))$$

Assume $G \in \mathit{PTGraph}$ and $T \in \mathit{Trans}(G)$. Then

a. $\beta(T) \in \mathit{Trans}(\beta(G))$
b. $(\beta(T))\ (\beta(I, O, E, \mu)) \sqsupseteq \beta(T(I, O, E, \mu))$

Proof sketch. Structural induction on T. Condition 1 ensures that "escapability projects through β": i.e., $\forall E_2, I_2, n$, if $e_2(E_2, I_2)(n)$, then $e_2(\beta(E_2), \beta(I_2))(n)$ too. (see Definition 3 on page 42). Condition 2 is a technicality: it ensures that the β-projection of an outside edge is still a valid outside edge (an outside edge must terminate in a load node).

The function application $(\beta(T))(\beta(I, O, E, \mu))$ operates similarly with $T(I, O, E, \mu)$ except that everything is projected through β. E.g., if $T(I, O, E, \mu)$ adds an inside edge $\langle n_1, f, n_2\rangle$, then $(\beta(T))(\beta(I, O, E, \mu))$ adds the edge $\langle \beta(n_1), f, \beta(n_2)\rangle$.

The use of inequality in statement 2 (instead of equality) is due to the fact that β is not necessarily injective: it may "merge" nodes and thus link together previously unconnected paths. Therefore, reachability on the right-hand side may strictly supersede reachability on the left-hand side. As `load` transformers use reachability internally, they may generate bigger results.

This lemma would have been harder to prove, or even incorrect, for transformers that use inequality tests: in general, non-injective projections do not preserve inequalities. □

Lemma 25. *If $G \in PTGraph$, $T \in Trans(G)$, and $k \in \mathbb{N}$, then*

1. $\alpha_k(T) \in Trans(\alpha_k(G))$
2. $(\alpha_k(T))\ (\alpha_k(I, O, E, \mu)) \sqsupseteq \alpha_k(T(I, O, E, \mu))$

Proof. Simple corollary of Lemma 24: the node morphism α_k (Figure 4-11 on page 58) satisfies the two conditions of Lemma 24 because it changes only the context of a node, not its kind (e.g., α transforms a load node into a load node). □

During the presentation of the analysis, we introduced the del_S operator that takes a node-based structure (e.g., a points-to graph) and returns a similar structure, but without any of the nodes from the set S (Figure 4-10).

Figure B-1 extends the del_S operator to handle inter-procedural transformers. $del_S(T)$ is the inter-procedural transformer that is similar to T, except that inside T, each atomic transformer that uses a node from S is replaced with `id`, the identity transformer.

Lemma 26. *Consider $G = \langle J, I, O, E, R, W\rangle \in PTGraph$, $\mu_1 \in Map$, $I_1 \in IEdges$, $O_1 \in OEdges$, $E_1 \in \mathcal{P}(CNode)$, $T \in Trans(G)$, such that G has well-formed outside edges and* $\text{NICE}(G, I_1, E_1, \mu_1)$ *holds. Let S be a subset of the captured nodes from G, $S \subseteq \{n \in nodes(G) \mid \neg e(G)(n)\}$. Then,*

$$(del_S(T))(del_S(I_1, O_1, E_1, \mu_1)) = del_S(T(I_1, O_1, E_1, \mu_1))$$

Proof. Structural induction on T:

$T = \texttt{id}$**:** Trivial

$T = T_1 \circ T_2$**:** We apply the induction hypothesis twice, for T_2 and T_1.

$T = T_1 \sqcup T_2$**:** We apply the induction hypothesis twice, for T_1 and T_2, and notice that del_S distributes over $\sqcup$: e.g., $del_S(I_a) \sqcup del_S(I_b) = del_S(I_a \sqcup I_b), \forall I_a, I_b \in IEdges$.

$T = \texttt{star}(T_1)$**:** As the lemma holds for T_1, we can use the same technique as for the case $\circ$ above to prove by induction on i that $\forall i \geq 0$, T^i satisfies the lemma too. Next, we can use the same technique as for the case $\sqcup$ above to prove that T satisfies the lemma.

$T = \texttt{store}(n_1, f, n_2)$**:** It is sufficient to compare the sets of inside edges on both sides of the equality we want to prove (the other elements are trivially equal). The set of inside edges on the right-hand side is

$$\begin{aligned} del_S(I_1 \ \cup\ \mu_1(n_1) \times \{f\} \times \mu_1(n_2)) = \\ del_S(I_1) \ \cup\ del_S(\mu_1(n_1)) \times \{f\} \times del_S(\mu_1(n_2)) \end{aligned}$$

$$
\begin{aligned}
del_S(T_1 \circ T_2) &= del_S(T_1) \circ del_S(T_2) \\
del_S(T_1 \sqcup T_2) &= del_S(T_1) \sqcup del_S(T_2) \\
del_S(\texttt{star}(T)) &= \texttt{star}(del_S(T)) \\
del_S(\texttt{id}) &= \texttt{id} \\
del_S(\texttt{store}(n_1, f, n_2)) &= \begin{cases} \texttt{store}(n_1, f, n_2) & \text{if } \{n_1, n_2\} \cap S = \emptyset \\ \texttt{id} & \text{otherwise} \end{cases} \\
del_S(\texttt{load}(n, f, n^L)) &= \begin{cases} \texttt{load}(n, f, n^L) & \text{if } \{n, n^L\} \cap S = \emptyset \\ \texttt{id} & \text{otherwise} \end{cases} \\
del_S(\texttt{gesc}(n)) &= \begin{cases} \texttt{gesc}(n) & \text{if } n \notin S \\ \texttt{id} & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure B-1: Definition of the function del_S for predicate transformers. Given a set of nodes S and an inter-procedural transformer T, $del_S(T)$ is the inter-procedural transformer similar to T, except that inside T, each atomic transformer that uses a node from S is replaced with `id`, the identity transformer. Figure 4-10 on page 57 defines the function del_S for other node-based structures.

If $n_1 \in S$, then $del_S(\texttt{store}(n_1, f, n_2)) = \texttt{id}$, and the left-hand side set of inside edges is simply $del_S(I_1)$. Also, $n_1 \in S$ implies that n_1 is captured in G; by Condition 1 of the NICE(G, I_1, E_1, μ_1) property, n_1 does not map to any other node, so $del_S(\mu_1(n_1)) = \emptyset$, implying equality of the two sets of inside edges. A similar reasoning is possible if $n_2 \in S$.

Finally, if $\{n_1, n_2\} \notin S$, then the set of inside edges on the left-hand side is

$$del_S(I_1) \cup (del_S(\mu_1))(n_1) \times \{f\} \times (del_S(\mu_1))(n_2)$$

Consider n_1' such that $\langle n_1, n_1' \rangle \in \mu_1$. If $n_1' = n_1$, then $n_1' \notin S$. If $n_1' \neq n_1$, then n_1' is the target of a non-reflexive mapping. By Condition 2 of the property NICE(G, I_1, E_1, μ_1), n_1' escapes. In both cases, $n_1' \notin S$. Hence, $del_S(\mu_1(n_1)) = (del_S(\mu_1))(n_1) = \mu_1(n_1)$. A similar relation holds for n_2, implying the equality of the two sets of inside edges.

$T = \texttt{gesc}(n)$**:** As $T \in \mathit{Trans}(G)$, $n \in E$. Hence, $n \notin S$ and $del_S(\texttt{gesc}(n)) = \texttt{gesc}(n)$.

We need to prove that

$$\texttt{gesc}(n)(del_S(I_1),\ del_S(O_1),\ del_S(E_1),\ del_S(\mu_1)) = \\ del_S(\ \texttt{gesc}(n)(I_1,\ O_1,\ E_1,\ \mu_1)\ )$$

It is sufficient to prove the equality of the sets of globally escaped nodes on the two sides of the equality, i.e., $del_S(E_1) \cup (del_S(\mu_1))(n) = del_S(E_1) \cup del_S(\mu_1(n))$. As in the case of `store`, as $n \notin S$, we can prove that $del_S(\mu_1(n)) = (del_S(\mu_1))(n) = \mu(n)$, which ends the proof of this case.

$T = \texttt{load}(n, f, n^L)$**:** As $T \in \mathit{Trans}(G)$, n is the source of a well-formed outside edge from the graph G; hence, $e_2(E, I)(n)$; the load node n^L escapes trivially; hence, $n \notin S$ and $n^L \notin S$ (S contains only captured nodes) and $del_S(\texttt{load}(n, f, n^L)) = \texttt{load}(n, f, n^L)$. We need to prove that

$$\texttt{load}(n, f, n^L)(del_S(I_1),\ del_S(O_1),\ del_S(E_1),\ del_S(\mu_1)) = \\ del_S(\texttt{load}(n, f, n^L)(I_1,\ O_1,\ E_1,\ \mu_1))$$

`load` may extend only the map and the set of outside edges. We examine the map first. On the right-hand side above, `load` adds a mapping from n^L to n^L if $\exists n_1 \in \mu_1(n)$ such that $e_2(E_1, I_1)(n_1)$. As $n^L \notin S$, this mapping is not affected by the application of del_S on the right-hand side.

Observation: Each node n_x that escapes according to $\langle E_1, I_1 \rangle$ does not appear in S: assume for the sake of contradiction that $n_x \in S \subseteq \mathit{nodes}(G)$. Then, by Lemma 22, $e_2(E, I)(n_x)$, which contradicts $n_x \in S$ (S contains only captured nodes).

Therefore, $n_1 \notin S$, and hence, the mapping from n to n_1 exists in $del_S(\mu_1)$. As $e_2(E_1, I_1)(n_1)$, there exists a path in I_1 that reaches n_1 from a node in $\mathit{CPNode} \cup \mathit{CLNode} \cup \mathcal{G} \cup E_1$. As each node from this path escapes according to $\langle E_1, I_1 \rangle$, by the observation above, none of them is in S. Therefore, this path is not affected by the removal of the nodes from S. Hence, the mapping from n^L to n^L is introduced on the left-hand side too.

The `load` transformer on the right-hand side may also add new mappings from n^L to n_2, for any edge $\langle n_1, f, n_2 \rangle \in I_1$ such that $n_1 \in \mu(n)$. Notice that $n_1 \notin S$: otherwise, n_1 would be a captured node from G and by NICE(G, I_1, E_1, μ_1) and Condition 2 of Def. 15, $n = n_1 \in S$; contradiction! Also, $n_2 \notin S$: assume for the sake of contradiction that $n_2 \in S$. As any load node escapes, $n_2 \neq n^L$. As `load` preserves the NICE property (Lemma 23) and n_2 becomes the target of a non-reflexive mapping, $e_2(E, I)(n_2)$, which contradicts $n_2 \in S$. In conclusion, none of the nodes used by the `load` transformer (n, n^L, n_1, n_2) appear in S. Therefore, the mapping from n^L to n_2 appears on both sides of the desired equality.

The proof for the outside edges is similar: it relies on the fact that none of the involved nodes is deleted by del_S. For brevity, we omit the details.

□

Lemma 27. *If $G \in PTGraph$, $T \in Trans(G)$, and S is a set of nodes, then $del_S(T) \in Trans(del_S(G))$.*

Proof. Trivial by structural induction on T. $del_S(G)$ removes any edge that has an endpoint in S; accordingly, $del_S(T)$ converts into `id` any atomic transformer that involves a node from S, etc. □

Lemma 28. *Consider an activation $A(m)$ of a method m, and an arbitrary interesting date $d \in ID_{A(m)}$. Let $G_d = \langle J_d, I_d, O_d, R_d, W_d \rangle$ be the points-to graph that the abstract semantics of $A(m)$ constructs for d. If d is not the termination date for the final RETURN from $A(m)$, then, $R_d = \emptyset$.*

Proof. Induction on the interesting dates, followed by case analysis on the instruction executed in each step. The initial points-to graph $G_{id_0} = G^m_{\text{init}}$ has an empty set of returned nodes. The abstract semantics transfer functions propagate this empty set. The only exception is the transfer function for the final RETURN from $A(m)$. □

Lemma 29. *Let $G = \langle L : J, I, O, E, R, W \rangle \in PTGraph$. If $n \in garbage(G)$, then $\neg e_2(E, I)(n)$.*

Proof. Direct application of the definitions of *garbage* (Figure 4-10 on page 57) and e_2 (Definition 3 on page 42). □

Lemma 30. *Consider an activation $A(m)$ of a method m, and an arbitrary interesting date $d \in ID_{A(m)}$. The points-to graph G_d that the abstract semantics of $A(m)$ constructs for d has well-formed outside edges.*

Proof. Induction on the interesting dates, followed by case analysis on the instruction executed in each step. The only non-trivial cases are those for LOAD and RETURN inside $A(m)$. The transfer function for a LOAD instruction adds only outside edges that start in escaped nodes. Consider a RETURN inside $A(m)$. Consider an outside edge $\langle n_1, f, n_2 \rangle$ in the points-to graph before the RETURN. By the induction hypothesis, n_1 is reachable along an *escaping path*: a path of inside edges that starts in a parameter node, a load node, or in a globally escaped node (by Lemma 28, there are no returned nodes). The garbage collection from the transfer function for the RETURN instruction removes only captured nodes (Lemma 29). Hence, this operation does not affect the escaping path for n_1. The node morphism α from the transfer function for a RETURN inside $A(m)$ projects the escaping path into another escaping path. In the resulting points-to graph, each outside edge starts in an escaped node. □

Lemma 31. $\forall k \in \mathbb{N}$, $\tau \circ \alpha_k = \alpha_{k+1} \circ \tau$.

Proof. τ and $\alpha_{\cdot}$ are overloaded symbols that work on a variety of node-based structures. Still, it suffices to prove the desired equality for the functions τ and $\alpha_{\cdot}$ operating on nodes. Consider a node (with context) $\langle n, c \rangle \in CNode$. If $c \leq k$, then

$$\begin{array}{rcccl} \tau(\alpha_k(\langle n, c \rangle)) & = & \tau(\langle n, c \rangle) & = & \langle n, c+1 \rangle \\ \alpha_{k+1}(\tau(\langle n, c \rangle)) & = & \alpha_{k+1}(\langle n, c+1 \rangle) & = & \langle n, c+1 \rangle \end{array}$$

If $c > k$, then

$$\begin{array}{rcccl} \tau(\alpha_k(\langle n, c\rangle)) & = & \tau(\langle n, k\rangle) & = & \langle n, k+1\rangle \\ \alpha_{k+1}(\tau(\langle n, c\rangle)) & = & \alpha_{k+1}(\langle n, c+1\rangle) & = & \langle n, k+1\rangle \end{array}$$

In both cases, $(\tau \circ \alpha_k)\ (\langle n, c\rangle) = (\alpha_{k+1} \circ \tau)\ (\langle n, c\rangle)$ □

B.4.2 Proof of Equation 6.13 on page 103

Background: Recall from Section 6.6 that Equation 6.13 is formulated in the context of an activation $A(m)$ of a method m. $ip_{2i} \in IP_{A(m)}$ is an even-numbered intra-procedural date of $A(m)$. At date ip_{2i}, the activation $A(m)$ starts executing a CALL instruction $P(lb_{ip_{2i}}) =$ "$v_R = v_0.s(v_1, \ldots, v_m)$" that invokes the method *callee*.

This CALL starts an activation of *callee*, $A(callee)$. The last intra-procedural date of $A(callee)$ is ip_{2i+1}, the date when the matching RETURN terminates. $G^{callee}_{ip_{2i+1}}$ is the points-to graph that the abstract semantics of $A(callee)$ constructs for ip_{2i+1}. $G_{ip_{2i}}$ and $G_{ip_{2i+1}}$ are the points-to graphs that the abstract semantics of $A(m)$ computes for the date ip_{2i}, respectively for the date ip_{2i+1}. Equation 6.13 has the form

$$interproc(G_{ip_{2i}},\ G^{callee}_{ip_{2i+1}},\ P(lb_{ip_{2i}}))\ \sqsupseteq\ G_{ip_{2i+1}}$$

Proof Idea: The abstract semantics of $A(m)$ constructs $G_{ip_{2i+1}}$ using a small-step strategy: it starts with $G_{ip_{2i}}$ and next applies the transfer functions for the CALL instruction and for all the instructions from $A(callee)$. Instead, *interproc* uses one big step: the inter-procedural transformer $T_{callee} = mct(\tau(gc(\rho(G^{callee}_{ip_{2i+1}}))))$. Intuitively, Equation 6.13 states that T_{callee} conservatively approximates the small-step processing of the abstract semantics of $A(m)$ for the instructions from $A(callee)$.

During our proof, we incrementally construct an inter-procedural transformer for each date inside $A(callee)$. We prove by induction that each such transformer conservatively approximates the abstract semantics of $A(m)$ up to the corresponding date. We also prove that the inter-procedural transformer for the end of $A(callee)$ is smaller than T_{callee}. Each inter-procedural transformer is a small change over the previous inter-procedural transformer. This fact keeps each induction step reasonably simple.

Notation: Let lb_0, lb_1, ... lb_l be the labels of the statements that $A(m)$ executes from the CALL to *callee* until the matching RETURN (in order); lb_0 is the label of the CALL and lb_l is the label of the matching RETURN. Instructions from labels lb_1, lb_2, ... lb_l belong to both $A(m)$ and $A(callee)$.

We use the superscript 0 for the points-to graphs constructed by the abstract semantics of $A(m)$ by processing the instructions from labels lb_0, lb_1, ... lb_l. Let $G^0_0 = G_{ip_{2i}}$ and $G^0_{k+1} = [\![lb_k]\!](G^0_k)$, $\forall k \in \{0, 1, \ldots, l\}$.

We use the superscript 1 for the points-to graphs constructed by the abstract

Notation: $\mu(L) = \lambda v.\ \mu(L(v)) = \lambda v.\ \bigcup_{n \in L(v)} \mu(n)$
$\mu([L_1, L_2, \ldots L_q]) = [\mu(L_1), \mu(L_2), \ldots \mu(L_q)]$

Let $\mu_0 = \{\langle n, n\rangle \mid n \in \mathit{CINode} \cup \mathcal{G}\} \cup \left(\bigcup_{j=0}^{m} \{n_{i,1}^P\} \times L(v_j)\right)$, and
$\langle I_k^2,\ O_k^2,\ E_k^2,\ \mu_k\rangle = T_k\ (I_0^0,\ O_0^0,\ E_0^0,\ \mu_0)$ in the following invariants:

$$\langle I_k^2,\ O_k^2,\ E_k^2\rangle \ \sqsupseteq\ \langle I_k^0,\ O_k^0,\ E_k^0\rangle \tag{B.1}$$

$$|J_k^0| \ =\ |J_k^1| + 1 \tag{B.2}$$

$$\mu_i(\tau(L_k^1 : J_k^1)) \ @\ [L_0^0] \ \sqsupseteq\ L_k^0 : J_k^0 \tag{B.3}$$

$$W_0^0 \ \cup \bigcup_{\langle n, f\rangle \in \tau(W_k^1)} (\mu_k(n) \setminus \mathit{CINode}) \times \{f\} \ \supseteq\ W_k^0 \tag{B.4}$$

$$T_k \ \in\ \mathit{Trans}(\tau(G_k^1)) \tag{B.5}$$

Figure B-2: Invariants for the proof of Equation 6.13.

semantics of $A(\mathit{callee})$:

$$\begin{aligned} G_1^1 &= G_{\text{init}}^{\mathit{callee}} = \langle (L_{\text{all-empty}}\left[p_j \mapsto \{n_{j,0}^P\}\right]_{0 \le j \le m}) : [],\ \emptyset,\ \emptyset,\ \emptyset,\ \emptyset,\ \emptyset\rangle \\ G_{k+1}^1 &= [\![lb_k]\!](G_k^1),\quad \forall k \in \{1, 2, \ldots, l\} \end{aligned}$$

With these notations, $G_{ip_{2i+1}}^{\mathit{callee}} = G_{l+1}^1$ and Equation 6.13 is equivalent to

$$\mathit{interproc}(G_0^0,\ G_{l+1}^1,\ \text{“}v_R = v_0.s(v_1, \ldots, v_m)\text{”}) \ \sqsupseteq\ G_{l+1}^0$$

Unless otherwise specified, for each points-to graph G_k^x, we use the same superscript x and subscript k to denote its sub-components:

$$G_k^x = \langle L_k^x : J_k^x,\ I_k^x,\ O_k^x,\ E_k^x,\ R_k^x,\ W_k^x\rangle$$

This notation identifies the top abstract local variable state L_i^x, the rest of the stack J_i^x, the set of inside edges I_i^x, etc.

For each $k \in \{1, 2, \ldots l\}$, our proof constructs an inter-procedural transformer T_k that conservatively approximates the abstract semantics of $A(m)$ up to that point. More specifically, if μ_0 is the initial inter-procedural node map from *interproc*, and $\langle I_k^2,\ O_k^2,\ E_k^2,\ \mu_k\rangle = T_k\ (I_0^0,\ O_0^0,\ E_0^0,\ \mu_0)$, then the Invariants B.1-B.5 from Figure B-2 hold.

Invariant B.1 states that the transformer T_k conservatively approximates the abstract semantics of $A(m)$, with respect to the inside edges, outside edges and the set of globally escaped nodes.

Invariant B.3 states that the μ_k-projection of the abstract stack from $\tau(G_k^1)$ conservatively approximates the abstract stack $L_k^0 : J_k^0$ created by the abstract semantics of $A(m)$. The append (“@”) operation on the left-hand side of Invariant B.3 is due to

Instruction $P(lb_k)$	T_{k+1}
STORE $v_1.f = v_2$	$\left(\bigsqcup \left\{\texttt{store}(\tau(n_1), f, \tau(n_2)) \mid n_1 \in L_k^1(v_1),\ n_2 \in L_k^1(v_2)\right\}\right) \circ T_k$
STATIC STORE $C.f = v$	$\left(\bigsqcup \left\{\texttt{gesc}(\tau(n)) \mid n \in L_k^1(v)\right\}\right) \circ T_k$
Unanalyzable CALL $v_R = v_0.s(v_1, \ldots v_q)$	$\left(\bigsqcup \left\{\texttt{gesc}(\tau(n)) \mid n \in \bigcup_{j=0}^{q} L_k^1(v_j)\right\}\right) \circ T_k$
THREAD START `start` v	$\left(\bigsqcup \left\{\texttt{gesc}(\tau(n)) \mid n \in L_k^1(v)\right\}\right) \circ T_k$
LOAD $v_2 = v_1.f$	**let** $B = \{\ n \in L_k^1(v_1) \mid e_2(E_k^1, I_k^1)(n)\ \}$ **in** **if** $B = \emptyset$ **then** T_k **else** $\left(\bigsqcup \left\{\texttt{load}(\tau(n), f, \tau(n^L_{lb_k, \lvert J_k^1\rvert})) \mid n \in B\right\}\right) \circ T_k$
RETURN inside $A(callee)$ `return` v	Assume $G_k^1 = \langle L_{k,1}^1 : L_{k,2}^1 : J_k^1,\ I_k^1,\ O_k^1,\ E_k^1,\ \emptyset\rangle$, and let $G = \langle L_{k,2}^1\left[v_R \mapsto L_{k,1}^1(v)\right] : J_k^1,\ I_k^1,\ O_k^1,\ E_k^1,\ \emptyset\rangle$, where v_R is the variable that stores the result of the matching CALL. Then, $T_{k+1} = \alpha_{\lvert J_k^1\rvert+1}(del_{\tau(garbage(G))}(T_k))$
other cases	T_k (unchanged)

Figure B-3: Definition of the inter-procedural transformers T_k, $k \in \{1, 2, \ldots, l\}$. $T_1 = \texttt{id}$. The other transformers are defined inductively using the rules from this figure.

the fact that $L_k^0 : J_k^0$ contains the additional element L_0^0 that models the state of the local variables from the method m. Invariant B.2 ensures that Invariant B.3 compares abstract stacks of equal heights (using element-wise comparison).

Invariant B.4 states that the μ_k-projection of the set of mutated abstract fields from the callee approximates the set of mutated abstract fields constructed by the abstract semantics of $A(m)$.

Figure B-3 presents the definition of the transformers T_k, $k \in \{1, 2, \ldots l\}$. Initially, $T_1 = \texttt{id}$. The other transformers T_k are defined inductively, based on the instruction from label lb_k. On a first reading, one can skip the precise definitions from Figure B-3. We discuss these definitions during the proof below.

The rest of this section has the following structure: First, we prove the Invariants B.1-B.5. Next, we prove Equation 6.13 using the invariants and several additional facts.

Proof of Invariants B.1-B.5

We do a proof by induction on $k \in \{1, 2, \ldots, l\}$. Invariant B.2 is trivial to prove: J_k^0 has an extra element, L_0^0, that models the state of the local variables from the method m. We focus on the other invariants.

Base Case: $i = 1$, $T_1 = \texttt{id}$. Invariant B.5 is trivially true: $\texttt{id} \in \mathit{Trans}(\tau(G_1^1))$. Invariant B.1 is trivially true (with equality), and $\mu_1 = \mu_0$. To prove Invariant B.3, notice that

$$\tau(L_1^1) = L_{\text{all–empty}}\left[p_j \mapsto \{\tau(n_{j,0}^P)\}\right]_{0\le j\le m} = L_{\text{all–empty}}\left[p_j \mapsto \{n_{j,1}^P\}\right]_{0\le j\le m}$$
$$L_1^0 = L_{\text{all–empty}}\left[p_j \mapsto L_0^0(v_j)\right]_{0\le j\le m} \quad \text{(def. of } [\![.]\!] \text{ for a CALL in Figure 6-5)}$$

and that μ_0 maps each parameter node $n_{j,1}^P$ to $L_0^0(v_j)$. Invariant B.4 is trivially true: $W_1^0 = W_0^0$.

Induction Step: We assume that Invariants B.1-B.5 hold at moment k and prove them at moment $k+1$. An important tool in our proof is that the property $\text{NICE}(\tau(G_k^1), I_0^0, E_0^0, \mu_0)$ (see Definition 15 on page 171) holds at each step k, $1 \le k \le l$:

- Only parameter nodes have non-reflexive mappings in μ_0.
- By Lemma 6 on page 101, $G_0^0 = G_{ip_{2i}}$ is an analysis points-to graph. Hence, all nodes from G_0^0 have context 0. Hence, all targets of non-reflexive mappings in μ_0 are nodes with context 0, that do not appear in $\tau(G_k^1)$ ($\tau(G_k^1)$ contains only nodes with context at least 1).
- I_0^0 and E_0^0 do not contain any nodes from $\tau(G_k^1)$ (distinct node contexts). This fact is the main reason we use node contexts in our formalism.

Therefore, our proof can use Lemmas 23, 21, 22, and 26. In particular, by Lemma 23 (preservation of the NICE property),

Fact 32. $\text{NICE}(\tau(G_k^1), I_k^2, O_k^2, \mu_k)$ *holds.*

Fact 32 implies several other useful results:

Fact 33. *Only parameter and load nodes can be sources of non-reflexive mappings in* μ_k*:* $\forall n_1, n_2.\ n_2 \in \mu_k(n_1) \ \wedge\ n_1 \neq n_2 \ \rightarrow\ n_1 \in \mathit{CPNode} \cup \mathit{CLNode}$.

Fact 34. *The map* μ_k *maps each captured node from* $\tau(G_k^1)$ *only to itself:*
$\forall n_1, n_2.\ (n_1 \in \mathit{nodes}(\tau(G_k^1))) \ \wedge\ \neg e_2(\tau(E_k^1), \tau(I_k^1))(n_1) \ \wedge\ (n_2 \in \mu_k(n_1)) \ \rightarrow\ n_1 = n_2$.

Fact 35. *If n is a captured node from $\tau(G_k^1)$, then the only node that μ_k maps to n is n itself:* $\forall n_1, n_2.\ (n_1 \in nodes(\tau(G_k^1)))\ \wedge\ \neg e_2(\tau(E_k^1), \tau(I_k^1))(n_1)\ \wedge\ (n_1 \in \mu_k(n_2))\ \rightarrow\ n_1 = n_2$.

Fact 33 is just Condition 1 of the property NICE$(\tau(G_k^1), I_k^2, O_k^2, \mu_k)$. Fact 34 is an immediate implication of Fact 33: parameter and load nodes always escape. Fact 35 is just Condition 2 of the property NICE$(\tau(G_k^1), I_k^2, O_k^2, \mu_k)$.

We do a case analysis on the instruction from label lb_k. We start with a few cases when the abstract semantics of $A(m)$ and the abstract semantics of $A(callee)$ proceed similarly. In each case, we focus on the elements that change: local variables that point to new nodes, new node mappings, etc.

Consider the case when the instruction from label lb_k is a COPY instruction "$v_1 = v_2$". In this case, $T_{k+1} = T_k$; hence, $\mu_{k+1} = \mu_k$. As the abstract semantics of COPY does not change the inside/outside edges nor the set of globally escaped nodes, the validity of Invariant B.1 propagates from moment k to moment $k+1$. For Invariant B.3, notice that

$$L_{k+1}^1 = L_k^1\,[v_2 \mapsto L_k^1(v_1)] \quad \text{which implies } \tau(L_{k+1}^1)\ =\ \tau(L_k^1)\,[v_2 \mapsto \tau(L_k^1(v_1))]\,;\ \text{and}$$
$$L_{k+1}^0 = L_k^0\,[v_2 \mapsto L_k^0(v_1)]$$

By the induction hypothesis, $\mu_k(\tau(L_k^1(v_1))) \supseteq L_k^0(v_1)$; hence, $\mu_{k+1}(\tau(L_{k+1}^1(v_2))) = \mu_k(\tau(L_{k+1}^1(v_2))) \supseteq L_{k+1}^0(v_2)$. As the state of the other local variables / stack frames does not change, Invariant B.3 is true at moment $k+1$.

For a NEW instruction "$v =$ `new` C", $T_{k+1} = T_k$, $\mu_{k+1} = \mu_k$, and the Invariant B.1 is trivially true (for lack of change from moment k). For Invariant B.3, notice that

$$\tau(L_{k+1}^1) = \tau(L_k^1)\left[v \mapsto \{\tau(n^I_{lb_k,|J_k^1|})\}\right] = \tau(L_k^1)\left[v \mapsto \{n^I_{lb_k,|J_k^1|+1}\}\right]$$
$$L_{k+1}^0 = L_k^0\left[v \mapsto \{n^I_{lb_k,|J_0^0|}\}\right]$$

By Invariant B.2, $|J_k^1| + 1 = |J_k^0|$ and the two inside nodes are identical. As the map μ_{k+1} is reflexive for inside nodes,[6] Invariant B.3 holds at moment $k+1$ too (apart from v, the other local variables do not change).

The cases of IF and NULLIFY are similar. The case of an analyzable CALL is also similar to the case of COPY: a CALL copies the values of the actual arguments into the formal parameters. The case of a STATIC LOAD is similar to the case of NEW: the semantics of $A(m)$ and $A(callee)$ set a local variable to point to a node from $\mathcal{G}$, and μ_k is reflexive for that node. For brevity, we skip the details of these cases.

[6] Map μ_0 is reflexive for nodes from $CINode \cup \mathcal{G}$, and T_{k+1} is extensive, by Lemma 18, so $\mu_{k+1} \supseteq \mu_k$ is also reflexive.

In the case of a STORE statement "$v_1.f = v_2$", by a simple examination of the definition of T_{k+1} (Figure B-3), we notice that $\mu_{k+1} = \mu_k$: the `store` transformers add new inside edges, but do not change the map. As STORE does not change the state of the local variables, the validity of Invariant B.3 trivially propagates at moment $k+1$. For Invariant B.1, only the set of inside edges changes:

$$\begin{array}{rcll} I^0_{k+1} & = & I^0_k \cup L^0_k(v_1) \times \{f\} \times L^0_k(v_2) & \text{By def. of } [\![.]\!]\text{; Fig. 6-5} \\ & \subseteq & I^2_k \cup L^0_k(v_1) \times \{f\} \times L^0_k(v_2) & \text{By Inv. B.1 at moment } k \\ & \subseteq & I^2_k \cup \mu_k(\tau(L^1_k(v_1))) \times \{f\} \times \mu_k(\tau(L^1_k(v_2))) & \text{By Inv. B.3 at moment } k \\ & \subseteq & I^2_{k+1} & \end{array}$$

To explain the last inclusion, notice that $T_{k+1} = F \circ T_k$, where F is a function such that $\forall n_1 \in L^1_k(v_1),\ n_2 \in L^1_k(v_2),\ F \sqsupseteq \texttt{store}(\tau(n_1), f, \tau(n_2))$. Each $\texttt{store}(\tau(n_1), f, \tau(n_2))$ generates the additional inside edges $\mu_k(\tau(n_1)) \times \{f\} \times \mu_k(\tau(n_2))$.

For Invariant B.4, notice that

$$\begin{array}{rcl} W^0_{k+1} & = & W^0_k \cup (L^0_k(v_1) \setminus \mathit{CINode}) \times \{f\}, \text{ and} \\ \tau(W^1_{k+1}) & = & \tau(W^1_k) \cup (\tau(L^1_k(v_1)) \setminus \mathit{CINode}) \times \{f\} \end{array}$$

Let $n \in L^0_k(v_1) \setminus \mathit{CINode}$. By Invariant B.3 at moment k, there exists a node n_2 such that $n_2 \in \tau(L^1_k(v_1))$ and $n \in \mu_k(n_2)$. As $n \notin \mathit{CINode}$, $n_2 \notin \mathit{CINode}$ either (otherwise, by Fact 33, $n = n_2 \in \mathit{CINode}$). Hence, $\langle n_2, f\rangle \in \tau(W^1_{k+1})$, $\langle n, f\rangle \in (\mu_{k+1}(n_2) \setminus \mathit{CINode}) \times \{f\}$, and, ultimately, Invariant B.3 is valid at moment $k+1$.

For Invariant B.5, notice that the new atomic transformers inside T_{k+1} correspond to the new inside edges from $\tau(G^1_{k+1})$; hence, $T_{k+1} \in \tau(G^1_{k+1})$.

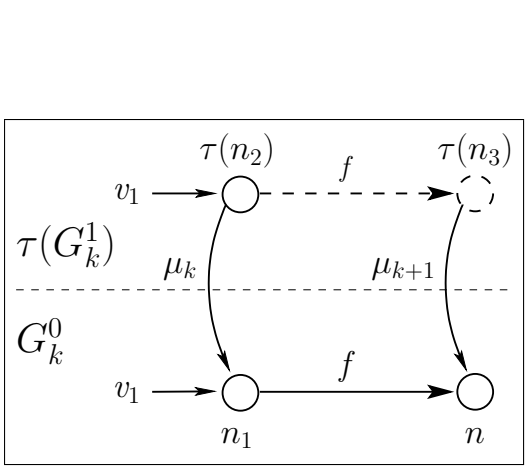
For a STATIC STORE "$C.f = v$", $\mu_{k+1} = \mu_k$: the `gesc` transformers do not change the map. Invariant B.3 is trivially true, for lack of change in the state of local variables. For Invariant B.1, only the set of globally escaped nodes changes. Using the definition of $[\![.]\!]$ and Invariants B.1 and B.3 at moment k:

$$E^0_{k+1} \ = \ E^0_k \cup L^0_k(v) \ \subseteq \ E^2_k \cup L^0_k(v) \ \subseteq \ E^2_k \cup \mu_k(\tau(L^1_k(v))) \ \subseteq \ E^2_{k+1}$$

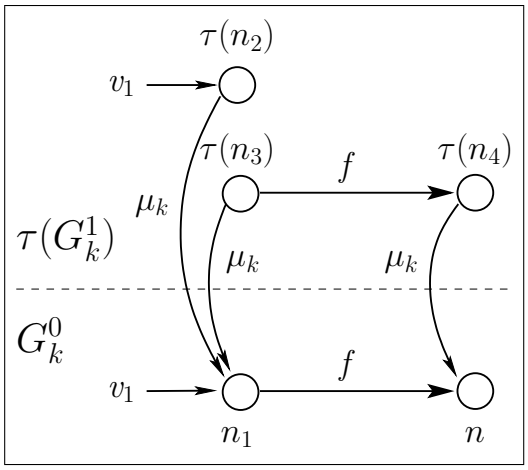
To explain the last inclusion, notice that $T_{k+1} = F \circ T_k$, where F is a function such that $\forall n \in L^1_k(v)$, $F \sqsupseteq \texttt{gesc}(\tau(n))$. Each $\texttt{gesc}(\tau(n))$ generates the additional globally escaped nodes $\mu_k(\tau(n))$. For Invariant B.5, notice that the new atomic transformers inside T_{k+1} correspond to the new globally escaped nodes from $\tau(G^1_{k+1})$; hence, $T_{k+1} \in \tau(G^1_{k+1})$.

The cases of an unanalyzable CALL or a THREAD START are similar to the case of STATIC STORE.

The remaining two cases (LOAD instructions and RETURNs inside $A(\mathit{callee})$) are more complex.



a. Case 1.1.



b. Case 1.2. $n_2 = n_3$ in the subcase 1.2.1; $n_2 \neq n_3$ in the subcase 1.2.2.

Figure B-4: Graphic representations for the case of a LOAD instruction. Solid circles represent general nodes, dashed circles represent load nodes, solid straight arcs represent inside edges, dashed straight arcs represent outside edges, and curved arcs represent node mappings.

LOAD: Consider a LOAD instruction "$v_2 = v_1.f$". This instruction affects the map μ_{k+1} (by adding potentially new mappings), the set of nodes pointed to by v_2, and the set of outside nodes.

We first prove that $L^0_{k+1}(v_2) \subseteq \mu_{k+1}(\tau(L^1_{k+1}(v_2)))$. We consider an arbitrary $n \in L^0_{k+1}(v_2)$ and prove that $n \in \mu_{k+1}(\tau(L^1_{k+1}(v_2)))$. From the definition of $[\![.]\!]$ for a LOAD statement (Figures 4-3 and 4-5), we identify two cases: (1) n is a node pointed to by an f-labeled inside edge that starts in a node pointed to by v_1, and (2) n is the load node attached to this statement (if any).

Case 1: $\exists n_1 \in L^0_k(v_1)$ such that $\langle n_1, f, n\rangle \in I^0_k$. By Invariant B.3 at moment k, $\exists n_2 \in L^1_k(v_1)$ such that $n_1 \in \mu_k(\tau(n_2))$.

Case 1.1: $\langle n_1, f, n\rangle \in I^0_0$. Figure B-4.a contains a graphic representation of this case. As n_1 appears in G^0_0, its context is zero. As the context of $\tau(n_2)$ is at least one, $n_1 \neq \tau(n_2)$. As $n_1 \in \mu_k(\tau(n_2))$, by Fact 33, n_2 must be a load or a parameter node and thus, $e_2(E^1_k, I^1_k)(n_2)$ trivially holds. Hence,

1. The abstract semantics of $A(\mathit{callee})$ makes v_2 point to the load node $n_3 = n^L_{lb_k,|J^1_k|}$: $n_3 \in L^1_{k+1}(v_2)$.
2. By the definition of T_{k+1} (Figure B-3), $T_{k+1} \sqsupseteq \texttt{load}(\tau(n_2), f, \tau(n_3)) \circ T_k$. As $n_1 \in \mu_k(\tau(n_2))$ and $\langle n_1, f, n\rangle \in I^0_k$, the `load` transformer (Figure 4-7 on page 55) adds a mapping from $\tau(n_3)$ to n: $n \in \mu_{k+1}(\tau(n_3))$.

Hence, $n \in \mu_{k+1}(\tau(n_3)) \subseteq \mu_{k+1}(\tau(L^1_{k+1}(v_2)))$

Case 1.2: $\langle n_1, f, n\rangle \in I_k^0 \setminus I_0^0 \subseteq I_k^2 \setminus I_0^0$. Figure B-4.b contains a graphic representation of this case. By Lemma 20, $\exists n_3, n_4$ such that $\langle n_3, f, n_4\rangle \in I_k^1$, $n_1 \in \mu_k(\tau(n_3))$, $n \in \mu_k(\tau(n_4))$. There are two subcases:

Case 1.2.1: $n_2 = n_3$. As $n_2 \in L_k^1(v_1)$ and $\langle n_2, f, n_4\rangle \in I_k^1$, $n_4 \in L_{k+1}^1(v_2)$. Hence, $n \in \mu_k(\tau(n_4)) \subseteq \mu_{k+1}(\tau(n_4)) \subseteq \mu_{k+1}(\tau(L_{k+1}^1(v_2)))$.

Case 1.2.2: $n_2 \neq n_3$. It suffices to prove that $e_2(E_k^1, I_k^1)(n_2)$; next, we can reason exactly as for Case 1.1 above: the abstract semantics of $A(\mathit{callee})$ introduces a load node that μ_{k+1} maps to n.

Assume for the sake of contradiction that $\neg e_2(E_k^1, I_k^1)(n_2)$, i.e., $\neg e_2(\tau(E_k^1), \tau(I_k^1))(\tau(n_2))$ (the graphs G_k^1 and $\tau(G_k^1)$ are isomorphic, as τ is injective). As $n_1 \in \mu_k(\tau(n_2))$, by Fact 34, $n_1 = \tau(n_2)$, and so, n_1 is a node that appears in $\tau(G_k^1)$ and $\neg e_2(\tau(E_k^1), \tau(I_k^1))(n_1)$. As $n_1 \in \mu_k(\tau(n_3))$, by Fact 35, $n_1 = \tau(n_3)$; hence, $\tau(n_2) = \tau(n_3) = n_1$; as τ is injective, $n_2 = n_3$; contradiction!

Case 2: n is the load node introduced by the abstract semantics of $A(m)$ to cope with the read from an escaped node. More specifically, $n = n^L_{lb_k, |J_k^0|} \in L_{k+1}^0(v_2)$, and $\exists n_1 \in L_k^0(v_1)$ such that $e_2(E_k^0, I_k^0)(n_1)$. We prove below that the abstract semantics of $A(\mathit{callee})$ adds the load node $\tau(n^L_{lb_k, |J_k^1|})$ to $\tau(L_k^1(v_2))$ and that μ_{k+1} contains a mapping between the two load nodes.

By Invariant B.3 at moment k, $\exists n_2 \in L_k^1(v_1)$ such that $n_1 \in \mu_k(\tau(n_2))$.

Assume $\neg e_2(E_k^1, I_k^1)(n_2)$. Hence, $\neg e_2(\tau(E_k^1), \tau(I_k^1))(\tau(n_2))$, and by Fact. 34, $n_1 = \tau(n_2)$. As $e_2(E_k^0, I_k^0)(n_1)$, by Lemma 22, $e_2(\tau(E_k^1), \tau(I_k^1))(\tau(n_2))$; contradiction! Hence, $e_2(E_k^1, I_k^1)(n_2)$, implying

$$\begin{array}{rcl} T_{k+1} & \sqsupseteq & \texttt{load}(\tau(n_2), f, \tau(n^L_{lb_k, |J_k^1|})) \circ T_k, \text{ and} \\ n^L_{lb_k, |J_k^1|} & \in & L_{k+1}^1(v_2) \end{array}$$

First, notice that $\tau(n^L_{lb_k, |J_k^1|}) = n^L_{lb_k, |J_k^1|+1} = n_{lb_k, |J_k^0|} = n$. Next, as $e_2(E_k^0, I_k^0)(n_1)$ and $\langle E_k^2, I_k^2\rangle \sqsupseteq \langle E_k^0, I_k^0\rangle$ (by Invariant B.1 at moment k), $e_2(E_k^0, I_k^0)(n_1)$ too. Finally, as $n_1 \in \mu_k(\tau(n_2))$, by the definition of the `load` transformer (Figure 4-7), $n \in \mu_{k+1}(n) \subseteq \mu_{k+1}(\tau(L_{k+1}^1(v_2)))$.

In all cases, $n \in \mu_{k+1}(\tau(L_{k+1}^1(v_2)))$, ending the proof for Invariant B.3 in the case of a LOAD statement.

The abstract semantics of $A(m)$ adds an outside edge from each $n_1 \in L_k^0(v_1)$ with the property $e_2(E_k^0, I_k^0)(n_1)$. However, as we proved in Case 2 above, for each such n_1, there exists a node $n_2 \in L_k^1(v_1)$ such that $n_1 \in \mu_k(\tau(n_2))$, and $e_2(E_k^1, I_k^1)(n_2)$. Hence, T_{k+1} contains a `load` transformer that introduces the outside edge from n_1. Therefore, $O_{k+1}^2 \sqsupseteq O_{k+1}^0$, which proves Invariant B.1 at moment $k+1$.

For Invariant B.5, notice that the new atomic transformers inside T_{k+1} correspond to the new outside edges from $\tau(G_{k+1}^1)$; hence, $T_{k+1} \in \tau(G_{k+1}^1)$. This completes the proof for the case of a LOAD instruction.

RETURN inside $A(callee)$**:** Consider the case of a RETURN inside $A(callee)$, "`return` v". Assume that

$$\begin{aligned} G_k^0 &= \langle L_{i,1}^0 : L_{i,2}^0 : J_k^0,\ I_k^0,\ O_k^0,\ E_k^0,\ \emptyset,\ W_k^0 \rangle \\ G_k^1 &= \langle L_{i,1}^1 : L_{i,2}^1 : J_k^1,\ I_k^1,\ O_k^1,\ E_k^1,\ \emptyset,\ W_k^1 \rangle \end{aligned}$$

and let

$$\begin{aligned} G_b^0 &= \langle J_b^0,\ I_k^0,\ O_k^0,\ E_k^0,\ \emptyset,\ W_k^0 \rangle \quad \text{where} \quad J_b^0 = L_{i,2}^0 \left[v_R \mapsto L_{i,1}^0(v) \right] : J_k^0 \\ G_b^1 &= \langle J_b^1,\ I_k^1,\ O_k^1,\ E_k^1,\ \emptyset,\ W_k^1 \rangle \quad \text{where} \quad J_b^1 = L_{i,2}^1 \left[v_R \mapsto L_{i,1}^1(v) \right] : J_k^1 \end{aligned}$$

v_R is the variable that stores the result of the corresponding CALL. As in the case of COPY, we can use Invariant B.3 at moment k to prove that

$$\mu_k(\tau(J_b^1)) \ @ \ L_0^0 \sqsupseteq J_b^0 \tag{B.6}$$

We write $h = |J_k^1|$, $D_0 = garbage(G_b^0)$, and $D_1 = garbage(\tau(G_b^1)) = \tau(garbage(G_b^1))$ (the last equality is due to the fact that G_b^1 and $\tau(G_b^1)$ are isomorphic, because τ is injective). With these notations:

$$\begin{aligned} G_{k+1}^0 &= \alpha_{|J_k^0|}(gc(G_b^0)) = \alpha_{h+1}(del_{D_0}(G_b^0)) \\ T_{k+1} &= \alpha_{h+1}(del_{D_1}(T_k)) \end{aligned}$$

We first prove that

$$\forall n \in D_1.\ (n \notin nodes(G_b^0)) \vee (n \in D_0) \tag{B.7}$$

Intuitively, Proposition B.7 states that any node that is garbage collected in $\tau(G_b^1)$ either does not appear in G_b^0 or is garbage collected in G_b^0 too. Therefore, for all practical purposes, the set of nodes that are garbage collected in $\tau(G_b^1)$ is a subset of the set of nodes that are garbage collected in G_b^0. As all garbage collected nodes are captured (Lemma 29), they are mapped only to themselves and no other node is mapped to them (Facts 34 and 35). We use these intuitive observations to prove that the garbage collected nodes "do not count".

Proof of Proposition B.7. Pick $n \in D_1$ such that $n \in nodes(G_b^0)$, and assume for the sake of contradiction that $n \notin D_0$. Hence, there exists a path of inside edges from I_k^0 that reaches n from a node $n_2 \in A \cup E_k^0 \cup nodes(J_b^0)$, where $A = CLNode \cup CPNode \cup \mathcal{G}$, and $nodes(J_b^0)$ is the set of nodes pointed to by local variables in J_b^0.

As $n \in D_1 = garbage(\tau(G_b^1))$, by Lemma 29, $\neg e_2(\tau(E_k^1), \tau(I_k^1))(n)$. Also, as $garbage(\tau(G_b^1)) \subseteq nodes(\tau(G_b^1))$, $n \in nodes(\tau(G_b^1))$.

If $n_2 \in A \cup E_k^0$, as $E_k^0 \subseteq E_k^2$ (by Invariant B.1 at moment k), $e_2(E_k^2, I_k^2)(n)$. As NICE$(\tau(G_k^1), I_k^2, E_k^2, \mu_k)$ holds (Fact 32), by Lemma 22, $e_2(\tau(E_k^1), \tau(I_k^1))(n)$; contradiction! Hence, it remains that $n_2 \in nodes(J_b^0)$.

At this point, we know that $n \in nodes(\tau(G_k^1))$, $\neg e_2(\tau(E_k^1), \tau(I_k^1))(n)$, n is reachable from n_2 using edges from $I_k^0 \subseteq I_k^2$, and NICE$(\tau(G_k^1), I_k^2, E_k^2, \mu_k)$ holds. By

Lemma 21, there exists a path of inside edges from $\tau(I_k^1)$ from n_2 to n, $n_2 \in nodes(\tau(G_k^1))$, and $\neg e_2(\tau(E_k^1), \tau(I_k^1))(n_2)$.

By Fact 35, the only node that maps to n_2 in μ_k is n_2 itself. We combine this fact with Equation B.6 and show that $n_2 \in nodes(\tau(J_b^1))$.[7] Therefore, there exists a path in $\tau(I_k^1)$ that reaches n from $nodes(\tau(J_b^1))$. Contradiction with $n \in D_1 = garbage(\tau(G_b^1))$. Hence, $n \in D_0$. □

To prove Invariant B.1 at moment $k+1$, we write

$$\langle I_{k+1}^2, O_{k+1}^2, E_{k+1}^2, \mu_{k+1}\rangle = T_{k+1}(I_0^0, O_0^0, E_0^0, \mu_0) = (\alpha_{h+1}(del_{D_1}(T_k)))\ (I_0^0, O_0^0, E_0^0, \mu_0)$$

As I_0^0, O_0^0, and E_0^0 contain only nodes with context 0, μ_0 contains only nodes with context 0 or 1, and $h \geq 0$, $\langle I_0^0, O_0^0, E_0^0, \mu_0\rangle = \alpha_{h+1}(I_0^0, O_0^0, E_0^0, \mu_0)$. By Lemma 25,

$$\begin{array}{rcl} \langle I_{k+1}^2, O_{k+1}^2, E_{k+1}^2, \mu_{k+1}\rangle & = & (\alpha_{h+1}(del_{D_1}(T_k)))\ (\alpha_{h+1}(I_0^0, O_0^0, E_0^0, \mu_0)) \\ & \sqsupseteq & \alpha_{h+1}(\ (del_{D_1}(T_k))(I_0^0, O_0^0, E_0^0, \mu_0)\ ) \end{array}$$

Nodes from I_0^0, O_0^0, and E_0^0 are disjoint from nodes from D_1 (different contexts: 0 vs. 1). Similarly, μ_0 does not contain any node from D_1: the only nodes with context 1 in μ_0 are parameter nodes and no parameter node can be in D_1.[8] Therefore, $\langle I_0^0, O_0^0, E_0^0, \mu_0\rangle = del_{D_1}(I_0^0, O_0^0, E_0^0, \mu_0)$. By Lemma 26,

$$\begin{array}{rcll} \langle I_{k+1}^2, O_{k+1}^2, E_{k+1}^2, \mu_{k+1}\rangle & \sqsupseteq & \alpha_{h+1}(\ (del_{D_1}(T_k))\ (del_{D_1}(I_0^0, O_0^0, E_0^0, \mu_0))\ ) & \\ & \sqsupseteq & \alpha_{h+1}(\ del_{D_1}(\ T_k(I_0^0, O_0^0, E_0^0, \mu_0))\ ) & \\ & \sqsupseteq & \alpha_{h+1}(\ del_{D_1}(I_k^2, O_k^2, E_k^2, \mu_k)\ ) & \text{(B.8)} \end{array}$$

Hence, $I_{k+1}^2 \supseteq \alpha_{h+1}(del_{D_1}(I_k^2))$. As $G_{k+1}^0 = \alpha_{h+1}(del_{D_0}(G_k^0))$, $I_{k+1}^0 = \alpha_{h+1}(del_{D_0}(I_k^0))$. By Invariant B.1 at moment k, $I_k^2 \supseteq I_k^0$. By Proposition B.7, $del_{D_1}(I_k^2) \supseteq del_{D_0}(I_k^0)$ (removal of fewer nodes from a larger structure[9]). As α_{h+1} is monotonic,

$$I_{k+1}^2 \supseteq \alpha_{h+1}(del_{D_1}(I_k^2)) \supseteq \alpha_{h+1}(del_{D_0}(I_k^0)) = I_{k+1}^0$$

The proofs that $O_{k+1}^2 \supseteq O_{k+1}^0$ and $E_{k+1}^2 \supseteq E_{k+1}^0$ are similar. Therefore, Invariant B.1 is valid at moment $k+1$.

Invariant B.3 at moment $k+1$ has the expression

$$\mu_{k+1}(\tau(L_{k+1}^1 : J_{k+1}^1))\ @\ [L_0^0]\ \sqsupseteq\ L_{k+1}^0 : J_{k+1}^0$$

---

[7] $n_2 \notin nodes(L_0^0)$, because n_2 has context at least 1 (as any node from $\tau(G_k^1)$) and nodes from $nodes(L_0^0)$ have context 0.

[8] Parameter nodes trivially escape, and all garbage collected nodes are captured by Lemma 29.

[9] More rigorously, consider $\langle n_1, f, n_2\rangle \in del_{D_0}(I_k^0)$. Equivalently, $\langle n_1, f, n_2\rangle \in I_k^0$, $n_1 \notin D_0$, and $n_2 \notin D_0$. Hence, $n_1 \in nodes(I_k^0) \subseteq nodes(G_b^0)$, and $n_1 \notin D_0$. By Proposition B.7, $n_1 \notin D_1$. Similarly, $n_2 \notin D_1$. As $I_k^0 \subseteq I_k^2$, $\langle n_1, f, n_2\rangle \in del_{D_1}(I_k^2)$.

Consider the following chain of equalities:

$$\begin{array}{rcll}
\tau(G^1_{k+1}) & = & \tau(\alpha_{|J^1_k|}(gc(G^1_b))) & \text{Def. of } [\![.]\!] \text{ (Figure 6-5)} \\
& = & \tau(\alpha_h(gc(G^1_b))) = \alpha_{h+1}(\tau(gc(G^1_b))) & \text{Lemma 31} \\
& = & \alpha_{h+1}(\tau(del_{garbage(G^1_b)}(G^1_b))) & \text{Def. of } gc \text{ (Figure 4-10)} \\
& = & \alpha_{h+1}(del_{\tau(garbage(G^1_b))}(\tau(G^1_b))) & G^1_b \text{ and } \tau(G^1_b) \text{ are isomorphic} \\
& = & \alpha_{h+1}(del_{D_1}(\tau(G^1_b))) & D_1 = \tau(garbage(G^1_b))
\end{array}$$

Hence, $\tau(L^1_{k+1} : J^1_{k+1}) = \alpha_{h+1}(del_{D_1}(\tau(J^1_b)))$. By Equation B.8, $\mu_{k+1} \sqsupseteq \alpha_{h+1}(del_{D_1}(\mu_k))$. Hence,

$$\begin{array}{rcll}
\mu_{k+1}(\tau(L^1_{k+1} : J^1_{k+1})) & \sqsupseteq & (\alpha_{h+1}(del_{D_1}(\mu_k)))\ (\alpha_{h+1}(del_{D_1}(\tau(J^b_1)))) & \\
& \sqsupseteq & \alpha_{h+1}(\ (del_{D_1}(\mu_k))\ (del_{D_1}(\tau(J^b_1)))\ ) & \text{(B.9)}
\end{array}$$

For the last inequality, it is sufficient to prove that $\forall u \in \mathbb{N}.\ \forall \mu \in Map.\ \forall n \in CNode.$ $\alpha_u(\mu)(\alpha_u(n)) \supseteq \alpha_u(\mu(n))$. Pick $n_1 \in \alpha_u(\mu(n))$. Hence, $\exists n_2$ such that $\langle n, n_2\rangle \in \mu$ and $n_1 = \alpha_u(n_2)$. Hence, $\alpha_u(\mu) \ni \langle \alpha_u(n), \alpha_u(n_2)\rangle = \langle \alpha_u(n), n_1\rangle$, and $n_1 \in \alpha_u(\mu)(\alpha_u(n))$.

Next, we prove that

$$(del_{D_1}(\mu_k))\ (del_{D_1}(\tau(J^b_1))) \ \sqsupseteq\ del_{D_1}(\mu_k(\tau(J^b_1))) \qquad \text{(B.10)}$$

Consider a node n pointed to by a local variable v from the l-th frame of $del_{D_1}(\mu_k(\tau(J^b_1)))$. Hence, $\exists n_2$ that is pointed to by v in the l-th stack frame of $\tau(J^b_1)$, $n \in \mu_k(n_2)$, and $n \notin D_1$. If $n_2 \in D_1$, then $\neg e_2(\tau(E^1_k), \tau(I^1_k))(n_2)$ (Lemma 29), and by Fact 34, $n_2 = n \notin D_1$, contradiction! So, $n_2 \notin D_1$, v points to n_2 in the l-th stack frame of $del_{D_1}(\tau(J^b_1))$, and $n \in (del_{D_1}(\mu_k))\ (n_2)$. Therefore, v points to n in the l-th frame of the left-hand side of Equation B.10.

We develop Equation B.9 as follows:

$$\begin{array}{rcl}
\mu_{k+1}(\tau(L^1_{k+1} : J^1_{k+1}))\ @\ [L^0_0] & \sqsupseteq & \alpha_{h+1}(del_{D_1}(\mu_k(\tau(J^b_1))))\ @\ [L^0_0] \\
& \sqsupseteq & \alpha_{h+1}(del_{D_1}(\ \mu_k(\tau(J^1_b))\ @\ [L^0_0]\ )) \\
& \sqsupseteq & \alpha_{h+1}(del_{D_1}(J^0_b)) \\
& \sqsupseteq & \alpha_{h+1}(del_{D_0}(J^0_b)) \ =\ L^0_{k+1} : J^0_{k+1}
\end{array}$$

The inequalities above are due to the following facts (in order): (1) α_{h+1} and del_{D_1} have no effect on L^0_0, as the context of all nodes from L^0_0 is zero; (2) Equation B.6 on page 186; (3) Proposition B.7 on page 186 (removal of fewer nodes).

To prove Invariant B.4, notice that for any points-to graph G, $garbage(G)$ contains only inside nodes (the other nodes are trivially reachable; see definition of *garbage* in Figure 4-10). As the sets of mutated abstract fields do not use any inside nodes, $W^0_{k+1} = \alpha_{h+1}(del_{D_0}(W^0_k)) = \alpha_{h+1}(W^0_k)$, and $\tau(W^1_{k+1}) = \alpha_{h+1}(del_{D_1}(\tau(W^1_k))) =$

$\alpha_{h+1}(\tau(W_k^1))$. By the induction hypothesis,

$$W_0^0 \ \cup \bigcup_{\langle n,f\rangle \in \tau(W_k^1)} (\mu_k(n) \setminus \mathit{CINode}) \times \{f\} \ \supseteq \ W_k^0 \tag{B.11}$$

We need to prove that

$$W_0^0 \ \cup \bigcup_{\langle n,f\rangle \in \tau(W_{k+1}^1)} (\mu_{k+1}(n) \setminus \mathit{CINode}) \times \{f\} \ \supseteq \ W_{k+1}^0 \tag{B.12}$$

We consider an arbitrary $\langle n, f\rangle \in W_{k+1}^0$ and prove that $\langle n, f\rangle$ is an element of the set from the left-hand side of Equation B.12. As $\langle n, f\rangle \in W_{k+1}^0 = \alpha_{h+1}(W_k^0)$, there exists n_2 such that $n = \alpha_{h+1}(n_2)$ and $\langle n_2, f\rangle \in W_k^0$. By Equation B.11, there are two cases:

1. If $\langle n_2, f\rangle \in W_0^0$, then n_2 has context 0. Hence, $n = n_2$, and $\langle n, f\rangle \in W_0^0$.

2. Otherwise, there exists a node n_3 such that $\langle n_3, f\rangle \in \tau(W_k^1)$ and $n_2 \in \mu_k(n_3) \setminus \mathit{CINode}$. As $n_2 \notin \mathit{CINode}$, $n_3 \notin \mathit{CINode}$ either (otherwise, by Fact 33, $n_2 = n_3 \in \mathit{CINode}$). Hence, n_2 and n_3 do not appear in D_1, $\langle n_3, n_2\rangle \in \mathit{del}_{D_1}(\mu_k)$, and $\langle \alpha_{h+1}(n_3), \alpha_{h+1}(n_2)\rangle \in \alpha_{h+1}(\mathit{del}_{D_1}(\mu_k)) \subseteq \mu_{k+1}$ (the last inclusion is due to Equation B.8). Moreover, $\langle n_3, f\rangle \in \tau(W_k^1)$ implies $\langle \alpha_{h+1}(n_3), f\rangle \in \alpha_{h+1}(\tau(W_k^1)) = \tau(W_{k+1}^1)$. Putting these facts together, we prove that $\langle \alpha_{h+1}(n_2), f\rangle = \langle n, f\rangle$ is an element of the set from the left-hand side of Equation B.12.

Finally, we need to prove Invariant B.5, i.e., $T_{k+1} \in \mathit{Trans}(\tau(G_{k+1}^1))$. By the corresponding rule from Figure B-3, $T_{k+1} = \alpha_{|J_k^1|+1}(\mathit{del}_{\tau(\mathit{garbage}(G_b^1))}(T_k))$. By Invariant B.5 at moment k, $T_k \in \mathit{Trans}(\tau(G_k^1))$. Notice that $\mathit{Trans}(\tau(G_k^1)) = \mathit{Trans}(\tau(G_b^1))$ (same sets of inside edges, outside edges, and globally escaped nodes). By Lemma 27 and Lemma 25 Part 1, $T_{k+1} \in \mathit{Trans}(\alpha_{h+1}(\mathit{del}_{\tau(\mathit{garbage}(G_b^1))}(\tau(G_b^1))))$. As we have already proved that $\tau(G_{k+1}^1) = \alpha_{h+1}(\mathit{del}_{\tau(\mathit{garbage}(G_b^1))}(\tau(G_b^1)))$, we obtain $T_{k+1} \in \mathit{Trans}(\tau(G_{k+1}^1))$.

Final Step

To prove Equation 6.13, we need to prove that

$$\mathit{interproc}(G_0^0,\ G_{l+1}^1,\ \text{“}v_R = v_0.s(v_1, \ldots, v_m)\text{”}) \ \sqsupseteq \ G_{l+1}^0 \tag{B.13}$$

We use the notation

$$\begin{aligned} G_l^0 &= \langle L_l^0 : L_0^0 : [],\ I_l^0,\ O_l^0,\ E_l^0,\ \emptyset,\ W_l^0\rangle \\ G_l^1 &= \langle L_l^1 : [],\ I_l^1,\ O_l^1,\ E_l^1,\ \emptyset,\ W_l^1\rangle \end{aligned}$$

By the definition of $[\![.]\!]$ (Figure 6-5),

$$\begin{aligned} G^0_{l+1} &= \alpha_0(gc(G^0_b)), \text{ where } G^0_b = \langle L^0_0\,[v_R \mapsto L^0_l(v)] : [],\ I^0_l,\ O^0_l,\ E^0_l,\ \emptyset,\ W^0_l \rangle \\ G^1_{l+1} &= \langle L^1_l : [],\ I^1_l,\ O^1_l,\ E^1_l,\ L^1_l(v),\ W^1_l \rangle \\ \rho(G^1_{l+1}) &= \langle L_{\text{all-empty}} : [],\ I^1_l,\ O^1_l,\ E^1_l,\ L^1_l(v),\ W^1_l \rangle \end{aligned}$$

[Recall from Section 4.4.3 that for each points-to graph G, $\rho(G)$ is a points-to graph similar to G, except that each local variable points to an empty set of nodes.] Let $D_0 = \mathit{garbage}(G^0_b)$. Hence, $gc(G^0_b) = \mathit{del}_{D_0}(G^0_b)$. By the definition of *interproc* (Figure 4-6 on page 53),

$$\begin{aligned} &\mathit{interproc}(G^0_0, G^1_{l+1}, \text{“}v_R = v_0.s(v_1, \ldots, v_j)\text{”}) = \\ &\qquad \alpha_0(\ \langle\ L^0_0\,[v_R \mapsto \mu_a(\tau(L^1_l(v)))] : [],\ I_a,\ O_a,\ E_a,\ \emptyset,\ W_a\ \rangle\ ) \end{aligned}$$

where $\langle I_a, O_a, E_a, \mu_a \rangle = T_{\mathit{callee}}(I^0_0, O^0_0, E^0_0, \mu_0)$, $T_{\mathit{callee}} = \mathit{mct}(\tau(gc(\rho(G^1_{l+1}))))$, and

$$W_a = W^0_0 \ \cup \bigcup_{\langle n,f\rangle \in \tau(W^1_l)} (\mu_a(n) \setminus \mathit{CINode}) \times \{f\}$$

[Note: we used the fact that the function gc does not remove any returned nodes (see Figure 4-10). Hence, $gc(\rho(G^1_{l+1}))$ and $\rho(G^1_{l+1})$ have the same set of returned nodes, $L^1_l(v)$. Similarly, as gc removes only captured inside nodes, $gc(\rho(G^1_{l+1}))$ has the same set of mutated abstract fields as $\rho(G^1_{l+1})$, W^1_l.]

With these notations, Equation B.13 is equivalent to the following statements:

$$\begin{aligned} \alpha_0(L^0_0\left[v_R \mapsto \mu_a(\tau(L^1_l(v)))\right]) &\sqsupseteq \alpha_0(\mathit{del}_{D_0}(L^0_0\left[v_R \mapsto L^0_l(v)\right])) \qquad \text{(B.14)} \\ \alpha_0(I_a, O_a, E_a, W_a) &\sqsupseteq \alpha_0(\mathit{del}_{D_0}(I^0_l, O^0_l, E^0_l, W^0_l)) \end{aligned}$$

As any node from $L^0_0\,[v_R \mapsto L^0_l(v)]$ is trivially reachable in G^0_b, del_{D_0} does not have any effect in Equation B.14 above. As α_0 is monotonic, it is sufficient to prove that

$$\mu_a(\tau(L^1_l(v))) \supseteq L^0_l(v) \qquad \text{(B.15)}$$

$$\langle I_a, O_a, E_a, W_a \rangle \sqsupseteq \mathit{del}_{D_0}(I^0_l, O^0_l, E^0_l, W^0_l) \qquad \text{(B.16)}$$

Let $D_1 = \mathit{garbage}(\tau(\rho(G^1_{l+1})))$. Consider $T_{l+1} = \mathit{del}_{D_1}(T_l)$. By Invariant B.5 at moment l, $T_l \in \mathit{Trans}(\tau(G^1_l))$. As G^1_l and $\rho(G^1_{l+1})$ have the same inside edges, outside edges, and directly globally escaped nodes, $\mathit{Trans}(\tau(G^1_l)) = \mathit{Trans}(\tau(\rho(G^1_{l+1})))$. By Lemma 27, $T_{l+1} \in \mathit{Trans}(\mathit{del}_{\mathit{garbage}(\tau(\rho(G^1_{l+1})))}(\tau(\rho(G^1_{l+1})))) = \mathit{Trans}(gc(\tau(\rho(G^1_{l+1})))) = \mathit{Trans}(\tau(gc(\rho(G^1_{l+1}))))$ (the last equality is due to the fact that for each points-to graph G, G and $\tau(G)$ are isomorphic). By Lemma 19, $T_{\mathit{callee}} = \mathit{mct}(\tau(gc(\rho(G^1_{l+1})))) \sqsupseteq T_{l+1}$, which implies

$$\langle I_a, O_a, E_a, \mu_a \rangle \ \sqsupseteq\ T_{l+1}(I^0_0, O^0_0, E^0_0, \mu_0) \ =\ (\mathit{del}_{D_1}(T_l))\ (I^0_0, O^0_0, E^0_0, \mu_0)$$

D_1 has no common nodes with I_0^0, O_0^0, E_0^0, and μ_0.[10] Hence, $\langle I_0^0, O_0^0, E_0^0, \mu_0\rangle = del_{D_1}(I_0^0, O_0^0, E_0^0, \mu_0)$. By Lemma 26,

$$\begin{aligned}\langle I_a, O_a, E_a, \mu_a\rangle &\sqsupseteq (del_{D_1}(T_l))\ (del_{D_1}(I_0^0, O_0^0, E_0^0, \mu_0))\\ &\sqsupseteq del_{D_1}(\ T_l(I_0^0, O_0^0, E_0^0, \mu_0)\ ) = del_{D_1}(I_l^2, O_l^2, E_l^2, \mu_l) \quad \text{(B.17)}\end{aligned}$$

Hence, $\mu_a \supseteq del_{D_1}(\mu_l)$. To prove Equation B.15, we first write

$$\mu_a(\tau(L_l^1(v))) \ \supseteq\ (del_{D_1}(\mu_l))\ (\tau(L_l^1(v)))$$

Notice the following two facts:

1. The nodes from $\tau(L_l^1(v))$ are trivially reachable in $\tau(\rho(G_{l+1}^1))$ (they are the returned nodes). Hence, they do not appear in D_1.

2. By Lemma 29, $\forall n \in D_1$, $\neg e_2(\tau(E_l^1), \tau(I_l^1))(n)$. Hence, by Fact 35, μ_l does not map any node from outside D_1 (e.g., from $\tau(L_l^1(v))$) to a node from D_1.

Therefore, the mappings involving nodes from D_1 are irrelevant in this context, i.e., $(del_{D_1}(\mu_l))\ (\tau(L_l^1(v))) = \mu_l(\tau(L_l^1(v)))$. By Invariant B.3 at moment l, $\mu_l(\tau(L_l^1(v)))\ \supseteq\ L_l^0(v)$, which completes the proof of Equation B.15.

To prove Equation B.16, notice that by Equation B.17, $I_a \supseteq del_{D_1}(I_l^2)$. By Invariant B.1 at moment l, $I_l^2 \supseteq I_l^0$. As in the case of a RETURN inside $A(m)$, we can prove that $\forall n \in D_1.(n \notin nodes(G_b^0)) \vee (n \in D_0)$.[11] Therefore, $I_a \supseteq del_{D_1}(I_l^2) \supseteq del_{D_0}(I_l^0)$ (removal of fewer nodes from a larger structure). Ultimately, we prove that $I_a \supseteq del_{D_0}(I_l^0)$. We can prove similar relations for O_a and E_a.

To prove that $W_a \supseteq del_{D_0}(W_l^0)$, notice that D_0 contains only inside nodes (other nodes are trivially reachable in G_b^0). Hence, $del_{D_0}(W_l^0) = W_l^0$. Next, as $\mu_a \supseteq del_{D_1}(\mu_l)$,

$$W_a\ \supseteq\ W_0^0\ \cup \bigcup_{\langle n,f\rangle \in \tau(W_l^1)} (\ (del_{D_1}(\mu_l))\ (n)\ \setminus\ CINode\ ) \times \{f\}$$

D_1 contains only inside nodes that do not appear in $\tau(W_l^1)$ (the analysis does not record the mutations on inside nodes). Additionally, as we already pointed out, μ_l

[10] Nodes from D_1 have context at least 1. The only nodes with context 1 in μ_0 are parameter nodes, that trivially escape and hence, by Lemma 29, cannot appear in D_1.

[11] The proof is similar to the proof of Proposition B.7: Pick $n \in D_1$ such that $n \in nodes(G_b^0)$ and assume for the sake of contradiction that $n \notin D_0$. Using the same techniques as in the proof of Proposition B.7, we find a node n_2 such that: (1) a local variable points to n_2 in G_b^0, (2) there exists a path of inside edges from $\tau(I_l^1)$ from n_2 to n, (3) $n_2 \in nodes(\tau(G_l^1))$, and (4) the only node that μ_l maps to n_2 is n_2 itself.

By fact (2), the context of n_2 is at least 1, which, by fact (1), implies that in G_b^0, v_R points to n_2 (the other local variables point to the same 0-context nodes as in L_0^0). Hence, $n_2 \in L_l^0(v)$. As $\mu_l(\tau(L_l^1(v))) \supseteq L_l^0(v)$ (Invariant B.3 at moment l), by fact (4), $n_2 \in \tau(L_l^1(v))$. We use fact (2) to obtain a contradiction with $n \in D_1 = garbage(\tau(G_b^1))$.

does not map any node from outside D_1 to a node from D_1. Hence, del_{D_1} is irrelevant in the relation above, i.e.,

$$W_a \;\supseteq\; W_0^0 \;\cup \bigcup_{\langle n,f\rangle \in \tau(W_l^1)} (\; \mu_l(n) \;\setminus\; \mathit{CINode} \;) \times \{f\}$$

Using Invariant B.4 at moment l, we prove that $W_a \supseteq W_l^0 = del_{D_0}(W_l^0)$.

This completes our proof of Equation 6.13.

Bibliography

[1] GNU Classpath, essential libraries for Java. Available from `http://www.gnu.org/software/classpath`.

[2] C. Scott Ananian. MIT Flex compiler infrastructure for Java. Available from `http://www.flex-compiler.lcs.mit.edu`, 1998-2004.

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

[4] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, 1996.

[5] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1996.

[6] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004.

[7] Bruno Blanchet. Escape analysis for object oriented languages. Application to Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1999.

[8] Bruno Blanchet. Escape Analysis for Java(TM). Theory and Practice. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(6), 2003.

[9] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9), September 1988.

[10] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, July 2002.

[11] Chandrasekhar Boyapati and Martin C. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the 16th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[12] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3), 1992.

[13] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. Technical Report NII-R0309, Computing Science Institute, Univ. of Nijmegen, March 2003.

[14] Brendon Cahoon and Kathryn S. McKinley. The Java Olden benchmark suite. Available online from `http://www-ali.cs.umass.edu/DaCapo/benchmarks.html`.

[15] Brendon Cahoon and Kathryn S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[16] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1995.

[17] Nestor Cataño and Marieke Huismann. ChAsE: a static checker for JML's assignable clause. In *Proceedings of the 4th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 2575 of *LNCS*, 2003.

[18] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 1990.

[19] Ramkrishna Chatterjee, Barbara Ryder, and William Landi. Relevant context inference. In *Proceedings of the 26th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, 1999.

[20] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, 1993.

[21] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1999.

[22] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. Technical Report RC22340, IBM Research, May 2002.

[23] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[24] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithms (2nd Edition)*. MIT Press and McGraw-Hill, 2001.

[25] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, 1999.

[26] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4), October 1991.

[27] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with ADABU. In *Proceedings of the 4th Workshop on Dynamic Analysis (WODA)*, 2006.

[28] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 1994.

[29] Alain Deutsch. On the complexity of escape analysis. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, 1997.

[30] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[31] Alexandru D. Sălcianu etal. `jpaul` – Java Program Analysis Utilities Library. Available from `http://jpaul.sourceforge.net`, 2005.

[32] Elliot Joel Berk etal. JLex: A lexical analyzer generator for Java. Available from `http://www.cs.princeton.edu/~appel/modern/java/JLex`.

[33] Scott Hudson etal. CUP parser generator for Java. Available from `http://www.cs.princeton.edu/~appel/modern/java/CUP`.

[34] Manuel Fähndrich. *BANE: A Library for Scalable Constraint-Based Program Analysis*. PhD thesis, University of California at Berkeley, May 1999.

[35] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 1998.

[36] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *Proceedings of the 9th International Conference on Compiler Construction (CC)*, volume 1781. Springer-Verlag, 2000.

[37] Ovidiu Gheorghioiu, Alexandru D. Sălcianu, and Martin C. Rinard. Interprocedural compatibility analysis for static object preallocation. In *Proceedings of the 30th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, 2003.

[38] Rakesh Ghiya and Laurie Hendren. Is it a tree, a DAG, or a cyclic graph? In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages (POPL)*, 1996.

[39] Benjamin Goldberg and Young G. Park. Escape analysis on lists. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 1992.

[40] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2001.

[41] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2001.

[42] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2001.

[43] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4), 1999.

[44] Michael Hind and Anthony Pioli. Which pointer analysis should I use? In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2000.

[45] Martin Hirzel, Amer Diwan, and Michael Hind. Pointer analysis in the presence of dynamic class loading. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP)*, 2004.

[46] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of lisp-like structures. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. Prentice-Hall, 1981.

[47] John Kodumal and Alex Aiken. Banshee: A scalable constraint-based analysis toolkit. In *Proceedings of the 12th International Static Analysis Symposium (SAS)*, 2005.

[48] Viktor Kuncak. *Decision Procedures for Modular Data Structure Verification*. PhD thesis, Massachusetts Institute of Technology, in preparation.

[49] Viktor Kuncak, Patrick Lam, Karen Zee, and Martin Rinard. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering*, 2006. Accepted for publication.

[50] Patrick Lam, Viktor Kuncak, and Martin C. Rinard. Hob: A tool for verifying data structure consistency. In *14th International Conference on Compiler Construction (tool demo)*, April 2005.

[51] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 29(9), 1979.

[52] William Landi and Barbara Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 1992.

[53] Gary T. Leavens. Advances and issues in JML. Presentation at the Java Verification Workshop, January 2002.

[54] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[55] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML. Technical Report 96-06p, Iowa State University, 2001.

[56] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[57] Ondřej Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, January 2006.

[58] Ondřej Lhoták and Laurie Hendren. Jedd: A BDD-based relational extension of Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[59] Ondřej Lhoták and Laurien Hendren. Scaling Java points-to analysis using spark. In *Proceedings of the 12th International Conference on Compiler Construction (CC)*, 2003.

[60] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for java. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2001.

[61] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[62] Roman Manevich, G. Ramalingam, John Field, Deepak Goyal, and Mooly Sagiv. Compactly representing first-order structures for static analysis. In *Proceedings of the 9th International Static Analysis Symposium (SAS)*, 2002.

[63] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of the 32nd Annual ACM Symposium on the Principles of Programming Languages (POPL)*, 2005.

[64] Darko Marinov, Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Martin C. Rinard. An evaluation of exhaustive testing for data structures. Technical Report MIT-LCS-TR-921, MIT CSAIL, Cambridge, MA, September 2003.

[65] Ana Milanova, Atanas Routev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, July 2002.

[66] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[67] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Massachusetts, 1994.

[68] Zhenyu Qian. A formal specification of Java Virtual Machine instructions for objects, methods and subrountines. In *Formal Syntax and Semantics of Java*, pages 271–312, 1999.

[69] Atanas Rountev. *Dataflow Analysis of Software Fragments*. PhD thesis, Rutgers University, 2002.

[70] Atanas Rountev. Precise identification of side-effect-free methods in Java. In *IEEE International Conference on Software Maintenance*, 2004.

[71] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. In *Proceedings of the 16th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[72] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3), 2002.

[73] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis problems. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[74] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.

[75] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodik. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

[76] Standard Performance Evaluation Corporation. SPECjvm98 Java virtual machine benchmark suite. `http://www.spec.org/osg/jvm98/jvm98/doc/index.html`, August 1998.

[77] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages (POPL)*, 1996.

[78] Alexandru D. Sălcianu. Pointer analysis and its applications to Java programs. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2001.

[79] Alexandru D. Sălcianu. `jppa` – a Java pointer and purity analysis tool. Available from `http://jppa.sourceforge.net`, 2006.

[80] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4), July 1998.

[81] Matthew S. Tschantz. Javari: Adding reference immutability to Java. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2006.

[82] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *Proceedings of the 20th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

[83] Frédéric Vivien and Martin C. Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2001.

[84] John Whaley and Monica Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[85] John Whaley and Martin C. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

[86] Robert Wilson and Monica Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 1995.

[87] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2004.