Pointer Analysis and its Applications for Java Programs

by

Alexandru D. Sălcianu

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

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2001

Author .....
Department of Electrical Engineering
and Computer Science
August 31, 2001

Certified by .....
Martin C. Rinard
Associate Professor
Thesis Supervisor

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

Pointer Analysis and its Applications for Java Programs

by

Alexandru D. Sălcianu

Submitted to the Department of Electrical Engineering
and Computer Science
on August 31, 2001, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

This thesis investigates the design and the correctness of a pointer analysis for the Java programming language. Although the analysis is based on a previous analysis, we made several important modifications that make it distinct from the original one. The major part of this thesis is dedicated to the correctness proof for the analysis.

The analysis is a flow-sensitive, compositional, inter-procedural pointer analysis. It is based on the abstraction of points-to graphs, which characterize how local variables and fields in objects point to other objects. Each points-to graph also contains escape information that characterizes how objects allocated in the analyzed part of the program can be accessed by other parts. The analysis computes a single, parameterized points-to graph for the exit point of each method, and instantiates the graph for each call site that might invoke that method. This points-to graph uses placeholders to abstract over the calling context.

We present three applications of the analysis: stack allocation, allocation in the thread-local heap, and synchronization removal. The analysis is able to detect that some objects are not used outside the method or the thread that allocates them. A compiler that uses the analysis can generate code that allocate these objects on the stack, respectively in a heap that is local to the current thread. These optimizations have the potential of reducing the garbage collection overhead. The compiler can also generate code that reduces, or even eliminates the cost of the synchronization operations executed on the objects that do not escape the thread that allocates them.

In a modern language like Java, code safety is very important. The main advantage of our analysis is that is comes with a correctness proof. The proof handles all the relevant features of the analysis. To the best of our knowledge, this is the first correctness proof for a flow-sensitive, compositional, inter-procedural pointer analysis.

Thesis Supervisor: Martin C. Rinard
Title: Associate Professor

To all those people who showed me
that human analysis is much more
complicated, and rewarding!, than
pointer analysis

Acknowledgments

Pointer analysis is a difficult topic, and without the help that I received from many people around me, I wouldn't be writing these lines now.

First of all, I would like to thank my family — my mother Maria, and my late father Eugen — for the education they gave me, and for the support they provided me throughout my life. Their help was enormous, and I can never repay it.

I would like to thank my advisor Martin Rinard, who believed in me and insisted that I come to MIT. Martin encouraged me to work in the pointer analysis research area, and provided me with many valuable research and life-experience lessons. His enthusiasm for research and publications helped me organize my ideas, learn many things, and finish tasks quickly. Finishing my SM thesis after the rather long period of two years is entirely due to my (exaggerated?) ambition to investigate new things, e.g., the correctness of the analysis.

I would like to thank the students from Martin's group: Darko Marinov, Maria-Cristina Marinescu, Viktor Kunčak, Wes Beebee, Brian Demsky, C. Scott Ananian, Radu Rugină, Chandrasekhar Boyapati, Karen Zee, Felix S. Klock, and Patrick Lam. They provided me with a friendly working environment. Darko was my usual partner for endless discussions on various topics: research, life, MIT, etc. I truly learned a lot from him. In addition, he took the painstaking task of reviewing drafts of my thesis. Maria-Cristina and Radu frequently joined my conversations with Darko. Brian, Scott, and Darko helped me adapt to the FLEX compiler infrastructure. Also, Brian and Scott helped me test various applications of the analysis. Viktor and Chandra exchanged many interesting research ideas with me. Viktor also provided me with a motivating example of what full commitment to research is! Wes implemented the Real Time Specification for Java in the FLEX compiler infrastructure. His implementation was very helpful for evaluating the benefits of pointer analysis for real time applications.

On the human life scale, two years is a long period. Two years at MIT is an even longer period, and I am deeply grateful to all those people who made this period a happy part of my life. In addition to the already mentioned people, I would like to thank the enthusiastic members of the two MIT clubs that I regularly frequented: International Film Club, and Romanian Student Association.

Contents

List of Figures

Chapter 1

Introduction

The presence of pointers in a programming language significantly complicates the analysis, optimization, and verification of programs that are written in that language, because the analysis system cannot determine the 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 instructions that use pointers. This limits the impact of program understanding and testing tools, and that of standard compiler optimizations such as constant propagation, common subexpression elimination, loop-invariant code motion, strength reduction, dead code removal, etc. For instance, without knowledge about the memory locations modified by a specific memory write instruction, no constants can be propagated accross that instruction. In the early decades of computing this was not a big problem, because pointers were rarely used. As modern object-oriented languages, such as Java [3], use pointers as their main datatype, the analysis of pointers becomes really important for an optimizing compiler.

In high-level languages, in addition to increasing the impact of standard compiler optimizations, precise knowledge about pointers enables new optimizations, such as removing unnecessary synchronizations, and optimizing memory allocation to reduce or even eliminate the garbage collection overhead.

In its most general definition, the pointer analysis field contains all the analyses that try to detect useful properties about pointers. As most of these properties are undecidable, each pointer analysis produces a conservative approximation of the precise result. There are at least four types of analyses that fall into this category: *points-to analyses*, *alias analyses*, *escape analyses*, and *shape analyses*. A points-to analysis, e.g., [6], identifies the memory locations which are pointed to by a specific pointer variable. An alias analysis, e.g., [10], determines the pairs of pointer expressions which are aliased, i.e., point to the same memory location. An escape analysis, e.g., [4], detects the memory locations that escape a given scope; the scope of such an analysis is usually a method, but other alternatives are possible: the current iteration of a loop, a group of methods, etc. Finally, a shape analysis, e.g., [19], detects the shape of the data structures manipulated by the program. Such an analysis checks

properties such as "if this method receives an acyclic list, it returns an acyclic list". In practice, all these analyses are closely related, and usually it is very difficult to decide whether a specific analysis is in one category or another.

Pointer analysis is a very active research area. In an invited talk at PASTE'01 [13], Michael Hind counted no fewer than seventy-five papers and nine PhD theses published on pointer analysis in the last twenty-one years[1]. This abundence is explained by the fact that pointer analysis is very useful but also very difficult. Being very useful, there is a lot of research interest on this problem and many researchers try to design analyses that solve it. On the other side, as the problem itself is very difficult, it is unlikely that any of these many analyses will solve it completely; instead, we have many algorithms solving various approximations of it. Another important observation is that only a tiny fraction of the published analyses have been proved to be correct.

In spite of its benefits and the impressive resources invested along more than two decades of research, pointer analysis remains in the stage of "exciting research problem". To the best of our knowledge, no industrial compiler uses a reasonably precise pointer analysis. Many reasons contributes to this situation: the difficulty of the problem itself, opportunities of big improvements using far simpler analyses, steady increases in hardware that made work on optimizing compiler look unnecessary, etc.

We believe that the ubiquitous use of modern programming languages will make pointer analysis critical for modern program understanding and verifying tools, as well as for optimizing compilers. As computers start to control critical aspects of human life, verification becomes increasingly important. Software companies will afford the high computation resources required by the pointer analysis if this can help them debug their products. Also, although big increases in hardware speed might make the necessity of compiler optimizations questionable for most applications, there will always be applications which require the additional speed provided by the optimizations that are enabled by pointer analysis. In our opinion, hardware improvements do not make pointer analysis, and program analysis in general, unnecessary; instead, they shift the focus of program analysis from optimization to verification.

Using pointer analysis in program verification tools implies the use of provably correct analyses. As most of the current pointer analyses do not have a correctness proof, a lot of work is necessary in this area.

This thesis investigates the design and the correctness of a pointer analysis for the Java programming language. Although the analysis was originally based on ideas from a previous analysis, published by Whaley and Rinard [25], we made several important modifications that make it distinct from the original analysis. The major part of this thesis is dedicated to the correctness proof for the analysis.

[1]Unfortunately, there is no available estimate of the number of SM theses. Whatever that number was, we incremented it!

The analysis is a flow-sensitive, compositional, inter-procedural pointer analysis. It is based on the abstraction of points-to graphs, which characterize how local variables and fields in objects point to other objects. The analysis uses the *object allocation site* model: all objects created by the same allocation instruction in the program are modeled by the same node. Each points-to graph also contains escape information that characterizes how objects allocated in the analyzed part of the program can be accessed by other parts. The analysis examines each method once[2] and generates a single, parameterized points-to graph for the end of the method. This points-to graph uses placeholders to abstract over the calling context. The analysis instantiates the points-to graph computed for the exit point of a method for each call site which may call that method.

We prove the correctness of the analysis with respect to three optimizations: stack allocation, allocation in the thread-local heap, and synchronization removal. We also obtain results that characterize the modeling relation between the points-to graphs and the heaps created by the execution of the analyzed program.

Initially, we worked on the correctness proof for the analysis of Whaley and Rinard [25]. However, to prove the correctness of that analysis, we had to reformalize it from scratch, obtaining a new analysis. Much of our new analysis was motivated by the need to make the formal specification of the analysis clearer and easier to reason about. The final design of the analysis was also motivated, in part, by correctness considerations. During the process of developing the proof for our analysis, we discovered several corner cases and found it necessary to augment the rules for the inter-procedural analysis to cover these corner cases and make our analysis correct.

The proof has a multi-layer structure. At the bottom level we have the concrete semantics of Java. The pointer analysis is the top layer. Due to the big difference in the complexity of these two layers, it was difficult to relate them directly. Our proof idea is to introduce an intermediate layer, the abstract semantics, between the two. For each relevant point from the execution of a program, the abstract semantics computes an abstract state that models the heap, and an explicit abstraction relation that records how nodes model objects. We prove that in this hierarchy, each layer is a conservative approximation of the layer beneath it. We split the proof in two parts: one set of invariants that relate the abstract semantics to the concrete semantics and another set of results that relate the pointer analysis to the abstract semantics.

Our proof has many things in common with the abstract semantics framework [8]. The main similarity is our view of the analysis, at least in an intermediate form of it, as an abstract execution of the program over a finite lattice. However, our proof is not based on the techniques specific to the abstract semantics framework. Instead, it uses simulation invariants, which is a popular methodology for the correctness proofs of distributed systems [15]: the abstract semantics simulates the concrete semantics with respect to a set of invariants.

This thesis makes the following contributions:

[2]However, each set of mutually recursive methods requires a fixed point algorithm.

new pointer analysis We give a complete, formal presentation of the modified analysis. The modified analysis is quite far from the initial one, although they are based on the same intuitive ideas. Therefore, it can be considered a new analysis.

correctness proof We present a formal correctness proof for the new pointer analysis. To the best of our knowledge, this is the first correctness proof for a flow-sensitive, compositional, inter-procedural pointer analysis.

The rest of this thesis is organized as follows. In Chapter 2, we present our new pointer analysis. Next, in Chapter 3, we describe a few applications of our pointer analysis. Chapter 4 is the core of the thesis: it presents a correctness proof for our analysis. We discuss related work in Chapter 5 and conclude in Chapter 6.

Chapter 2

Analysis Presentation

This chapter gives a formal presentation of our pointer analysis. We start by presenting the main design features of the analysis in Section 2.1. In Section 2.2, we describe the general mathematical notations that we use in this thesis. Next, we present the representation of the analyzed program in Section 2.3. Section 2.4 introduces the sets and the notations that are used in the main section of this chapter, Section 2.5, to formally present our analysis. Finally, Section 2.6 gives a high-level description of an algorithm for computing the analysis.

2.1 Analysis Features

Our pointer analysis is a *flow-sensitive, forward dataflow may-analysis.* The analysis is *compositional* and can *analyze incomplete programs.* The heap is represented by using the *object allocation site* model. These are all classic terms from the program analysis theory [16]. Here is a brief explanation of them:

flow-sensitivity The analysis attaches to each program point a *points-to graph* that models the heaps created by the execution paths that end in that point. We contrast this with a *flow insensitive* analysis, which would compute a single points-to graph for the entire program.

object allocation site model All objects created by executions of a given object allocation instruction from the source program are modeled by the same node.

forward dataflow analysis The analysis uses the control flow of a method to propagate information from the entry point of the method to its exit point. In contrast, a reverse dataflow analysis would use the reverse flow of the method to propagate information from the exit point of the method to its entry point.

may-analysis The points-to graph that our analysis computes for a given program point models the *union* of the heaps created by *all* the execution paths that end in that point. For example, if some path creates a heap reference, the analysis will report it, even if some other paths do not create it. A *must*-analysis would model the *intersection* of those heaps; in the previous example, it would report

only the references which are created along all the execution paths which end in that point.

compositionality The analysis analyzes each method once[1], without knowing its calling context. The analysis of a method produces a parameterized result that is later instantiated for each call site where that method might be called.

analysis of incomplete programs The analysis is able to analyze parts of the program that call methods whose code is unavailable, e.g., native methods, or for which we do not know the calling context.

Analysis Scope

As we mentioned previously, for each program point, the analysis computes a points-to graph that models the heap at that point. The points-to graph computed by the analysis for a program point will also give information about which objects escape, i.e., are reachable from outside the analysis scope. In the case of the pure intra-procedural analysis, which do not analyze any call site, the scope of the analysis is the method m that contains that program point, up to the program point. The inter-procedural analysis extends this scope from method m to m plus the methods it transitively calls.

2.2 General Mathematical Notations

In the presentation of the analysis and in the rest of this thesis, we use the following mathematical notations: "$\{a_0, a_1, \ldots, a_k\}$" represents the set that contains the distinct elements $a_0, a_1, \ldots, a_k$. "$[a_0, a_1, \ldots, a_k]$" is a 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" is the set of all the lists of elements from the set A.

"$\{a_i \mapsto b_i\}_{i \in I}$" denotes a partial function f such that $f(a_i) = b_i, \forall i \in I$, and f is undefined in the other points. If $f : A \to B$ is a function from A to B, $a \in A$, and $b \in B$, the notation "$f\,[a \mapsto b]$" denotes a function that has the value b in the point a, and behaves exactly like f in the other points of the domain A.

If $\mu \subseteq A \times B$ is a relation from A to B, then, if $a \in A$, $\mu(a) = \{b \mid \langle a, b\rangle \in \mu\}$. Furthermore, if $S \subseteq A$, $\mu(S) = \bigcup_{a \in S} \mu(a)$.

2.3 Program Representation

Although the analysis handles the full Java language [3], for simplicity, we use just a subset of it. We indicate the missing parts and how the analysis handles them.

Figure 2-1 presents the notations used in the representation of the analyzed program. The analyzed program is composed of a set of classes, *Class*, which define a

[1]Recursive methods still require a fixed-point algorithm.

set of methods, *Method*. Among these methods, there is a distinguished one, denoted m_{main}, which is the root of the program: the execution of the program starts with the first instruction from this method. Each method has a list of parameters, a set of local variables, and a body consisting of a list of instructions. We make the following simplifiying convention: if a method m has $k = arity(m)$ parameters, then these parameters are, in order, $p_0, p_1, \ldots, p_{k-1}$. For non-static methods, p_0, i.e., the first parameter, is the "`this`" parameter. In Java, parameters are just a special case of local variables, i.e., $p_i \in V$.

The body of the method m is a list of instructions from the set *Statement*. Each instruction has a unique label $lb \in Label$ obtained by pairing m with the address a of the instruction, which is simply the index of the instruction into m's list of instructions. The indexing starts from zero; the first instruction of method m has the label $\langle m, 0\rangle$. Given a label $lb = \langle m, a\rangle$, the auxiliary function *next* computes the label immediately succeeding lb, $next(lb) = \langle m, a+1\rangle$. The statement associated with the label lb is obtained with the help of the auxiliary function $P : Label \to Statement$.

Also, each class $C \in Class$ has a set of fields $fields(C) = \{f_0, f_1, \ldots, f_{q-1}\}$. Some of the fields are static: a static field is attached to the class C, not to a specific instance of C. We can view the static fields as some sort of global variables. In our notation, we do not make the distinction between static and non-static fields, but we have distinct instructions for manipulating them. We ignore all access modifiers: `public`, `private`, etc.

The field [] is an artificial field; it cannot appear in a normal program, but the analysis uses it internally to model array cells. We discuss more on this issue in Section 2.4.

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

Figure 2-1: Sets and notations for the program representation

We suppose that prior to the analysis, the program is parsed and converted into an intermediate representation containing only the instructions from Figure 2-2.

A Java program manipulates data of primitive types — integers, booleans, floating point types — and non-primitive (or pointer) types: simple objects and arrays. To simplify the presentation, we consider only the instructions that might manipulate pointers; all the other instructions are assimilated with NOP (and their transfer functions will be the identity function). For example, we ignore all the instructions for integer arithmetic, and the instructions that copy an integer value from one local variable to another. We also suppose that procedures have only pointer-type parameters and return values. Extending the analysis to handle primitive types is straightforward.

COPY	$v_1 = v_2$
NEW	$v =$ `new` C
NEW ARRAY	$v =$ `new` $C[k]$
NULLIFY	$v =$ `null`
STORE	$v_1.f = v_2$
STATIC STORE	$C.f = v$
ARRAY STORE	$v_1[i] = v_2$
LOAD	$v_2 = v_1.f$
STATIC LOAD	$v = C.f$
ARRAY LOAD	$v_2 = v_1[i]$
IF	`if` $(\ldots)$ `goto` a_t
CALL	$v_R = v_0.s(v_1, \ldots, v_j)$
RETURN	`return` v
THREAD START	`start` v
NOP	`nop`, other irrelevant instructions

Figure 2-2: Instructions in the analyzed program

We give a formal semantics of the instructions at the beginning of the correctness proof in Chapter 4. At this point, we give just the informal semantics. A COPY instruction "$v_1 = v_2$" copies the value of the local variable v_2 into v_1. A NEW instruction "$v =$ `new` C" creates a new object of class C; all the non-static fields of non-primitive type from the newly-created object are initialized to `null`. Similarly, the NEW ARRAY instruction creates a new array object. If the cells of the newly created array are of a non-primitive type, they are all initialized to `null`. A NULLIFY instruction simply sets a variable to `null`.

The STORE instructions store values into memory. There are three types of STORE instructions: the first two assign a value to a non-static, respectively static field; the third one, ARRAY STORE, stores a value in a specific array cell. The LOAD instructions read values from memory. There are three types of LOAD instructions: the first two read a non-static, respectively static field; the third one reads the value

stored in a specific array cell.

Normally, the intra-procedural control flow goes from label $lb = \langle m, a \rangle$ to label $next(lb) = \langle m, a + 1 \rangle$. This normal flow of control can be altered with the help of an IF instruction. An IF instruction is basically a conditional jump to a specific address in the same method; the specific condition tested by an IF instruction is not important for the analysis and we intentionally left it unspecified.

The inter-procedural flow of control is handled by CALL and RETURN. CALL calls a virtual method: it calls the method named s, defined for the class C of the receiver object, i.e., the object pointed to by v_0. There are two steps: the first step, called *dynamic dispatch*, determines the specific method that has to be called (the callee); the second step does the actual call. The parameter passing semantics is call-by-value; however, keep in mind that each parameter has pointer type, i.e., it is the address of an object. RETURN returns the control from the callee back into the caller. Also, a RETURN instruction "`return` v" copies the value of the local variable v from the callee into the variable v_R from the caller, where v_R is the variable that receives the result of the corresponding CALL instruction "$v_R = v_0.s(v_1, \ldots, v_j)$".

To simplify the presentation, we did not give a specific instruction for calls to static methods; these calls are simpler because they skip the dynamic dispatch phase. As the analysis is not concerned with the dynamic dispatch (we suppose a conservative call graph is constructed before the analysis), the calls to static methods will be treated similarly to the virtual calls. We also did not give any mechanism for calls to native methods. However, the analysis will deal with the more general case of unanalyzable calls, calls that we cannot or do not want to analyze[2]. Once we give the analysis for the virtual calls, it is easy to extend it to cover the calls to static and native methods. It is also trivial to extend it for methods that do not return any result, i.e., methods whose result has type `void`.

Unlike Java, where a thread is started by calling a special native method — `public native java.lang.Thread.start()` — we start a thread by executing the THREAD START instruction "`start` v". This, too, does not reduce the generality of our analysis: each call site which may call that native method can be considered a potential thread start instruction and treated as such.

In a Java program, no local variable can be used before being initialized. We suppose this is already checked by an earlier stage in the compiler.

In Figure 2-2, we did not present any I/O instructions. These instructions are supposed to be implemented as calls to specific native methods.

Termination and Result In our model, a Java program terminates when all its threads of execution terminate. It is possible to have infinite executions. In this thesis, we are not directly interested in the result of a program but rather in the heap structures manipulated by the program during its execution. As we prove later, after we do the optimizations enabled by the pointer analysis, the program manipulates

[2]For example, because the code of the callees is not available or their analysis would be too expensive.

the same heap structures, and so, it will ultimately obtain the same result. This guarantees that our optimizations preserve the semantics.

We assume that we have a control flow graph for each method appearing in the program and a call graph for the entire program. These structures are frequently used by any realistic compiler, and it is reasonable to suppose that they are constructed by a phase that precedes the analysis. For the sake of completeness, we briefly describe both of them below.

Control Flow Graph The control flow graph of a method m conservatively approximates all the possible execution paths inside m. Given a method m, the *control flow graph* of m, denoted CFG_m, is a directed graph $CFG_m = \langle A, E \rangle$, where

- the set of vertices, A, is the set of labels appearing in the body of m plus one special label $entry_m$ for m's starting point and one special label $exit_m$ for m's exit point, i.e., $A = labels(m) \cup \{entry_m, exit_m\}$;
- the set of arcs $E \subseteq A \times A$ contains:
  - an arc from $entry_m$ to $\langle m, 0 \rangle$ (the label of the first instruction from m);
  - an arc from lb to $next(lb)$, for every label lb such that the instruction at label lb is not a RETURN;
  - an arc from lb to $\langle m, a_t \rangle$ for every label lb such that the instruction at label lb is "`if` $(\ldots)$ `goto` a_t";
  - an arc from lb to $exit_m$, for every label lb such that the instruction at label lb is a RETURN.

The two special labels $entry_m$ and $exit_m$ guarantee that m contains an isolated entry point (no arc points to it) and a single exit point. The control flow graph has the property that 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 suppose that the intermediate representation handles exceptions *explicitly*. Such a representation adds an explicit test in front of each instructions which might throw an exception (e.g. a pointer dereferencing), and after each call which might propagate an exception. So, every dynamic execution path translates into a path in the statically constructed control flow graph.

In a control flow graph CFG_m, we define the functions $pred, succ : A \to \mathcal{P}(A)$ that return the predecessors, respectively successors of a given label lb, as follows:

$$\begin{aligned} pred(lb) &= \{lb' \mid \langle lb', lb \rangle \in E\} \\ succ(lb) &= \{lb' \mid \langle lb, lb' \rangle \in E\} \end{aligned}$$

Call Graph A call graph is a function $CG : Label \rightarrow \mathcal{P}(Method)$ that, for each CALL instruction, gives the set of methods that that CALL instruction might call in a concrete execution. The call graph is *conservative*: for a given CALL at label lb, $CG(lb)$ contains *all* the methods that might be called by that CALL in *any* given concrete execution, As in general, a call graph construction algorithm is imprecise, the set $CG(lb)$ might contain some other methods, too. The smaller this set is, the more precise the call graph algorithm is said to be.

There are many algorithms for computing the call graph of a program [1, 9]. The correctness of the analysis is not affected by the precision of the call graph used in a particular implementation of the analysis. However, for achieving good precision, a good call graph is recommended. It is also possible to combine the pointer analysis and the call graph construction; we have not investigated this issue yet.

2.4 Sets and Notations

Figure 2-3 presents the sets and the notations used by the analysis. In the following paragraphs, we give an intuitive explanation of our abstractions.

Nodes

We introduce the *node* abstraction to model objects created during the concrete execution of the program. A node from the abstract semantics models one or more objects from the concrete execution. There are several disjoint kinds of nodes: inside nodes, placeholder nodes, result nodes, static nodes, and the special node $n_{\texttt{null}}$ for modeling the special pointer value `null`.

Inside nodes model objects created by the analyzed method m or by one of the methods transitively called by it. We introduce one inside node n^I_{lb} for each label that corresponds to a NEW or an ARRAY NEW instruction. n^I_{lb} models all the objects created by executions of the instruction from label lb. Note that as lb can be in a loop, n^I_{lb} might represent more than one object.

Suppose we analyze a method m. Method m is not restricted to manipulate only objects created by itself. For compositionality reasons, the analysis of one method cannot look "outside" it. Hence, we need some placeholder nodes to model those objects that are manipulated by m, but which might be created outside it. Later, in the inter-procedural analysis we try to find the actual nodes that each placeholder node stands for. There are two disjoint kinds of placeholder nodes: the parameter nodes and the load nodes.

The parameter nodes are used to model the objects that are passed as parameters to a given activation (i.e., invocation) of a method m. To obtain a compositional analysis, we assume that the parameters are maximally unaliased by introducing one parameter node $n^P_{m,i}$ for each formal parameter p_i of the analyzed method m. If later in the inter-procedural analysis we discover that two parameters are aliased, we merge the aliased placeholder nodes by mapping them to the same nodes. Intuitively, it is always easier to treat the more general case of maximally unaliased parameters

$$
\begin{array}{rcl}
n & \in & Node = INode \uplus NodePlaceholder \uplus RNode \uplus \\
 & & SNode \uplus \{n_{\texttt{null}}\} \\
n^I_{lb} & \in & INode = \{\texttt{INSIDE}\} \times Label \\
n^I_{lb} & \stackrel{\text{def}}{\equiv} & \langle \texttt{INSIDE}, lb \rangle \\
 & & \\
 & & NodePlaceholder = LNode \uplus PNode \\
 & & \\
n^L_{lb} & \in & LNode = \{\texttt{LOAD}\} \times Label \\
n^L_{lb} & \stackrel{\text{def}}{\equiv} & \langle \texttt{LOAD}, lb \rangle \\
 & & \\
n^P_{m,i} & \in & PNode = \{\texttt{PARAM}\} \times Method \times \mathbb{N} \\
n^P_{m,i} & \stackrel{\text{def}}{\equiv} & \langle \texttt{PARAM}, m, i \rangle \\
 & & \\
n^R_{lb} & \in & RNode = \{\texttt{RETURN}\} \times Label \\
n^R_{lb} & \stackrel{\text{def}}{\equiv} & \langle \texttt{RETURN}, lb \rangle \\
 & & \\
n^S_C & \in & SNode = \{\texttt{STATIC}\} \times Class \\
n^S_C & \stackrel{\text{def}}{\equiv} & \langle \texttt{STATIC}, C \rangle \\
 & & \\
I^a & \in & InsideEdges^a = \mathcal{P}(\ (Node \setminus \{n_{\texttt{null}}\}) \times Field \times Node\ ) \\
O^a & \in & OutsideEdges^a = \mathcal{P}(\ (Node \setminus \{n_{\texttt{null}}\}) \times Field \times LNode\ ) \\
L^a & \in & LocVar^a = V \rightarrow \mathcal{P}(Node) \\
 & & \\
G & \in & PTGraph^a = InsideEdges^a \times OutsideEdges^a \times LocVar^a \times \\
 & & \mathcal{P}(Node) \times \mathcal{P}(Node)
\end{array}
$$

Figure 2-3: Sets and notations used by the analysis

and to merge nodes later, than to work with fewer nodes and to split later! For a given activation of a method, a parameter node models a single object: the object sent as actual argument. We use the notation $ParamNodes(m)$ to indicate the set of parameter nodes of method $m \in Method$:

$$ParamNodes(m) = \{n^P_{m,i} \mid 0 \leq i \leq arity(m) - 1\}$$

Another reason for introducing placeholders is the need to cope with the LOAD instructions that read references from *escaping* nodes. Escaping nodes are those nodes that might be accessed from outside the current activation of m. As the analysis does not know what the other parts of the program might write in that object, it does not know what is actually read in the concrete execution. In this case we use the second kind of placeholders, the load nodes. We introduce at most one load node n^L_{lb} for each label lb that corresponds to a LOAD instruction, i.e., $P(lb) =$ "$v_1 = v_2.f$". n^L_{lb} models the objects read by that instruction from objects that might be accessed from outside the current activation of method m. As a LOAD instruction might be in a loop, a load node might model multiple objects.

Some methods cannot be analyzed, because their code is not available, as is the case of the native methods, or because their analysis would take too long. However, when m calls such a method, we need to model somehow the object that is returned from it. We use a special kind of nodes, the result nodes, exactly for this purpose. If the instruction at label lb corresponds to an unanalyzable CALL[3], the result node n^R_{lb} models the object that might be returned by the called method. Due to the loops, a result node might model multiple objects.

To model static fields, we create a static node n^S_C for each class C that has static fields. This static node acts as a "wrapper" for the static fields of the associated class C. For example, if a class C has several static fields, they will be modeled as normal fields of the node n^S_C. The existence of a single static node for a class is the consequence of the fact that the static fields are associated with the class. Other ideas are possible for modeling the static fields, e.g., introducing a special mathematical structure for each static field, but we prefer the static nodes because they make the static fields look similar to the normal, non-static fields, and lead to a more uniform formalism.

Nodes and threads In Java, with the exception of the main thread, each thread of execution corresponds to a "thread object", i.e., a normal object that implements the `Runnable` interface. As objects are modeled by nodes, the threads that are different from the main thread are modeled by nodes, too.

[3]We do not give any formal definition of an unanalyzable CALL. We simply suppose that, due to various reasons, some CALLs are unanalyzable.

Edges

Suppose that in the heap created by the execution of the program, the field f of object o_1 has as value the address of object o_2. In this case we say that o_1 points to o_2 through the field f, or, in other words, we have the heap reference $\langle o_1, f, o_2 \rangle$. Heap references are modeled in the analysis by edges: triples from the set $(Node \setminus \{n_{\texttt{null}}\}) \times Field \times Node$. In the previous example, the heap reference $\langle o_1, f, o_2 \rangle$ will be modeled by the heap edge $\langle n_1, f, n_2 \rangle$, where n_1 is a node that models o_1 and n_2 is a node that models o_2. There are two kinds of edges: inside edges, which model the heap references created by the analyzed scope, and outside edges, which model the heap references read by the the analyzed scope from objects that might be accessed from outside it. An outside edge always ends in a load node. As a program cannot write to nor read from a null pointer, an edge cannot start from $n_{\texttt{null}}$.

Modeling of array cells In Java, arrays are just a special kind of objects. By consequence, they are modeled by nodes. If an array has cells of non-primitive type, the values stored in these cells are addresses of heap objects. To model these references, we use the artificial field [] that represents all the cells of an array. We do not distinguish between the different cells of an array. For example, if object o_1 is an array modeled by node n_1 such that $o_1[0]$ is a reference to the object o_2, and $o_1[1]$ is a reference to the object o_3, then the field [] of node n_1 points to nodes n_2 and n_3, where node n_2 models o_2 and node n_3 models o_3.

If we extend the analysis to include some form of integer analysis and treat the array cells in a more precise way, we have to use more than one field for representing the array cells. However, we should be aware that this is not a trivial thing to do: there is currently only one pointer analysis which distinguishes the array fields [18].

State of the Local Variables

The state of the local variables of the analyzed method m is modeled as a function $L^a : V \to \mathcal{P}(Node)$ that assigns to each local variable v the set of nodes that v might point to. Note that although at any given moment in the program execution, v has only one value, $L^a(v)$ might have more than one element. This is both due to the analysis imprecisions, and to the fact that the statically computed $L^a(v)$ must model the value of v on all possible dynamic execution paths that reach that program point.

We also use L^a to keep track of the objects that might be returned from the analyzed method m. For this purpose, we introduce a dummy variable v_{ret} that is put to point to the objects that are returned from m[4]. $L^a(v_{ret})$ is the set of nodes that might be returned from m. An alternative is to have a separate set of returned nodes; for simplicity reasons, we decided to minimize the number of components of a points-to graph.

[4]This is equivalent to splitting a RETURN instruction `return` v in three phases: copy v into v_{ret}, pop the stack frame of the callee, and copy v_{ret} into v_R, where v_R is the variable from the caller that has to receive the result of the corresponding CALL instruction: "$v_R = v_0.s(v_1, \ldots, v_{k-1})$."

Points-to Graphs

The points-to graph that the analysis computes for a given program point conservatively models the program state created by any execution path that reaches that point. We are interested in the values of the local variables that have pointer types, and in that part of the heap that the analyzed scope creates or accesses. For example, we are not interested in the specific value of an integer variable. The points-to graph models the heap and the state of the local variables and also offers information on which nodes escape from the analyzed scope.

Formally, a points-to graph $G \in PTGraph^a$ is a five-tuple

$$G = \langle I^a, O^a, L^a, S^a, U^a \rangle$$

consisting of the set of *inside* edges I^a, the set of *outside* edges O^a, the abstract state of the local variables L^a, the set of the threads started by the analyzed scope S^a, and the set of nodes that directly escaped into an unanalyzed method U^a.

We have already examined the first three components. We now explain the other two. The objects created by the analyzed scope might escape it (i.e., be manipulated from outside the analyzed scope), because they are reachable from one of the following *escapability sources*:

- Static fields (which behave as global variables);
- Caller (i.e., they are reachable from the parameters or from the object that is returned to the caller);
- Threads started by the analyzed scope;
- Methods called by unanalyzable CALL instructions from the analyzed scope; as we do not know what these methods do, we have to be very cautious and consider the object potentially accessible to the entire program (e.g., one of these methods might store a reference to the object in a static field).

As an object might be manipulated from outside the analyzed scope only if it is reachable from the outside, escapability is basically a reachability property and has to be propagated along the heap references. The heap references already being modeled by the sets of inside and outside edges, we just need to keep track of the nodes that model the objects that directly escape (the sources of escapability).

The static fields are already modeled by the static nodes. The nodes directly reachable from the caller are already modeled too: the objects received as parameters are modeled by the parameter nodes and the objects that might be returned to the caller are modeled by the nodes from $L^a(v_{ret})$. The sets S^a and U^a have the purpose of modeling the remaining two source of escape information. The set S^a models the threads started by the analyzed scope. The set U^a models the set of objects that are passed as arguments to the unanalyzed methods. The objects received as return values from these unanalyzed methods are already modeled by nodes from *RNode*.

Escape Predicate

Having modeled both the sources of escape information and the heap edges, we can propagate the escape information along the inside and the outside edges to see if a particular node escapes from the analyzed scope of not. Given a points-to graph, we can define an *escape predicate* that tells whether a node is reachable from one of the sources of escape information. Before giving the definition of the escape predicate, we give a formal definition of reachability from a set or roots over a set of edges.

Definition 1 (Reachability predicate). *Let $R \subseteq A$ be a set of* root *vertices from the set A and $E \subseteq A \times Field \times A$ be a set of arcs, labeled by fields. We define the predicate*

$$reachable(R, E) : A \to \{\texttt{true}, \texttt{false}\}$$

as the least fixed point of the following constraints:

$$\frac{a \in R}{reachable(R, E)(a)} \tag{2.1}$$

$$\frac{\langle a_1, f, a_2 \rangle \in E,\ reachable(R, E)(a_1)}{reachable(R, E)(a_2)} \tag{2.2}$$

The previous definition implicitly uses the usual ordering relation on boolean predicates; if $p_1, p_2 : B \to \{\texttt{true}, \texttt{false}\}$ are two boolean predicates over an arbitrary set B, then:

$$p_1 \sqsubseteq p_2 \ \text{ iff } \ \forall b \in B, p_1(b) \to p_2(b)$$

Note that we did not specify the set A of vertices. In the next definition, it will be *Node* but the definition of reachability is more general. Basically, a vertex a is reachable from the set R of roots if it is one of the roots or if it is pointed to by an arc which starts in an already reachable vertex.

Definition 2 (Escape predicate). *Given a method m with the associated parameter nodes $ParamNodes(m) = \{n^P_{m,0}, \ldots, n^P_{m,k-1}\}$, and a points-to graph $G = \langle I^a, O^a, L^a, S^a, U^a \rangle$ computed by the analysis for some program point inside m, we define the escape predicate*

$$e^a(G) : Node \to \{\texttt{true}, \texttt{false}\}$$

as follows:

$$e^a(G)(n) = reachable(N, I^a \cup O^a)(n)$$

where $N = ParamNodes(m) \cup SNode \cup L^a(v_{ret}) \cup S^a \cup U^a \cup RNode$.

Notation: We say that a node n *escapes in the points-to graph* G iff $e^a(G)(n)$ is true. Otherwise, we say that n *is captured in* G. We omit the points-to graph G when it is obvious from the context.

Ordering Relations

In the next section, we define our analysis in the Monotone Framework from [16]. In that framework, an analysis attaches an element from a property space to each program point from the program. In our case, the property space is the set $PTGraph^a$. To comply with the Monotone Framework, we need to define an ordering relation $\sqsubseteq$ on it such that $\langle PTGraph^a, \sqsubseteq \rangle$ is a join semi-lattice. Furthermore, to ensure termination of the fixed-point algorithms used for implementing our analysis, $\langle PTGraph^a, \sqsubseteq \rangle$ should respect the Ascending Chain Property: any ascending chain has to stabilize at some point.

The ordering relation on points-to graphs is defined component-wise. We first introduce ordering relations for the components of points-to graphs. A points-to graph $G = \langle I^a, O^a, L^a, S^a, U^a \rangle$ is composed of a few sets — I^a, O^a, S^a, and U^a — and a function $L^a : V \to \mathcal{P}(Node)$. The ordering relation between sets is the set inclusion relation and the associated join operation is the set union. For elements from the set of functions $LocVar^a = V \to \mathcal{P}(Node)$, we use the classic ordering between functions:

$$L_1^a \sqsubseteq L_2^a \text{ iff } \forall v \in V, L_1^a(v) \sqsubseteq L_2^a(v) \text{ i.e., } \forall v \in V, L_1^a(v) \subseteq L_2^a(v)$$

The associated join operation is

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

Definition 3 (Ordering relation on $PTGraph^a$). *Given two points-to graphs*

$$G_1 = \langle\ I_1^a,\ O_1^a,\ L_1^a,\ S_1^a,\ U_1^a\ \rangle$$
$$G_2 = \langle\ I_2^a,\ O_2^a,\ L_2^a,\ S_2^a,\ U_2^a\ \rangle$$

G_1 is said to be smaller than G_2, i.e. $G_1 \sqsubseteq G_2$ iff

$$I_1^a \subseteq I_2^a,\ O_1^a \subseteq O_2^a,\ L_1^a \sqsubseteq L_2^a,\ S_1^a \subseteq S_2^a, \text{ and } U_1^a \subseteq U_2^a$$

Lemma 1. *$\langle PTGraph^a, \sqsubseteq \rangle$ is a join semi-lattice with the associated join operator $\sqcup$ defined as follows*

$$\langle\ I_1^a,\ O_1^a,\ L_1^a,\ S_1^a,\ U_1^a\ \rangle \sqcup \langle\ I_2^a,\ O_2^a,\ L_2^a,\ S_2^a,\ U_2^a\ \rangle =$$
$$\langle\ I_1^a \cup I_2^a\ ,\ O_1^a \cup O_2^a\ ,\ \lambda v.\, (L_1^a(v) \cup L_2^a(v))\ ,\ S_1^a \cup S_2^a\ ,\ U_1^a \cup U_2^a\ \rangle$$

and the least element $\bot_{PTGraph^a} = \langle\ \emptyset,\ \emptyset,\ \lambda v.\emptyset,\ \emptyset,\ \emptyset\ \rangle$.

Proof: Trivial. □

Lemma 2. *For a given analyzed program, the join semi-lattice* $\langle PTGraph^a, \sqsubseteq \rangle$ *satisfies the Ascending Chain Property, i.e.,* $G_1 \sqsubseteq G_2 \sqsubseteq \ldots$ *implies* $\exists i$ *such that* $G_i = G_{i+1} = \ldots$.

Proof: For any given program, the number of nodes we can define is bounded: we have one parameter node for each formal parameter, one inside node for each NEW statement, at most one load node for each LOAD statement, one return node for each unanalyzed CALL and one static node for each class that has static fields. By consequence, $PTGraph^a$ is finite and $\langle PTGraph^a, \sqsubseteq \rangle$ trivially respects the Ascending Chain Condition. □

2.5 Formal Presentation of the Analysis

We define our pointer analysis in the context of the Monotone Framework for dataflow analysis from [16, Chapter 2].

Given a method m, with k parameters, $p_0, p_1, \ldots, p_{k-1}$, the analysis instance for method m computes a pair $\langle \circ A, A\circ \rangle$ of two functions $\circ A, A\circ : Label \rightarrow PTGraph^a$, such that for each label lb from the body of m, $\circ A(lb)$ is the points-to graph attached to the program point right before lb and $A\circ(lb)$ is the points-to graph attached to the program point right after lb[5].

Here are the elements that define our analysis:

- The analysis is a *forward* analysis that propagates information from the method entry point along the edges of the control flow graph. The set E of *extremal labels* contains just the entry label of m, $entry_m$, and the flow used by the analysis is the control flow graph for method m, CFG_m.

- The *extremal value* for the extremal label, i.e., the points-to graph associated with the beginning of m, is:

$$G_{init} = \langle \emptyset,\ \emptyset,\ L^a_{init},\ \emptyset,\ \emptyset \rangle$$

  where

$$L^a_{init}(p_i) = \{n^P_{m,i}\}, \forall i \in \{0, 1, \ldots, k-1\} \text{ and } L^a_{init}(v) = \emptyset \text{ otherwise.}$$

- The property lattice is $\langle PTGraph^a, \sqsubseteq \rangle$.

- The transfer functions associated with the labels are presented in Figure 2-4.

[5] In [16, Chapter 2], the authors use the notations $A_\circ, A_\bullet$. We replaced them with our own notations — $\circ A$, respectively $A\circ$ — which make the position of the program point more explicit: *before*, respectively *after* the label.

The analysis for method m can be expressed as a set of equations:

$$\begin{array}{rcl} {}^{\circ}A(lb) & = & \begin{cases} G_{init} & \text{if } lb \equiv entry_m \\ \bigsqcup\{A^{\circ}(lb') \mid lb' \in pred(lb)\} & \text{otherwise} \end{cases} \\ A^{\circ}(lb) & = & [\![lb]\!]^a({}^{\circ}A(lb)) \end{array} \tag{2.3}$$

We prove later in this section that the transfer functions are monotonic. Once we know this, we can solve the analysis equations using any of the many standard methods for monotone dataflow analysis [16], ranging from Chaotic Iteration (the easiest to implement) to Worklist Algorithms to Iterating Through Strong Components (the most efficient). The solution we obtain is the least fixed point of the equations; for historic reasons, it is called *MFP*, which stands for *Maximal Fixed Point.*

As the join operation on $PTGraph^a$ is essentially a component-wise set-union, the analysis is a *may*-analysis: the points-to graph associated with a program point inside m conservatively approximates the information that is propagated along *any* execution path that reaches that point.

2.5.1 Transfer Functions

Figure 2-4 presents the transfer functions associated with the labels from the analyzed program. The transfer function $[\![lb]\!]^a$ takes as argument the points-to graph for the program point just before label lb and returns the points-to graph for the program point right after lb. We define the functions $[\![lb]\!]^a$ on a case by case basis, based on the instruction from label lb. Figure 2-4 does not cover the case of an analyzable CALL; we study this case later in Section 2.5.2, which we dedicate entirely to the inter-procedural analysis. In the next paragraphs, we give an intuitive description of the transfer functions associated with the labels/instructions from the program.

As a general rule, assignments to variables are *destructive*, i.e., assigning something to v "removes" all the previous values of $L^a(v)$, while assignments to node fields are *non-destructive*[6]: assigning something to $n_1.f$ does not remove the existing edges that start from n_1. The reason is that nodes might represent multiple locations 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 concern different objects! For example, it is possible that the assignment to $n_1.f$ models an assignment to $o_1.f$, where object o_1 is modeled by n_1, and the edge $\langle n_1, f, n_2\rangle$ models an edge in the concrete heap that starts from the object $o_2 \neq o_1$, where o_2 too is modeled by n_1. This imprecision is due to the fact that the same node models potentially many objects, and is unavoidable in a static analysis that needs to represent a possibly infinite structure, the heap, in some finite way.

[6]An equivalent term is *weak updates*; the opposed term, *strong updates*, denotes the updates that removes the previous edges.

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

$[\![entry_m]\!]^a$ and $[\![exit_m]\!]^a$ are the identity function;
the other cases are presented below:

$P(lb)$	$[\![lb]\!]^a(G = \langle I^a, O^a, L^a, S^a, U^a \rangle)$
$v_1 = v_2$	$\langle I^a, O^a, L^a\left[v_1 \mapsto L^a(v_2)\right], S^a, U^a \rangle$
$v = \texttt{new}\ C$	$\langle I_2^a, O^a, L^a\left[v \mapsto \{n_{lb}^I\}\right], S^a, U^a \rangle$ where $I_2^a = I^a \cup \{\langle n_{lb}^I, f, n_{\texttt{null}} \rangle\}_{f \in fields(C)}$
$v = \texttt{new}\ C[k]$	$\langle I^a \cup \{\langle n_{lb}^I, [], n_{\texttt{null}} \rangle\}, O^a, L^a\left[v \mapsto \{n_{lb}^I\}\right], S^a, U^a \rangle$
$v = \texttt{null}$	$\langle I^a, O^a, L^a\left[v \mapsto \{n_{\texttt{null}}\}\right], S^a, U^a \rangle$
$v_1.f = v_2$	$\langle I^a \cup ((L^a(v_1) \setminus \{n_{\texttt{null}}\}) \times \{f\} \times L^a(v_2)), O^a, L^a, S^a, U^a \rangle$
$C.f = v$	$\langle I^a \cup (\{n_C^S\} \times \{f\} \times L^a(v)), O^a, L^a, S^a, U^a \rangle$
$v_1[i] = v_2$	$\langle I^a \cup ((L^a(v_1) \setminus \{n_{\texttt{null}}\}) \times \{[]\} \times L^a(v_2)), O^a, L^a, S^a, U^a \rangle$
$v_2 = v_1.f$	$process\_load(G, v_2, L^a(v_1) \setminus \{n_{\texttt{null}}\}, f, lb)$
$v = C.f$	$process\_load(G, v, \{n_C^S\}, f, lb)$
$v_2 = v_1[i]$	$process\_load(G, v_2, L^a(v_1) \setminus \{n_{\texttt{null}}\}, [], lb)$
`if (...) goto` a_t	$\langle I^a, O^a, L^a, S^a, U^a \rangle$ (unmodified)
$v_R = v_0.s(v_1, \ldots, v_j)$	**Case 1:** analyzable call Studied later in Section 2.5.2 --- **Case 2:** unanalyzable call $\langle I^a, O^a, L^a\left[v_R \mapsto n_{lb}^R\right], S^a, U^a \cup \bigcup_{i=0}^{j} L^a(v_i) \rangle$
`return` v	$\langle I^a, O^a, L^a\left[v_{ret} \mapsto L^a(v)\right], S^a, U^a \rangle$ where v_{ret} is the dummy variable used to store the return value of m.
`start` v	$\langle I^a, O^a, L^a, S^a \cup L^a(v), U^a \rangle$
`nop`	$\langle I^a, O^a, L^a, S^a, U^a \rangle$ (unmodified)

Figure 2-4: Definition of the family of analysis transfer functions $[\![lb]\!]^a, lb \in Label$

The two special labels we introduced, $entry_m$ and $exit_m$, do not correspond to any concrete instruction and hence do not have any impact on the program execution. The transfer function for them is naturally the identity function on the set $PTGraph^a$. This is also the case for the labels that correspond to IF, NOP, or some other instruction that does not manipulate pointers.

The processing of a NULLIFY instruction "v = `null`" simply sets v to point to the special node $n_{\texttt{null}}$. A COPY instruction "$v_1 = v_2$" makes v_1 point to all the nodes that v_2 might point to. As previously mentioned, the analysis "forgets" the previous value of $L^a(v_1)$.

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}, and records the fact that all the fields of n^I_{lb} point to $n_{\texttt{null}}$. Because assignments to node fields are non-destructive, the analysis does not remove the previous inside edges corresponding to those fields[7].

The transfer functions associated with the three STORE instructions — STORE, STATIC STORE, and ARRAY STORE — are similar. In all cases, the analysis updates the set of inside edges by adding the edges $A \times \{f\} \times B$, where the specific sets of nodes A and B depend on the STORE instruction from label lb. For a normal STORE "$v_1.f = v_2$", $A = L^a(v_1) \setminus \{n_{\texttt{null}}\}$ and $B = L^a(v_2)$; the analysis creates inside edges labeled with the field f from any node v_1 might point to, except $n_{\texttt{null}}$, to any node v_2 might point to. As the program cannot write at the address `null`, the analysis does not create edges from $n_{\texttt{null}}$. The case of an ARRAY STORE instruction "$v_1[i] = v_2$" is similar, except that we use the special field [] which models the references coming from all the cells of the array. For a STATIC STORE "$C.f = v$," A is the singleton containing the special static node n^S_C, which serves as a wrapper for all the static fields of class C, and $B = L^a(v)$.

The case of the three LOAD instructions is more complicated than the cases we have seen so far. When the program loads a reference from a node, if that node is reachable from outside the analyzed scope, the analysis does not know all the references that start from that node: some part of the program that is outside the analyzed scope might create a new reference that the analysis is unaware of. In this situation, the analysis introduces outside edges to model the references that the program reads, and a load node that is a placeholders for the actually loaded nodes.

The processing of the three types of LOAD instruction is similar: in all three cases the analysis loads a field from a set A of nodes and makes a certain target variable point to the loaded nodes. For LOAD and STATIC LOAD, the analysis loads the field f that is mentioned in the instruction. For an ARRAY LOAD, the analysis loads the special field [] that models the references coming from all the cells of an array

[7]Such edges might exist if the `new` instruction is in a loop; of course they refer to some other object, created in a previous iteration of the loop, but n^I_{lb} models that object too.

$process\_load$: $PTGraph^a \times V \times \mathcal{P}(Node) \times Field \times Label \rightarrow PTGraph^a$

PARAMETERS:

Points-to graph $G = \langle I^a, O^a, L^a, S^a, U^a \rangle$ right before the LOAD;
Local variable v that is the target of the LOAD;
Set of nodes A that the program loads from;
Field f that the program loads;
Label lb of the LOAD instruction (used to generate a load node).

RESULT:

Points-to graph after the LOAD instruction.

$$\begin{array}{rcl} \text{Let } B & = & \{n \in Node \mid \exists n_1 \in A, \langle n_1, f, n \rangle \in I^a\} \\ E & = & \{n \in A \mid e^a(G)(n)\} \end{array}$$

Case 1: $E = \emptyset$

$$process\_load(G, v, A, f, lb) = \langle I^a, O^a, L^a\left[v \mapsto B\right], S^a, U^a \rangle$$

Case 2: $E \neq \emptyset$

$$process\_load(G, v, A, f, lb) = \langle I^a, O^a_2, L^a\left[v \mapsto \left(B \cup \{n^L_{lb}\}\right)\right], S^a, U^a \rangle$$

where $O^a_2 = O^a \cup (E \times \{f\} \times \{n^L_{lb}\})$

Figure 2-5: Definition of auxiliary function *process_load*

object. The set A is obviously $L^a(v_1) \setminus \{n_{\texttt{null}}\}$ in the case of LOAD and ARRAY LOAD; for a STATIC LOAD "$v = C.f$," it is the singleton consisting of the static node n^S_C associated to the class C. Notice that as the program cannot read from the address `null`, the analysis does not load anything from $n_{\texttt{null}}$.

Figure 2-5 presents the formal definition of the auxiliary function *process_load* that does the actual work for the three LOAD instructions. *process_load* must receive the following arguments: the points-to graph G right before the LOAD operation, the set A of nodes that the program loads from, the loaded field f, the target variable v to be set, and the label lb of the LOAD instruction. In the points-to graph returned by *process_load*, the target variable v points to all the nodes that are pointed to by the field f of nodes from A. If A contains nodes that escape the analyzed scope (i.e., nodes n such that $e^a(G)(n)$), the target variable also points to the load node n^L_{lb} attached to the analyzed label/instruction. Also, in this case, the analysis introduces an outside edge from every escaping node from A to n^L_{lb}; the label of each such outside edge is the field loaded by the analyzed instruction, i.e., f.

The last three instructions that we have to discuss — unanalyzable CALL, START THREAD, and RETURN — create potentially new sources of escapability. The processing of these instructions updates one of the sets U^a, S^a, or $L^a(v_{ret})$. In the

case of an unanalyzable CALL "$v_R = v_0.s(v_1, \ldots, v_j)$", the analysis adds all the nodes pointed to by $v_0, \ldots, v_j$ to the set of nodes that are passed to unanalyzable CALLs. Also, in the points-to graph after the unanalyzable CALL instruction, v_R points to the result node n^R_{lb} that models the object that is returned by the call at label lb. For a START THREAD instruction "`start` v", the analysis adds all nodes pointed to by v to the set of potentially started nodes. Finally, for a RETURN instruction "`return` v", the dummy local variable v_{ret} is put to point to the nodes that are returned.

Example 1. Consider the following small method:

```
   C2 m(C p0) {
0:   v0 = p0.f;
1:   if (v0 == null) goto 3;
2:   return v0;
3:   v1 = new C2;
4:   p0.f = v1;
5:   return v1;
   }
```

lb	$\circ A(lb)$	$A \circ (lb)$
$entry_m$	G_0	G_0
0	G_0	G_1
1	G_1	G_1
2	G_1	G_2
3	G_1	G_3
4	G_3	G_4
5	G_4	G_5
$exit_m$	G_6	G_6

For simplicity, we use integer labels. Method `m` checks whether the field `f` of its only parameter is null or not. If it is, then it sets this field to point to a newly created object. In both cases, it returns the (possibly new) non-null value of the field `f`.

The table on the right side of the code presents the result of the pointer analysis for method `m`. Figure 2-6 presents the graphical representation of the points-to graphs $G_0, \ldots, G_6$ that appear in the aforementioned table.

As the analyzed method does not contain any loop, the computation of the analysis is straighforward: it starts with the points-to graph for the beginning of the method, and propagates the information down the flow by applying the analysis transfer functions.

In the points-to graph G_0 for the beginning of the method (Figure 2-6.a), `p0` points to the parameter node n^P_0; all the other components of G_0 are empty. The transfer function for the LOAD instruction from label 0 loads the field `f` of the escaped node n^P_0. The analysis uses the load node n^L_0 as a placeholder for the actual nodes that the instruction might read. It puts `v0` to point to n^L_0, and creates the outside edge $\langle n^P_0, \mathtt{f}, n^L_0 \rangle$. The resulting points-to graph, which is valid for the program point right after label 0, is G_1 (Figure 2-6.b). As the transfer function associated with an IF is the identity function, G_1 is also the points-to graph for the program point right before label 2 and for the program point right before label 3. The RETURN from label 2 records the fact that the method might return n^L_0 by setting the dummy variable v_{ret} to point to this node (Figure 2-6.c). On the other branch of the IF instruction, the NEW instruction from label 3 puts `v1` to point to the inside node associated with this instruction, i.e., n^I_3 (Figure 2-6.d). We supposed that the class `C2` does not have any pointer-type fields; otherwise, we would have several inside edges from n^I_3 to $n_{\mathtt{null}}$. The STORE from label 4 creates an inside edge from n^P_0 to n^I_3 (Figure 2-6.e). The RETURN from label 5 sets v_{ret} to point to n^I_3 (Figure 2-6.f). The label $exit_m$

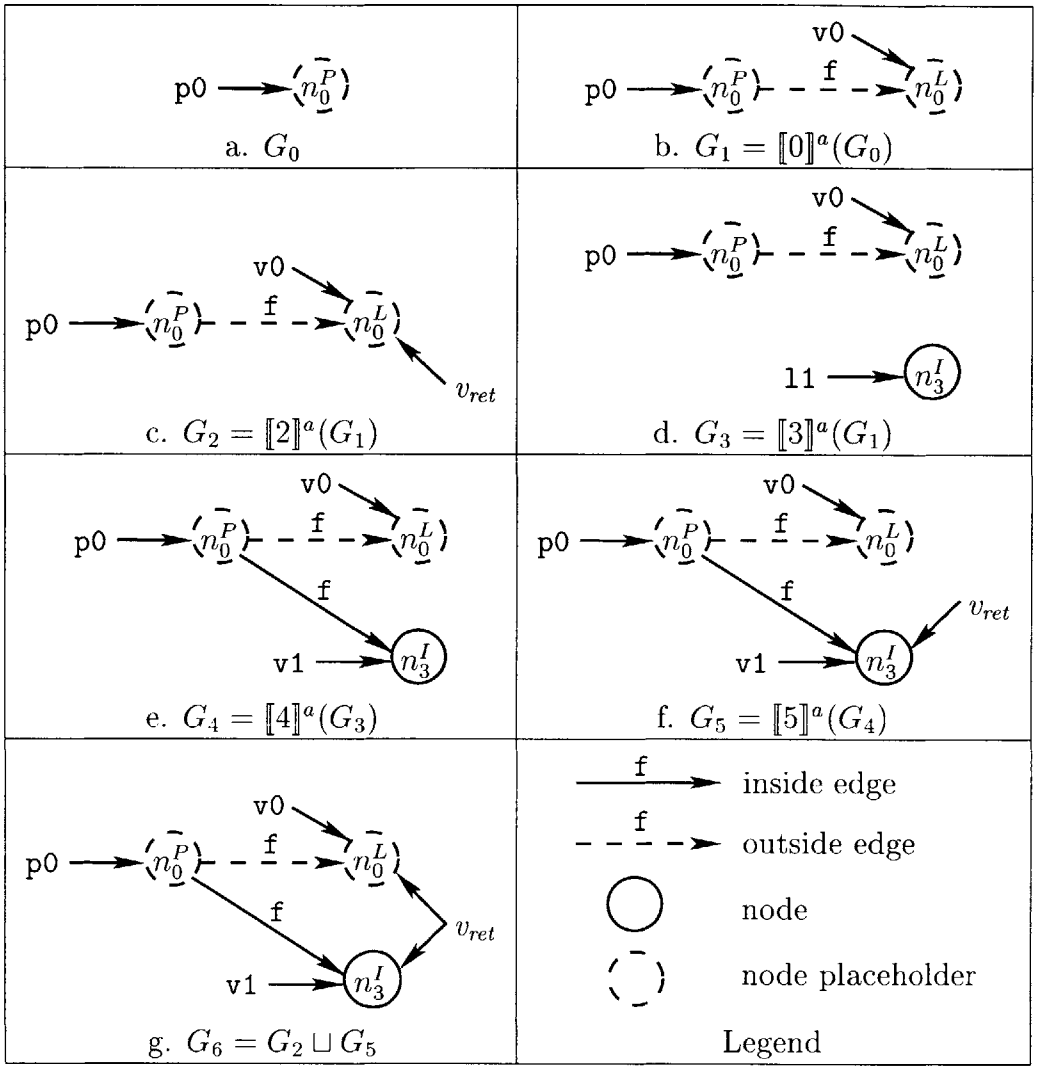


Figure 2-6: Points-to Graphs for Example 1

associated with the exit of the program has two predecessors: labels 2 and 5 that correspond to the two RETURN instructions. Accordingly, the points-to graph for the end of the method is $G_6 = G_2 \sqcup G_5$ (Figure 2-6.g). △

To comply with the Monotone Framework, we need to verify that the transfer functions are monotonic. Ignoring the case of an analyzable CALL, which has not been covered yet, we obtain the following lemma:

Lemma 3 (Monotonicity of the transfer functions). *For every label lb inside m, which does not correspond to an analyzable CALL,* $[\![lb]\!]^a : PTGraph^a \rightarrow PTGraph^a$ *is monotonic, with respect to the ordering relation from the join semi-lattice* $\langle PTGraph^a, \sqsubseteq \rangle$.

Proof hint: $[\![entry_m]\!]^a$ and $[\![exit_m]\!]^a$ are clearly monotonic; for the other labels, we do a case analysis on the type of the instruction at label lb, and we use the appropriate definition of $[\![lb]\!]^a$ from Figure 2-4. The only non-trivial cases are those of the LOAD instructions. In these cases, note that if $G_1 \sqsubseteq G_2$, then $e^a(G_1)(n) \rightarrow e^a(G_2)(n)$. □

After we present the transfer function for an analyzable CALL, we extend Lemma 3 to cover that case too.

2.5.2 Inter-procedural Analysis

In Java, after every NEW instruction, the program calls a constructor that initializes the newly created object. If we want to analyze real Java programs, our analysis has to provide some degree of inter-procedurality; otherwise, all our nodes would escape the analyzed scope and we wouldn't be able to obtain precise information about any of them. This subsection presents the transfer function for the labels that correspond to analyzable CALLs.

Suppose we have a CALL instruction at label lb, having the form

$$v_R = v_0.s(v_1, \ldots, v_j)$$

The transfer function $[\![lb]\!]^a$ for label lb receives as argument the points-to graph for the program point right before the CALL, and returns the points-to graph for the program point right after the CALL. It is defined as follows:

$$[\![lb]\!]^a(G) = \bigsqcup_{callee \in CG(lb)} interproc(G, \circ A(exit_{callee}), lb, callee) \tag{2.4}$$

For each possibly called method $callee \in CG(lb)$, the analysis uses the auxiliary function *interproc* to compute a points-to graph for the program point after the CALL, valid in the case when *callee* is called. As the statically computed call graph cannot guess which one of the possible callees is called in a specific execution of the CALL instruction, the analysis has to conservatively join the points-to graphs computed for all the possible callees.

Figure 2-7 presents the definition of the function *interproc*. It has four arguments:

- Points-to graph G for the program point right before the CALL;
- Points-to graph $G_{callee} = \circ A(exit_{callee})$ for the exit point of method *callee*;
- Label lb_c of the analyzed CALL instruction;
- Called method *callee*.

The analysis combines G and G_{callee} to compute the points-to graph valid for the program point after the CALL, in the case when *callee* is called. This computation has three steps:

$$
\begin{array}{l}
\textit{interproc}:\ \textit{PTGraph}^a \times \textit{PTGraph}^a \times \textit{Label} \times \textit{Method} \rightarrow \textit{PTGraph}^a \\
\\
\textit{interproc}(G, G_{\textit{callee}}, lb_{\mathrm{c}}, \textit{callee}) = \\
\quad \textbf{let } \mu' = \textit{mapping}(G, G_{\textit{callee}}, lb_{\mathrm{c}}, \textit{callee}) \ \textbf{in} \\
\qquad \textit{simplify}(\textit{combine}(G, G_{\textit{callee}}, \mu', v_R)) \\
\\
\quad \text{where } P(lb_{\mathrm{c}}) = \text{“}v_R = v_0.s(v_1, \ldots, v_j)\text{”}.
\end{array}
$$

Figure 2-7: Definition of function *interproc*

1. First, the analysis computes a mapping $\mu' \in \textit{Mapping} = \textit{Node} \times \textit{Node}$ that maps the nodes from $G_{\textit{callee}}$ to nodes that appear in the final graph. Note that this mapping is not necessarily the identity function: some of the placeholders from the callee (parameter and load nodes) might represent other nodes. The construction of the mapping μ' uses the points-to graph before the CALL to disambiguate as many node placeholders as possible.

2. Next, the analysis uses the mapping μ' to combine the two points-to graphs, G and $G_{\textit{callee}}$.

3. Finally, the analysis simplifies the resulting points-to graph by removing superfluous load nodes and outside edges. It removes only those parts of the graph that are not critical for the correctness, i.e., they are present there only as a result of the analysis imprecisions.

Each step is implemented with the help of an auxiliary function. These functions are, in order: *mapping*, *combine*, and *simplify*. We present them in detail in the next paragraphs.

Construction of the Node Mapping

The function *mapping* constructs a mapping $\mu' \in \textit{Mapping}$ that maps the nodes from $G_{\textit{callee}}$ to nodes that appear in the final graph. Figure 2-8 presents its formal definition.

In a first step, *mapping* constructs an intermediate mapping $\mu \in \textit{Mapping}$ that maps the node placeholders from the callee to nodes from the points-to graph for the program point right before the CALL. This step is the core of our inter-procedural analysis. We define μ as the least fixed point of the constraints 2.5, 2.6, 2.7, and 2.8.

The first two constraints initialize the mapping, the other two extend it. In a real implementation, the first two constraints will be applied to initialize the mapping and only the next two constraints will be considered by the iterations of the fixed-point algorithm.

Constraint 2.5 records the fact that the parameter node $n^P_{\textit{callee},i}$ represents the nodes actually pointed to by v_i, the i^{th} argument passed to *callee*. These nodes are

$mapping : PTGraph^a \times PTGraph^a \times Label \times Method \rightarrow Mapping$

$Mapping = Node \times Node$

PARAMETERS:

Points-to graph right before the CALL, $G = \langle I^a, O^a, L^a, S^a, U^a \rangle$;
Points-to graph from *callee*, $G_{callee} = \langle I^a_{callee}, O^a_{callee}, L^a_{callee}, S^a_{callee}, U^a_{callee} \rangle$;
Label lb_c of the CALL instruction; $P(lb_c) =$ "$v_R = v_0.s(v_1, \ldots, v_j)$";
Called method *callee*.

RESULT:

Mapping $\mu' \in Mapping$, computed as follows:

1. Let $\mu \in Mapping$ be the least fixed point of the following constraints:

$$L^a(v_i) \subseteq \mu(n^P_{callee,i}),\ \forall i \in \{0, 1, \ldots j\} \tag{2.5}$$

$$n^S_C \in \mu(n^S_C),\ \forall C \in Class \tag{2.6}$$

$$\frac{\langle n_1, f, n_2 \rangle \in O^a_{callee},\ \langle n_3, f, n_4 \rangle \in I^a,\ n_3 \in \mu(n_1)}{n_4 \in \mu(n_2)} \tag{2.7}$$

$$\frac{\begin{array}{c}\langle n_1, f, n_2 \rangle \in O^a_{callee},\ \langle n_3, f, n_4 \rangle \in I^a_{callee}, \\ ((\mu(n_1) \cup \{n_1\}) \cap (\mu(n_3) \cup \{n_3\})) \setminus \{n_{\texttt{null}}\} \neq \emptyset, \\ (n_1 \neq n_3) \vee (n_1 \in LNode)\end{array}}{\mu(n_4) \cup (\{n_4\} \setminus ParamNodes(callee)) \subseteq \mu(n_2)} \tag{2.8}$$

2. Extend μ to obtain μ' as follows:

$$\mu'(n) = \begin{cases} \mu(n) & \text{if } n \in ParamNodes(callee) \\ \mu(n) \cup \{n\} & \text{otherwise} \end{cases}$$

Figure 2-8: Definition of function *mapping*

the nodes from the set $L^a(v_i)$. Constraint 2.6 completes the initialization by mapping each static node to itself[8]: $n_C^S \in \mu(n_C^S)$, $\forall C \in Class$.

The other two constraints extend the mapping by matching outside edges (i.e., LOAD instructions that read references) against inside edges (which model heap edges created by STORE instructions). As more and more mappings are discovered, the *mapping* function goes deeper and deeper into the points-to graphs, and more edge matchings can be applied; accordingly, the fixed point algorithm repeatedly applies the constraints 2.7 and 2.8 till no further progress is possible and it reaches the least fixed point.

Constraint 2.7 matches the outside edge $\langle n_1, f, n_2 \rangle \in O^a_{callee}$ from the callee against the inside edge $\langle n_3, f, n_4 \rangle \in I^a$ from the caller, under the condition that n_1 might represent n_3: $n_3 \in \mu(n_1)$. Figure 2-9.a presents a graphic representation of this situation. As n_1 might be n_3, the outside edge read from n_1 might be the inside edge $\langle n_3, f, n_4 \rangle$, and the load node n_2 might be the node n_4. The analysis extends the mapping to record this fact. This constraint deals with the cases when the callee reads references from the heap structures created by the caller.

Constraint 2.8 matches an outside edge from the callee against an inside edge from the callee. This constraint deals with the aliasing present in the calling context. Suppose we have the outside edge $\langle n_1, f, n_2 \rangle \in O^a_{callee}$ that corresponds to a LOAD from the escaped node n_1, and an inside edge from the caller $\langle n_3, f, n_4 \rangle \in I^a_{callee}$. If $(\mu(n_1) \cap \mu(n_3)) \setminus \{n_{\texttt{null}}\} \neq \emptyset$, then n_1 and n_3 might represent the same node $n_5 \in \mu(n_1) \cap \mu(n_3)$, $n_5 \neq n_{\texttt{null}}$. The case $n_5 = n_{\texttt{null}}$ is not interesting, because no edge can start from $n_{\texttt{null}}$: the program cannot write to/read from the address `null`. Figure 2-9.b presents a graphic representation of this situation. As the analysis for the callee was not aware of the aliasing between n_1 and n_3, it could not detect the fact that the LOAD operation that created the outside edge might have read the reference modeled by the inside edge. Once we know that n_1 and n_3 might be placeholders for the same node, we are able to infer that n_2 might be n_4. We record this fact by enforcing $\{n_4\} \in \mu(n_2)$ (the set difference is a technical detail that we explain later). Also, as n_4 is a node from the callee, it might be a node placeholder that represents some other nodes (it might be in the domain of the μ relation). Therefore, node n_2 might represent not only n_4 but also some nodes represented by this node. The analysis updates the mapping to record this too, such that $\mu(n_4) \subseteq \mu(n_2)$.

The same reasoning is valid in the case when n_1 might be a placeholder for n_3, i.e., $n_3 \in \mu(n_1)$ (Figure 2-9.c), or n_3 might be a placeholder for n_1, i.e., $n_1 \in \mu(n_3)$ (Figure 2-9.d). Instead of having three rules, we extended the condition $(\mu(n_1) \cap \mu(n_3)) \setminus \{n_{\texttt{null}}\} \neq \emptyset$ as follows:

$$((\mu(n_1) \cup \{n_1\}) \cap (\mu(n_3) \cup \{n_3\})) \setminus \{n_{\texttt{null}}\} \neq \emptyset$$

[8]In a real implementation, only the static nodes that actually appear in G_{callee} need to be considered.

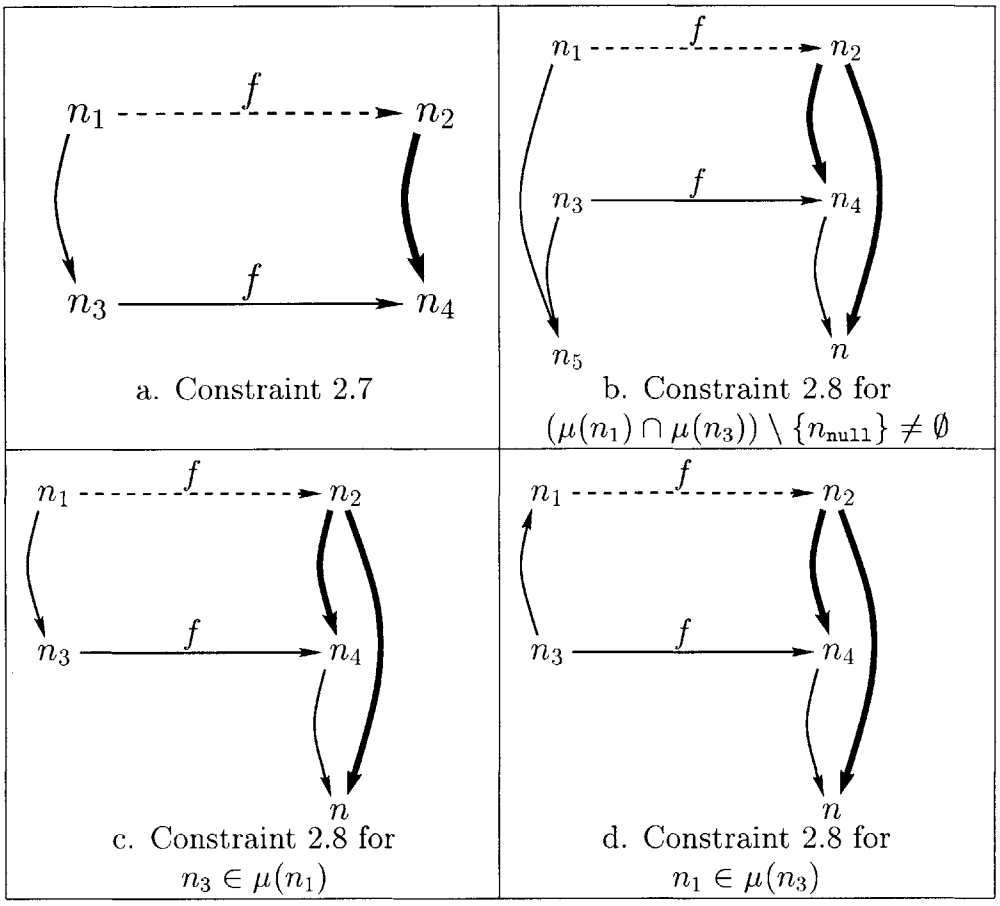


Figure 2-9: Graphic representation of Constraint 2.7 and Constraint 2.8

To obtain a reasonable precision, we do not want to have too many fake mappings. The set difference "$\setminus\{n_{\texttt{null}}\}$" from the previous condition is a first attempt to reduce the fake mappings, based on the observation that a program cannot write to/read from a null pointer. The third part of the precondition reduces the applicability of Constraint 2.8 even further:

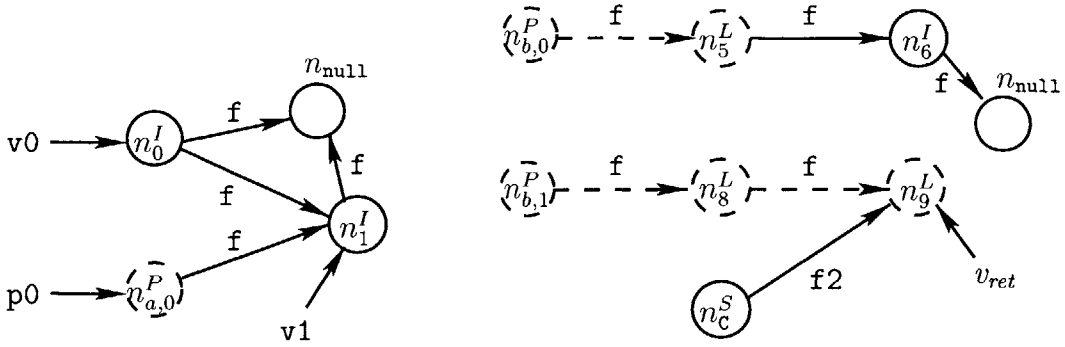
$$(n_1 \neq n_3) \vee (n_1 \in LNode)$$

The correctness proof will show that this condition does not prevent the analysis from detecting all the real mappings. Intuitively, if the condition is not satisfied, the analysis of method *callee* would have already detected that the LOAD operation that corresponds to the outside edge might load the node n_4.

Once we have the mapping μ that disambiguates the node placeholders from *callee*, we extend it to obtain the final mapping μ' by mapping each non-parameter node to itself. Intuitively, the inside and the return nodes from the callee model objects manipulated by the callee during its execution, and we want these nodes to be present in the points-to graph after the CALL: $n \in \mu'(n)$. The mapping μ already totally disambiguates the parameter nodes, i.e., the analysis identified all the nodes they might stand for. So, they are unnecessary in the resulting points-to graph. As a consequence, we do not apply the previous map extension for the parameter nodes. This is also the reason why in Constraint 2.8, instead of $\mu(n_4) \cup \{n_4\} \subseteq \mu(n_2)$, we have $\mu(n_4) \cup (\{n_4\} \setminus ParamNodes(callee)) \subseteq \mu(n_2)$: as we do not want the callee parameter nodes to appear in the resulting points-to graph, we avoid creating mappings to them.

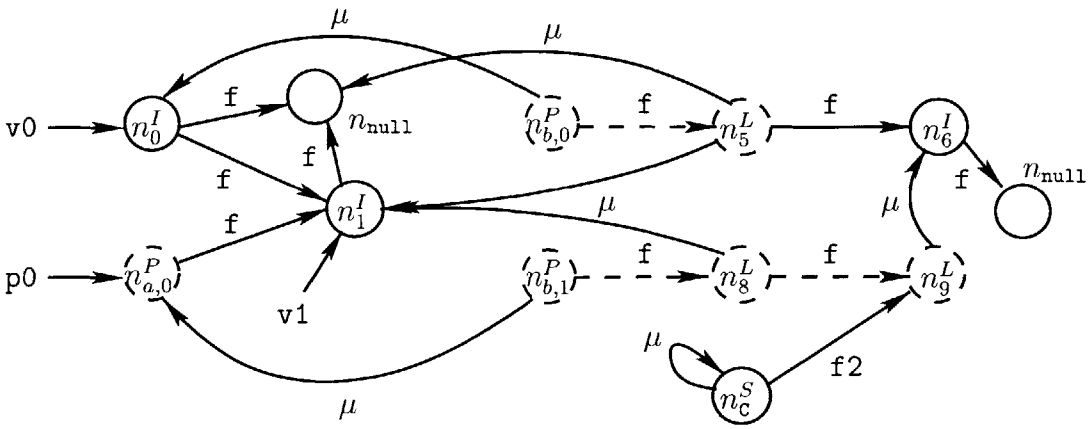
However, we cannot say the same thing about the load nodes: some of these nodes have to be present in the resulting points-to graph. Each load node is a placeholder for the nodes that a specific LOAD instruction might have loaded from an escaped node. But that escaped node might remain an escaped node even in the points-to graph after the CALL. For example, consider the case of a node n that is reachable from one of the parameter nodes of *callee*, $n^P_{callee,i}$ and suppose that while constructing the mapping μ, we found out that that $n^P_{callee,i}$ stands for a node from the caller — the method m — that is reachable from one of m's parameter nodes. In this case, n escapes even in the resulting points-to graph. Therefore, we have to preserve the load nodes in the resulting graph, in order to represent the nodes that might have been loaded from the escaped nodes. In a first phase, we preserve all the load nodes. We see later in this section how to remove some of them.

Example 2. Consider the following piece of code:



a. Points-to graph before the call

b. Points-to graph for the end of b



c. Node mapping μ

Figure 2-10: Points-to graphs and node mapping for Example 2

```
   C a(C p0) {
0:   v0 = new C;
1:   v1 = new C;
2:   v0.f = v1;
3:   p0.f = v1;
4:   v2 = v0.b(p0);
     ...
   }
```

```
   C b(C p0, C p1) {
5:   v0 = p0.f;
6:   v1 = new C;
7:   v0.f = v1;
8:   v1 = p1.f;
9:   v1 = v1.f;
10:  C.f2 = v1;
11:  return v1;
   }
```

For simplicity, we use integer labels. Method a creates a new object at label 0, and sets both the field f of its single parameter and the field f of the newly created object to point to another new object, which it creates at label 1. Figure 2-10.a presents the points-to graph computed by the analysis for the program point right before the CALL instruction from label 4. As usual, we use solid arcs for inside edges and dashed arcs for outside edges. We also use continuous circles for nodes and dashed circles for node placeholders.

The CALL from label 4 invokes the method b with the arguments n_0^I and $n_{a,0}^P$. It

is easy to understand what `b` does by studying the points-to graph computed by the analysis for the exit point of `b`, which we present in Figure 2-10.b. We did not present the values of the local variables of `b`, because the callee local variables are insignificant for the mapping construction, and the inter-procedural analysis in general. In its first three instructions, `b` reads the `f` field of its first parameter, which is modeled by $n^P_{b,0}$, and creates a new reference from the loaded object, modeled by n^L_5, to a newly created object, modeled by n^I_6. This creates the upper "chain" of nodes from Figure 2-10.b. In the next three instructions, i.e., two LOADs and a STATIC STORE, method `b` sets the static field `C.f2` to point to the object pointed to by `p1.f.f`, which is modeled by n^L_9. This creates the bottom "chain" of nodes from Figure 2-10.b.

Notice that method `b` was analyzed under the assumption that its arguments are maximally unaliased. In particular, the analysis of `b` did not know that for our CALL `p1.f` and `p2.f` are aliased. As n^L_5 and n^L_8 might be the same node, the reference that `b` reads at label 9 might be the reference that it created at label 7; therefore, the placeholder n^L_9 might be n^I_6. The construction of the mapping relation μ identifies all these cases.

Figure 2-10.c presents the mapping μ. This mapping is constructed as follows. Constraint 2.5 maps $n^P_{b,0}$ to n^I_0 and $n^P_{b,1}$ to n^I_0. Next, Constraint 2.7 matches the outside edge $\langle n^P_{b,0}, \mathtt{f}, n^L_5\rangle$ from the callee against the inside edge $\langle n^I_0, \mathtt{f}, n^I_1\rangle$ from the caller and maps n^L_5 to n^I_1. Similar applications of Constraint 2.7 map n^L_5 to $n_{\mathtt{null}}$ and n^L_8 to n^I_1.

At this moment, the analysis is aware that the placeholders n^L_5 and n^L_8 might represent the same node. By consequence, Constraint 2.8 matches the outside edge $\langle n^L_8, \mathtt{f}, n^L_9\rangle$ against the inside edge $\langle n^L_5, \mathtt{f}, n^I_6\rangle$, and maps n^L_9 to n^I_6. Note that in this last constraint application, both edges are from the callee points-to graph, i.e., they are created for the *same* analysis scope.

At this point, the analysis cannot extend the mapping any longer, i.e., it reached the least fixed point of the mapping constraints. The analysis extends μ to obtain μ'. The following table presents the final values of μ and μ' for the relevant nodes, i.e., the nodes that appear in the points-to graph for the end of `b`:

n	μ	μ'
$n^P_{b,0}$	$\{ n^I_0 \}$	$\{ n^I_0 \}$
$n^P_{b,1}$	$\{ n^P_{a,0} \}$	$\{ n^P_{a,0} \}$
n^L_5	$\{ n^I_1, n_{\mathtt{null}}\}$	$\{ n^I_1, n_{\mathtt{null}}, n^L_5 \}$
n^L_8	$\{ n^I_1 \}$	$\{ n^I_1, n^L_8 \}$
n^L_9	$\{ n^I_6 \}$	$\{ n^I_6, n^L_9 \}$
n^I_6	$\emptyset$	$\{ n^I_6 \}$
$n^S_{\mathtt{C}}$	$\{ n^S_{\mathtt{C}} \}$	$\{ n^S_{\mathtt{C}} \}$
$n_{\mathtt{null}}$	$\emptyset$	$\{ n_{\mathtt{null}}\}$

$\triangle$

$$
\begin{array}{l}
\textit{combine}:\ PTGraph^a \times PTGraph^a \times Mapping \times V \rightarrow PTGraph^a \\
\\
\textit{combine}(\langle I^a, O^a, L^a, S^a, U^a\rangle, \langle I^a_{callee}, O^a_{callee}, L^a_{callee}, S^a_{callee}, U^a_{callee}\rangle, \mu', v_R) = \\
\quad \textbf{let}\ \ I^a_2 = I^a \cup I^a_{callee}[\mu'] \\
\qquad\quad O^a_2 = O^a \cup O^a_{callee}[\mu'] \\
\qquad\quad L^a_2 = L^a\,[v_R \mapsto \mu'(L^a_{callee}(v_{ret}))] \\
\qquad\quad S^a_2 = S^a \cup \mu'(S^a_{callee}) \\
\qquad\quad U^a_2 = U^a \cup \mu'(U^a_{callee})\ \ \textbf{in} \\
\qquad\quad \langle I^a_2, O^a_2, L^a_2, S^a_2, U^a_2\rangle \\
\\
\quad \text{where} \\
\qquad I^a_{callee}[\mu'] = \bigcup_{\langle n_1, f, n_2\rangle \in I^a_{callee}} (\mu'(n_1) \setminus \{n_{\texttt{null}}\}) \times \{f\} \times \mu'(n_2) \\
\qquad O^a_{callee}[\mu'] = \bigcup_{\langle n, f, n^L\rangle \in O^a_{callee}} (\mu'(n) \setminus \{n_{\texttt{null}}\}) \times \{f\} \times \{n^L\}
\end{array}
$$

Figure 2-11: Definition of function *combine*

Combining the Points-to Graphs

Once we have the mapping μ', we combine the points-to graph for the program point right before CALL, G, with the points-to graph from the end of *callee*, G_{callee}, to obtain the points-to graph for the point right after the CALL. The combination is done by the function *combine*, presented in Figure 2-11. *combine* has four arguments:

- points-to graph $G = \langle I^a, O^a, L^a, S^a, U^a\rangle$ for the program point right before the CALL;
- points-to graph $G_{callee} = \langle I^a_{callee}, O^a_{callee}, L^a_{callee}, S^a_{callee}, U^a_{callee}\rangle$ for the exit point of *callee*;
- mapping μ';
- variable v_R to store the value returned by the callee.

Intuitively, *combine* returns the union between G and the projection of G_{callee} through the mapping μ'.

The equations from the definition of *combine* require some explanation. The heap references that existed before the CALL might exist after the CALL too, and so, I^a should be included in the set of inside edges in the points-to graph after the call, I^a_2. If *callee* created the heap edge $\langle n_1, f, n_2\rangle$, where n_1 may be any of the nodes from $\mu'(n_1)$, and n_2 may be any of the nodes from $\mu'(n_2)$, then *callee* might have created any of the edges from the set $(\mu'(n_1) \setminus \{n_{\texttt{null}}\}) \times \{f\} \times \mu'(n_2)$ and all these edges should appear in I^a_2. As usual, we do not introduce edges starting in $n_{\texttt{null}}$. We obtain

the following expression for I_2^a:

$$I_2^a = I^a \cup I^a_{callee}[\mu']$$

Similary, for the set of outside edges O^a after the CALL, the analysis takes the union of the set of outside edges right before the CALL, O^a, and the *semi*-projection[9] through μ' of the outside edges from the end of *callee*, O^a_{callee}. It is natural to ask why the analysis does not do a full projection, as in the case of the inside edges, i.e., why the analysis does not project the target of an outside edge. Here's why: an outside edge $\langle n, f, n^L_{lb}\rangle$ from the callee models the action of loading the field f from the node n, action done by the LOAD instruction at label *lb*. Moreover, n escaped from *callee*, and the placeholder n^L_{lb} was introduced to model the nodes that might have been read in that instruction. As n may be any of the nodes from $\mu'(n)$, the read operation may take place from any of these nodes, hence the need for projecting the node n. However, n^L_{lb} has the same meaning: it models the nodes read in the instruction at label *lb*. As the analysis introduces at most one load node per LOAD instruction, and not one load node per each possible source of the read, it has to keep n^L_{lb} unchanged.

The abstract state of the local variables after the CALL, L_2^a, is pretty much like the state before the CALL, L^a, except that now v_R — the local variable that receives the value returned from the callee — must point to the nodes returned from *callee*. In G_{callee}, the dummy variables v_{ret} points to these nodes. Accordingly, the analysis takes the set of nodes $L^a_{callee}(v_{ret})$, projects it through μ', and puts v_R to point to the resulting set:

$$L_2^a = L^a\,[v_R \mapsto \mu'(L^a_{callee}(v_{ret}))]$$

The projection operation is necessary because some of the returned nodes might be placeholders from *callee* that μ' disambiguates.

The set S_2^a of started threads is the union of the set of threads that were started before the CALL, S^a, and the projection through μ' of the set of threads started by *callee*. Intuitively, if *callee* started a node n and n can be any of the nodes from the set $\mu'(n)$, then *callee* might have started any of those nodes, and we have to enforce $\mu'(S^a_{callee}) \subseteq S_2^a$. The case of U_2^a is identical.

Example 3. (Example 2 continued) Figure 2-12 presents the result of the *combine* function in the context of Example 2.

The construction of the combined points-to graph is straightforward. Notice that in the resulting graph, the parameter nodes of the callee, i.e., the method `b`, disappeared. Notice also that in the resulting graph, there is no inside edge from $n_{\texttt{null}}$ to n^I_6, although $n_{\texttt{null}} \in \mu'(n^L_5)$, and $\langle n^L_5, \texttt{f}, n^I_6\rangle$ is an inside edge in the points-to graph of the callee. △

[9]The analysis projects just the start node of an outside edge, the target remains unmodified.

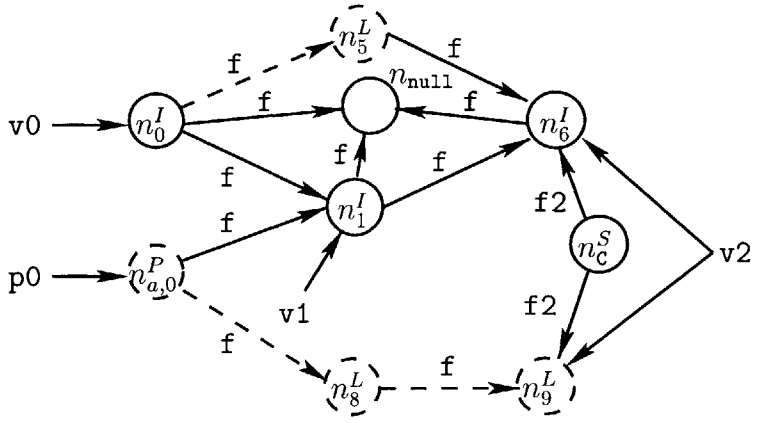


Figure 2-12: Combined points-to graph for Example 3

$$
\begin{array}{l}
\mathit{simplify} : \mathit{PTGraph}^a \to \mathit{PTGraph}^a \\
\\
\mathit{simplify}(G = \langle I^a, O^a, L^a, S^a, U^a \rangle) = \\
\quad \textbf{let } A = \{n \in \mathit{LNode} \mid \neg e^a(G)(n)\} \textbf{ in} \\
\qquad \textbf{let } I^a_s = I^a \setminus \{\langle n_1, f, n_2 \rangle \mid \{n_1, n_2\} \cap A \neq \emptyset\} \\
\qquad\qquad O^a_s = O^a \setminus \{\langle n, f, n^L \rangle \mid (\{n, n^L\} \cap A \neq \emptyset) \vee \neg e^a(G)(n)\} \\
\qquad\qquad L^a_s = \lambda v.\, (L^a(v) \setminus A) \\
\qquad\qquad S^a_s = S^a \setminus A \\
\qquad\qquad U^a_s = U^a \setminus A \textbf{ in} \\
\qquad\qquad\quad \langle I^a_s, O^a_s, L^a_s, S^a_s, U^a_s \rangle
\end{array}
$$

Figure 2-13: Definition of function *simplify*

Points-to Graph Simplification

As we mentioned earlier, it is possible to simplify a bit the points-to graph obtained by combining G and G_{callee}. The analysis removes the superfluous load nodes and outside edges. The simplification is done by the auxiliary function *simplify* from Figure 2-13.

The simplification uses two ideas:

1. The analysis introduces a load node as a placeholder for the nodes that are read from an escaped node. So, a load node inherently escapes somewhere. If in the points-to graph returned by the *combine* function, we have a captured load node n^L_{lb}, all the nodes that the program loaded n^L_{lb} from, i.e., the nodes that point to it through some outside edge from O^a_2, are captured too (remember that escapability propagates along heap edges). So, the analysis had complete information about those nodes and already identified the nodes loaded from

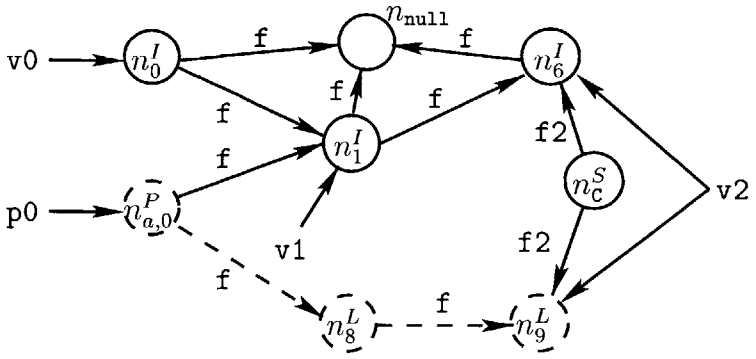


Figure 2-14: Simplified points-to graph for Example 4

them. The load node is superfluous; its presence is just the result of the analysis imprecisions. Therefore, the analysis can safely remove all the captured load nodes together with all the edges pointing to/from them. In Figure 2-13, we colect these nodes in the set A.

2. An outside edge models a reference that was loaded from an escaped node. If the source node of an outside edge is captured, the outside edge is superfluous and the analysis can remove it.

Example 4. (Example 2 continued) Figure 2-14 presents the final points-to graph for the program point right after the CALL at label 4 from Example 2. This graph is the simplified version of the graph from Figure 2-12.

In the graph from Figure 2-12, the load node n_5^L is captured. This happens because the inside node n_0^I, which is the only node that points to n_5^L, is captured, too. Therefore, the part of the program that is "outside" the analyzed scope cannot access n_0^I. Thus, the analysis has already identified all the nodes that the program might load from this node and there is no need for the placeholder n_5^L. The simplification removes n_5^L, together with all the surrounding edges. The other load nodes and outside edges cannot be removed. For instance, as $n_{a,0}^P$ escapes, the analysis cannot be sure that when the program reads the field f of $n_{a,0}^P$, it reads only n_1^I: some other thread running in parallel might write something else in that field. Therefore, the simplification preserves the load node n_8^L, and the outside edge $\langle n_{a,0}^P, \texttt{f}, n_8^L \rangle$. △

Compatibility with the Monotone Framework

After we introduced the transfer function for analyzable CALL instructions, we need to check that the transfer functions still satisfy the conditions of the Monotone Framework. For this, we need to prove that the transfer function associated with a CALL instruction is monotonic.

Lemma 4. *Let lb be the label of an analyzable CALL instruction. Then, the transfer function* $[\![lb]\!]^a$ *given by the Equation 2.4 is monotonic in its input G (the points-to graph right before CALL) and also in its implicit input(s)* $G_{callee} = \circ A(exit_{callee}), callee \in CG(lb)$ *(the points-to graphs computed by the analysis for the exit point of the possible callee(s)).*

Proof: The transfer function for a CALL is the composition of a few functions. All these functions turn out to be monotonic. *mapping* is clearly monotonic in G and G_{callee}, because its result is defined as the least fixed point of a group of monotonic set constrains. It is also trivial to show that *combine* is monotonic in G, G_{callee}, and μ'. Proving the monotonicity of *simplify* is a bit more delicate due to the set differences that appear in its definition. Consider two points-to graphs $G_1 \sqsubseteq G_2$. It is trivial to see that any node that escapes in G_1 escapes in G_2: all the escapability sources from G_1 are present in G_2 and all paths from G_1 still exist in G_2. As a consequence, all the load nodes and outside edges that are preserved in the simplified G_1, are preserved in the simplified version of G_2, too. So, *simplify* is monotonic, too. This finishes the proof of Lemma 4. □

The fact that $[\![lb]\!]^a$ is monotonic in G_1 guarantees that the intra-procedural analysis of a method terminates. Additional monotonicity of $[\![lb]\!]^a$ in its implicit input(s) $G_{callee} = \circ A(exit_{callee}), callee \in CG(lb)$, guarantees the termination of the inter-procedural analysis for the entire program.

2.6 Analysis Algorithm

The analysis that we presented as a set of dataflow equations can be computed by using any of the standard algorithms for computing a dataflow analysis. [16, Chapters 2 and 6] presents a good survey of these algorithms. As these algorithms are well studied, we do not enter into low-level technical details. Instead, we present a high-level view of an algorithm that we recommend.

We recommend using a variant of the "Iterating Through Strong Components" algorithm. The algorithm that we recommend contains an outer loop for computing the inter-procedural analysis, and nested inside it, an inner loop for computing the intra-procedural analysis.

The inter-procedural computation processes the strongly connected components of the call graph, i.e., the groups of mutually recursive[10] methods, in increasing topological order, i.e., from the leaf of the call graph to the main method. For each such set of mutually recursive methods, the algorithm uses a worklist to iterate over the set of methods till it reaches the least fixed point. At the beginning of the processing of a strongly connected component, the worklist contains all the methods from that component. In each iteration, the algorithm takes a method from the worklist and

[10]In practice, many of these groups will be singletons.

calls the inner computation, i.e., the intra-procedural computation, to analyze the method. If the points-to graph for the end of the method changed, all the possible callers of the method that are in the current strong component are added to the worklist. The inter-procedural computation for a component terminates when the worklist is empty.

The intra-procedural computation is very similar to the inter-procedural computation, except that it operates with instructions instead of methods: it processes the strongly connected components of the control flow graph of the method in decreasing topological order[11], and iterates over each such component by using a worklist.

[11]The asymmetry between the outer computation that uses the *increasing* topological order, and the inner computation that uses the *decreasing* topological order is due to the fact that although the analysis propagates through the call graph in a *bottom-up* fashion, the inter-procedural analysis is a forward analysis that propagates information *down* the flow.

Chapter 3

Analysis Applications

This chapter presents several applications of our pointer analysis: stack allocation, allocation in the thread-local heap, and synchronization removal.

3.1 Stack Allocation

If an object is not reachable after the end of its *allocating method*, i.e., the method instance that allocated it, it can be allocated in the stack frame of the allocating method instead of in the garbage collected heap. As the compiler does a static analyis of the program, it is not aware of the particular objects created during the program execution. By consequence, instead of deciding whether to stack allocate an individual objects, the compiler makes its decisions at the level of the object allocation sites, i.e., the NEW instructions. If the compiler detects that all the objects allocated by a specific NEW instruction can be stack-allocated, it replaces the NEW instruction with a small sequence of instructions that allocates an object in the stack frame.

This transformation has the following benefits:

less garbage collection overhead The objects allocated in the stack frame will be implicitly deallocated, without any execution time overhead, when the method returns and the stack rolls back.

cheap memory allocation Allocating an object from the stack requires a simple adjustment of the stack pointer — addition or subtraction, depending on the processor type.

better memory locality A method is likely to use the objects allocated by it more often than other objects. Allocating objects in the method stack frame will increase the memory locality and will also take advantage of the memory hierarchy because the stack frame will tend to be resident in the cache.

enabling more optimizations If the compiler can precisely compute the location of an object on the stack, it can generate code to access its fields directly, removing one level of memory indirection.

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, and for those allocation sites that create large objects. As a consequence, the compiler should avoid stack allocating these sites.

Simple Strategy for Stack Allocation

In the simplest case, if the inside node that models the objects created by a specific NEW instruction from a method is captured in the points-to graph computed by the analysis for the end of that method, then all these objects can be allocated in the method stack frame. Formally, consider a NEW instruction at label lb inside method m, and let $G = \circ A(exit_m)$. If $\neg e^a(G)(n^I_{lb})$, the compiler converts the NEW instruction into a sequence of instructions that allocates space from the stack.

Use of Inlining to Enhance the Stack Allocation

The compiler can significantly improve the effectiveness of stack allocation by using method inlining. Method 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.

For example, consider a method m_1 that calls method m_2. Suppose that one object o that is allocated in m_2 escapes from this method, because it is reachable from m_2's return value. If o is unreachable after the end of m_1, i.e., the caller of m_2, its lifetime is included into the lifetime of m_1's stack frame. In this case, if the compiler inlines m_2 into m_1, o can be stack allocated.

Theoretically, we can extend this technique to arbitrarily long call chains. However, abusive method inlining can dramatically increase the code size. To avoid this, we impose a limit on the length of the inlined call chains and inline only the call chains that are beneficial for stack allocation, i.e., those call chains that cause a non-empty set of inside nodes to become captured. To avoid increasing the memory consumption, the compiler should avoid inlining the CALL instructions placed in statically unbounded loops.

Additional Implementation Issues

As the stack allocation increases the stack size, programs that used to work with a given maximal stack size might terminate with a stack overflow errors after they are "enhanced" by the stack allocation optimization. A technique that eliminates this problem is to allocate the stack allocatable objects in a separate stack. When a method terminates, not only the normal stack, but also this additional stack will roll

back[1]. To avoid overflowing the second stack, the sequence of instructions that does the stack allocation should start by testing that the second stack is not full yet. If it is, the object will be allocated from the heap.

For this transformation to interoperate correctly with the rest of the system, the garbage collector must recognize stack-allocated objects. The recognition mechanism is simple — it examines the address of the object to determine if it is allocated on a stack or in the heap. During a collection, the collector still scans stack-allocated objects normally, but it does not attempt to move or collect stack-allocated objects.

3.2 Allocation in the Thread-Local Heap

The allocation in the thread-local heap is a generalization of the stack allocation. Suppose that, in addition to the global heap, we have a heap for each thread of execution. Consider an object that cannot be stack allocated. If its lifetime is included in the lifetime of the *allocating thread*, i.e., the thread that allocates it, the program can allocate the object in the heap that corresponds to the allocating thread. When that thread terminates its execution, its heap is atomically collected without any additional overhead from the garbage collector. This optimization is particularly effective for the applications that create many short-lived threads of execution.

To apply this optimization, the compiler needs to detect the object creation sites that create only objects that are local to their allocating thread. For this task, we apply the same idea as for detecting whether inlining a method can enhance the stack allocation. More specifically, suppose we have a NEW instruction at label *lb* inside method m. We consider the corresponding inside node n^I_{lb}, and check that on any reverse call path[2] that starts in m, n^I_{lb} becomes captured at some level. The only difference between this case and the case of the stack allocation via method inlining is that now, as we do not do any inlining, we can go back much deeper into the call graph. To cope with cycles in the call graph, i.e., mutually recursive methods, we still limit the length of the call chain, but this limit is much bigger than the limit that we use for the stack allocation via method inlining. For each NEW instruction that satisfies the condition, the compiler converts the NEW instruction into a code sequence that allocates space from the thread-local heap. If the thread-local heap does not have enough free space, allocation proceeds as in the case of a regular NEW.

In addition, the compiler needs to insert a code sequence before each thread start instruction to create the heap attached to that thread. Also, the compiler needs to add a code sequence to the thread exit code; this last code sequence will atomically deallocate the thread-local heap when the thread terminates. As in the case of the stack allocation, the garbage collector needs to be modified to identify the objects that are allocated in the thread-local heaps. This can be accomplished by using some address-based mechanism. The garbage collector scans the objects allocated in the

[1]The compiler needs to generate code for setting the pointer of the second stack to its previous value only for those methods that might do some stack allocation.

[2]"Reverse" means going from the callee to the caller.

thread-local heaps but it does not attempt to move or collect them.

This optimization is very effective for programs that create many short-lived threads of execution but may cause other programs to run out of memory. For example, consider a program with a single thread where all the objects are obviously thread-local. In this case, no deallocation or garbage collection will ever take place! In order to apply this optimization in a safe and beneficial way, the compiler requires user-provided design information about the application.

3.3 Synchronization Removal

The compiler can allocate an object in a thread-local heap only if the object is not reachable from outside its allocating thread. As this object is manipulated by a single thread, all the lock acquire/release operations executed on it will always succeed.

The compiler can use this observation to optimize the program as follows. For each allocation site that can be modified to allocate objects in the thread-local heap, the compiler adds a short sequence of instructions that sets a special flag in the newly allocated object. Also, the compiler modifies each lock acquire/release to test whether the object it is performed on has this flag set or not. If yes, then the operation succeeds immediately; otherwise, it proceeds in its usual way. To preserve the semantics of the Java memory model on machines that implement weak memory consistency models, the compiler inserts a memory barrier before the test.

This optimization makes the synchronization operations cheaper when performed on some objects but more expensive when performed on the rest of objects. If the number of the objects that have the flag set is very small, or a synchronization operation is very inexpensive, the "optimized" program might actually be slower than the original one. However, there are cases when this optimization is very effective. A distributed computing environment is a perfect example of such a case. In this environment, synchronizations require network traffic; therefore, they are orders of magnitude more expensive than a flag test.

Chapter 4

Correctness Proof

This chapter presents a correctness proof for the analysis and for the optimizations that use the analysis results. The major part of the chapter deals with the correctness of the stack allocation optimization. We also discuss the correctness of the other two optimizations we presented in Chapter 3: allocation in the thread-local heap and synchronization removal. Most of the chapter is dedicated to the proof of the following theorem:

Theorem 5 (Correctness of the stack allocation hints). *Consider a method m, and let $G = \langle I^a, O^a, L^a, S^a, U^a \rangle$ be the points-to graph that the pointer analysis computes for the exit point of m, i.e., $G = \circ A(exit_m)$. Let n^I_{lb} be an inside node correspoding to the NEW instruction at label lb from method m, or from one of its callees. If $\neg e^a(G)(n^I_{lb})$, then:*

- **Correctness of the basic stack allocation:** *If lb is a label from m, then each time an instance of m executes the NEW instruction from label lb, it is safe to allocate the newly created object in the stack frame of that instance of m.*

- **Correctness of the stack allocation enhanced by method inlining:** *If lb is a label from callee, one of the methods transitively called by m, then each time an instance of m transitively calls callee, for each execution of the NEW instruction from label lb, it is safe to allocate the newly created object in the stack frame of that instance of m.*

Note that the text of the theorem is not very formal. In particular, we do not say what we mean by "it is safe". Intuitively, it is safe to allocate an object in a stack frame iff the lifetime of the object is included into the lifetime of the stack frame. All these notions will become more clear along the proof.

During the proof of this theorem we also prove results which describe how the nodes and the edges from the points-to graphs model the objects and the heap references from the concrete execution.

Proof outline

To study the correctness of the analysis, we first specify a precise semantics for the analyzed language. This semantics allows us to replace the intuitive view we have about the execution of a program with a precise, mathematical view. To separate this semantics from some other, higher-level semantics, we call it the *concrete* semantics. The concrete semantics precisely defines the possible executions of a program[1]. The core of the concrete semantics is the transition relation between concrete states.

To simplify the proofs, we work with *SmallJava*, a subset of the instructions from Figure 2-2. Although it is a simplified version of Java, *SmallJava* contains all the features which are important for the analysis. We believe that extending the proof to handle the rest of the instructions is just a matter of time and space, which does not affect the proof ideas.

In order to prove the correctness of our pointer analysis, we need to investigate the link between the points-to graphs produced by the analysis and the concrete states constructed by the concrete semantics. Unfortunately, it seems that due to the big difference between the concrete semantics and the pointer analysis, we cannot investigate the link between them in a single step.

The analysis is different from the concrete semantics in three aspects:

1. The analysis uses the "object creation site" model to abstract the objects created in the concrete execution of the program.

2. Unlike the concrete semantics, the analysis does not "step into" the methods called by the analyzed method. Instead, it uses the points-to graphs computed for the end of the callees to pass directly from the points-to graph for the program point right before a CALL instruction to the points-to graph for the program point right after it (by using the inter-procedural analysis).

3. The analysis is performed statically; its results should be valid for *all* the possible concrete executions.

We introduce an intermediate layer between the concrete semantics and the pointer analysis, the *abstract semantics*. The abstract semantics takes one of the possible concrete executions and constructs, for each *interesting point*[2] of the execution, an abstract state which models the concrete state in a way which is very similar to the pointer analysis. A common property of the abstract semantics and the analysis is that they both use the "object creation site" model. However, the abstract semantics is close to the concrete semantics in the other two aspects: it "steps into" the callees and is not computed statically, i.e., it refers to a specific execution. Intuitively, the abstract semantics is a pointer analysis-like instrumentation of a concrete execution trace.

[1]Due to the possibility of having multiple threads of execution, there might be many possible executions for the same program.

[2]The precise definition of these "interesting points" is not important now; we present it later.

This intermediate layer allows us to split the proof in two halfs: first, we study the way the abstract semantics models the concrete semantics and prove some valuable results about this modeling relation. As the abstract semantics turns out to be very close to the pointer analysis for the methods which do not contain any analyzable CALL instructions, we can interpret this first half as a correctness proof for the intra-procedural analysis. Next, we show that the pointer analysis safely approximates the abstract semantics of *any* possible concrete execution. We can view this second part as a correctness proof for the inter-procedural analysis. By composing the two parts, we finally characterize the relation between the analysis and the concrete semantics of our analyzed language.

The rest of this chapter is structured as follows. In Section 4.1 we present the concrete semantics of *SmallJava*. In Section 4.2, we present the second layer of our proof, the abstract semantics. We investigate the relation between the concrete and the abstract semantics in Section 4.3. That section ends with a first sufficient condition for safe stack allocation. Next, in Section 4.4 we prove that the pointer analysis is a safe approximation of the abstract semantics. In Section 4.5 we combine the results of Section 4.3 and Section 4.4 to prove the correctness of the three optimizations enabled by our analysis. In Section 4.6 we discuss the modeling relation between the points-to graphs that the analysis computes and the possible heaps from a concrete execution. We conclude this chapter in Section 4.7 with a discussion on several proof details.

4.1 Concrete Semantics

To simplify the proofs, instead of analyzing the semantics of the full Java language, we restrict ourselves to a subset of it, called *SmallJava*. *SmallJava* contains those Java instructions that manipulate pointers. To make the language realistic, we also consider the IF instruction that allows us to handle the intra-procedural control flow, the two instructions for managing the inter-procedural control flow — CALL and RETURN — and the THREAD START instruction that starts a new thread of execution. Figure 4-1 presents the *SmallJava* instructions.

4.1.1 Sets and Notations

Figure 4-2 presents the sets and notations that we use in the definition of the concrete semantics of *SmallJava*. These notations are an extension of those presented in Figure 2-1, which we used in Section 2.3 to describe the program representation.

The state of the program at a given moment during its execution is a triple containing a thread agenda $A \in \mathit{ThreadAgenda}$, a heap $H \in \mathit{Heap}$ and a type function $TY \in \mathit{OTypes}$:

$$\Xi = \langle A, H, TY \rangle$$

COPY	$v_1 = v_2$
NEW	$v =$ `new` C
NULLIFY	$v =$ `null`
STORE	$v_1.f = v_2$
LOAD	$v_2 = v_1.f$
IF	`if` $(\ldots)$ `goto` a_t
CALL	$v_R = v_0.s(v_1, \ldots, v_{j-1})$
RETURN	`return` v
THREAD START	`start` v

Figure 4-1: *SmallJava* instructions

The thread agenda A maintains the state of the different threads of execution of the program. The heap H records the references between the objects created by the program. Finally, the type function TY assigns to each object its type (i.e., class); this function is introduced only for the purpose of dynamic dispatch.

During its execution, a program might create objects by executing NEW instructions. We emphasize the fact that each time the program executes such an instruction, it creates a new, fresh object. Although in a Java Virtual Machine, the garbage collector might free some heap space from time to time and two objects might share (at different moments in time), the same memory space, the two objects are distinct, each one having its own identity. In addition to this "normal" objects, the set *Object* contains two special elements: $o_{\texttt{null}}$ and o_{main}. We use $o_{\texttt{null}}$ to model the null pointers. We explain the purpose of o_{main} later in this section.

The heap H is a curryfied function that attaches to a given object o_1 and a given field f, the object $o_2 = H(o_1)(f)$ that the field f of o_1 points to. Considering the heap to be a function has the advantage of emphasizing the fact that any field of any object points to at most one object. A heap H is a partial function: we are not interested by the value of $H(o_1)(f)$ for objects o_1 that were not created yet, or for innexistent fields f.

Notation: For convenience, we sometimes use the notation $\langle o_1, f, o_2 \rangle \in H$ instead of $H(o_1)(f) = o_2$. However, keep in mind that for any heap H, object o_1 and field f, there is at most one object o_2 such that $\langle o_1, f, o_2 \rangle \in H$. We also write that $\langle o_1, f, o_2 \rangle \in H$ is a "heap reference", a "heap edge", or a "concrete heap edge".

The thread agenda A maintains the local state of each thread of execution. A is a function that attaches to a thread identifier t the stack that represents the local state of the corresponding thread. To simplify the notation, the identifier of a thread is the thread object itself. This is possible because in *SmallJava*, the program starts a new thread of execution by executing `start` on an object that implements the

$$
\begin{aligned}
\Xi &\in State = ThreadAgenda \times Heap \times OTypes \\
o &\in Object = \{o_{\texttt{null}}, o_{main}, o_0, o_1, \ldots\} \\
A &\in ThreadAgenda = ThreadId \rightarrow JavaStack \\
t &\in ThreadId = Object \\
J &\in JavaStack = \texttt{list of } (LocVar \times Label) \\
L &\in LocVar = V \rightarrow Object \\
lb &\in Label = Method \times Address \\
H &\in Heap = Object \rightarrow Field \rightarrow Object \\
TY &\in OTypes = Object \rightarrow Class \\
T &\in Trace = Date \rightarrow State \\
d &\in Date = \mathbb{N} \\
getMeth &: Class \times MethodName \rightarrow Method
\end{aligned}
$$

Figure 4-2: Sets and notations for the concrete semantics

interface `Runnable`, i.e., an object that has a method called "run"[3]. Also, as the program cannot execute `start` twice on the same object, the thread identifiers are distinct. To handle the main thread, which is the only one that the Java Virtual Machine starts without using the previously described mechanism, we introduce a dummy object/thread identifier $o_{main} \in Object = ThreadId$.

The stack of a thread t is a list of stack frames, each stack frame corresponding to a method from the current call chain in that thread. A stack frame is a pair composed of the state of the local variables for the corresponding method and the current address inside that method. The state of the local variables of a method is a function L from local variables to objects from the heap. In this chapter, we ignore any datatypes except pointers; it is straightforward to extend the concrete semantics to handle the case when local variables can have primitive (i.e., integer, boolean etc.) values. In our notation, we sometimes indicate the top stack frame, i.e., the frame of the currently executed method of thread t by writing $J = \langle L, lb \rangle : J_{tail}$.

The last component of a state is a function that assigns to each heap object its type, i.e., class. This function is used when the program calls a method named s on an object o. In this case, the program needs to do a dynamic dispatch, i.e., to identify the method to call. The dynamic dispatch has two steps: first, the TY function returns the class of o, $C = TY(o)$; next, the auxiliary function $getMeth$ uses C to provide the method $m = getMeth(C, s)$.

[3]We have already commented in Section 2.3 on the equivalence between the instruction `start` and the call to the special native method `java.lang.Thread.start()`.

4.1.2 Concrete Semantics Transitions

A concrete execution trace T of a program is a series of states, indexed by date. We use a discreet model of time: a date is a natural number, $Date = \mathbb{N}$. Throughout the proof, we interchangeably use the terms "date", "time", and "moment".

Any trace T starts with the initial state $T(0) = \Xi_0$ that has a single thread — the main thread — whose stack contains a single frame, which corresponds to the main method m_{main}. To simplify the things, we suppose that m_{main} does not have any parameter, i.e., all parameters are hard-coded into the program. Formally:

$$\Xi_0 = \langle\ A\,[t \mapsto [\ \langle\ \{\}\ ,\ \langle m_{main}, 0\rangle\ \rangle\ ]\ ],\ \{\},\ \{\}\ \rangle \tag{4.1}$$

where $t = o_{main}$ is the dummy object for the main thread. No local variable of the main method has been initialized yet. The current label inside the main method is its first label, $\langle m_{main}, 0\rangle$. Finally, as no object has been created yet, the initial heap and object type function are not defined for any object.

At each step, the concrete semantics selects a thread t from the thread agenda agenda and executes its current instruction. Formally, if the stack of thread t is $J = \langle L, lb\rangle : J_{tail}$, then the concrete semantics executes the instruction $P(lb)$ from label lb. As the initial thread agenda contains only the main thread, the first executed instruction is always the instruction found at label $\langle m_{main}, 0\rangle$. However, the thread selection is non-deterministic and so, in general we cannot say anything about the next instructions. A thread terminates when its root method (the `run()` method of the thread object) returns. If no further transition is possible (because all the threads terminated), the concrete execution stops. We allow the possibility of infinite executions.

Throughout the proof, we adopt the following notation convention: the state Ξ_d represents the state at moment d, i.e., the state right after the d^{th} instruction and right before the $(d+1)^{th}$ instruction. The execution of the $(d+1)^{th}$ instruction is responsible for the transition $\Xi_d \Rightarrow \Xi_{d+1}$.

Figure 4-3 presents the transition relation $\Rightarrow$ for the concrete semantics. Due to the selection of the thread t, the transition relation is inherently non-deterministic. The transition relation is defined function of the instruction that the concrete semantics executes. In the next paragraphs, we explain the processing done for each type of instruction.

In the case of a COPY instruction "$v_1 = v_2$" the transition updates the local state of the topmost method of the thread t such that the local variable v_1 points to whatever v_2 points to. Similarly, a NULL instruction $v =$ `null` sets v to point to the special object $o_{\texttt{null}}$. In the case of NEW instruction "`new` C", the transition creates a fresh object o, updates TY to reflect o's class, and extends H to put all the fields of o to point to $o_{\texttt{null}}$. LOAD simply puts v_2 to point to the object pointed to by the field f of the object pointed to by v_1; in the new state, v_2 points to $H(o_1)(f)$. An IF instruction breaks the normal flow of execution if its condition is satisfied: instead of going from label lb to the consecutive label $next(lb)$, as is normally the case, the

Instruction $P(lb)$	Transition
COPY $v_1 = v_2$	$\langle A\,[t \mapsto \langle L, lb\rangle : J]\,, H, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle L\,[v_1 \mapsto L(v_2)]\,, next(lb)\rangle : J]\,, H, TY\rangle$
NEW $v =$ **new** C	$\langle A\,[t \mapsto \langle L, lb\rangle : J]\,, H, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle L\,[v \mapsto o]\,, next(lb)\rangle : J]\,, H_2, TY_2\rangle$ where o is a fresh object, $H_2 = H\big[o \mapsto \{f \mapsto o_{\texttt{null}}\}_{f \in fields(C)}\big]$ and $TY_2 = TY\,[o \mapsto C]$
NULLIFY $v =$ **null**	$\langle A\,[t \mapsto \langle L, lb\rangle : J]\,, H, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle L\,[v \mapsto o_{\texttt{null}}]\,, next(lb)\rangle : J]\,, H, TY\rangle$
STORE $v_1.\mathrm{f} = v_2$	$\langle A\,[t \mapsto \langle L, lb\rangle : J]\,, H, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle L, next(lb)\rangle : J]\,, H_2, TY\rangle$ where $H_2 = H[L(v_1) \mapsto H(L(v_1))\,[f \mapsto L(v_2)]]$
LOAD $v_2 = v_1.\mathrm{f}$	$\langle A\,[t \mapsto \langle L, lb\rangle : J]\,, H, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle L\,[v_2 \mapsto H(L(v_1))(f)]\,, next(lb)\rangle : J]\,, H, TY\rangle$
IF **if** $(\ldots)$ **goto** a_t	$\langle A\,[t \mapsto \langle L, lb\rangle : J]\,, H, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle L, lb_2\rangle : J]\,, H, TY\rangle$ where $lb = \langle m, a\rangle$ and $lb_2 = \begin{cases} \langle m, a_t\rangle & \text{if the condition is true} \\ next(lb) & \text{otherwise} \end{cases}$
CALL $v_R = v_0.s(v_1, \ldots, v_j)$	$\langle A\,[t \mapsto \langle L, lb\rangle : J]\,, H, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle L_{callee}, \langle callee, 0\rangle\rangle : \langle L, next(lb)\rangle : J]\,, H, TY\rangle$ where $callee = getMeth(s, TY(L(v_0)))$ $L_{callee} = \{p_i \mapsto L(v_i)\}_{0 \le i \le j}$
RETURN **return** v	$\langle A\,[t \mapsto \langle L_{callee}, lb\rangle : \langle L, lb_{ret}\rangle : J]\,, H, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle L\,[v_R \mapsto L_{callee}(v)]\,, lb_{ret}\rangle : J]\,, H, TY\rangle$ where v_R is the variable in which to store the result of the corresponding method call. --- $\langle A\,[t \mapsto [\langle L, lb\rangle]]\,, H, TY\rangle \Rightarrow \langle A, H, TY\rangle$
THREAD START **start** v	$\langle A\,[t \mapsto \langle L, lb\rangle : J]\,, H, TY\rangle \Rightarrow$ $\langle A\,[t \mapsto \langle L, next(lb)\rangle : J]\,[L(v) \mapsto J_{startee}]\,, H, TY\rangle$ where $startee = getMeth(\text{“}run\text{”}, TY(L(v)))$ $J_{startee} = [\langle\{p_0 \mapsto L(v)\}, \langle startee, 0\rangle\rangle]$

Figure 4-3: Transition relation $\Rightarrow$ for the concrete semantics

execution jumps to address a_t from the same method.

The transition for STORE is a bit difficult to understand at a first look. Intuitively, it updates the heap to make $H(o_1)(f) = o_2$ where $o_1 = L(v_1)$ and $o_2 = L(v_2)$. It does not change the value of H for other combinations of locations and fields are unchanged.

As previously explained, the transition for a CALL instruction uses $getMeth$ and TY to call the method named s of the object $L(v_0)$, i.e., the method $callee = getMeth(s, TY(L(v_0)))$. It creates a new stack frame for method *callee*, where the *callee*'s formal parameters point to the objects that are actually sent: $p_i \mapsto L(v_i), 0 \leq i \leq j$. When the concrete semantics selects the thread t again, it will execute the first instruction of *callee*.

The transition rule for a THREAD START instruction `start` v adds a new thread to the thread agenda. The thread identifier of the new thread is the "started" object: $t = L(v)$. The topmost, and only, method of this newly created thread is the method *startee*, which is the method named `run()` of the object $L(v)$: $startee = getMeth(\text{"}run\text{"}, TY(L(v)))$. The only formal parameter of this method, the `this` parameter, points to the thread object $L(v)$.

The transition for a RETURN instruction pops the topmost stack frame and passes the returned value into the caller. There is a single exception from this rule: the case of a RETURN from the root method of a thread. The instruction that started that method was not a CALL but a THREAD START. Hence, there is no caller stack frame to return the result to. Instead, such a RETURN instruction terminates the execution of its thread t. In the concrete semantics transition, we remove the thread t from the thread agenda. In this case, the returned value is not used. It is possible to add a special RETURN without value; in this thesis, we preferred working with a minimal instruction set.

Notation: If we work in the context of a trace T, we sometimes write $\Rightarrow_T$ instead of simply $\Rightarrow$ to indicate that we refer to a transition that is actually done in the trace T, not just a possible transition.

Observation: For the sake of simplicity, we did a number of additional simplifications:

- The concrete semantics that we presented does not handle the errors that might appear during the execution. In particular, the concrete semantics does not prevent the execution of instructions that read from/write to the object $o_{\texttt{null}}$. We work only with programs that avoid these situations, with the help of explicit checks before each instruction that might commit these errors. As a consequence, in the concrete heaps that we examine, $o_{\texttt{null}}$ acts as a "sink" object: there might be references toward it, but no references from it.

- We have no explicit instructions for synchronization between threads. However, this is not a big issue: the possible schedulings we allow in the absence of synchronizations are a superset of the schedulings that would be possible if we had explicit thread synchronization instructions.

4.1.3 Object Lifetime

In the rest of this section, we introduce the concept of object lifetime. Intuitively, we define the lifetime of an object to be the interval of time when that object is reachable from the program variables. This is precisely the definition used by a garbage collector: an object is considered dead (and the memory space occupied by it is collected) when it is no longer reachable. A more precise definition would be to consider an object dead if the program execution never uses it in the future (although it might still be reachable), but this additional precision turns out not to be necessary in our case.

Definition 4 (Reachability in a concrete state). *Given a concrete state* $\Xi \in State$*, we define the reachability predicate* $reachable(\Xi) : Object \to \{\texttt{true}, \texttt{false}\}$ *as the least fixed point of the following constraits:*

$$\frac{\Xi = \langle A\,[t \mapsto J_1 : \langle L\,[v \mapsto o]\,, lb\rangle : J_2]\,, H, TY\rangle}{reachable(\Xi)(o)} \tag{4.2}$$

$$\frac{\Xi = \langle A, H, TY\rangle,\ \langle o_1, f, o_2\rangle \in H,\ reachable(\Xi)(o_1)}{reachable(\Xi)(o_2)} \tag{4.3}$$

Intuitively, an object o is reachable in a given state Ξ if it is pointed to by a local variable from a stack frame of one of the threads (Constraint 4.2) or if the heap contains an edge from an already reachable object to o (Constraint 4.3).

For the following three definitions, consider a possibly infinite concrete execution trace T: $\Xi_0 \Rightarrow_T \Xi_1 \Rightarrow_T \ldots \Rightarrow_T \Xi_d \Rightarrow_T \Xi_{d+1} \Rightarrow_T \ldots$

Definition 5 (Object creation date). *Given an object* $o \in Object$*, we define the* object creation date*, denoted* $d_C(o)$*, to be the date* $d \in Date$ *such that the transition* $\Xi_{d-1} \Rightarrow_T$ *created* o*, i.e., the* d^{th} *instruction was the NEW that created* o*). By convention, if such a date does not exist, e.g., for* $o_{\texttt{null}}$*,* $d_C(o) = \infty$*.*

The following lemma tells that if at some point after its creation date, an object becomes unreachable, it remains unreachable for ever.

Lemma 6. $\forall o \in Object \setminus \{o_{\texttt{null}}\}$*, if* o *is not reachable at date* $d \geq d_C(o)$ *then* o *is not reachable at any other later date* $d' > d$*, i.e.,*

$$\neg reachable(\Xi_d)(o) \to \neg reachable(\Xi_{d'})(o)$$

Proof: It is sufficient to prove the implication for $d' = d + 1$. Suppose for the sake of contradiction that the implication does not hold, i.e., $\neg reachable(\Xi_d)(o) \wedge reachable(\Xi_{d+1})(o)$. So, the $(d+1)^{th}$ instruction caused o to become reachable again. By a case analysis of all types of instruction, we see that COPY, LOAD, START THREAD, IF, CALL, NULLIFY and STORE can at most create new paths to objects that were already reachable before the instructions. None of them can make o to

become reachable again. As $d \geq d_C(o)$, the case of NEW is also irrelevant and we obtain a contradiction. □

Definition 6 (Object death date). *Given an object $o \in Object$, we define the* object death date, *denoted $d_D(o)$, to be the date d, such that*

$$reachable(\Xi_d)(o) \wedge \neg reachable(\Xi_{d+1})(o)$$

If such a date does not exist, we put $d_D(o) = \infty$ by default.

Lemma 6 makes sure the previous definition is well-formed, i.e., for any object o, there is at most one d that satisfies the required implication.

Definition 7 (Lifetime of an object). *The lifetime of the object $o \in Object$ is the time interval $[d_C(o), d_D(o)]$.*

4.2 Abstract Semantics

The abstract semantics we present for *SmallJava* is concerned with a specific activation, i.e., invocation, of a method m. Given a concrete execution trace T and an activation $A(m)$ of m, the abstract semantics computes an abstract state for each "relevant" moment of the execution of $A(m)$. We say that we work with the "abstract semantics for/of activation $A(m)$". The abstract state attached to a specific date models only the part of the concrete state at that date that is relevant for $A(m)$. E.g., we are not interested in the state of the local variables of some thread running in parallel with $A(m)$, nor in the heap references that such a thread creates, and $A(m)$ does not read. In general, the abstract semantics of $A(m)$ does not look "outside" $A(m)$.

The rest of this section is organized as follows. First, in Section 4.2.1, we explain which are the interesting dates for the execution of an activation. Next, in Section 4.2.2 we define the concrete escape predicates, an auxiliary notion that we use in the definition of the abstract semantics. Section 4.2.3 presents the sets and the notations used for the abstract semantics. Finally, in Section 4.2.4, we explain how the abstract semantics computes the abstract states for the interesting dates of an activation.

4.2.1 Method Activation and Interesting Dates

As other threads might interrupt the execution of $A(m)$, not all the moments between the CALL that started $A(m)$ and the corresponding RETURN which ends $A(m)$ are worth looking at. In general, it is worth looking only at those dates when $A(m)$ executes an instruction. In the next paragraphs, we define the important dates for the execution of a specific method activation. The abstract semantics computes an abstract state only for these dates.

Suppose we have a concrete execution trace T, and we know that the first instruction from an invocation of method m occured in thread t, in the transition

$\Xi_{d_0} \Rightarrow_T \Xi_{d_0+1}$. By looking into the trace T, we can easily identify all the dates when the program executes instructions from that invocation of method m, or from methods it transitively calls. We simply look into the concrete trace T, starting with d_0, and select those transitions $\Xi_d \Rightarrow_T \Xi_{d+1}$ where the concrete semantics selects thread t for execution. We stop as soon as we find a RETURN instruction that returns the control from the invocation of m that started at d_0, i.e., the first RETURN instruction that the program executes in thread t, with the same stack depth as at date d_0. For each such transition $\Xi_d \Rightarrow_T \Xi_{d+1}$ that corresponds to an instruction from $A(m)$, we consider both d and $d+1$ to be *interesting dates.*

Some CALLs might be unanalyzable. We do not want to include the instructions of the methods invoked by these unanalyzable CALLs (and those of the methods transitively called by them) into $A(m)$. For each such CALL, we skip all the dates d between the CALL and the corresponding RETURN[4]. We do consider interesting the date when the CALL instruction starts and the date when the corresponding RETURN finishes.

All the interesting dates are introduced in pairs; we end up with a list which is the concatenation of two-elements lists of the form $[id_{2i}, id_{2i+1}]$. An interesting date will be denoted id_j, where j is the index into the list of interesting dates. In general, id_{2i} is the moment right before executing an instruction and id_{2i+1} is the moment right after. For each i such that $P(lb_{id_{2i},t})$, i.e., the current instruction at date id_{2i} of the thread t where $A(m)$ takes place, is not an unanalyzable CALL, the transition $\Xi_{id_{2i}} \Rightarrow_T \Xi_{d_{2i+1}}$ corresponds to the execution of exactly one instruction from $A(m)$ and $id_{2i+1} = id_{2i} + 1$. If $P(lb_{id_{2i},t})$ is an unanalyzable CALL, the chain of transitions $\Xi_{id_{2i}} \Rightarrow_T^* \Xi_{id_{2i+1}}$ contains all the instructions of the method invoked by that CALL (and those of the methods that it transitively calls). It is important to note that between id_{2i+1} and $id_{2(i+1)}$, the program executes only instructions from threads other than t.

Definition 8 (Interesting Dates). *The list*

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

which we construct as explained previously, is the list of interesting dates *of the activation* $A(m)$.

Note that as instructions from outside $A(m)$ might separate two consecutive instructions from $A(m)$, it is not generally the case that $id_{2i+1} = id_{2(i+1)}$. All we can say is that $id_{2i+1} \leq id_{2(i+1)}$.

The last two interesting dates — id_{2r} and id_{2r+1} — correspond to the execution of the RETURN instruction which terminates the activation $A(m)$.

Observation: We study only method activations that are finite, i.e., we are not concerned with activations that end up in an infinite loop. Hence, the last transition from $A(m)$, $\Xi_{id_{2r}} \Rightarrow_T \Xi_{id_{2r+1}}$, corresponds to a RETURN instruction. As the ultimate

[4]Determined by looking at the stack depth.

goal of our proof is to prove the correctness of the stack allocation hints produced by the analysis, this does not introduce any limitation: an activation of infinite length never returns, its stack frame is never removed and so, any stack allocation hints are correct for it.

Notation: For convenience, we use the notation:

$$\Xi_d = \langle A_d\,[t \mapsto \langle L_{d,t}, lb_{d,t}\rangle : J_{d,t}]\,, H_d, TY_d\rangle$$

to indicate the concrete state at the interesting date $d \in ID_{A(m)}$ — d is id_{2i} or id_{2i+1} — where the thread t has the local state $\langle L_{d,t}, lb_{d,t}\rangle : J_{d,t}$. The chain of transition(s) $\Xi_{id_{2i}} \Rightarrow_T^* \Xi_{id_{2i+1}}$, which has more than one transition only for an unanalyzable CALL, starts with the instruction $P(lb_{id_{2i},t})$.

4.2.2 Concrete Escape Predicate

When we study the execution of an activation $A(m)$, it is useful to identify the concrete locations that can be accessed from outside $A(m)$. For this purpose, we introduce one escape predicate in each interesting date. An escape predicate tells us which objects might be reachable and thus, might be accessed from outside $A(m)$. We use the escape predicates later in the this section, when we define the abstract semantics transfer function.

Being a reachability-like property, escapability propagates along the heap references. However, it turns out to be more convenient for the later proofs if we propagate it only along the heap references which are created by $A(m)$. This is not a limitation: if a part of the program outside $A(m)$ creates a reference, then it was able to access both ends of the newly created reference and so, they were already escaped. We start by formally defining the set of concrete heap edges that the activation $A(m)$ creates.

Definition 9 (Edges created by activation $A(m)$). *We define the family of sets of edges $H_d^{A(m)} \subseteq Object \times Field \times Object$, $d \in ID_{A(m)}$, in an inductive way, by the following equations:*

$$H_{id_0}^{A(m)} = \emptyset$$

$$H_{id_{2i+1}}^{A(m)} = \begin{cases} H_{id_{2i}}^{A(m)} \cup \{\langle L_{id_{2i},t}(v_1), f, L_{id_{2i},t}(v_2)\rangle\} & \text{if } P(lb_{id_{2i},t}) = \text{“}v_1.f = v_2\text{”} \\ H_{id_{2i}}^{A(m)} \cup \{\langle o, f, o_{\texttt{null}}\rangle \mid f \in fields(C)\} & \\ \qquad \text{if } P(lb_{id_{2i},t}) = \text{“}v = \texttt{new } C\text{”}; \ o \text{ is the newly created location} & \\ H_{id_{2i}}^{A(m)} & \text{otherwise} \end{cases}$$

$$H_{id_{2(i+1)}}^{A(m)} = H_{id_{2i+1}}$$

Intuitively, $H_d^{A(m)}$ is the set of concrete heap edges that $A(m)$ creates in its execution up to the date $d \in ID_{A(m)}$. The sets $H_d^{A(m)}$ are cummulative, $H_{id_0}^{A(m)}$ is empty, and STORE and NEW are the only instructions that can add new edges. Now, we are able to define the escape predicates:

Definition 10 (Concrete Escape Predicates). *We define the family of concrete escape predicates* e_d : *Object* $\rightarrow$ $\{\texttt{true}, \texttt{false}\}, d \in ID_{A(m)}$, *as the least fixed point of the following constraints (the ordering relation* $\sqsubseteq$ *for boolean predicates is given in the first constraint):*

$$\forall d < d', e_d \sqsubseteq e_{d'}, \ i.e., \ \forall o, e_d(o) \rightarrow e_{d'}(o) \tag{4.4}$$

$$\frac{d_C(o) \leq d, o \text{ was not created by } A(m)}{e_d(o)} \tag{4.5}$$

$$\frac{\langle o_1, f, o_2 \rangle \in H_d^{A(m)}, \ e_d(o_1)}{e_d(o_2)} \tag{4.6}$$

$$\frac{P(lb_{id_{2r},t}) = \text{“}\texttt{return } v\text{”}, o = L_{id_{2r},t}(v)}{e_{id_{2r+1}}(o)} \tag{4.7}$$

$$\frac{P(lb_{id_{2i},t}) = \text{“}\texttt{start } v\text{”}, o = L_{id_{2i},t}(v)}{e_{id_{2i+1}}(o)} \tag{4.8}$$

$$\frac{P(lb_{id_{2i},t}) = \text{“}v_R = v_0.s(v_1, \ldots, v_j)\text{” is unanalyzable}, o_j = L_{id_{2i},t}(v_j)}{e_{id_{2i+1}}(o_j)} \tag{4.9}$$

Constraint 4.4 makes sure the escape predicates are cummulative: once an object escapes, it escapes forever. Constraint 4.5 takes care of the locations created outside $A(m)$, which trivially escape. As escapability is a reachability property, it propagates over the heap edges, as required by Constraint 4.6.

The last three constraints are relevant when $d = id_{2i}$, i.e., right before the execution of an instruction from $A(m)$. These constraints indicate how the instruction from id_{2i} affects the escape information at date id_{2i+1}, immediately after the instruction.

There are three types of instructions which can "escape" locations outside $A(m)$: the final RETURN of $A(m)$, THREAD START instructions, and unanalyzable CALLs. We discuss here just the first one; the others are similar. Suppose the last RETURN instruction from $A(m)$, the one from the transition from date id_{2r} to id_{2r+1}, has the form "`return` v". After the execution of this instruction, $o = L_{id_{2r}}(v)$ is reachable from the caller; accordingly, Constraint 4.7 sets $e_{id_{2r+1}}(o)$ to true.

Before presenting a lemma that shows that the definition of the escape predicates corresponds to our intuition, we introduce one more definition:

Definition 11. *Consider a date d, the corresponding state* Ξ_d *from the trace T, and an activation* $A(m)$ *in thread t. We define the state* $outside_{A(m)}(\Xi_d)$ *to be the state that has the same heap and type function as* Ξ_d *and where the thread agenda is as follows:*

- *We modify the stack for the thread t by removing the stack frames corresponding to methods in $A(m)$. In other words, we keep only those stack frames below the first stack frame of $A(m)$ (the one generated by the CALL instruction that started $A(m)$) and those stack frames above the first stack frame generated by an unanalyzable CALL, if any (by looking at the return label for each frame, we can find the CALL that generated it).*

- *The state of the other threads is unmodified.*

Notice that the state $outside_{A(m)}(\Xi_d)$ is not a result of a valid execution (the stack of thread t has a "gap"). We defined it simply for the purpose of our proofs.

Now, the lemma that links the definition of the escape predicate to our intuition:

Lemma 7. *$\forall o \in Object, \forall d \in ID_{A(m)}$ such that $d \geq d_C(o)$, if $\Xi'_d = outside_{A(m)}(\Xi_d)$, then*

$$reachable(\Xi'_d)(o) \rightarrow e_d(o)$$

Proof: Induction on the list of interesting dates $ID_{A(m)}$ plus case analysis on the type of the instruction executed in the current transition. □

4.2.3 Sets and Notations

Figure 4-4 presents the sets and the notations for the abstract semantics. For each interesting date $d \in ID_{A(m)}$, the abstract semantics computes an abstract state $\Xi^{\#} \in State^{\#}$ and an abstraction relation $\rho \in Abstr$. It also maintains a calling context $c \in Context$. We describe all these mathematical objects in the next paragraphs.

Calling Context

Formally, a calling context is a list of dates. As this definition is too generic, we try to give it a more intuitive shape. The calling context at date $d \in ID_{A(m)}$ represents the dates when we executed the CALLs which created the "relevant" part of the current call stack of thread t, the thread where $A(m)$ takes place. To understand what "relevant" means, it is easier to understand what it does not refer to: there is no need to consider the stack frames which existed when $A(m)$ started nor the stack frame of the root of $A(m)$, because these stack frames are present in the stack during the entire execution of $A(m)$.

As a short example, suppose $A(m)$ starts its execution and, at date d_1, executes its first CALL. Next, it executes instructions from the called method including, at date d_2, a new CALL. Let's consider the first instruction from the method called by the last CALL. This instruction was reached due to the CALL at date d_1 and the CALL at date d_2. The calling context at the moment when the program executes this instruction is $c = [d_2, d_1]$. At that moment, the stack of the thread t where $A(m)$ takes place contains all the stack frames which were there when $A(m)$ begun, the stack frame of the instance of m which is the "root" of $A(m)$, the stack frame of the

$$
\begin{array}{rcl}
c & \in & \mathit{Context} = \texttt{list of } \mathit{Date} \\
\\
n & \in & \mathit{Node}^{\#} = \mathit{Node} \times \mathit{Context} \\
\\
\mathcal{N} & = & \{n_{\texttt{null}}\} \times \mathit{Context} \\
\mathit{INode}^{\#} & = & \mathit{INode} \times \mathit{Context} \\
\mathit{LNode}^{\#} & = & \mathit{LNode} \times \mathit{Context} \\
\mathit{PNode}^{\#} & = & \mathit{PNode} \times \mathit{Context} \\
\mathit{RNode}^{\#} & = & \mathit{RNode} \times \mathit{Context} \\
\\
n^{I}_{lb,c} & \stackrel{\text{def}}{=} & \langle n^{I}_{lb}, c\rangle \in \mathit{INode}^{\#} \\
n^{L}_{lb,c} & \stackrel{\text{def}}{=} & \langle n^{L}_{lb}, c\rangle \in \mathit{LNode}^{\#} \\
n^{P}_{m,i,c} & \stackrel{\text{def}}{=} & \langle n^{P}_{m,i}, c\rangle \in \mathit{PNode}^{\#} \\
n^{R}_{lb,c} & \stackrel{\text{def}}{=} & \langle n^{R}_{lb}, c\rangle \in \mathit{RNode}^{\#} \\
n_{\texttt{null},c} & \stackrel{\text{def}}{=} & \langle n_{\texttt{null}}, c\rangle \in \mathcal{N} \\
\\
I^{\#} & \in & \mathit{IEdges}^{\#} = \mathcal{P}((\mathit{Node}^{\#} \setminus \mathcal{N}) \times \mathit{Field} \times \mathit{Node}^{\#}) \\
O^{\#} & \in & \mathit{OEdges}^{\#} = \mathcal{P}((\mathit{Node}^{\#} \setminus \mathcal{N}) \times \mathit{Field} \times \mathit{LNode}^{\#}) \\
L^{\#} & \in & \mathit{LocVar}^{\#} = V \to \mathcal{P}(\mathit{Node}^{\#}) \\
J^{\#} & \in & \mathit{JavaStack}^{\#} = \texttt{list of } \mathit{LocVar}^{\#} \\
\\
\rho & \in & \mathit{Abstr} = \mathcal{P}(\mathit{Object} \times \mathit{Node}^{\#}) \\
\\
\Xi^{\#} & \in & \mathit{State}^{\#} = \mathit{IEdges}^{\#} \times \mathit{OEdges}^{\#} \times \mathit{JavaStack}^{\#} \times \\
 & & \quad \mathcal{P}(\mathit{Node}^{\#}) \times \mathcal{P}(\mathit{Node}^{\#})
\end{array}
$$

Figure 4-4: Sets and notations for the abstract semantics

method called at date d_1, and, on the top, the stack frame of the method called at date d_2.

The abstract semantics transfer function maintains the calling context as a stack: it is empty at the beginning of $A(m)$, each CALL instruction pushes the current date on the top of the context and each RETURN instruction pops the topmost element of the context.

Comment: At a first look, the concept of calling context seems strange. The reason for its existence is difficult to explain before examining the proofs for the inter-procedural analysis from Section 4.4. For the time being, let's say that the calling context has the purpose of making the distinction between the current instance of a method and its previous instances. This way, the nodes associated with the objects manipulated by the current instance of a method will be distinct from all the nodes previously created. A previous attempt to prove the analysis correctness failed because the abstract semantics was not making this distinction. We discuss more on this issue in Section 4.7.

Abstract State

The abstract states computed by the abstract semantics are very similar to the points-to graphs computed by the pointer analysis with the modification that now, all nodes are specialized function of the calling context in which they are created. Instead of using nodes from the set *Node*, the abstract semantics uses "nodes with context":

$$n = \langle n', c \rangle \in Node^{\#} = Node \times Context$$

We employ the term "node" for both simple nodes and nodes with context. The real meaning of the term should be clear from the surrounding text.

In general, for each set of nodes $X \in \{INode, LNode, PNode, RNode\}$ from the analysis, $X^{\#}$ represents the equivalent set of nodes from the abstract semantics. E.g., $INode^{\#} = INode \times Context$. The set $\mathcal{N}$ contains the null node with all the possible contexts. Figure 4-4 also contains the notations that we use for the nodes with context. E.g., $n^I_{lb,c} \in INode^{\#}$ is the inside node n^I_{lb} paired with the context c.

A second modification is that instead of a single state for the local variables $L^a \in LocVar^a$, an abstract state has a stack $J^{\#} \in JavaStack^{\#}$. The pointer analysis does not *"step into"* a called method. As it remains inside the analyzed method m, it needs to model just the state of the local variables of m. On the other side, the abstract semantics sequentially processes the instructions of the transitively called methods, one by one, and hence, it has to maintain the state of the local variables of all the methods that are in the call chain between the root method of the activation $A(m)$, i.e., m, and the top-most method. Intuitively, the stack $J^{\#}$ from an abstract state models the upper part of the concrete stack associated with the thread t where $A(m)$ takes place.

All the other structures are similar to the corresponding ones from the pointer analysis except that now, they are based on nodes with context, instead of simple

nodes. Similar to a points-to graph, an abstract state

$$\Xi^{\#} = \langle I^{\#}, O^{\#}, J^{\#}, S^{\#}, U^{\#} \rangle$$

contains a set $I^{\#}$ of inside edges, a set $O^{\#}$ of outside nodes, a set $S^{\#}$ of started threads, and a set $U^{\#}$ of nodes passed as arguments to unanalyzable CALLs.

As in the case of the points-to graphs, we can attach an escape predicate to each abstract state computed by the abstract semantics. This predicate will tell us which nodes might be reachable from outside the analyzed activation of method m.

Definition 12 (Escape predicate for the abstract semantics). *Consider an abstract state $\Xi^{\#} = \langle I^{\#}, O^{\#}, L^{\#} : J^{\#}, S^{\#}, U^{\#} \rangle$ computed by the abstract semantics of some activation of a method m. We define the escape predicate*

$$e^{\#}(\Xi^{\#}) : Node^{\#} \rightarrow \{\texttt{true}, \texttt{false}\}$$

as follows:

$$e^{\#}(\Xi^{\#})(n) = reachable(N, I^{\#} \cup O^{\#})(n)$$

where the auxiliary predicate reachable is as described in Definition 1 and N, the set of escapability sources, is

$$N = PNode^{\#} \cup L^{\#}(v_{ret}) \cup S^{\#} \cup U^{\#} \cup RNode^{\#}$$

The parameter nodes (now with context) and the returned nodes (pointed to by the dummy local variable v_{ret}) are trivially reachable from the caller of $A(m)$. Note that in an abstract state computed by the abstract semantics of an activation $A(m)$, the only parameter nodes which might occur are m's parameter nodes, $n^{P}_{m,i,[]}$. The started thread nodes are reachable from the code of their execution threads, while the nodes passed to unanalyzable methods and the nodes which model the objects returned from these methods, escape into the unanalyzable methods. Like reachability, escapability propagates along the inside and the outside edges.

The following easy lemma proves that the escape predicate for the abstract semantics is monotonic:

Lemma 8. *Consider two abstract states*

$$\begin{aligned} \Xi^{\#}_1 &= \langle I^{\#}_1, O^{\#}_1, L^{\#}_1 : J^{\#}_1, S^{\#}_1, U^{\#}_1 \rangle \\ \Xi^{\#}_2 &= \langle I^{\#}_2, O^{\#}_2, L^{\#}_2 : J^{\#}_2, S^{\#}_2, U^{\#}_2 \rangle \end{aligned}$$

If $I^{\#}_1 \subseteq I^{\#}_2$, $O^{\#}_1 \subseteq O^{\#}_2$, $L^{\#}_1(v_{ret}) \subseteq L^{\#}_2(v_{ret})$, $S^{\#}_1 \subseteq S^{\#}_2$ and $U^{\#}_1 \subseteq U^{\#}_2$, then

$$\forall n, e^{\#}(\Xi^{\#}_1)(n) \rightarrow e^{\#}(\Xi^{\#}_2)(n)$$

Proof: By the conditions of the lemma, each escapability source and each path that exists in $\Xi^{\#}_1$ exists in $\Xi^{\#}_2$ too. □

Note: The previous lemma states a property that is stronger than monotonicity: we do not impose any condition on the stacks of the two abstract states. We use this additional power very frequently in our proofs. In general, the abstract semantics transfer functions augment the sets of the inside/outside edges, the set of started threads and/or the set of nodes passed to unanalyzable CALL sites. Based on the previous lemma, if a node escapes at a certain point, it will escape in the future too.

Abstraction Relation

For each interesting date $d \in ID_{A(m)}$, the abstract semantics computes an abstraction relation $\rho_d \in Abstr$ with the following meaning: $o\ \rho_d\ n$ if at date d, location o is modeled, i.e., abstracted, by node n. By defining ρ_d as an arbitrary relation instead of a function, we allow the same node to model multiple locations, and multiple nodes to model the same location. It is important to note that the abstraction relation is not part of the pointer analysis: anyway, being attached to a specific execution trace, it cannot even be computed by a static analysis. We introduced it just for the purpose of the correctness proof.

4.2.4 Abstract Execution of $A(m)$

The abstract semantics starts with the initial abstract state $\Xi^{\#}_{id_0}$, initial abstraction relation ρ_{id_0}, and initial calling context c_{id_0}, defined by the following equations:

$$\Xi^{\#}_{id_0} = \langle\ \emptyset,\ \emptyset,\ [\{p_i \mapsto n^P_{m,i,[]}\}_{0\le i\le k-1}],\ \emptyset,\ \emptyset\ \rangle \tag{4.10}$$

$$\rho_{id_0} = \{\langle o_i, n^P_{m,i,[]}\rangle\}_{0\le i\le k-1} \tag{4.11}$$

$$c_{id_0} = [] \text{ (empty context)} \tag{4.12}$$

where $n^P_{m,0,[]}, n^P_{m,1,[]}, \ldots n^P_{m,k-1,[]}$ are the k parameter nodes for method m (with empty context) and $o_0, o_1, \ldots o_{k-1}$ are the k objects the method m receives as actual arguments. In the initial abstract state, the sets of inside and outside edges are empty and each formal parameter p_i points to its corresponding parameter node $n^P_{m,i,[]}$. $A(m)$ has not started any thread, nor lost any node through an unanalyzable CALL. In the initial abstraction relation, we record the fact that the parameter node $n^P_{m,i,[]}$ models the object o_i that is the i^{th} actual parameter in the call that generated activation $A(m)$.

The abstract semantics computes the rest of the tuples $\langle \Xi^{\#}_d, \rho_d, c_d\rangle$, $d \in ID_{A(m)}$ as follows:

$$\langle \Xi^{\#}_{id_{2i+1}}, \rho_{id_{2i+1}}, c_{id_{2i+1}}\rangle = [\![id_{2i}]\!]^{\#}(\langle \Xi^{\#}_{id_{2i}}, \rho_{id_{2i}}, c_{id_{2i}}\rangle) \tag{4.13}$$

$$\langle \Xi^{\#}_{id_{2(i+1)}}, \rho_{id_{2(i+1)}}, c_{id_{2(i+1)}}\rangle = \langle \Xi^{\#}_{id_{2i+1}}, \rho_{id_{2i+1}}, c_{id_{2i+1}}\rangle \tag{4.14}$$

As we are concerned only with instructions from $A(m)$, the abstract state, the

$[\![.]\!]^{\#} : Date \rightarrow State^{\#} \times Abstr \times Context \rightarrow State^{\#} \times Abstr \times Context$

$P(lb_{d,t})$	Definition of $[\![d]\!]^{\#}$
analyzable CALL $v_R = v_0.s(v_1, ..., v_j)$	$[\![d]\!]^{\#}(\langle\ \langle I^{\#}, O^{\#}, L^{\#} : J^{\#}, S^{\#}, U^{\#}\rangle,\ \rho,\ c\rangle) =$ $\langle\ \langle I^{\#}, O^{\#}, L^{\#}_{callee} : L^{\#} : J^{\#}, S^{\#}, U^{\#}\rangle,\ \rho,\ d : c\rangle$ where $L^{\#}_{callee} = \{p_i \mapsto L^{\#}(v_i)\}_{0 \le i \le j}$
RETURN inside $A(m)$ **return** v	$[\![d]\!]^{\#}(\langle\ \langle I^{\#}, O^{\#}, L^{\#}_{callee} : L^{\#} : J^{\#}, S^{\#}, U^{\#}\rangle,\ \rho,\ d_c : c\rangle) =$ $\langle\ \alpha(d_c)(\Xi^{\#}_2),\ \alpha(d_c)(\rho),\ c\rangle$ where $\Xi^{\#}_2 = \langle I^{\#}, O^{\#}, L^{\#}\left[v_R \mapsto L^{\#}_{callee}(v)\right] : J^{\#}, S^{\#}, U^{\#}\rangle$ and v_R is the receiver variable for the corresponding CALL
otherwise	$[\![d]\!]^{\#}(\langle\ \Xi^{\#},\ \rho,\ c\ \rangle) = \langle\ \Xi^{\#}_2,\ \rho_2,\ c\ \rangle$ where $\Xi^{\#}_2 = [\![lb_{d,t}, c]\!](\Xi^{\#})$ $\rho_2 = update\_\rho(\langle d, lb_d, \Xi^{\#}, c\rangle)(\rho)$

Figure 4-5: Definition of abstract semantics transfer function $[\![.]\!]^{\#}$

abstraction relation, and the calling context do not change from date id_{2i+1} to date $id_{2(i+1)}$. The interesting things happen when we pass from id_{2i} to id_{2i+1}.

Transfer Function $[\![.]\!]^{\#}$

Figure 4-5 presents the formal definition of the abstract semantics transfer function $[\![.]\!]^{\#}$. Given a date $d = id_{2i}$ when an instruction of $A(m)$ starts its execution, the function $[\![d]\!]^{\#}$ takes the current abstract state, abstraction relation and calling context and returns their updated version for the date d_{2i+1}. There are three cases in the definition of $[\![d]\!]^{\#}$.

When it processes an analyzable CALL, the abstract semantics "step into" the called method. If the instruction was "$v_R = v_0.s(v_1, \ldots, v_j)$", the abstract semantics initializes the state of the local variables of the called method to be $L^{\#}_{callee} = \{p_i \mapsto L^{\#}(v_i)\}_{0 \le i \le j}$ where $L^{\#}$ models the state of the local variables of the caller, and pushes $L^{\#}_{callee}$ on the abstract stack. Also, the abstract semantics pushes the date d on top of the calling context.

$$\alpha(d)(n) = \begin{cases} \langle n', c \rangle & \text{if } n = \langle n', d : c \rangle, n' \in Node \\ n & \text{otherwise} \end{cases}$$

$$\alpha(d)(\langle\ I^\#\ ,\ O^\#\ ,\ J^\#\ ,\ S^\#\ ,\ U^\#\ \rangle) = \langle\ \alpha(d)(I^\#)\ ,\ \alpha(d)(O^\#)\ ,\ \alpha(d)(J^\#)\ ,\ \alpha(d)(S^\#)\ ,\ \alpha(d)(U^\#)\ \rangle$$

$$\begin{aligned} \alpha(d)(I^\#) &= \{\langle \alpha(d)(n_1), f, \alpha(d)(n_2)\rangle \mid \langle n_1, f, n_2\rangle \in I^\#\} \\ \alpha(d)(O^\#) &= \{\langle \alpha(d)(n_1), f, \alpha(d)(n_2)\rangle \mid \langle n_1, f, n_2\rangle \in O^\#\} \\ \alpha(d)(J^\# = [L_0^\#, \ldots, L_l^\#]) &= [\alpha(d)(L_0^\#), \ldots, \alpha(d)(L_l^\#)] \\ \alpha(d)(L^\#) &= \lambda v.\alpha(d)(L^\#(v)) \\ \alpha(d)(A) &= \{\alpha(d)(n) \mid n \in A\}, \forall A \subseteq Node^\# \\ \alpha(d)(\rho) &= \{\langle o, \alpha(d)(n)\rangle \mid \langle o, n\rangle \in \rho\} \end{aligned}$$

Figure 4-6: Definition of conversion $\alpha(d)$, $d \in Date$

A RETURN inside $A(m)$ (i.e., not the RETURN which terminates $A(m)$) is the opposite of an analyzable CALL: it returns from the callee into the caller. Suppose the RETURN instruction has the form "`return` v", and the abstract semantics executed the corresponding CALL, which has the form "$v_R = v_0.s(v_1, \ldots, v_j)$", at date d_c. Due to the nesting of the CALL/RETURN instructions and to the way we maintain the calling context, the top element of the calling context is exactly d_c. In this case, the abstract semantics operates in two steps:

1. It constructs a new abstract state $\Xi_2^\#$. In $\Xi_2^\#$, the state of the local variables of the callee, $L_{callee}^\#$, is no longer on the abstract stack. Also, in $\Xi_2^\#$, the local variable v_R from the caller points to the nodes that the callee's local variable v points to.

2. The abstract semantics removes d_c from the top of the calling context. In addition, the conversion $\alpha(d_c)$ removes it from the top of the context of any node appearing in the abstraction relation ρ or in the abstract state $\Xi_2^\#$. Those nodes which do not contain d_c at the top of their context are unaffected by the conversion. We describe the conversion $\alpha(d_c)$ later in this section.

Finally, for the other instructions, the calling context remains the same, but the abstract semantics computes a new abstract state and a new abstraction relation by using the auxiliary functions $[\![.,.]\!]$ (Figure 4-7) and $update\_\rho$ (Figure 4-8).

Node Conversion $\alpha(d)$

Figure 4-6 presents the definition of the conversion $\alpha(d), d \in \mathit{Date}$. When given a node n, $\alpha(d)$ behaves as follows:

- If the context of n starts with d, i.e. $n = \langle n', d : c \rangle$, $\alpha(d)$ returns a "new" nodes which is similar to n, but has a context without the date d at its top, i.e., $\langle n', c \rangle$.
- Otherwise, $\alpha(d)$ behaves like the identity function.

When it receives a more complicated structure — an abstract state, a set of inside/outside edges, a set of nodes, an abstraction relation, etc. — $\alpha(d)$ propagates deep into it and replaces every node n with $\alpha(d)(n)$.

Auxiliary Function $[\![.,.]\!]$

For each label lb and context c, $[\![lb, c]\!]$ is a function which takes the current abstract state, and returns the abstract state after the execution of the instruction from label lb, in the calling context c. Figure 4-7 presents the definition og $[\![lb, c]\!]$ depending on the instruction from the label lb. $[\![lb, c]\!]$ is almost identical to the analysis transfer function $[\![lb]\!]^{\#}$ (Figure 2-4), except that the newly created node, if any, has the context c. E.g., if the instruction at label lb is a NEW, $[\![lb, c]\!]$ uses the inside node with context $n^I_{lb,c} = \langle n^I_{lb}, c \rangle$ instead of the "simple" node n^I_{lb}. In the case of a LOAD instruction, we have simply "inlined" the definition of the function *process_load* (Figure 2-5), and adjusted it to work with nodes with context.

Similar to the pointer analysis, the abstract semantics does not create edges starting from $n_{\mathtt{null},c}, \forall c \in \mathit{Context}$, and does not load references from these nodes. This is due to the fact that a program cannot read from/write to a null address.

Auxiliary Function *update_ρ*

Figure 4-8 presents the definition of the function *update_ρ*. If, at date $d = id_{2i}$, the program is about to execute the instruction from label lb, the current abstract state is $\Xi^{\#}$ and the current calling context is c, then, the function *update_ρ*$(\langle d, lb, \Xi, c \rangle)$ takes the current abstraction relation ρ, and returns the abstraction relation for the date id_{2i+1}.

The definition of *update_ρ* is closely related to that of $[\![.,.]\!]$. While it is possible to define them as a single function, we preferred to separate the part which is "inherited" from the pointer analysis, i.e., the function $[\![.,.]\!]$, from the abstraction relation maintenance part which is specific to the abstract semantics.

Most of the instructions do not modify ρ. There are four exceptions: NEW, NULLIFY, LOAD (in a special case), and unanalyzable CALL. In the case of a NEW instruction which creates the new object o, the abstract semantics extends ρ to record the fact that the node $n^I_{lb,c}$ models the object o. For both NEW and NULLIFY, the abstract semantics extends ρ to record the fact that the node $n_{\mathtt{null},c}$ models the special $o_{\mathtt{null}}$, which represents the null pointers. The processing for a LOAD instruction

$[\![.,.]\!] : \mathit{Label} \times \mathit{Context} \to \mathit{State}^{\#} \to \mathit{State}^{\#}$

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

$P(lb)$	$[\![lb, c]\!](\Xi^{\#} = \langle I^{\#},\ O^{\#},\ L^{\#} : J^{\#},\ S^{\#},\ U^{\#}\rangle)$
$v_1 = v_2$	$\langle I^{\#},\ O^{\#},\ L^{\#}\left[v_1 \mapsto L^{\#}(v_2)\right] : J^{\#},\ S^{\#},\ U^{\#}\rangle$
$v = \texttt{new}\ C$	$\langle I_2^{\#},\ O^{\#},\ L^{\#}\left[v \mapsto \{n^I_{lb,c}\}\right] : J^{\#},\ S^{\#},\ U^{\#}\rangle$ where $I_2^{\#} = I^{\#} \cup \{\langle n^I_{lb,c}, f, n_{\texttt{null},c}\rangle\}_{f \in \mathit{fields}(C)}$
$v = \texttt{null}$	$\langle I^{\#},\ O^{\#},\ L^{\#}[v \mapsto \{n_{\texttt{null},c}\}] : J^{\#},\ S^{\#},\ U^{\#}\rangle$
$v_1.f = v_2$	$\langle I_2^{\#},\ O^{\#},\ L^{\#} : J^{\#},\ S^{\#},\ U^{\#}\rangle$ where $I_2^{\#} = I^{\#} \cup ((L^{\#}(v_1) \setminus \mathcal{N}) \times \{f\} \times L^{\#}(v_2))$
$v_2 = v_1.f$	Let $B = \{n \mid \exists n_1 \in L^{\#}(v_1) \setminus \mathcal{N}, \langle n_1, f, n\rangle \in I^{\#}\}$ $E = \{n \in L^{\#}(v_1) \setminus \mathcal{N} \mid e^{\#}(\Xi^{\#})(n)\}$ **Case 1:** $E = \emptyset$ $\langle I^{\#},\ O^{\#},\ L^{\#}[v \mapsto B] : J^{\#},\ S^{\#},\ U^{\#}\rangle$ --- **Case 2:** $E \neq \emptyset$ $\langle I^{\#},\ O_2^{\#},\ L_2^{\#} : J^{\#},\ S^{\#},\ U^{\#}\rangle$ where $L_2^{\#} = L^{\#}\left[v \mapsto \left(B \cup \{n^L_{lb,c}\}\right)\right]$ $O_2^{\#} = O^{\#} \cup (E \times \{f\} \times \{n^L_{lb,c}\})$
$\texttt{if}\ (\ldots)\ \texttt{goto}\ a_t$	$\Xi^{\#}$ (unmodified)
$v_R = v_0.s(v_1, \ldots, v_{j-1})$	**Case 1:** analyzable call $[\![.,.]\!]$ will never be called in such a case --- **Case 2:** unanalyzable call $\langle I^{\#},\ O^{\#},\ L_2^{\#} : J^{\#},\ S^{\#},\ U_2^{\#}\rangle$ where $L_2^{\#} = L^{\#}\left[v_R \mapsto n^R_{lb,c}\right]$ $U_2^{\#} = U^{\#} \cup \bigcup_{i=0}^{j-1} L^{\#}(v_i)$
$\texttt{return}\ v$	$\langle I^{\#},\ O^{\#},\ L^{\#}\left[v_{ret} \mapsto L^{\#}(v)\right],\ S^{\#},\ U^{\#}\rangle$ where v_{ret} is the dummy variable that stores the return value of m.
$\texttt{start}\ v$	$\langle I^{\#},\ O^{\#},\ L^{\#} : J^{\#},\ S^{\#} \cup L^{\#}(v),\ U^{\#}\rangle$

Figure 4-7: Definition of $[\![.,.]\!]$

$update\_\rho : Date \times Label \times State^{\#} \times Context \rightarrow Abstr \rightarrow Abstr$

$P(lb)$	$update\_\rho(\langle d, lb, \Xi^{\#} = \langle I^{\#}, O^{\#}, L^{\#}, S^{\#}, U^{\#} \rangle, c \rangle)(\rho)$
$v_1 = v_2$	ρ (unmodified)
$v =$ **new** C	$\rho \cup \{\langle o, n^I_{lb,c} \rangle\} \cup \{\langle o_{\texttt{null}}, n_{\texttt{null},c} \rangle\}$ where o is the object created in the concrete execution at date d
$v =$ `null`	$\rho \cup \{\langle o_{\texttt{null}}, n_{\texttt{null},c} \rangle\}$
$v_1.f = v_2$	ρ (unmodified)
$v_2 = v_1.f$	Let $E = \{n \in L^{\#}(v_1) \setminus \mathcal{N} \mid e^{\#}(\Xi^{\#})(n)\}$ $\langle o_1, f, o_2 \rangle$ be the heap edge read by the concrete execution at date d **Case 1:** $E = \emptyset$ ρ (unmodified) **Case 2:** $E \neq \emptyset$ ρ_2 where $\rho_2 = \begin{cases} \rho \cup \{\langle o_2, n^L_{lb,c} \rangle\} & \text{if } e_d(o_1) \neq \emptyset \\ \rho \text{ (unmodified)} & \text{otherwise} \end{cases}$
`if (...) goto` a_t	ρ (unmodified)
$v_R = v_0.s(v_1, \ldots, v_j)$	**Case 1:** analyzable call $update\_\rho$ will never be called in such a case **Case 2:** unanalyzable call $\rho \cup \{\langle o, n^R_{lb,c} \rangle\}$ where o is the object returned by the call in the concrete execution.
`return` v	ρ (unmodified)
`start` v	ρ (unmodified)

Figure 4-8: Definition of $update\_\rho$

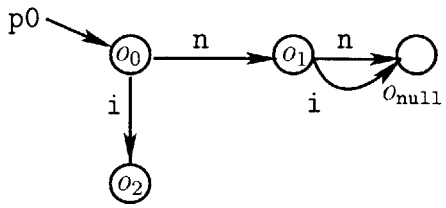
might introduce a load node $n^L_{lb,c}$ (see definition of $[\![.,.]\!]$ in Figure 4-7). In this case, if the LOAD instruction read the heap reference $\langle o_1, f, o_2\rangle$ and o_1 was an escaped node, i.e., $e_d(o_1)$ was true, then, the abstract semantics extends ρ to record the fact that $n^L_{lb,c}$ models o_2. Finally, in the case of an unanalyzable CALL, the abstract semantics extends the abstraction ρ to record the fact that the return node $n^R_{lb,c}$ models the returned object o.

Example 5. Consider the following piece of code:

```
  Cell a(Cell p0) {
0:  v0 = p0;
1:  if(v0 == null) goto 4;
2:  v0 = v0.b();
3:  if(true) goto 1;
4:  return p0;
  }

  Cell b(Cell p0) {
5:  v0 = new Info;
6:  p0.f = v0;
7:  v1 = p0.n;
8:  return v1;
  }

Cell {
  Info i;
  Cell n;
}
```



Initial concrete heap at the beginning of a

For simplicity, we use integer labels. Method a uses method b to scan a null terminated linked list, and to update the information of all the list cells. The field n of a list cell (class Cell) points to the next cell from the list. Method b, which is a method of class Cell, does all the processing for a list cell: it sets the field i of the the cell to point to a newly created object, and returns a pointer to the next cell. For simplicity, we suppose that the class Info does not have any pointer field.

Table 4.1 presents a possible concrete execution that contains an activation of a, $A(\mathtt{a})$. We suppose that the argument sent by the CALL that starts $A(\mathtt{a})$ is a pointer to the first cell of a list of two elements, o_0 and o_1, as presented in the figure right under the code. The field i of o_0 points to the object o_3, while the field i of o_1 is null. Table 4.1 presents only the dates when $A(\mathtt{a})$ starts to execute an instruction.

The list of interesting dates of $A(\mathtt{a})$ is:

$$ID_{A(\mathtt{a})} = [10,11,\ 11,12,\ 20,21,\ 21,22,\ \ldots,\ 43,44]$$

We look closer at several of these dates. Each row of Figure 4-9 presents the concrete heap (the left side), the abstract state (the right side) and the abstract relation that the abstract semantics computes for a specific date. For simplicity, Figure 4-9 ignores the value of the local variables. We use continuous circles for objects and nodes, and dashed circles for node placeholders. Similarly, we use solid arcs for heap references and inside edges, and dashed arcs for outside edges.

date	label: instruction	context
	other instructions	
10	`0: v0 = p0;`	[]
11	`1: if(v0 == null) goto 4;`	[]
	other threads	
20	`2: v0 = v0.b();`	[]
21	`5: v0 = new Info;`	[20]
22	`6: p0.f = v0;`	[20]
23	`7: v1 = p0.n;`	[20]
24	`8: return v1;`	[20]
25	`3: if(true) goto 1;`	[]
30	`1: if(v0 == null) goto 4;`	[]
31	`2: v0 = v0.b();`	[]
32	`5: v0 = new Info;`	[31]
33	`6: p0.f = v0;`	[31]
34	`7: v1 = p0.n;`	[31]
	other threads	
40	`8: return v1;`	[31]
41	`3: if(true) goto 1;`	[]
42	`1: if(v0 == null) goto 4;`	[]
43	`4: return p0;`	[]

Table 4.1: Execution of $A(\mathtt{a})$ from Example 5. For each interesting date d listed in the first column, the second column presents the instruction about to be executed by $A(\mathtt{a})$ at date d, and its label; the third column presents the calling context at date d.

At the beginning of $A(\mathtt{a})$ (first row of Figure 4-9), the abstract state contains only the parameter node $n^P_{\mathtt{a},0,[]}$ that models the object o_0, and the calling context is []. At date 21, the abstract semantics "steps into" the method `b` that is called with the parameter o_0 in the concrete semantics, respectively $n^P_{\mathtt{a},0,[]}$ in the abstract semantics. At this point, the calling context becomes [20]. We briefly treat the next instructions. The object o_3 created by `b` is modeled by the inside node $n^I_{5,[20]}$. In the concrete semantics, `b` reads the field `n` of o_0, i.e., o_1. The abstract semantics introduces the load node $n^L_{7,[20]}$ to model the loaded object o_1 (Figure 4-9.b).

The next instruction is a RETURN that terminates this first invocation of `b`. The node conversion $\alpha(20)$ transforms $n^L_{7,[20]}$ into $n^L_{7,[]}$, and $n^I_{5,[20]}$ into $n^I_{5,[]}$ (Figure 4-9.c).

Skipping several instructions, Figure 4-9.d presents the concrete state, abstract state and the abstraction relation at the end of the second invocation of `b`, right before returning to `a`. Notice that the load node $n^L_{7,[31]}$, and the inside node $n^I_{5,[31]}$ that the abstract semantics uses inside this second invocation of `b` are distinct from all the nodes used before, including the nodes from the first invocation of `b`. The RETURN instruction that follows applies the conversion $\alpha(31)$. As a result, $n^L_{7,[31]}$ is merged

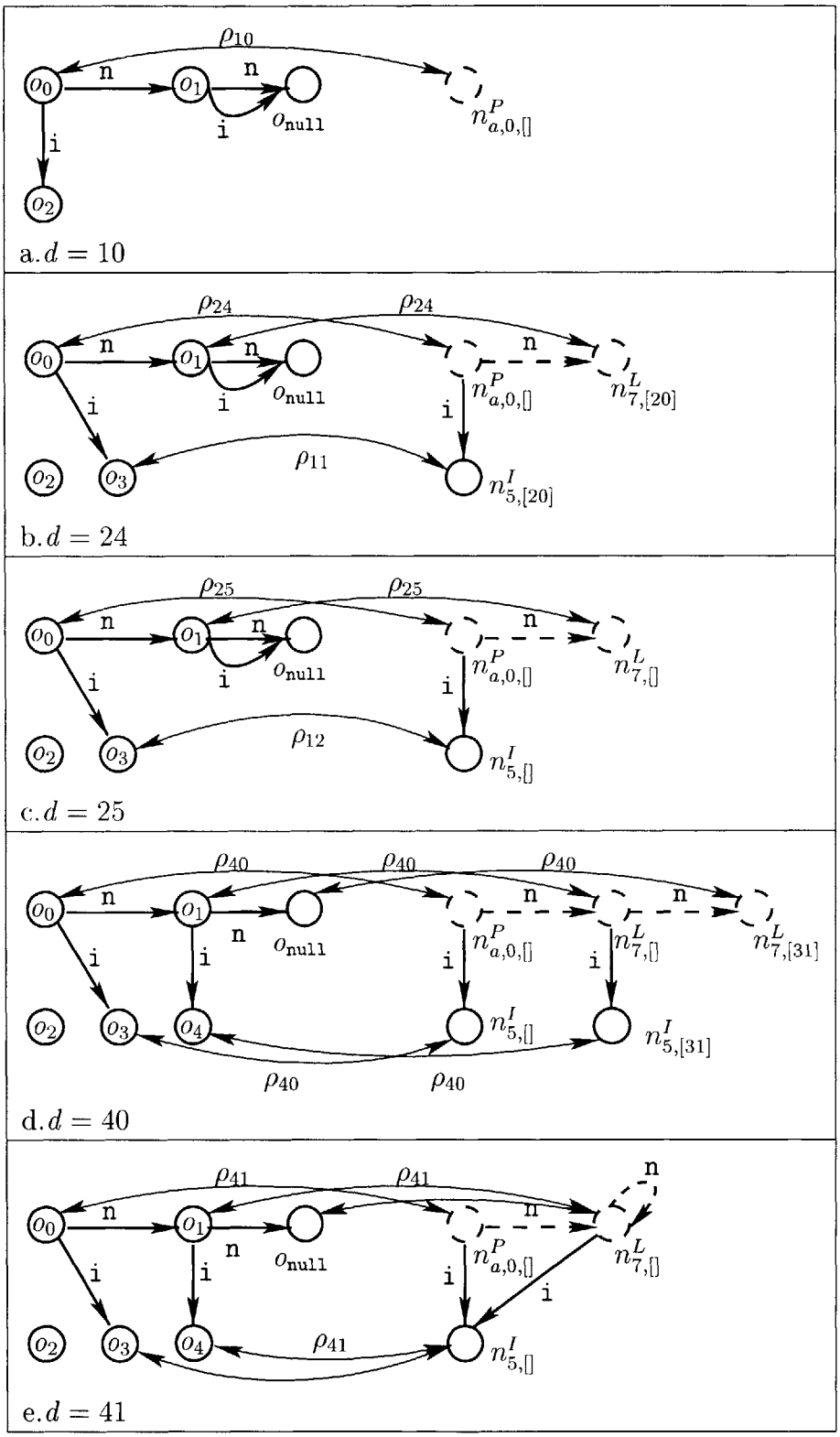


Figure 4-9: Abstract execution of $A(\mathtt{a})$ from Example 5

with $n^L_{7,[]}$, and $n^I_{5,[31]}$ is merged with $n^I_{5,[]}$ (Figure 4-9.e). So, once the current instance of b finishes, its nodes are merged with the nodes from the previous instances.

Several interesting facts are emphasized by this simple example:

- Objects that are not manipulated by $A(\mathtt{a})$, e.g., o_2, are ignored by the abstraction relation.
- The nodes introduced by the abstract semantics while processing an instance of a method are different from the nodes introduced for the previous instances.
- A node can model several objects. E.g., at date 41, $n^I_{5,[]}$ models both o_3 and o_4.
- At the dates that correspond to executions of instructions from the "root level" of $A(\mathtt{a})$, i.e., not from a callee, the current calling context, and the contexts of all the nodes that appear in the abstract state are empty.

△

4.3 Abstract Semantics Invariants

In this section, we prove a few invariants that describe the connection between the concrete and the abstract sematics. The section concludes with a sufficient condition for stack allocation.

The invariants are valid in any interesting date $d \in ID_{A(m)}$ of the execution of the studied activation of method m. For the rest of the section, we use the following notations for the concrete and the abstract state at date d:

$$\begin{aligned} \Xi_d &= \langle A_d\,[t \mapsto \langle L_{d,t}, lb_{d,t}\rangle : J_{d,t}]\,, H_d, TY_d\rangle \\ \Xi^\#_d &= \langle I^\#_d, O^\#_d, L^\#_d : J^\#_d, S^\#_d, U^\#_d\rangle \end{aligned}$$

In the concrete state Ξ_d, we emphasize the local state of thread t, the thread of $A(m)$; $L_{d,t}$ is the state of the local variables of top-most method from $A(m)$. In the abstract state $\Xi^\#_d$, we emphasize the structure $L^\#_d$ which describes the state of the local variables of the topmost method in the call stack at date d; $L^\#_d$ models $L_{d,t}$. Also, we use the notation ρ_d for the abstraction relation at date d.

We start with two easy invariants which will be used in later proofs:

Invariant 1. $\forall d \in ID_{A(m)},\ \forall o \in Object,\ \forall n = n_{\mathtt{null},c} \in \mathcal{N},\ o\ \rho_d\ n \rightarrow o = o_{\mathtt{null}}$.

Proof: Induction on the list of interesting dates plus inspection of the definition of *update_ρ* (Figure 4-8). □

Intuitively, this invariant tells that a null node, i.e., $n_{\mathtt{null}}$ paired with some context, models only the null object $o_{\mathtt{null}}$.

Invariant 2. $\forall d \in ID_{A(m)}, \langle n, f, n_L\rangle \in O^\#_d \rightarrow e^\#(\Xi^\#_d)(n) \wedge e^\#(\Xi^\#_d)(n_L)$.

Proof: As escapability propagates along the outside edge $\langle n, f, n_L \rangle$, it is enough to prove $e^{\#}(\Xi_d^{\#})(n)$. We do so by induction on the list of interesting dates. The invariant is trivially satisfied at the beginning of $A(m)$, and each transition preserves the invariant as follows. The transfer function for a LOAD instruction adds only outside edges starting from nodes which escape. A RETURN inside $A(m)$ projects the abstract state through $\alpha(d)$; $\alpha(d)$ converts each path from the abstract state before the RETURN into a path in the abstract state after the RETURN. So, every node that escapes in the state before the RETURN escapes in the state after it too and our invariant is preserved. The other instructions are irrelevant because they do not add outside edges and preserve existing paths, and, by consequence, the escapability status of the nodes. □

Invariant 3. *$\forall d \in ID_{A(m)}$, $\forall o \in Object \setminus \{o_{\texttt{null}}\}$, if $d_C(o) \leq d$ and $\neg e_d(o)$ then $\{n \mid o \ \rho_d \ n\} = \{n^I_{lb,c}\}$ where lb is the label of the instruction that created o (the instruction executed in the transition $\Xi_{d_C(o)-1} \Rightarrow_T \Xi_{d_C(o)}$) and c is some calling context.*

Proof: As $\neg e_d(o)$, location o was created by $A(m)$ (otherwise, Constraint 4.5 would set $e_d(o)$ to be true). So, the NEW instruction from label lb that created o was processed by the abstract semantics, and an inside node corresponding to lb was put to model o. This means that $\{n \mid o \ \rho_d \ n\} \cap INode^{\#} = \{n^I_{lb,c}\}$ where the calling context c depends on the context at the date when the abstract semantics executed the NEW instruction, and on the RETURN instructions which followed. However, for the purpose of this proof we are not interested in the particular formula of c. Suppose for the sake of contradiction that there are some other nodes that model o. There are three types of instructions which might extend the set $\{n \mid o \ \rho_d \ n\}$: NEW, LOAD, and unanalyzed CALL. Some RETURN instructions might "adjust" the context of some of the nodes from the set, but they do not add new nodes; NULLIFY is also irrelevant because $o \neq o_{\texttt{null}}$. There is a single relevant NEW instruction, the one that created o, and the node that it introduced is already in the set. We prove that the other two cases cannot occur.

Suppose that $A(m)$ executed at moment $d' < d$ a LOAD instruction that introduced a load node to model o. This means that $A(m)$ read the heap edge $\langle o', f, o \rangle$, where $e_{d'}(o')$ was true. If this edge was created by $A(m)$, it is present in the set $H_d^{A(m)}$, and by constraints 4.6 and 4.4, we immediately have that $e_d(o)$ is true; contradiction! Otherwise, if the edge was created by a STORE from some other part of the program, at the date of the execution of that STORE, o was reachable from "outside" $A(m)$. By Lemma 7, o escaped at that time and, as escapability is a cummulative property, $e_d(o)$ is true; again, contradiction!

Finally, suppose that $A(m)$ executed at moment $d_1 < d$ an unanalyzable CALL that returned o at moment d_2, $d_1 < d_2 \leq d$. In the abstract semantics, the return node $n^R_{lb,c}$ models the object o. In the transition from date $d_2 - 1$ to d_2, the method called by the unanalyzed CALL executed a RETURN instruction, "`return` v"; in the stack frame corresponding to that method, at moment $d_2 - 1$, v points to o. So, at moment $d_2 - 1$, o is reachable from outside $A(m)$.

By the construction of $A(m)$, $d_1, d_2 \in ID_{A(m)}$; hence, e_{d_1} and e_{d_2} are defined. As $\neg e_d(o)$ and escapability is a cummulative property (Invariant 4.4), we obtain $\neg e_{d'}(o), \forall d' \leq d$. In particular, $\neg e_{d_1}(o)$ and $\neg e_{d_2}(o)$. As $\neg e_{d_2}(o)$, o is unreachable from the parameters that are passed to the unanalyzable CALL (otherwise, constraints 4.6 and 4.9 would set $e_{d_2}(o)$ to true).

By Lemma 7, as $\neg e_{d_1}(o)$, o is unreachable from outside $A(m)$ at d_1. By applying the previous observation, we have that o is unreachable from outside $A(m)$ even at $d_1 + 1$. As from $d_1 + 1$ to $d_2 - 1$, only instructions from outside $A(m)$ are executed, o remains unreachable from outside $A(m)$ at $d_2 - 1$ (we can prove this formally, by induction, in the style of Lemma 6). Hence, o cannot be returned from the unanalyzed CALL. Contradiction! This completes the proof of Invariant 3. □

The following three invariants are very closely related: the validity of one depends on the validity of the others. For this reason, we prove them together, in a single proof by induction.

Invariant 4. $\forall d \in ID_{A(m)}, \langle o_1, f, o_2\rangle \in H_d^{A(m)} \rightarrow \exists n_1, n_2 \in Node, (o_1\ \rho_d\ n_1) \wedge (o_2\ \rho_d\ n_2) \wedge (\langle n_1, f, n_2\rangle \in I_d^{\#})$.

Invariant 5. $\forall d \in ID_{A(m)}, \forall v \in V, L_{d,t}(v) = o \rightarrow \exists n \in Node, (o\ \rho_d\ n) \wedge (n \in L_d^{\#}(v))$.

Invariant 6. $\forall d \in ID_{A(m)}, \forall o \in Object \setminus \{o_{\texttt{null}}\}, \forall n \in Node^{\#}, e_d(o) \wedge (o\ \rho_d\ n) \rightarrow e^{\#}(\Xi_d^{\#})(n)$.

Proof for Invariants 4, 5, 6: As $d \in ID_{A(m)}$, $\exists j, 0 \leq j \leq 2r+1$ such that $d = id_j$. We prove the three invariants by induction on j.

Initial state In the initial state $d = id_0$, $H_{id_0}^{A(m)} = \emptyset$ and Invariant 4 is trivially satisfied. $L_{id_0,t}$ is defined only for parameters: $L_{id_0,t}(p_i) = o_i$ where o_i's are the objects actually sent as arguments in the call that started $A(m)$. As $\rho_{id_0} = \{\langle o_i, n_{m,i,[]}^P\rangle\}$, Invariant 5 is satisfied. Finally, as trivially $e^{\#}(\Xi_{id_0}^{\#})(n_{m,i,[]}^P)$, Invariant 6 is satisfied, too.

Induction step Assume that the three invariants hold for $d \in \{id_0, \ldots, id_j\}$. We will prove that they hold for $d = id_{j+1}$ too. If $j = 2i + 1$, then :

$$\langle \Xi_{id_{2(i+1)}}^{\#}, \rho_{id_{2(i+1)}}, c_{id_{2(i+1)}}\rangle = \langle \Xi_{id_{2i+1}}^{\#}, \rho_{id_{2i+1}}, c_{id_{2i+1}}\rangle$$

By the induction hypothesis, the three invariants are trivially satisfied for $d = id_{j+1} = id_{2(i+1)}$. In the rest of this proof, we work on the more interesting case of $j = 2i$, when passing from id_j to id_{j+1} corresponds to a real transition:

$$\begin{aligned} d &= id_{2i+1} \\ \langle \Xi_d^{\#}, \rho_d, c_d\rangle &= \langle \Xi_{id_{2i+1}}^{\#}, \rho_{id_{2i+1}}, c_{id_{2i+1}}\rangle = [\![lb_{id_{2i},t}]\!]^{\#}(\langle \Xi_{id_{2i}}^{\#}, \rho_{id_{2i}}, c_{id_{2i}}\rangle) \end{aligned}$$

The method for proving the three invariants at date id_{2i+1} is the same: case analysis on the type of the instruction $P(lb_{id_{2i},t})$. For all the instructions except the unanalyzable CALL, this is the instruction from the transition $\Xi^{\#}_{id_{2i}} \Rightarrow_T \Xi^{\#}_{id_{2i+1}}$. For the unanalyzable CALL, it is the first instruction from the chain of transitions $\Xi^{\#}_{id_{2i}} \Rightarrow^*_T \Xi^{\#}_{id_{2i+1}}$; the other instructions from that transition chain are from the method transitively invoked by the unanalyzable CALL, and so, they cannot affect the state of the local variables of $A(m)$ nor the set of heap edges $H^{A(m)}$.

Before examining the invariants, we prove the following auxiliary lemma:

Lemma 9. *Let $id_{2i} \in ID_{A(m)}$ be an even-indexed date that corresponds to the beginning of the transition chain $\Xi^{\#}_{id_{2i}} \Rightarrow^* \Xi^{\#}_{id_{2i+1}}$. If the instruction at label $lb_{id_{2i},t}$ (the instruction executed in that transition chain) is not a RETURN inside $A(m)$, then $\rho_{id_{2i}} \subseteq \rho_{id_{2i+1}}$.*

Proof of Lemma 9: Simple inspection of the definition of the transfer function $[\![.]\!]^{\#}$ for the abstract semantics (Figure 4-5), and the definition of *update_*ρ (Figure 4-8). □

We continue the proof of the three invariants.

Invariant 4 The transfer function for a RETURN inside $A(m)$ modifies both the set of inside nodes and the abstraction relation by adjusting the context of the nodes appearing in those structures. As this adjustment is done in the same way in both structures, the validity of Invariant 4 extends from id_{2i} to id_{2i+1}.

From the remaining instructions, only NEW and STORE create new heap references. All the other instructions leave the heap and the set of inside edges unchanged; as $\rho_{id_{2i}} \subseteq \rho_{id_{2i+1}}$ (Lemma 9), they trivially preserve Invariant 4.

The NEW instruction "v = `new` C" creates the heap edges $\{\langle o, f, o_{\texttt{null}}\rangle \mid f \in \mathit{fields}(C)\}$ in order to initialize all the fields of the newly created location o. From the definition of the transfer function for updating the abstract states $[\![., .]\!]$ (Figure 4-7), we have:

$$I^{\#}_{id_{2i+1}} = I^{\#}_{id_{2i}} \cup \{\langle n^I_{lb}, f, n_{\texttt{null},c}\rangle \mid f \in \mathit{fields}(C)\} \text{ and}$$
$$\rho_{id_{2i+1}} = \rho_{id_{2i}} \cup \{\langle o, n^I_{lb}\rangle\} \cup \{\langle o_{\texttt{null}}, n_{\texttt{null},c}\rangle\}$$

and Invariant 4 is clearly valid at date id_{2i+1}.

So, suppose that the last instruction was the STORE instruction "$v_1.f = v_2$" and that $L_{id_{2i}}(v_1) = o_1 \neq o_{\texttt{null}}$ (the program cannot write at `null`), $L_{id_{2i}}(v_2) = o_2$. The only new heap edge is $\langle o_1, f, o_2\rangle$. By our induction hypothesis, Invariant 5 is valid at moment id_{2i}, which implies

$$\exists n_1 \text{ such that } (o_1\ \rho_{id_{2i}}\ n_1) \wedge (n_1 \in L^{\#}_{id_{2i}}(v_1)) \text{ and}$$
$$\exists n_2 \text{ such that } (o_2\ \rho_{id_{2i}}\ n_2) \wedge (n_2 \in L^{\#}_{id_{2i}}(v_2))$$

Also, as $o_1 \neq o_{\texttt{null}}$, by Invariant 1, $n_1 \notin \mathcal{N}$. Due to the abstract semantics of the STORE instruction, $\langle n_1, f, n_2 \rangle \in I^{\#}_{id_{2i+1}}$. As $\rho_{id_{2i}} \subseteq \rho_{id_{2i+1}}$ (Lemma 9), Invariant 4 is valid at date id_{2i+1}.

Invariant 5 The cases of STORE, IF, THREAD START, and the final RETURN (the one that terminates $A(m)$) are easy to handle. By a quick inspection of the transition relation for the concrete semantics (Figure 4-3), we notice that these instructions do not modify the state of the local variables: $L_{id_{2i+1}} = L_{id_{2i}}$. Also, in the case of the abstract semantics (Figure 4-7), $L^{a}_{id_{2i+1}} = L^{a}_{id_{2i}}$ for all of these instructions except RETURN. The abstract semantics of the final RETURN affects only the special dummy variable v_{ret} which we use to store the result of the method; as this variable does not exist in the concrete semantics, it does not introduce any complication. As $\rho_{id_{2i}} \subseteq \rho_{id_{2i+1}}$, the validity of Invariant 5 propagates from id_{2i} to id_{2i+1}.

In the case of a COPY instruction "$v_1 = v_2$", we just have to look at the new value of v_1: $L_{id_{2i+1}}(v_1) = o = L_{id_{2i}}(v_2)$. By our induction hypothesis, Invariant 5 holds at date id_{2i}, which proves that $\exists n$ such that $(o\ \rho_{id_{2i}}\ n) \wedge (n \in L^{\#}_{id_{2i}}(v_2))$. By our definition of $[\![.]\!]$ for a COPY instruction, we have $n \in L^{\#}_{d_{2i+1}}(v_1)$, and hence, Invariant 5 is valid at date id_{2i+1}.

A NEW instruction "$v = \texttt{new}\ C$" modifies L by setting $L_{id_{2i+1}}(v) = o$, where o is the new object created by this instruction. The abstract semantics sets $L^{\#}_{id_{2i+1}}(v) = \{n^{I}_{lb,c}\}$, where lb is the label of the NEW instruction and c is the current calling context. It also extends the abstraction relation such that $o\ \rho_{id_{2i+1}}\ n^{I}_{lb,c}$. The case of a NULLIFY instruction is similar: $L_{id_{2i+1}}(v) = o_{\texttt{null}}$, $L^{\#}_{id_{2i+1}}(v) = n_{\texttt{null},c}$, and $o_{\texttt{null}}\ \rho_{id_{2i+1}}\ n_{\texttt{null},c}$. In both cases, Invariant 5 is valid at date id_{2i+1}.

The case of an unanalyzed CALL "$v_R = v_0.s(v_1, \ldots, v_j)$" is a bit different from the others, because id_{2i+1} is not simply $id_{2i} + 1$: although the abstract semantics executes a single transition[5], there are many transitions in the concrete semantics, corresponding to the execution of the instructions of the methods transitively invoked by the unanalyzable CALL and the instructions of threads other than t. However, from all these many instructions, only the last one, the RETURN from the transition $\Xi_{id_{2i+1}-1} \Rightarrow_T \Xi_{id_{2i+1}}$, modifies the state of the local variables of $A(m)$. It sets $L_{id_{2i+1}}(v) = o$, where o is the location returned by the unanalyzed CALL. In the abstract semantics, $L^{\#}_{id_{2i+1}}(v) = \{n^{R}_{lb,c}\}$, where lb is the label of the CALL instruction, and c is the current calling context. Since $o\ \rho_{id_{2i+1}}\ n^{R}_{lb,c}$, Invariant 5 holds at date id_{2i+1}.

[5]I.e., one application of the abstract transfer function $[\![.]\!]^{\#}$.

In the case of a RETURN instruction inside $A(m)$, the transfer function $[\![.]\!]^{\#}$ (Figure 4-5) closely models the transition from the concrete semantics (Figure 4-3). Consider a RETURN instruction "`return` v", and let v_R be the variable that stores the result of the corresponding CALL instruction. The abstract semantics pops from the stack the state of the callee local variables, and in the state of the local variables for the caler, puts v_R to point to the nodes which were pointed to by v in the callee. The validity of Invariant 5 at date $d = id_{2i+1}$ for variable v_R follows from the validity of Invariant 5 for variable v at date id_{2i}. The meticulous reader might notice that the transfer function also adjusts the context of the nodes. As this adjustment is also done in the abstraction relation, it does not harm our proof. The other variables remained unmodified both in the concrete and the abstract semantics from the date of the corresponding CALL and so, by the induction hypothesis, they preserve Invariant 5 too.

If we look only at the state of the local variables, an analyzable CALL is very similar to a series of COPY instructions: it copies local variables from the caller into the parameters of the callee. By the same arguments as in the case of COPY, Invariant 5 is valid at date id_{2i+1}.

We left for the end the most difficult case: a LOAD instruction "$v_2 = v_1.f$". Suppose that right before its execution, $L_{id_{2i}}(v_1) = o_1$ and $\langle o_1, f, o_2 \rangle \in H_{id_{2i}}$; by our hypothesis that the analyzed program is correct, $o_1 \neq o_{\texttt{null}}$. With these notations, the LOAD instruction reads the heap edge $\langle o_1, f, o_2 \rangle$ and sets $L_{id_{2i+1}}(v_2) = o_2$. By the induction hypothesis, Invariant 5 is valid at date id_{2i} and so,

$$\exists n_1 \text{ such that } (n_1 \in L^{\#}_{id_{2i}}(v_1)) \wedge (o_1 \ \rho_{id_{2i}} \ n_1)$$

There are two cases:

1. If $e_{id_{2i}}(o_1)$, then by Invariant 6 at date id_{2i}, $e^{\#}(id_{2i})(n_1)$ is true. In this case, the abstract semantics puts $n^L_{lb,c} \in L^{\#}_{id_{2i+1}}(v_2)$, where lb is the label of the LOAD instruction, and c is the current calling context, and updates ρ such that $o_2 \ \rho_{id_{2i+1}} \ n^L_{lb,c}$. This preserves Invariant 5.

2. If $\neg e_{id_{2i}}(o_1)$, by Invariant 3 at date id_{2i}, $\{n \mid o \ \rho_{id_{2i}} \ n\} = \{n^I_{lb,c}\} = \{n_1\}$; so, the only node that abstracts location o at date id_{2i} is $n_1 = n^I_{lb,c}$.

   Also $\neg e_{id_{2i}}(o_1)$ implies that o_1 cannot be accessed from outside $A(m)$ (Lemma 7), the heap edge $\langle o_1, f, o_2 \rangle$ was created by $A(m)$, i.e., $\langle o_1, f, o_2 \rangle \in H^{A(m)}_{id_{2i}}$. By the induction hypothesis, Invariant 4 is valid at date id_{2i} which, together with the fact that $\{n \mid o \ \rho_{id_{2i}} \ n\} = \{n_1\}$, gives us that:

   $$\exists n_2 \text{ such that } (o_2 \ \rho_{id_{2i}} \ n_2) \wedge (\langle n_1, f, n_2 \rangle \in I^{\#}_{id_{2i}})$$

   By the definition of the abstract semantics, $n_2 \in L^{\#}_{id_{2i+1}}(v_2)$. As $\langle o_2, n_2 \rangle \in \rho_{id_{2i}} \subseteq \rho_{id_{2i+1}}$, Invariant 4 is valid at date id_{2i+1}.

In all cases, Invariant 5 is valid at date id_{2i+1}.

Invariant 6 Recall that the (abstract) escape predicate $e^{\#}(\Xi_d^{\#})$ (Definition 12) tells whether in the abstract state $\Xi_d^{\#}$ for date d, a specific node is reachable from a set of *escapability sources* (the set N from the definition) along a path of inside and outside edges. Similary, in the concrete semantics, the concrete escape predicate e_d (Definition 10) tells whether an object is reachable in $H_d^{A(m)}$ (the heap references created by $A(m)$) from one of the objects directly escaped by one of the constraints 4.5, 4.7, 4.8, and 4.9.

COPY, NULLIFY, and IF preserve the invariant: in the concrete semantics, they do not directly 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 the set of escapability sources and all the paths consisting of inside/outside edges. As a result, all the nodes which escaped at date id_{2i} still escape at date id_{2i+1}. As the abstraction relation is not modified by these instructions — the extension done by NULLIFY is irrelevant, because $o \neq o_{\texttt{null}}$ — they preserve Invariant 6. We can apply the same reasoning for an analyzable CALL and for a RETURN inside $A(m)$ (i.e., not the last RETURN which ends $A(m)$)[6].

A NEW instruction updates the abstraction relation such that $o\ \rho_{id_{2i+1}}\ n^I_{lb,c}$, where o is the newly created object. As o does not escape anywhere yet, i.e., $\neg e_{id_{2i+1}}(o)$, the invariant is preserved. A LOAD instruction might introduce a load node $n^L_{lb,c}$ to model the loaded object[7]. As $e^{\#}(\Xi^{\#}_{id_{2i+1}})(n^L_{lb,c})$ from the very beginning[8], this instruction too preserves the invariant.

RETURN, THREAD START, and unanalyzable CALL escape nodes, due to one of the constraints 4.7, 4.8, and 4.9 from Definition 10. This escape info propagates along the heap edges by Constraint 4.6. We treat only the case of the unanalyzable CALL; the other two cases are similar.

Suppose for the sake of contradiction that the invariant is not true at date id_{2i+1}, i.e., there exist an object o and a node n such that $e_{id_{2i+1}}(o)$, $o\ \rho_{id_{2i+1}}\ n$, and $\neg e^{\#}(\Xi^{\#}_{id_{2i+1}})(n)$. n cannot be the return node $n^R_{lb,c}$ which corresponds to the unanalyzable CALL because a return node trivially escapes. As a consequence, n models o even at date id_{2i}. Furthermore, n does not escape in the state $\Xi^{\#}_{id_{2i}}$; otherwise, by Lemma 8, it would escape in $\Xi^{\#}_{id_{2i+1}}$ too. By our induction hypothesis, Invariant 6 is valid at date id_{2i}; hence, $\neg e_{id_{2i}}(o)$. Also note that due to Definition 9, $H^{A(m)}_{id_{2i+1}} = H^{A(m)}_{id_{2i}}$. The only reason o escapes at date id_{2i+1} is that it is reachable from

[6]Again, the meticulous reader might notice that a RETURN inside $A(m)$ adjusts the context of some nodes. However, as this adjustement is done uniformly, into the abstract state and the abstraction relation alike, it does not affect our proof.

[7]c is the current calling context.

[8]The load node is reachable via the newly introduced outside edges from one or more escaped nodes.

one of the nodes that directly escaped into the unanalyzable CALL by Constraint 4.9, by a path of edges from $H^{A(m)}_{id_{2i}}$.

Hence, there exists a path, possibly of length zero, $o_0, o_1, \ldots, o_p = o$, $\forall j, \langle o_j, f_j, o_{j+1}\rangle \in H^{A(m)}_{id_{2i+1}}$, that reaches o from a location o_0 that is pointed to by one of the parameters passed to the unanalyzable CALL. As $\neg e_{id_{2i}}(o)$ and escapability is propagated along the edges from $H^{A(m)}_{id_{2i}}$, $\neg e_{id_{2i}}(o_j)$ for any object o_j from the path. By Invariant 3, for each of the objects o_j, there is a single node n_j that models it, i.e., $o_j\ \rho_{id_{2i}}\ n_j$, where $n_p = n$. Therefore, by Invariant 4, which is valid at moment id_{2i} by the induction hypothesis, we have a corresponding path along the inside edges from the abstract state at moment id_{2i}: $n_0, n_1, \ldots, n_p = n$, $\langle n_j, f_j, n_{j+1}\rangle \in I^{\#}_{id_{2i}} \subseteq I^{\#}_{id_{2i+1}}$.

Now, by Invariant 5 at moment id_{2i}, as one of the parameters from the unanalyzable CALL, say v points to o_0 in the concrete semantics, in the abstract state at date id_{2i}, v points to the unique node that models o_0, n_0, i.e., $n_0 \in L^{\#}_{id_{2i}}(v)$. By the definition of $[\![.,.]\!]$ in the case of an unanalyzable CALL, $n_0 \in U^{\#}_{id_{2i+1}}$; therefore, n_0 escapes in the state $\Xi^{\#}_{id_{2i+1}}$. As the abstract escape information propagates along the edges from $I^{\#}_{id_{2i}}$, $n_p = n$ escapes in the abstract state $\Xi^{\#}_{id_{2i+1}}$. Contradiction.

The only remaining case is that of a STORE instruction "$v_1.f = v_2$". This instruction does not create new objects nor extend the abstraction relation. However, it creates a new edge, which can generate new paths in $H^{A(m)}_{id_{2i+1}}$. Suppose for the sake of contradiction that Invariant 6 is not valid at date id_{2i+1}, i.e., there exist an object o and a node n such that $e_{id_{2i+1}}(o)$, $o\ \rho_{id_{2i+1}}\ n$, and $\neg e^{\#}(\Xi^{\#}_{id_{2i+1}})(n)$. We immediately obtain that $o\ \rho_{id_{2i}}\ n$ and $\neg e^{\#}(\Xi^{\#}_{id_{2i}})(n)$ and as Invariant 6 is valid at date id_{2i} (by the induction hypothesis), we have that $\neg e_{id_{2i}}(o)$. The reason o became escaped at date id_{2i+1} is that the edge introduced by the STORE instruction made o reachable from one of the objects which were already escaped at date id_{2i}.

Let us consider one of the shortest paths in $H^{A(m)}_{id_{2i+1}}$ from an object o_0 such that $e_{id_{2i}}(o_0)$ is true, to o. Let the objects from this path be $o_0, o_1, \ldots, o_p = o$. Due to the way we selected this path, with the exception of o_0, no other object from this path escapes at date id_{2i}. By applying the same idea as in the case of an unanalyzable CALL, we obtain 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 $H^{A(m)}_{id_{2i}}$. But it appears in the set $H^{A(m)}_{id_{2i+1}}$, which means that the STORE instruction created it, i.e., $L_{id_{2i},t}(v_1) = o_0$, and $L_{id_{2i},t}(v_2) = o_1$. By Invariant 5 at date id_{2i}, valid by our induction hypothesis, $L^{\#}_{id_{2i}}(v_1)$ contains one of the nodes which models o_0, call it n_0, and $L^{\#}_{id_{2i}}(v_2)$ contains the unique node which models o_1, the node n_1. Thus, at date id_{2i+1}, we have a path of inside edges from n_0 to n.

As Invariant 6 is valid at date id_{2i}, $e^{\#}(\Xi^{\#}_{id_{2i}})(n_0)$ is true. So, n_0 escapes in state $\Xi^{\#}_{id_{2i+1}}$. As n is reachable from an already escaped node, it escapes too. But we supposed that n is captured in $\Xi^{\#}_{id_{2i+1}}$! Contradiction.

In all cases Invariant 6 is valid at date id_{2i+1}. This completes our proof by induction for invariants 4, 5, and 6. □

Lemma 10. *Consider an interesting date for $A(m)$, $d \in ID_{A(m)}$. Let $\Xi^{\#}_d$ and c_d be the abstract state, respectively the context constructed by the abstract semantics of $A(m)$ for date d. If $n = \langle n', c\rangle$ is a node appearing in $\Xi^{\#}_d$, then its context c is a suffix of c_d.*

Proof: Induction on the index inside the list of interesting dates of $A(m)$. The property is clearly true at the beginning of $A(m)$. The execution of an instruction that is not a RETURN inside $A(m)$ preserves the property for the nodes which already exist; such an instruction might also create new nodes, but these nodes have the context c_d which is clearly a suffix of itself. A RETURN inside $A(m)$ uniformly removes the date of the corresponding CALL from the head of the current context c_d and from the head of each node context; hence, it preserves the property too. □

It is easy to prove that due to the nesting of the CALL/RETURN instructions and due to our way of maintaining the calling context, at the end of the activation $A(m)$, i.e. at date $d = id_{2r+1}$, the context is empty: $c_{id_{2r+1}} = []$. So, by Lemma 10, in the abstract state $\Xi^{\#}_{id_{2r+1}}$, all the nodes have empty contexts. In particular, the inside node which represents the objects allocated at label lb is $n^I_{lb,[]}$. Now, we are ready to prove the following theorem.

Theorem 11. *Consider a concrete execution trace T, an activation $A(m)$ of method m and suppose that, at some date in its execution, $A(m)$ allocated an object o by executing the NEW instruction from label lb. Also, consider the list of interesting dates for $A(m)$, $ID_{A(m)} = [id_0, \ldots, id_{2r+1}]$ and suppose the abstract semantics obtained the following abstract state for the final date id_{2r+1}:*

$$\Xi^{\#}_{id_{2r+1}} = \langle I^{\#}_{id_{2r+1}}, O^{\#}_{id_{2r+1}}, L^{\#}_{id_{2r+1}} : J^{\#}_{id_{2r+1}}, S^{\#}_{id_{2r+1}}, U^{\#}_{id_{2r+1}}\rangle$$

If $\neg e^{\#}(id_{2r+1})(n^I_{lb,[]})$, then the lifetime of object o is included into the execution time interval for $A(m)$, i.e. $[d_C(o), d_D(o)] \subseteq [id_0, id_{2r}]$.

Proof: As $A(m)$ created o, $d_C(o) \geq id_0$. To complete the proof, we need to prove that $d_D(o) \leq id_{2r}$. Let $\Xi_{id_{2r+1}}$ be the concrete state at date id_{2r+1}. By Lemma 7, we have that

$$reachable(\Xi'_{id_{2r+1}})(o) \rightarrow e_{id_{2r+1}}(o)$$

where $\Xi'_{id_{2r+1}} = outside_{A(m)}(\Xi_{id_{2r+1}})$. It is easy to note that $outside_{A(m)}(\Xi_{id_{2r+1}}) = \Xi_{id_{2r+1}}$: the stack frame created by $A(m)$ was removed by the RETURN from the transition $\Xi_{id_{2r}} \Rightarrow_T \Xi_{id_{2r+1}}$. Therefore,

$$reachable(\Xi_{id_{2r+1}})(o) \rightarrow e_{id_{2r+1}}(o)$$

When the abstract semantics processes the NEW instruction at label lb that created object o, it puts the inside node $n^I_{lb,c}$ to model o, where c is the current calling context

at that moment. At the end of $A(m)$, all the nodes have an empty calling context; hence, $o\ \rho_{id_{2r+1}}\ n^I_{lb,[]}$. By Invariant 6,

$$e_{id_{2r+1}}(o) \rightarrow e^{\#}(id_{2r+1})(n^I_{lb,[]})$$

Combining these two implications we have that

$$reachable(\Xi_{id_{2r+1}})(o) \rightarrow e^{\#}(id_{2r+1})(n^I_{lb,[]})$$

and so, as $\neg e^{\#}(id_{2r+1})(n^I_{lb,[]})$, o is no longer reachable at date id_{2r+1}. By the definition of $d_D(o)$, this implies $d_D(o) < id_{2r+1} = id_{2r} + 1$, which completes our proof. □

As the stack frame for the instance of method m which is the "root" of $A(m)$ is created at date id_0 and destroyed at date id_{2r}, we have the following obvious corollary, which represents a sufficient condition for stack allocation:

Corollary 12. *In the conditions of Theorem 11, if $\neg e^{\#}(id_{2r+1})(n^I_{lb,[]})$, then all the objects that $A(m)$ creates by executing the NEW instruction from label lb can be safely allocated in the stack frame of the instance of method m which is the "root" of $A(m)$.*

4.4 Analysis *vs.* Abstract Semantics

In the previous section, we proved that the abstract semantics conservatively models the concrete semantics, with respect to a set of invariants. This enabled us to obtain Corollary 12, which gives a sufficient condition for the stack allocation of the objects allocated by the activation $A(m)$. That condition is defined for the abstract state computed for the end of $A(m)$. However, we defined the abstract semantics just for the purpose of the correctness proof; we cannot even compute it statically. What we actually need is a condition defined on the points-to graphs computed by the pointer analysis.

In this section, we prove that the pointer analysis is a conservative approximation of the abstract semantics, and hence, of the concrete semantics, too. This will enable us to extend the sufficient condition of Corollary 12 into a proof for Theorem 5.

Consider a method m and an activation $A(m)$. As the condition from Corollary 12 was expressed on the abstract state for the end of $A(m)$, $\Xi^{\#}_{id_{2r+1}}$, we are particulary interested in proving that the points-to graph G that the pointer analysis computes for the exit point of method m, conservatively approximates $\Xi^{\#}_{id_{2r+1}}$. At a first look, if we ignore the inter-procedural aspects, the pointer analysis and the abstract semantics look very similar, except that the first one works with nodes from the set *Node*, while the other one operates with nodes with context from the set $Node^{\#} = Node \times Context$. However, we have already proved that in the abstract state for the end of $A(m)$, all nodes have the context []. Therefore, the isomorphism $\beta(n) = \langle n, [] \rangle$ allows us to compare analysis data structures against abstract semantics equivalent structures.

The major contribution of this section is the following theorem:

Theorem 13. *Let $\Xi^{\#}_{id_{2r+1}}$ be the abstract state that the abstract semantics computes for date id_{2r+1} (the end of $A(m)$) and let $G = \circ A(exit_m)$ be the points-to graph that the pointer analysis computes for the exit point of method m. Then,*

$$\Xi^{\#}_{id_{2r+1}} \sqsubseteq \beta(G)$$

The rest of this section is organized as follows. First, in Section 4.4.1, we present several auxiliary notions and results, and give the formal definition of the node conversion β. Next, in Section 4.4.2, we prove Theorem 13. The proof is quite long and difficult. In order to simplify it, we give a high level proof which uses an auxiliary result, Equation 4.17. We prove the correctness of this auxiliary result in Section 4.4.3. As this proof is very long, too, once again, we give a high-level proof which uses two auxiliary results: Equation 4.21 and Equation 4.23. Section 4.4.4 presents several results about the node mappings. In Section 4.4.5, we use these results to prove the correctness of Equation 4.21. This proof uses all the constraints from the definition of the *mapping* function (Figure 2-8) which is the core of the inter-procedural analysis. The proof for Equation 4.23 is rather technical and uninteresting; for completeness, we present it in Appendix A.

4.4.1 Auxiliary Notions

Notations: Throughout this section, we suppose that we have a concrete execution trace T and, in T, an activation $A(m)$ of a method m which starts at date d_s and takes place in thread t. We also suppose that the list of interesting dates for $A(m)$ is

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

The activation $A(m)$ is terminated by the RETURN instruction executed in the transition from id_{2r} to id_{2r+1}. We also use the following notation simplification: if $\Xi^{\#}_x$ is an abstract state (where x is an arbitrary subscript), then we denote all the components of $\Xi^{\#}_x$ by using the same subscript, i.e.,

$$\Xi^{\#}_x = \langle I^{\#}_x,\ O^{\#}_x,\ L^{\#}_x : J^{\#}_x,\ S^{\#}_x,\ U^{\#}_x \rangle$$

We emphasize $L^{\#}_x$, the state of the local variables of the current method (the topmost method from the call chain), by separating it from the rest of the stack, $J^{\#}_x$. We formulate all the results from this section in the context of these notations. We explicitly indicate any case where these conventions do not apply.

As the pointer analysis does not step into the callees (as the abstract semantics does), not all the dates from $ID_{A(m)}$ are relevant for it: for the dates inside the execution of a callee, the abstract semantics constructs an abstract state while no corresponding points-to graph is created by the analysis of method m. For this reason, we select from $ID_{A(m)}$ just the dates that correspond to the execution inside the instanciation of m that is the root of $A(m)$. Formally, we have the following definition:

Definition 13 (Intra-procedural interesting dates). *Consider an execution trace T and an activation $A(m)$ of method m, whose list of interesting dates are $ID_{A(m)} = [id_0, \ldots, id_{2r+1}]$. We construct the list of* intra-procedural interesting dates, *$IP_{A(m)}$, as follows:*

1. *We scan the dates id_{2i} in increasing order: $id_0, id_2, \ldots, id_{2r}$. In each such date $A(m)$ starts the execution of an instruction.*

2. *For each examined date id_{2i}, if the instruction executed by $A(m)$ is not an analyzable CALL, we add both id_{2i} and id_{2i+1} to $IP_{A(m)}$. If it's an analyzable CALL, we add to $IP_{A(m)}$ both the date id_{2i} when the CALL was executed and the date id_{2j+1} when the corresponding RETURN instruction terminates; in this case, we also skip all the dates in between, i.e., we don't step into the callees.*

The additions to $IP_{A(m)}$ are always done in pairs. Therefore,

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

The list $IP_{A(m)}$ is the concatenation of small lists of two dates $[ip_{2i}, ip_{2i+1}]$ which corresponds to the execution path of $A(m)$ inside *method*: 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 an analyzable CALL. Notice that we are not interested in the instructions executed by the callees of $A(m)$. The only difference between $ID_{A(m)}$ and $IP_{A(m)}$ is that the construction of $IP_{A(m)}$ supposes that all the CALL instructions of m are unanalyzable. Finally, note that the last two dates from $IP_{A(m)}$ and $ID_{A(m)}$ are the same: id_{2r} and id_{2r+1}; the transition from id_{2r} to id_{2r+1} executes the RETURN which terminates $A(m)$.

Lemma 14. *For any intra-procedural interesting date, $d \in IP_{A(m)}$, the context c_d computed by the abstract semantics of $A(m)$ for d is empty: $c_d = []$.*

Proof sketch: An analyzable CALL adds the current date to the head of the context; the corresponding RETURN removes this date. Hence, at any date, the current context contains (in reverse order), the dates of the analyzable CALLs which have not been matched by a corresponding RETURN yet. At each intra-procedural date $d \in IP_{A(m)}$, all the CALLs have returned. Therefore, $c_d = []$. □

Now, we are ready to prove that not only at the end of $A(m)$, but in any intra-procedural interesting date $d \in IP_{A(m)}$, the abstract state computed by the abstract semantics contains only nodes with an empty context:

Corollary 15. *Consider an intra-procedural interesting date for $A(m)$, $d \in IP_{A(m)}$ and let $\Xi^\#_d$ be the abstract state that the abstract semantics of $A(m)$ constructed for date d. If $n = \langle n', c \rangle$ is a node appearing in $\Xi^\#_d$, then its context is empty, i.e., $c = []$.*

Proof: By Lemma 10, the context of any node appearing in the abstract state $\Xi^\#_d$ is a suffix of the calling context c_d. Combining this with Lemma 14, we obtain the corollary. □

As a result of Corollary 15, it is possible to compare the abstract state from the end of $A(m)$ with the points-to graph that the pointer analysis computes for the end of m. For this, we need the following node conversion:

Definition 14. *Let β be the conversion $\beta(n) = \langle n, [] \rangle$. This conversion takes a data stucture specific to the pointer analysis and propagates deep inside it (in the same style like $\alpha(d)$), returning an equivalent abstract semantics data structure.*

A first, easy result that we obtain is the following:

Lemma 16. *If lb is the label of an instruction that is not an analyzable CALL or a RETURN inside $A(m)$, then*

$$\beta([\![lb]\!]^a(G)) = [\![lb, []]\!](\beta(G)), \forall G \in PTGraph^a$$

Proof: By a quick inspection of the definition of $[\![.,.]\!]$ (Figure 4-7) we note that $[\![lb, []]\!]$ is exactly like $[\![lb]\!]^a$, except that instead of manipulating node n, it manipulates node $\beta(n)$. Hence the lemma. □

Definition 15 (Call depth of an activation). *For each interesting date $d \in ID_{A(m)}$, let $\Xi_d^\# = \langle I_d^\#, O_d^\#, J_d^\#, S_d^\#, U_d^\# \rangle$ be the abstract state that the abstract semantics of $A(m)$ computes for date d. The* call depth *of the activation $A(m)$ is the maximal height of the stack $J_d^\#$, $d \in ID_{A(m)}$.*

Intuitively, the call depth of an activation is the length of the maximal call chain from the activation. For example, an activation that does not call any method has a call depth of zero; an activation which calls a method which itself calls another method, has a call depth of at least two, etc.

4.4.2 Proof of Theorem 13

Proof of Theorem 13: Recall that we want to prove that $\Xi_{id_{2r+1}}^\# \sqsubseteq \beta(G)$ where $\Xi_{id_{2r+1}}^\#$ is the abstract state for the end of $A(m)$, and G is the points-to graph for the end of m. We do a proof by induction on *depth*, the call depth of the activation $A(m)$.

Initial case: *depth* $= 0$ In this case, $A(m)$ does not execute any analyzable CALL (otherwise, its call depth would be at least one). Let $IP_{A(m)} = [ip_0, \ldots, ip_{2q+1}]$ and let lb_j be the current label inside m reached by the execution of $A(m)$ at date $ip_j, 0 \leq j \leq 2q+1$. More formally, if in the concrete semantics, at date d, the topmost frame of the stack of the thread t, i.e., the thread where activation $A(m)$ takes place, is $\langle L_{d,t}, lb_{d,t} \rangle$, then:

$$\begin{aligned} lb_j &= lb_{ip_j,t}, \forall j \in \{0, 1, \ldots, 2q\} \\ \text{By convention, } lb_{2q+1} &= exit_m \end{aligned}$$

We prove by induction on j that $\Xi^{\#}_{ip_j} \sqsubseteq \beta(\circ A(lb_j)), \forall j \in \{0, 1, \ldots, 2q+1\}$. This proof by induction on j is nested inside the big, outer proof by induction on the call depth of $A(m)$.

- **Initial case** $j = 0$. Trivial by the definition of the initial points-to graph and the initial abstract state.
- **Induction step** $j \to j+1$. If j is odd, i.e., $j = 2i+1$, then $\Xi^{\#}_{ip_{2(i+1)}} = \Xi^{\#}_{ip_{2i+1}}$, $lb_{2(i+1)} = lb_{2i+1}$ (because the program does not execute any instruction of the thread t is between ip_{2i+1} and $ip_{2(i+1)}$) and the property we want to prove is trivially preserved. Now, suppose we have an even j, $j = 2i$. By Lemma 14, we have that $c_{ip_{2i}} = []$; hence,

$$\Xi^{\#}_{ip_{2i+1}} = [\![lb_{2i}, []]\!](\Xi^{\#}_{ip_{2i}}) \tag{4.15}$$

In the pointer analysis

$$\begin{aligned} \circ A(lb_{2i+1}) &= \bigsqcup\{A\circ(lb) \mid lb \in pred(lb_{2i+1})\} \\ &\sqsupseteq A\circ(lb_{2i}) = [\![lb_{2i}]\!]^a(\circ A(lb_{2i})) \end{aligned} \tag{4.16}$$

Using the monotonicity of β, Equation 4.16 and Lemma 16, we obtain

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

By the induction hypothesis, $\beta(\circ A(lb_{2i})) \sqsupseteq \Xi^{\#}_{ip_{2i}}$. As $[\![lb, []]\!]$ is a monotonic function, just as the analysis transfer function $[\![lb]\!]^{\#}$ was, we have that

$$[\![lb_{2i}, []]\!](\beta(\circ A(lb_{2i}))) \sqsupseteq [\![lb_{2i}, []]\!](\Xi^{\#}_{ip_{2i}}) = \Xi^{\#}_{ip_{2i+1}}$$

Combining these last two relations, we obtain the desired result:

$$\beta(\circ A(lb_{2i+1})) \sqsupseteq \Xi^{\#}_{ip_{2i+1}}$$

We've just finished proving that $\Xi^{\#}_{ip_j} \sqsubseteq \beta(\circ A(lb_j)), \forall j \in \{0, 1, \ldots, 2q+1\}$. If we put $j = 2q+1$, as $id_{2r+1} = ip_{2q+1}$ [9], we obtain that $\Xi^{\#}_{id_{2r+1}} \sqsubseteq \beta(\circ A(exit_m))$.

Induction step: Suppose that Theorem 13 is true for any activation having a call depth strictly smaller than *depth*. We shall prove that Theorem 13 is true for an activation of call depth *depth*.

As in the case of the initial case $depth = 0$, we do an inner proof by induction on j to prove that $\Xi^{\#}_{ip_j} \sqsubseteq \beta(\circ A(lb_j)), \forall j \in \{0, 1, \ldots, 2q+1\}$. For simplicity, we keep the same notations as in the previous case.

The only modification in the inner proof is that now, $A(m)$ might execute an analyzable CALL instruction. If we prove the induction step of the inner proof for

[9] Remember that $ID_{A(m)}$ and $IP_{A(m)}$ have the same last two elements.

the case when $j = 2i$ and the instruction at label is an analyzable CALL, then we finish the proof of Theorem 13.

Suppose that $\Xi^{\#}_{ip_{2i}} \sqsubseteq \beta(\circ A(lb_{2i}))$. We shall prove that $\Xi^{\#}_{ip_{2i+1}} \sqsubseteq \beta(\circ A(lb_{2i+1}))$. In the same way as in the case of $depth = 0$, we have that

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

Let *callee* be the method that is called at date ip_{2i}. $callee \in CG(lb_{2i})$, where CG is the (correct) call graph of the program. By the definition of the transfer function $[\![lb]\!]^a$ for an analyzable CALL instruction

$$\begin{aligned}[\![lb_{2i}]\!]^a(\circ A(lb_{2i})) &= \bigsqcup_{m_2 \in CG(lb_{2i})} interproc(\circ A(lb_{2i}), \circ A(exit_{m_2}), lb_{2i}, m_2) \\ &\sqsupseteq interproc(\circ A(lb_{2i}), \circ A(exit_{callee}), lb_{2i}, callee)\end{aligned}$$

We combine the last two relations and the monotonicity of β to obtain

$$\beta(\circ A(lb_{2i+1})) \sqsupseteq \beta(interproc(\circ A(lb_{2i}), \circ A(exit_{callee}), lb_{2i}, callee))$$

The analyzable CALL that we deal with starts a new activation of method *callee*, which we name $A(callee)$. This activation has its own list of interesting dates, which is of course a sublist of $ID_{A(m)}$ because each instruction of $A(callee)$ is also an instruction of $A(m)$. Let

$$ID_{callee} = [id_{callee,0}, \ldots, id_{callee,2k}, id_{callee,2k+1}, \ldots, id_{callee,2u+1}]$$

The abstract semantics of $A(callee)$ attaches to each interesting date of $id_{callee,k} \in ID_{callee}, 0 \leq k \leq 2u+1$ an abstract state $\Xi^{\#}_{2,id_{callee,k}}$. We use the subscript 2 to make the distinction between the abstract state $\Xi^{\#}_{2,id_{callee,k}}$ that the abstract semantics of $A(callee)$ computes for date $id_{callee,k}$ and the the abstract state $\Xi^{\#}_{id_{callee,k}}$ that the abstract semantics of $A(m)$ computes for the same date.

Let's examine the abstract states $\Xi^{\#}_{id_{2i}}$, i.e., the abstract state right before the CALL, and $\Xi^{\#}_{2,id_{callee,2u+1}}$, i.e., the abstract state that the abstract semantics of $A(callee)$ computes for the end of $A(callee)$. Both of them contains only nodes with empty context. Suppose we have a function $interproc^{\#}$ identical to *interproc* except that it works with nodes with context. We'll give its precise definition later, for the moment let's focus on our proof. As $interproc^{\#}$ is exactly like $interproc^{\#}$ except that it manipulates nodes with context, it is easy to prove a result similar to Lemma 16:

$$\begin{aligned}&\beta(interproc(\circ A(lb_{2i}), \circ A(exit_{callee}), lb_{2i}, callee)) = \\ &\quad interproc^{\#}(\beta(\circ A(lb_{2i})), \beta(\circ A(exit_{callee})), lb_{2i}, callee, [])\end{aligned}$$

Similar to *interproc*, $interproc^{\#}$ is monotonic in its first 2 inputs. By the induction hypothesis of the inner proof, $\beta(\circ A(lb_{2i})) \sqsupseteq \Xi^{\#}_{ip_{2i}}$. As the call depth of $A(callee)$ is strictly smaller than that of $A(m)$, by the induction hypothesis of the outer proof,

$interproc^{\#}$: $State^{\#} \times State^{\#} \times Label \times Method \times Context \to State^{\#}$

$interproc^{\#}(\Xi^{\#}, \Xi^{\#}_{callee}, lb_c, callee, c) =$
  let $\mu' = mapping^{\#}(\Xi^{\#}, \Xi^{\#}_{callee}, lb_c, callee, c)$ **in**
    $simplify^{\#}(combine^{\#}(\Xi^{\#}, \Xi^{\#}_{callee}, \mu', v_R))$

  where $P(lb_c) =$ "$v_R = v_0.s(v_1, \ldots, v_j)$".

Figure 4-10: Definition of function $interproc^{\#}$

$\beta(\circ A(exit_{callee})) \sqsupseteq \Xi^{\#}_{2,id_{callee},2u+1}$. Combining all these, we obtain

$$\beta(\circ A(lb_{2i+1})) \sqsupseteq interproc^{\#}(\Xi^{\#}_{ip_{2i}}, \Xi^{\#}_{2,id_{callee},2u+1}, lb_{2i}, callee, c)$$

Suppose we have a proof that

$$interproc^{\#}(\Xi^{\#}_{ip_{2i}}, \Xi^{\#}_{2,id_{callee},2u+1}, lb_{2i}, callee, c) \sqsupseteq \Xi^{\#}_{ip_{2i+1}} \tag{4.17}$$

Then, we finally obtain

$$\beta(\circ A(lb_{2i+1})) \sqsupseteq \Xi^{\#}_{ip_{2i+1}}$$

and we finish the inner and the outer induction proofs, together with the proof of Theorem 13 itself! □

The rest of this section provides the missing parts of the previous proof.

4.4.3 Proof of Equation 4.17

To prove Equation 4.17, we first have to give the promised definition of the auxiliary function $interproc^{\#}$. We do so in Figure 4-10. The definition of $interproc^{\#}$ uses the auxiliary functions $mapping^{\#}$, $combine^{\#}$ and $simplify^{\#}$. We define them in Figure 4-11, Figure 4-12, respectively Figure 4-13.

As previously mentioned, $interproc^{\#}$ and its auxiliary functions are identical to their equivalents from the pointer analysis, with the exception of a few technical details:

- They work with structures built up of nodes with context, instead of plain nodes.
- $interproc^{\#}$ and $mapping^{\#}$ have an additional argument: $c \in Context$. $interproc^{\#}$ just passes it down to $mapping^{\#}$, which uses it to identify the parameter nodes of *callee* in Constraint 4.18: instead of $n^P_{callee,i}$, it uses $n^P_{callee,i,c}$. E.g., if $\Xi^{\#}_{callee}$ is the abstract state that the abstract semantics of $A(callee)$ computes for the end of $A(callee)$, we use $c = []$.

- There is a "cosmetic" change inside $combine^{\#}$: instead of working with the state of local variables L, it works with the single-element stack $[L^{\#}]$ (similar for $L^{\#}_{callee}$).

Now, we are ready to attack the proof of Equation 4.17. Using the notations from the proof of Theorem 13, we want to prove that

$$\Xi^{\#}_{ip_{2i+1}} \sqsubseteq interproc^{\#}(\Xi^{\#}_{ip_{2i}}, \Xi^{\#}_{2,id_{callee,2u+1}}, lb_{2i}, callee, [])$$

Intuitively, we want to prove that the processing done by the inter-procedural analysis, which in this case is more appropriate to call inter-procedural combination of abstract states, is a conservative (i.e., safe) approximation of the abstract semantics which steps into the code of the callee and individually processes its instructions.

Proof sketch: Most of the proofs that we have presented so far were based on the same idea: induction on the list of interesting dates. In each case, we verified that the desired property was true for the first interesting date and next, we proved that each transition preserved the property. As the abstract semantics is a small step semantics, each induction step worked with a single, simple transition[10]. Therefore, the complexity of the proofs was not very big. The case of Equation 4.17 is different, at least at a first view. Now, we have to "jump" in a single step from ip_{2i} to ip_{2i+1} without individually processing all the transitions that the astract semantics of $A(m)$ makes inside $A(callee)$.

However, it is possible to go back to a small step semantics. The key idea is the following: for each interesting date of $A(callee)$, we can "freeze" the execution of the callee at that moment and do the inter-procedural combination of abstract states by using, instead of $\Xi^{\#}_{2,id_{callee,2u+1}}$, the current abstract state as computed by the abstract semantics for $A(callee)$, as if $A(callee)$ ended at that date. So, for each $d \in ID_{callee}$, we compute a result of the inter-procedural combination of abstract states and prove that it is more conservative than the abstract state that the abstract semantics for $A(m)$ computes for that date. This way, we process the instructions of the callee one by one and we can apply our standard proof technique: induction on a small step semantics. Hence, we can prove that *near the end*[11] of $A(callee)$, the result of the inter-procedure analysis combination of states conservatively approximates the abstract state that abstract semantics of $A(m)$ computes for that date. For technical reasons, we process the last instruction of $A(callee)$, i.e., the RETURN which finishes it, in a different way.

Proof of Equation 4.17: As we explained before, we would like to prove that at any interesting date d inside $A(callee)$, the result of the inter-procedural combination of abstract states is a conservative approximation of the abstract state $\Xi^{\#}_{d}$. However, we have two technical problems:

[10]Even in the case of an unanalyzable CALL, when we can have many concrete instructions, we still have a single transition in the abstract semantics.

[11]The meaning of this expression will become obvious in the next phrase.

$mapping^{\#} : State^{\#} \times State^{\#} \times Label \times Method \times Context \rightarrow Mapping^{\#}$

$Mapping^{\#} = Node^{\#} \times Node^{\#}$

PARAMETERS:

Points-to graph right before the CALL, $\Xi^{\#} = \langle I^{\#}, O^{\#}, [L^{\#}], S^{\#}, U^{\#} \rangle$;
Points-to graph from *callee*, $\Xi^{\#}_{callee} = \langle I^{\#}_{callee}, O^{\#}_{callee}, [L^{\#}_{callee}], S^{\#}_{callee}, U^{\#}_{callee} \rangle$;
Label lb_c of the CALL instruction:

$$P(lb_c) = \text{“}v_R = v_0.s(v_1, \ldots, v_j)\text{”};$$

Called method *callee*;
Context $c \in Context$ for the parameter nodes of *callee*.

RESULT:

Mapping $\mu' \in Mapping$, computed as follows:

1. Let $\mu \in Mapping^{\#}$ be the least fixed point of the following constraints:

$$L^{\#}(v_i) \subseteq \mu(n^{P}_{callee,i,c}),\ \forall i \in \{0, 1, \ldots j\} \tag{4.18}$$

$$\frac{\langle n_1, f, n_2 \rangle \in O^{\#}_{callee},\ \langle n_3, f, n_4 \rangle \in I^{\#},\ n_3 \in \mu(n_1)}{n_4 \in \mu(n_2)} \tag{4.19}$$

$$\frac{\begin{array}{c}\langle n_1, f, n_2 \rangle \in O^{\#}_{callee},\ \langle n_3, f, n_4 \rangle \in I^{\#}_{callee}, \\ ((\mu(n_1) \cup \{n_1\}) \cap (\mu(n_3) \cup \{n_3\})) \setminus \mathcal{N} \neq \emptyset, \\ (n_1 \neq n_3) \vee (n_1 \in LNode^{\#})\end{array}}{\mu(n_4) \cup (\{n_4\} \setminus ParamNodes^{\#}(callee, c)) \subseteq \mu(n_2)} \tag{4.20}$$

where $ParamNodes^{\#}(callee, c) = \{n^{P}_{callee,0,c}, \ldots, n^{P}_{callee,k,c}\}$

2. Extend μ to obtain μ' as follows:

$$\mu'(n) = \begin{cases} \mu(n) & \text{if } n \in ParamNodes^{\#}(callee, c) \\ \mu(n) \cup \{n\} & \text{otherwise} \end{cases}$$

Figure 4-11: Definition of function $mapping^{\#}$

$$
\begin{array}{l}
combine^{\#}\colon State^{\#} \times State^{\#} \times Mapping^{\#} \times V \to State^{\#} \\
\\
combine^{\#}(\langle I^{\#}, O^{\#}, [L^{\#}], S^{\#}, U^{\#}\rangle, \langle I^{\#}_{callee}, O^{\#}_{callee}, [L^{a}_{callee}], S^{\#}_{callee}, U^{\#}_{callee}\rangle, \mu', v_R) = \\
\quad \textbf{let} \quad I^{\#}_2 = I^{\#} \cup I^{\#}_{callee}[\mu'] \\
\qquad\quad O^{\#}_2 = O^{\#} \cup O^{\#}_{callee}[\mu'] \\
\qquad\quad L^{\#}_2 = L^{\#}\left[v_R \mapsto \mu'(L^{\#}_{callee}(v_{ret}))\right] \\
\qquad\quad S^{\#}_2 = S^{\#} \cup \mu'(S^{\#}_{callee}) \\
\qquad\quad U^{\#}_2 = U^{\#} \cup \mu'(U^{\#}_{callee}) \quad \textbf{in} \\
\qquad\quad \langle I^{\#}_2, O^{\#}_2, [L^{\#}_2], S^{\#}_2, U^{\#}_2\rangle \\
\\
\quad \text{where} \\
\qquad I^{\#}_{callee}[\mu'] = \bigcup_{\langle n_1, f, n_2\rangle \in I^{\#}_{callee}} (\mu'(n_1) \setminus \mathcal{N}) \times \{f\} \times \mu'(n_2) \\
\qquad O^{\#}_{callee}[\mu'] = \bigcup_{\langle n, f, n^L\rangle \in O^{\#}_{callee}} (\mu'(n) \setminus \mathcal{N}) \times \{f\} \times \{n^L\}
\end{array}
$$

Figure 4-12: Definition of function $combine^{\#}$

$$
\begin{array}{l}
simplify^{\#} : State^{\#} \to State^{\#} \\
\\
simplify^{\#}(\Xi^{\#} = \langle I^{\#}, O^{\#}, [L^{\#}_0, L^{\#}_1, \ldots, L^{\#}_{j-1}], S^{\#}, U^{\#}\rangle) = \\
\quad \textbf{let } A = \{n \in LNode \mid \neg e^{\#}(\Xi^{\#})(n)\} \textbf{ in} \\
\qquad \textbf{let} \quad I^{\#}_s = I^{\#} \setminus \{\langle n_1, f, n_2\rangle \mid \{n_1, n_2\} \cap A \neq \emptyset\} \\
\qquad\quad O^{\#}_s = O^{\#} \setminus \{\langle n, f, n^L\rangle \mid (\{n, n^L\} \cap A \neq \emptyset) \vee \neg e^{\#}(\Xi^{\#})(n)\} \\
\qquad\quad L^{\#}_{i,s} = \lambda v.\,(L^{a}_i(v) \setminus A)\,, \forall i \in \{0, 1, \ldots, j-1\} \\
\qquad\quad S^{\#}_s = S^{\#} \setminus A \\
\qquad\quad U^{\#}_s = U^{\#} \setminus A \quad \textbf{in} \\
\qquad\qquad \langle I^{\#}_s, O^{\#}_s, [L^{a}_{0,s}, L^{\#}_{1,s}, \ldots, L^{\#}_{j-1,s}], S^{\#}_s, U^{\#}_s\rangle
\end{array}
$$

Figure 4-13: Definition of function $simplify^{\#}$

$$interproc_2^{\#}: State^{\#} \times State^{\#} \times Label \times Method \times Context \rightarrow State^{\#}$$

$$interproc_2^{\#}(\Xi^{\#}, \Xi^{\#}_{callee}, lb_c, callee, c) =$$
$$\quad \textbf{let } \mu' = mapping^{\#}(\Xi^{\#}, \Xi^{\#}_{callee}, lb_c, callee, c) \textbf{ in}$$
$$\quad\quad simplify^{\#}(combine_2^{\#}(\Xi^{\#}, \Xi^{\#}_{callee}, \mu'))$$

Figure 4-14: Definition of function $interproc_2^{\#}$

1. The abstract semantics of $A(callee)$ starts with an empty context while the abstract semantics of $A(m)$ starts processing the instructions from $A(callee)$ with the context $[ip_{2i}]$ (ip_{2i} is the date of the CALL which starts $A(callee)$). So, each time the abstract semantics of $A(callee)$ uses the context c, the abstract semantics of $A(m)$ uses the context $c@[ip]$. Consider the case of a NEW instruction from label lb. The abstract semantics for $A(callee)$ creates the inside node $n^I_{lb,c}$ while the abstract semantics for $A(m)$ creates the inside node $n^I_{lb,c@[ip_{2i}]}$. As a result, some nodes appear in both $\Xi^{\#}_{2,d}$ and $\Xi^{\#}_d$ but they have the context c in the first state and $c@[ip_{2i}]$ in the second one. Therefore, it is generally not possible to compare the abstract state $\Xi^{\#}_d$ and the result of the inter-procedural combination of abstract states.

2. The function $interproc^{\#}$ does not work for the abstract states $\Xi^{\#}_{2,d}$ that correspond to dates inside the execution of $A(m)$ because these states might have a stack with more than one frame ($A(callee)$ might contain some analyzable CALLs).

The first problem can be solved very easily by processing $\Xi^{\#}_{2,d}$ to adjust the context of the nodes appearing inside it: context c will be turned into $c@[ip_{2i}]$. To this purpose, we define the conversion γ as follows:

$$\gamma(\langle n, c\rangle) = \langle n, c@[ip_{2i}]\rangle$$

Similar to $\alpha(d)$ and β, γ propagates deep into the structure it receives as argument. If we use $\gamma(\Xi^{\#}_{2,d})$, instead of $\Xi^{\#}_{2,d}$, into the inter-procedural combination of abstract states, all nodes have the right context and we can do the desired comparison.

We solve the second problem by defining a version of $interproc^{\#}$, called $interproc_2^{\#}$, which is very similar to $interproc^{\#}$, but does a more general processing for the stack. Figure 4-14 presents the formal definition of $interproc_2^{\#}$. It uses the same auxiliary functions *mapping* and *simplify* which were present in the definition of $interproc^{\#}$. However, instead of $combine^{\#}$, it uses the auxiliary function $combine_2^{\#}$ from Figure 4-15.

The function $combine_2^{\#}$ is able to deal with abstract states $\Xi^{\#}_{callee}$ whose stack $J^{\#}_{callee}$ might have more than one frame. All the stack frames are projected through

$$
\begin{array}{l}
combine_2^{\#}\colon State^{\#} \times State^{\#} \times Mapping^{\#} \to State^{\#} \\
\\
combine_2^{\#}(\langle I^{\#}, O^{\#}, [L^{\#}], S^{\#}, U^{\#}\rangle, \langle I^{\#}_{callee}, O^{\#}_{callee}, J^{\#}_{callee}, S^{\#}_{callee}, U^{\#}_{callee}\rangle, \mu') = \\
\quad \textbf{let} \quad I_2^{\#} = I^{\#} \cup I^{\#}_{callee}[\mu'] \\
\qquad\quad O_2^{\#} = O^{\#} \cup O^{\#}_{callee}[\mu'] \\
\qquad\quad J_2^{\#} = L^{\#} : J^{\#}_{callee}[\mu'] \\
\qquad\quad S_2^{\#} = S^{\#} \cup \mu'(S^{\#}_{callee}) \\
\qquad\quad U_2^{\#} = U^{\#} \cup \mu'(U^{\#}_{callee}) \quad \textbf{in} \\
\qquad\quad \langle I_2^{\#}, O_2^{\#}, [L_2^{\#}], S_2^{\#}, U_2^{\#}\rangle \\
\\
\quad \text{where} \\
\qquad\qquad I^{\#}_{callee}[\mu'] = \bigcup_{\langle n_1, f, n_2\rangle \in I^{\#}_{callee}} (\mu'(n_1) \setminus \mathcal{N}) \times \{f\} \times \mu'(n_2) \\
\qquad\qquad O^{\#}_{callee}[\mu'] = \bigcup_{\langle n, f, n^L\rangle \in O^{\#}_{callee}} (\mu'(n) \setminus \mathcal{N}) \times \{f\} \times \{n^L\} \\
\qquad (J^{\#}_{callee} = [L^{\#}_0, \ldots, L^{\#}_l])[\mu'] = [\ L^{\#}_0[\mu'] \ , \ \ldots \ , \ L^{\#}_l[\mu'] \ ] \\
\qquad\qquad L^{\#}_k[\mu'] = \lambda v \ . \ \mu'(L^{\#}_k(v)), \ \forall k \in \{0, 1, \ldots, l\}
\end{array}
$$

Figure 4-15: Definition of function $combine_2^{\#}$

μ':

$$(J^{\#}_{callee} = [L^{\#}_0, \ldots, L^{\#}_l])[\mu'] = [\ L^{\#}_0[\mu'] \ , \ \ldots \ , \ L^{\#}_l[\mu'] \ ]$$

As we did not encountered the final RETURN of $A(callee)$ yet, we do not set the variable v_R that stores the result of the callee and do not throw away the stack frame(s) from the callee. Instead, we compute the stack of the resulting abstract state by adding $L^{\#}$, i.e., the abstract state of the local variables of m, at the end (i.e., bottom) of $J^{\#}_{callee}[\mu']$: $J_2^{\#} = (J^{\#}_{callee}[\mu'])$ @ $[L^{\#}]$.

Suppose we have a proof that

$$\Xi^{\#}_{id_{callee,k}} \sqsubseteq interproc_2^{\#}(\Xi^{\#}_{ip_{2i}}, \gamma(\Xi^{\#}_{2,id_{callee,k}}), lb_{2i}, callee, [ip_{2i}]), \quad \forall k \in \{0, ..., 2u\} \tag{4.21}$$

We introduce the following notations:

$$
\begin{array}{lcl}
\Xi^{\#}_{3,d} & = & \gamma(\Xi^{\#}_{2,d}) \\
\Xi^{\#}_{4,d} & = & interproc_2^{\#}(\Xi^{\#}_{ip_{2i}}, \Xi^{\#}_{3,d}, lb_{2i}, callee, [ip_{2i}])
\end{array}
$$

To save some space, we also use the notation $d' = id_{callee,2u}$. With this notations, if

we put $k = 2u$ in Equation 4.21, we obtain $\Xi^{\#}_{d'} \sqsubseteq \Xi^{\#}_{4,d'}$ where

$$\begin{aligned}
\Xi^{\#}_{d'} &= \langle I^{\#}_{d'}, O^{\#}_{d'}, [L^{\#}_{d',callee}, L^{\#}_{d'}], S^{\#}_{d'}, U^{\#}_{d'} \rangle \\
\Xi^{\#}_{4,d'} &= \langle I^{\#}_{4,d'}, O^{\#}_{4,d'}, [L^{\#}_{4,d',callee}, L^{\#}_{4,d'}], S^{\#}_{4,d'}, U^{\#}_{4,d'} \rangle
\end{aligned}$$

The abstract semantics transfer function for the RETURN which finishes $A(callee)$, has two steps: it first alters the abstract state by appropriately setting v_R and throwing away the stack frame of the callee; next, it applies the function $\alpha(ip_{2i})$ to the state produced by the first step. Suppose the RETURN instruction is "`return` v". The first step constructs the following abstract states:

$$\begin{aligned}
\Xi^{\#}_{a} &= \langle I^{\#}_{d'}, O^{\#}_{d'}, [L^{\#}_{d'} \left[v_R \mapsto L^{\#}_{callee,d'}(v) \right]], S^{\#}_{d'}, U^{\#}_{d'} \rangle \\
\Xi^{\#}_{b} &= \langle I^{\#}_{4,d'}, O^{\#}_{4,d'}, [L^{\#}_{4,d'} \left[v_R \mapsto L^{\#}_{4,callee,d'}(v) \right]], S^{\#}_{4,d'}, U^{\#}_{4,d'} \rangle
\end{aligned}$$

Obviously, $\Xi^{\#}_{d'} \sqsubseteq \Xi^{\#}_{4,d'}$ implies $\Xi^{\#}_{a} \sqsubseteq \Xi^{\#}_{b}$. As $\alpha(ip_{2i})$ is clearly monotonic, we obtain the following inequality:

$$\alpha(ip_{2i})(\Xi^{\#}_{a}) \sqsubseteq \alpha(ip_{2i})(\Xi^{\#}_{b}) \tag{4.22}$$

By the definition of $\Xi^{\#}_{a}$, $\alpha(ip_{2i})(\Xi^{\#}_{a})$ is simply $\Xi^{\#}_{ip_{2i+1}}$: the conversion $\alpha(ip_{2i})$ is the remaining part of the abstract semantics transfer function for a RETURN inside $A(m)$. The other term of the inequality is very familiar too: due to the way the abstract semantics of $A(callee)$ processes the final RETURN, and to the definitions of $interproc^{\#}$ and $interproc^{\#}_{2}$, we have that:

$$\begin{aligned}
\Xi^{\#}_{b} &= interproc^{\#}(\Xi^{\#}_{ip_{2i}}, \Xi^{\#}_{3,id_{callee,2u+1}}, lb_{2i}, callee) \\
&= interproc^{\#}(\Xi^{\#}_{ip_{2i}}, \gamma(\Xi^{\#}_{2,id_{callee,2u+1}}), lb_{2i}, callee)
\end{aligned}$$

Suppose we have a proof that

$$\begin{aligned}
&\alpha(ip_{2i})(interproc^{\#}(\Xi^{\#}_{ip_{2i}}, \gamma(\Xi^{\#}_{2,id_{callee,2u+1}}), lb_{2i}, callee, [ip_{2i}])) \sqsubseteq \\
&\quad interproc^{\#}(\Xi^{\#}_{ip_{2i}}, \Xi^{\#}_{2,id_{callee,2u+1}}, lb_{2i}, callee, [])
\end{aligned} \tag{4.23}$$

Combining the last relations with Equation 4.22, we obtain the desired inequality

$$\Xi^{\#}_{ip_{2i+1}} \sqsubseteq interproc^{\#}(\Xi^{\#}_{ip_{2i}}, \Xi^{\#}_{2,id_{callee,2u+1}}, lb_{2i}, callee, [])$$

which finishes the proof of Equation 4.17, provided that we are able to prove Equation 4.21 and Equation 4.23. We prove Equation 4.21 in Subsection 4.4.5, and Equation 4.23 in Appendix A. □

4.4.4 Properties of the Node Mappings

In this subsection, we study the properties of the mappings internally used by the functions $interproc^{\#}$ and $interproc_2^{\#}$. The results we obtain are used in the proofs of Equation 4.21 (Subsection 4.4.5) and Equation 4.23 (Appendix A).

Lemma 17. *Consider a CALL instruction at label lb_c, that might call the method callee, a date $d \in Date$, a context $c \in Context$, and two abstract states $\Xi^{\#}, \Xi^{\#}_{callee} \in State^{\#}$. Let μ be the mapping obtained as the least fixed point of the three constraints from Figure 4-11. Then,*

$$\forall n, \mu(n) \neq \emptyset \rightarrow n \in LNode^{\#} \cup PNode^{\#}$$

Proof: Although the proof is very easy, it illustrates a powerful technique that we use in future proofs.

A possible algorithm for computing the least fixed point of a set of constraints is *Chaotic Iteration* [16]. This algorithm works as follows:

1. Start with the smallest possible mapping, $\mu_0 = \emptyset$.

2. Iterate till no longer possible: pick one applicable instance of a constraint that can extend the mapping, and apply it. The $(k+1)^{\text{th}}$ iteration extends μ_k into a strictly bigger mapping μ_{k+1}, $\mu_{k+1} \supset \mu_k$.

3. When no constraint can extend the mapping, stop the algorithm. The mapping at that moment is the result of the algorithm.

It is a classic result from the program analysis theory that if the constraints are monotonic (as in our case), the Chaotic Iteration algorithm terminates and its result is the least fixed point of the constraints. Therefore, μ can be obtained by a finite sequence of applications of the constraints. If we prove by induction on the iteration index k that each mapping μ_k satisfies the property from the text of the lemma, the last mapping, i.e., μ, satisfies them too.

The induction proof is trivial: the initial, empty mapping satisfies the desired property and each application of a constraint creates new mappings from a parameter node with context, (Constraint 4.18), or from a load node with context (Constraint 4.19). □

Lemma 18. *Consider a CALL instruction at label lb_c that might call the method callee, a date $d \in Date$, and two abstract states $\Xi^{\#}, \Xi^{\#}_{callee} \in State^{\#}$ such that*

- *no node appears in both $\Xi^{\#}$ and $\Xi^{\#}_{callee}$, and*

- *the parameter nodes that appear in $\Xi^{\#}_{callee}$ are only the nodes from the set $ParamNodes^{\#}(callee, c)$, where $c \in Context$.*

Let

$$\begin{aligned}
\mu_1' &= mapping^{\#}(\Xi^{\#}, \Xi^{\#}_{callee}, lb_c, callee, c) \\
\mu_2' &= mapping^{\#}(\alpha(d)(\Xi^{\#}), \alpha(d)(\Xi^{\#}_{callee}), lb_c, callee, \alpha(d)(c)) \\
&\quad where\ \alpha(d)(c) = \begin{cases} c_2 & if\ c = d : c_2 \\ c & otherwise \end{cases}
\end{aligned}$$

Let μ_1 and μ_2 be the mappings computed internally by the two applications of $mapping^{\#}$, i.e., the results of the first step of the algorithm from Figure 4-11. These mappings are defined as the least fixed point of the three constraints from Figure 4-11; μ_1' is obtained by enlarging μ_1 to contain the pairs $\langle n, n\rangle$ for any node n that is not a parameter node; similarly for μ_2'. With these notations $\alpha(d)(\mu_1) \subseteq \mu_2$ and $\alpha(d)(\mu_1') \subseteq \mu_2'$ where

$$\alpha(d)(\mu') = \{\langle \alpha(d)(n_1), \alpha(d)(n_2)\rangle \mid \langle n_1, n_2\rangle \in \mu'\}$$

Note: The two conditions that we imposed on $\Xi^{\#}$ and $\Xi^{\#}_{callee}$ are not hard to meet. We'll use this lemma only with $\Xi^{\#} = \Xi^{\#}_{ip_{2i}}$, $\Xi^{\#}_{callee} = \gamma(\Xi^{\#}_{2,d})$, and $c = [ip_{2i}]$. These abstract states clearly use disjoint nodes: ip_{2i} makes the difference. Also, all the parameter nodes from $\gamma(\Xi^{\#}_{2,d})$ have the context $[ip_{2i}]$.

Proof: All we have to prove is $\alpha(d)(\mu_1) \subseteq \mu_2$; the second relation is an easy implication of the first one.

As we explained in the proof of Lemma 17, we can compute μ_1 and μ_2 with the help of the *Chaotic Iteration* algorithm that iteratively applies constraints till a fixed point is reached. Consider the computation of μ_1': we start with an empty mapping $\mu_{1,0} = \emptyset$ and apply a series of constraint instances till no further progress is possible. Let $\mu_{1,k}, k \in \{0, 1, \ldots, z\}$ be the successive mappings that the algorithm constructs.

We prove by induction on k that, if we start with $\mu_{2,0} = \emptyset$ and apply the same constraint instances, one by one, in the same order, but this time projected through $\alpha(d)$ (i.e., n becomes $\alpha(d)(n)$, $I^{\#}$ becomes $\alpha(d)(I^{\#})$, etc.), we obtain the mappings $\mu_{2,1}, \ldots, \mu_{2,k}, \ldots, \mu_{2,z}$ that respect the condition $\alpha(d)(\mu_{1,k}) \subseteq \mu_{2,k}, 0 \leq k \leq z$.

The initial step $k = 0$ is trivial because both mappings are empty. For the induction step, we suppose $\alpha(d)(\mu_{1,k}) \subseteq \mu_{2,k}$ and we prove that the same relation holds for $k + 1$. We do a case analysis on the type of the constraint that we applied for extending $\mu_{1,k}$ into $\mu_{1,k+1}$.

The constraints 4.18 and 4.19 are easy to deal with: we simply apply the same constraint instance, but this time everything is projected through $\alpha(d)$. For brevity, we skip these two cases.

The most difficult case is that of Constraint 4.20. Consider the nodes n_1, n_2, n_3, and n_4, and the field f such that the precondition of Constraint 4.19 is satisfied. The constraint extends $\mu_{1,k}$ by mapping n_2 to

$$M = \mu_{1,k}(n_4) \cup (\{n_4\} \setminus ParamNodes^{\#}(callee, c))$$

If the constraint can be applied for $\mu_{2,k}$, for the same nodes but projected through

$\alpha(d)$, then $\mu_{2,k}$ will be similarly extended by mapping $\alpha(d)(n_2)$ to $\alpha(d)(M)$ and, with the help of the induction hypothesis, we ultimately prove that $\alpha(d)(\mu_{1,k+1}) \subseteq \mu_{k+1}$.

For convenience, we use the notation $n'_i = \alpha(d)(n_i), i \in \{1,2,3,4\}$. To finish the lemma, we have to prove that if the precondition of Constraint 4.20 is valid for $n_1, n_2, n_3, n_4, I^{\#}_{callee}, O^{\#}_{callee}$, it is valid for $n'_1, n'_2, n'_3, n'_4, \alpha(d)(I^{\#}_{callee}), \alpha(d)(O^{\#}_{callee})$, too.

As Constraint 4.20 was applicable, we have that

1. $\langle n_1, f, n_2\rangle \in O^{\#}_{callee}$ and $\langle n_3, f, n_4\rangle \in I^{\#}_{callee}$;
2. $((\mu_{1,k}(n_1) \cup \{n_1\}) \cap (\mu_{1,k}(n_3) \cup \{n_3\})) \setminus \mathcal{N} \neq \emptyset$;
3. The predicate $P(n_1, n_3) =$"$(n_1 \neq n_3) \vee (n_1 \in LNode^{\#})$" is satisfied.

Projecting the first condition through $\alpha(d)$ is trivial: $\langle n_1, f, n_2\rangle \in O^{\#}_{callee}$ implies $\langle n'_1, f, n'_2\rangle \in \alpha(d)(O^{\#}_{callee})$, etc. We can combine the induction hypothesis and the second condition to show that $((\mu_{2,k}(n'_1) \cup \{n'_1\}) \cap (\mu_{2,k}(n'_3) \cup \{n'_3\})) \setminus \mathcal{N} \neq \emptyset$. Showing that the predicate $P(n'_1, n'_3)$, the last part of the precondition, is satisfied is more delicate. We have two cases:

1. If $n_1 \in LNode^{\#}$, then, as the $\alpha(d)$ conversion does not change the type of a node, the same relation is true about n'_1 and $P(n'_1, n'_3)$ is trivially satisfied.
2. Otherwise, $n_1 \notin LNode^{\#}$; as $P(n_1, n_3)$ is valid, $n_1 \neq n_3$.
   (a) If $n_3 \in LNode^{\#}$, as nodes n_1 and n_3 are from disjoint sets, n'_1 and n'_3 are from disjoint sets too, and hence different. Once again, $P(n'_1, n'_3)$ is satisfied.
   (b) If $n_3 \in PNode^{\#}$, suppose $n'_1 = n'_3$. Thus, $n_1 \in PNode^{\#}$ too. As all the parameter nodes from $\Xi^{\#}_{callee}$ have the same context c, $n'_1 = n'_3$ implies $n_1 = n_3$. Contradiction! Therefore, $n'_1 \neq n'_3$, and the predicate $P(n'_1, n'_3)$ is satisfied.
   (c) Otherwise, if $n_3 \notin LNode^{\#} \cup PNode^{\#}$, by Lemma 17, $\mu_{2,k}(n_3) = \emptyset$. Therefore, Condition 2 becomes:

$$(\mu_{1,k}(n_1) \cup \{n_1\}) \cap (\mu_{1,k}(n_3) \cup \{n_3\}) = \mu_{1,k}(n_1) \cap \{n_3\}$$

   As the previous intersection is non-empty, $n_3 \in \mu_{1,k}(n_1)$. We already supposed that in this case, $n_1 \notin LNode^{\#}$. If $n_1 \notin PNode^{\#}$, by Lemma 17, $\mu_{2,k}(n_1) = \emptyset$, and the previous intersection is empty. Contradiction! Therefore, n_1 is a parameter node. Let's note that the only constraint that can add mappings for a parameter node is Constraint 4.18. Furthermore, this constraint adds only mappings from a parameter node from $\Xi^{\#}_{callee}$ to a node from $\Xi^{\#}$. As $\Xi^{\#}$ and $\Xi^{\#}_{callee}$ do not have any common node, a parameter node from $\Xi^{\#}_{callee}$, such as n_1, cannot map to nodes from $\Xi^{\#}_{callee}$. Contradiction with $n_3 \in \mu_{1,k}(n_1)$. Therefore, this case is impossible.

In conclusion, the third part of the precondition is valid too when projected through $\alpha(d)$ and so, Constraint 4.20 can be applied for $\mu_{2,k}$ too. So, if z is the iteration when the computation of μ_1 finished

$$\alpha(d)(\mu_1) = \alpha(d)(\mu_{1,z}) \subseteq \mu_{2,z} \subseteq \mu_2$$

and the proof of 18 is complete. □

Notations: We introduce two new notations. If $d \in ID_{callee}$ is an interesting date for $A(callee)$, then let μ'_d be

$$\mu'_d = mapping^{\#}(\Xi^{\#}_{ip_{2i}}, \Xi^{\#}_{3,d}, lb_{2i}, callee, [ip_{2i}])$$

The computation of $mapping^{\#}$ (Figure 4-11) has in two steps: in the first step, we compute the least fixed point of the three constraints from Figure 4-11. Next, we extend the result of the first step to obtain μ'_d. Let μ_d be the result of the first step. The relation between is μ'_d and μ_d is:

$$\mu'_d(n) = \begin{cases} \mu'_d(n) & \text{if } n = n^P_{callee,i,[ip_{2i}]} \\ \mu'_d(n) \cup \{n\} & \text{otherwise} \end{cases} \tag{4.24}$$

Lemma 19. *Consider the dates $d_1 = id_{callee,2l}$ and $d_2 = id_{callee,2l+1}$ (where $0 \leq 2l < 2l+1 \leq 2u+1$). Using the convention of this section, let μ_{d_1} and μ_{d_2} be the mappings computed for date d_1, respectively d_2:*

$$\begin{aligned} \mu'_{d_1} &= mapping^{\#}(\Xi^{\#}_{ip_{2i}}, \Xi^{\#}_{3,d_1}, lb_{2i}, callee, [ip_{2i}]) \\ \mu'_{d_2} &= mapping^{\#}(\Xi^{\#}_{ip_{2i}}, \Xi^{\#}_{3,d_2}, lb_{2i}, callee, [ip_{2i}]) \end{aligned}$$

Using the previously introduced notations, μ'_{d_1} and μ'_{d_2} are the extended version of μ_{d_1} and μ_{d_2}.

1. *If the instruction executed in the abstract semantics transition from d_1 to d_2 is not a RETURN inside $A(callee)$, then $\mu_{d_1} \subseteq \mu_{d_2}$ and $\mu'_{d_1} \subseteq \mu'_{d_2}$.*

2. *In the case of a RETURN inside $A(callee)$, we have a "cosmetized" version of the previous relation: $\alpha(d)(\mu_{d_1}) \subseteq \mu_{d_2}$ and $\alpha(d)(\mu'_{d_1}) \subseteq \mu'_{d_2}$, where d is the date when the corresponding CALL was executed.*

Proof: The first case is very easy: as the abstract semantics maintains the sets of inside/edges in a cummulative way (it just extend them with new elements), all the applications of the Constraints 4.18, 4.19 and 4.20 at date $id_{callee,2l}$, are still possible at date $id_{callee,2l+1}$ and so, μ_{2l+1} contains all the mappings from μ_{2l}: $\mu_{2l} \subseteq \mu_{2l+1}$. Obviously, $\mu'_{2l} \subseteq \mu'_{2l+1}$, too.

In the case of a RETURN inside $A(callee)$, let's notice that the matching CALL is inside $A(callee)$ too and therefore $d > ip_{2i}$. We apply Lemma 18 and notice that:

- $\alpha(d)(\Xi^{\#}_{ip_{2i}}) = \Xi^{\#}_{ip_{2i}}$ because, as $ip_{2i} < d$, none of the nodes from $\Xi^{\#}_{ip_{2i}}$ contains d in its context.
- As $\alpha(d)$ works at the head of a context and γ works at its tail, they commute in a nice way: $\alpha(d)(\gamma(\Xi^{\#}_{2,d_1})) = \gamma(\alpha(d)(\Xi^{\#}_{2,d_1}))$. In general, $\alpha(d)(\Xi^{\#}_{2,d_1}) \neq \Xi^{\#}_{2,d_2}$: the processing associated with a RETURN inside $A(callee)$ also supposes some modifications to the stack. However, as the mapping construction uses just the sets of edges from the callee, in this case we can consider them to be equal.
- $\alpha(d)([ip_{2i}]) = [ip_{2i}]$

This completes the proof of Lemma 19. □

Lemma 20. *Consider an interesting date of $A(callee)$, $d \in ID_{callee}$, and, with the previously introduced notations, the corresponding mapping μ_d. If n_a is a node of the form $n_a = \gamma(n_b)$ and $\exists n_c \in Node^{\#}$ such that $n_a \in \mu_d(n_c)$ (i.e., $\langle n_c, n_a \rangle \in \mu$), then $e^{\#}(\Xi^{\#}_{3,d})(n_a)$.*

Proof: Using the same idea as in the proof of Lemma 17, we have that the mapping μ_d is the limit of an increasing chain of mappings $\mu_{d,0} \subset \mu_{d,1} \subset \ldots \mu_{d,z} = \mu_d$ that is obatianed by applying a finite series of constraint instances. We prove by induction on k that any $\mu_{d,k}$, including the final one, which is μ_d, satisfies the property stated in the lemma.

Initial case: $k = 0$. The empty mapping trivially satisfies the property.

Induction step: $k \rightarrow k + 1$. We do a case analysis on the constraint applied in iteration k. Constraints 4.18 and 4.19 are irrelevant because they create mappings toward nodes from $\Xi^{\#}_{ip_{2i}}$, that cannot have the date ip_{2i} in their context and hence, cannot be of the form $\gamma(n_b)$. The relevant case is that of Constraint 4.20

It is worth examining only the nodes that are now targeted[12] for the first time by the mapping. For the other nodes, the property is true by the induction hypothesis. Using the node notations from Figure 4-11, after we apply the constraint, n_2 is mapped, in addition to its previous mappings, to n_4 (if n_4 is not a parameter node) and to the nodes that n_4 was previously mapped to, i.e., the nodes from the set $\mu_{d,k}(n_4)$. The only node that might become targeted now is n_4. We prove that $e^{\#}(\Xi^{\#}_{3,d})(n_4)$. There are two cases:

1. If $\mu_{d,k}(n_3) \neq \emptyset$, by Lemma 17, $e^{\#}(\Xi^{\#}_{3,d})(n_3)$.
2. Otherwise, $\mu_{d,k}(n_3) = \emptyset$, and we have two subcases:
   (a) If $n_1 \neq n_3$, then, as $(\mu_{d,k}(n_1) \cup \{n_1\}) \cap (\mu_{d,k}(n_3) \cup \{n_3\}) \neq \emptyset$, we have that $n_3 \in \mu_{d,k}(n_1)$. As n_3 is a node from $\Xi^{\#}_{3,d}$, it is of the form $\gamma(n)$ and, by the induction hypothesis, $e^{\#}(\Xi^{\#}_{3,d})(n_3)$.

[12] I.e., those nodes n_a such that exists a node n_b that is mapped to n_a.

(b) If $n_1 = n_3$, then n_3 trivially escapes because it is the source of an outside edge (Invariant 2).

In all cases, $e^{\#}(\Xi^{\#}_{3,d})(n_3)$. As escapability propagates over the inside edge from n_3 to n_4, $e^{\#}(\Xi^{\#}_{3,d})(n_4)$ too. This terminates the proof of Lemma 20. □

Lemma 21. *If n is not a parameter nodes, then $n \in \mu'_d(n), \forall d \in ID_{callee}$.*

Proof: Obvious by the construction of μ'_d (Equation 4.24). □

Corollary 22. $\forall d \in ID_{callee}, n \notin LNode^{\#} \cup RNode^{\#} \rightarrow \mu'_d(n) = \{n\}$.

Proof: Direct application of Lemma 21 and Lemma 17. □

Corollary 23. $\forall d \in ID_{callee}, \forall n$ *that appear in* $\Xi^{\#}_{3,d}$, $\neg e^{\#}(\Xi^{\#}_{3,d})(n) \rightarrow \mu'_d(n) = \{n\}$.

Proof: As n appears in $\Xi^{\#}_{3,d}$, and is captured there, it cannot be a load or a parameter node. We directly apply Corollary 22. □

Corollary 24. *For any interesting date of $A(callee)$, $d \in ID_{callee}$, if $n' = \gamma(n)$ and n' does not escape in the state $\Xi^{\#}_{3,d}$ (i.e., $\neg e^{\#}(\Xi^{\#}_{3,d})(n')$), then no other node but itself is mapped to n': $\{n_1 \mid n' \in \mu'_d(n_1)\} = \{n'\}$.*

Proof: Direct application of Lemma 21 and Lemma 20. □

4.4.5 Proof of Equation 4.21

As a quick reminder, we want to prove that

$$\forall k \in \{0, \ldots, 2u\}, \text{ if } d = id_{callee,k} \text{ then}$$
$$\Xi^{\#}_d \sqsubseteq \Xi^{\#}_{4,d} = interproc^{\#}_2(\Xi^{\#}_{ip_{2i}}, \gamma(\Xi^{\#}_{2,d}), lb_{2i}, callee, [ip_{2i}])$$

Proof: The final step in the definition of $interproc^{\#}_2$ (Figure 4-14), is a call to $simplify^{\#}$. However, remember that $simplify^{\#}$ removes only those load nodes and outside edges that cannot be produced by the abstract semantics. Therefore, if we are able to prove Equation 4.21 in the case when $interproc^{\#}_2$ does not call $simplify^{\#}$, we also prove it for the "real" definition of $interproc^{\#}_2$ because none of the load node and the outside edges from $\Xi^{\#}_d$ are removed by the simplification. For the rest of the proof of Equation 4.21, we ignore the existence of $simplify^{\#}$. Hence, using our previous notations

$$\begin{aligned} \Xi^{\#}_{3,d} &= \gamma(\Xi^{\#}_{2,d}) \\ \Xi^{\#}_{ip_{2i}} &= \langle I^{\#}_{ip_{2i}}, O^{\#}_{ip_{2i}}, [L^{\#}_{ip_{2i}}], S^{\#}_{ip_{2i}}, U^{\#}_{ip_{2i}} \rangle \end{aligned}$$

we have the following definitions for the components of $\Xi^{\#}_{4,d}$:

$$\begin{aligned} I^{\#}_{4,d} &= I^{\#}_{ip_{2i}} \cup I^{\#}_{3,d}[\mu'_d] \\ O^{\#}_{4,d} &= O^{\#}_{ip_{2i}} \cup O^{\#}_{3,d}[\mu'_d] \\ J^{\#}_{4,d} &= (J^{\#}_{3,d}[\mu'_d]) \ @ \ [L^{\#}_{ip_{2i}}] \\ S^{\#}_{4,d} &= S^{\#}_{ip_{2i}} \cup \mu'_d(S^{\#}_{3,d}) \\ U^{\#}_{4,d} &= U^{\#}_{ip_{2i}} \cup \mu'_d(U^{\#}_{3,d}) \end{aligned}$$

Note that by the definition of $J^{\#}_{4,d}$, its topmost element, $L^{\#}_{4,d}$ is always the projection of the topmost element of $J^{\#}_{3,d}$, $L^{\#}_{3,d} = \gamma(L^{\#}_{2,d})$:

$$L^{\#}_{4,d} = L^{\#}_{3,d}[\mu'_d]$$

We do a proof by induction on k.

Initial case: $k = 0, d = id_{callee,0}$. In this case, ip_{2i}, $ip_{2i}+1$ and d are consecutive interesting dates for $A(m)$: $A(m)$ executes the CALL that starts $A(callee)$ in the transition from date ip_{2i} to date $ip_{2i}+1$, and next, at date $d = id_{callee,0}$, it starts the execution of the first instruction of *callee*. By the definition of the transfer function $[\![.]\!]^{\#}$ in the case of a CALL instruction (Figure 4-5), we obtain

$$\Xi^{\#}_{d} = \Xi^{\#}_{ip_{2i}+1} = \langle \ I^{\#}_{ip_{2i}} \ , \ O^{\#}_{ip_{2i}} \ , \ [\ \{p_w \mapsto L^{\#}_{ip_{2i}}(v_w)\}_{0 \le w \le j} \ , \ L^{\#}_{ip_{2i}} \ ] \ , \ S^{\#}_{ip_{2i}} \ , \ U^{\#}_{ip_{2i}} \ \rangle$$

The abstract state $\Xi^{\#}_{2,d}$ at the beginning of $A(callee)$ is

$$\Xi^{\#}_{2,d} = \langle \ \emptyset \ , \ \emptyset \ , \ [\ \{p_w \mapsto n^{P}_{callee,w}\}_{0 \le w \le j} \ ] \ , \ \emptyset \ , \ \emptyset \ \rangle$$

As the set $I^{\#}_{2,d}$ is empty, $I^{\#}_{3,d} = \gamma(I^{\#}_{2,d})$ is empty too, and $I^{\#}_{4,d} = I^{\#}_{d}$. Similar relations can be obtained for $O^{\#}_{4,d}$, $S^{\#}_{4,d}$ and $U^{\#}_{4,d}$. As $L^{\#}_{ip_{2i}}(v_a) \subseteq \mu'_0(n^{P}_{callee,w}), 0 \le w \le j$ by Constraint 4.18 from the definition of $mapping^{\#}$, $J^{\#}_{d} \sqsubseteq J^{\#}_{4,d}$. As a consequence, $\Xi^{\#}_{d} \sqsubseteq \Xi^{\#}_{4,d}$ [13].

Induction step: $k \to k+1$. The case of an odd k, $k = 2l+1$ is trivial because the abstract states do not change when moving from date $id_{callee,2l+1}$ to date $id_{callee,2(l+1)}$. So, we focus on the case of an even k, $k = 2l$. To save some space, we denote $d1 = id_{callee,2l}$, and $d2 = id_{callee,2l+1}$. The abstract states $\Xi^{\#}_{d2}$ and $\Xi^{\#}_{2,d2}$ are obtained from $\Xi^{\#}_{d1}$, respectively $\Xi^{\#}_{2,d1}$ with the help of the transfer function $[\![.]\!]^{\#}$, which might use the auxiliary function $[\![.,.]\!]$. We do a case analysis on the type of the instruction

[13] As $mapping^{\#}$ computes the least fixed point of three constraints out of which only Constraint 4.18 is applicable, we can even prove they are equal. However, this is not necessary for our proof.

from the transition from date $d1$ to date $d2$.

Lemma 19 simplifies a lot of our work: as $\mu'_{d1} \subseteq \mu'_{d2}$, in each case, we need to examine the things that actually change in the transition from $d1$ to $d2$, i.e., the new inside/outside edges, the new started thread nodes, the new nodes passed to unanalyzable CALLs, and the variables whose value changes.

A COPY instruction "$v_1 = v_2$" modifies just the value of the local variable v_1 from the top-most stack frame, as follows: $L^{\#}_{d2}(v_1) = L^{\#}_{d1}(v_2)$ and $L^{\#}_{2,d2}(v_1) = L^{\#}_{2,d1}(v_2)$ (which immediately implies $L^{\#}_{3,d2}(v_1) = L^{\#}_{3,d1}(v_2)$). By the induction hypothesis

$$L^{\#}_{d1}(v_2) \subseteq L^{\#}_{4,d1}(v_2) = \mu'_{d1}(L^{\#}_{3,d1}(v_2))$$

Furthermore, as $\mu'_{d1} \subseteq \mu'_{d2}$ and $L^{\#}_{3,d2}(v_1) = L^{\#}_{3,d1}(v_2)$, we have that

$$\mu'_{d1}(L^{\#}_{3,d1}(v_2)) \subseteq \mu'_{d2}(L^{\#}_{3,d1}(v_1)) = L^{\#}_{4,d2}(v_1)$$

Therefore, $L^{\#}_{d2}(v_1) = L^{\#}_{d1}(v_2) \subseteq L^{\#}_{4,d2}(v_1)$, and, by consequence, $\Xi^{\#}_{d2} \sqsubseteq \Xi^{\#}_{4,d2}$.

The case of a NULLIFY instruction "$v_1 = \texttt{null}$" is trivial: as $L^{\#}_{d2}(v_1) = \{n_{\texttt{null}}\}$, $L^{\#}_{2,d2}(v_1) = \{n_{\texttt{null}}\}$, and $n_{\texttt{null}} \in \mu'_{d2}(n_{\texttt{null}})$ (Lemma 21), we obtain

$$L^{\#}_{4,d2}(v_1) = \mu'_{d2}(L^{\#}_{3,d2}(v_1)) = \mu'_{d2}(\{n_{\texttt{null}}\}) \supseteq \{n_{\texttt{null}}\} = L^{\#}_{d2}(v_1)$$

By consequence, $\Xi^{\#}_{d2} \sqsubseteq \Xi^{\#}_{4,d2}$.

A NEW instruction "$v = \texttt{new}\ C$" changes the value of the local variable v and adds a few inside edges. The abstract semantics for $A(\mathit{callee})$ creates the inside node $n^I_{lb,c}$ while the abstract semantics for $A(m)$ creates the inside node $n^I_{lb,c@ip_{2i}}$. We study first the value of the local variable v:

$$L^{\#}_{4,d2}(v) = \mu'_{d2}(L^{\#}_{3,d2}(v)) = \mu'_{d2}(\gamma(L^{\#}_{2,d2}(v))) = \mu'_{d2}(\{n^I_{lb,c@[ip_{2i}]}\})$$

As $n^I_{lb,c@[ip_{2i}]} \in \mu'_{d2}(n^I_{lb,c@[ip_{2i}]})$ (Lemma 21), we have that

$$L^{\#}_{4,d2}(v) \supseteq \{n^I_{lb,c@[ip_{2i}]}\} = L^{\#}_{d2}(v).$$

The abstract semantics of $A(m)$ extends $I^{\#}_{d2}$ with the following edges:

$$\{n^I_{lb,c@[ip_{2i}]}\} \times \mathit{fields}(C) \times \{n_{\texttt{null},c@[ip_{2i}]}\}$$

The same edges appear in $I^{\#}_{3,d2}$ [14] and, as $n \in \mu'_{d_2}(n), \forall n$ (Lemma 21), those edges appear in $I^{\#}_{4,d2}$, too. Overall, $\Xi^{\#}_{d2} \sqsubseteq \Xi^{\#}_{4,d2}$.

In the case of a STORE instruction "$v_1.f = v_2$", the abstract semantics of $A(m)$ extends $I^{\#}_{d1}$ with the following edges:

$$E = (L^{\#}_{d1}(v_1) \setminus \mathcal{N}) \times \{f\} \times L^{\#}_{d1}(v_2) \subseteq (\mu'_{d1}(L^{\#}_{3,d1}(v_1)) \setminus \mathcal{N}) \times \{f\} \times \mu'_{d1}(L^{\#}_{3,d1}(v_2))$$

The inclusion is due to the fact that $L^{\#}_{d1} \sqsubseteq L^{\#}_{4,d1} = L^{\#}_{3,d1}[\mu'_{d1}]$ by the induction hypothesis. As $\mu_1(n) = \{n\}, \forall n \in \mathcal{N}$ (Corollary 22), we can prove that $\mu'_{d1}(A) \setminus \mathcal{N} = \mu'_{d1}(A \setminus \mathcal{N}) \setminus \mathcal{N}$ for any set of nodes $A \subseteq Node^{\#}$. Therefore,

$$E \subseteq (\mu'_{d1}(L^{\#}_{3,d1}(v_1) \setminus \mathcal{N}) \setminus \mathcal{N}) \times \{f\} \times \mu'_{d1}(L^{\#}_{3,d1}(v_2))$$

Furthermore, by the processing of the STORE instruction in the abstract semantics of $A(callee)$,

$$(L^{\#}_{3,d1}(v_1) \setminus \mathcal{N}) \times \{f\} \times L^{\#}_{3,d1}(v_2) \subseteq I^{\#}_{3,d2}; \text{ hence,}$$
$$(\mu'_{d2}(L^{\#}_{3,d1}(v_1) \setminus \mathcal{N}) \setminus \mathcal{N}) \times \{f\} \times \mu'_{d2}(L^{\#}_{3,d1}(v_2)) \subseteq I^{\#}_{3,d2}[\mu'_{d2}] \subseteq I^{\#}_{4,d2}$$

As $\mu'_{d1} \subseteq \mu'_{d2}$, we have that $E \subseteq I^{\#}_{4,d2}$, $I^{\#}_{d2} \subseteq I^{\#}_{4,d2}$, and finally, $\Xi^{\#}_{d2} \sqsubseteq \Xi^{\#}_{4,d2}$.

In the abstract semantics of $A(m)$, a THREAD START instruction "`start` v" extends the set of started threads as follows: $S^{\#}_{d2} = S^{\#}_{d1} \cup L^{\#}_{d1}(v)$. As always, it is sufficient to study the newly started threads, the set $L^{\#}_{d1}(v)$. By the induction hypothesis, $L^{\#}_{d1}(v) \subseteq L^{\#}_{4,d1}(v) = \mu'_{d1}(L^{\#}_{3,d1}(v))$. On the other side, in the abstract semantics of $A(callee)$, $L^{\#}_{2,d1}(v) \subseteq S^{\#}_{2,d2}$ and so, $L^{\#}_{3,d1}(v) \subseteq S^{\#}_{3,d2}$. Combining these facts with $\mu'_{d1} \subseteq \mu'_{d2}$ (Lemma 19), we obtain:

$$L^{\#}_{d1}(v) \subseteq \mu'_{d1}(L^{\#}_{3,d1}(v)) \subseteq \mu'_{d1}(S^{\#}_{3,d2}) \subseteq \mu'_{d2}(S^{\#}_{3,d2}) \subseteq S^{\#}_{4,d2}$$

This proves that $S^{\#}_{d2} \subseteq S^{\#}_{4,d2}$, and finally, $\Xi^{\#}_{d2} \sqsubseteq \Xi^{\#}_{4,d2}$.

An unanalyzable CALL "$v_R = v_0.s(v_1, \ldots, v_{q-1})$" modifies the set of nodes passed as parameters in an unanalyzable CALL and sets the local variable v_R to point to a corresponding return node. Using the same ideas we used in the case of THREAD START, COPY and NEW, we can easily prove that $\Xi^{\#}_{d2} \sqsubseteq \Xi^{\#}_{4,d2}$.

An analyzable CALL is very similar to a series of COPY instructions: we copy values from the variables that are passed as actual arguments to the corresponding

[14] $I^{\#}_{2,d2}$ contains the edges from the set $\{n^I_{lb,c}\} \times \mathit{fields}(C) \times \{n_{\mathtt{null},c}\}$ and the conversion γ adds the missing ip_{2i} at the end of the context c.

parameters. The only difference is that the destination of these transfers (the parameters of the called method) are now in a newly created stack frame). As the other stack frames are unchanged, we can use the same ideas as in the case of COPY to prove that $\Xi^{\#}_{d2} \sqsubseteq \Xi^{\#}_{4,d2}$.

Consider the case of a RETURN inside $A(callee)$ "`return` v". We can split this instruction in two parts:

1. Pop off the stack the state of the local variables of the called method and set v_R in the stack frame of the caller.

2. Eliminate d_c from the head of the node contexts with the help of the $\alpha(d_c)$ conversion, where d_c is the date of the corresponding CALL.

If we ignore the conversion $\alpha(d_c)$, what remains from the RETURN instruction is very similar to a COPY and we can use the same ideas to prove that $\Xi^{\#}_{d2} \sqsubseteq \Xi^{\#}_{4,d2}$. As $\alpha(d_c)$ is uniformly applied in the abstract semantics for both $A(m)$ and $A(callee)$, and, by Lemma 19, $\alpha(d_c)(\mu'_{d1}) \subseteq \mu'_{d2}$, it doesn't harm our proof. We examine only the sets of inside edges. The proofs for the other components of the abstract states are similar.

In the abstract semantics for $A(callee)$, $I^{\#}_{2,d2} = \alpha(d_c)(I^{\#}_{2,d1})$ and so, $I^{\#}_{3,d2} = \alpha(d_c)(I^{\#}_{3,d1})$. As $\alpha(d_c)(\mu'_{d1}) \subseteq \mu'_{d2}$, we can write the following chain of relations:

$$\begin{array}{rcl} I^{\#}_{4,d2} & = & I^{\#}_{ip_{2i}} \cup I^{\#}_{3,d2}[\mu'_{d2}] \;=\; I^{\#}_{ip_{2i}} \cup \alpha(d_c)(I^{\#}_{3,d1})[\mu'_{d2}] \\ & \supseteq & I^{\#}_{ip_{2i}} \cup \alpha(d_c)(I^{\#}_{3,d1})[\alpha(d_c)(\mu'_{d1})] \end{array}$$

Also, as no node from $\Xi^{\#}_{ip_{2i}}$ has d_c in its context (because $ip_{2i} < d_c$), $\alpha(d_c)(I^{\#}_{ip_{2i}}) = I^{\#}_{ip_{2i}}$. Using the definitions, we can prove that

$$\alpha(d_c)(I^{\#}_{3,d1})[\alpha(d_c)(\mu'_{d1})] \supseteq \alpha(d_c)(I^{\#}_{3,d1}[\mu'_{d1}])$$

We continue the previous chain of relations:

$$\begin{array}{rcll} I^{\#}_{4,d2} & \supseteq & I^{\#}_{ip_{2i}} \cup \alpha(d_c)(I^{\#}_{3,d1}[\mu'_{d1}]) & \\ & = & \alpha(d_c)(I^{\#}_{ip_{2i}} \cup I^{\#}_{3,d1}[\mu'_{d1}]) & \alpha(d_c) \text{ is irrelevant for } I^{\#}_{ip_{2i}} \\ & = & \alpha(d_c)(I^{\#}_{4,d1}) & \text{Construction of } I^{\#}_{4,d1} \\ & \supseteq & \alpha(d_c)(I^{\#}_{d1}) = I^{\#}_{d2} & \text{Induction hypothesis} \end{array}$$

We have proved that $I^{\#}_{d2} \subseteq I^{\#}_{4,d2}$. The case of the other components of an abstract state is similar. We finally obtain $\Xi^{\#}_{d2} \sqsubseteq \Xi^{\#}_{4,d2}$.

Finally, we arrive at the most difficult case: the case of a LOAD instruction "$v_2 = v_1.f$" having the label lb. The proof for this case directly uses the two yet unused constraints from the definition of the function $mapping^{\#}$, i.e., Constraint 4.19, and Constraint 4.20 (Figure 4-11). The processing for the LOAD instruction changes the

value of the local variable v_2 and might add new outside edges. In a first phase, we prove that $L^{\#}_{d2}(v_2) \subseteq L^{\#}_{4,d2}(v_2)$; we examine the outside edges later. Let

$$\begin{aligned} B &= \{n \in Node^{\#} \mid \exists n_1 \in L^{\#}_{d1}(v_1) \setminus \mathcal{N}, \langle n_1, f, n\rangle \in I^{\#}_{d1}\} \\ L &= \begin{cases} \{n^L_{lb,c@[ip_{2i}]}\} & \text{if } \exists n_1 \in L^{\#}_{d1}(v_1) \setminus \mathcal{N} \text{ such that } e^{\#}(\Xi^{\#}_{d1})(n_1) \\ \emptyset & \text{otherwise} \end{cases} \end{aligned}$$

With these notations, $L^{\#}_{d2}(v_2) = B \cup L$. Consider $n \in L^{\#}_{d2}(v_2)$. We want to prove that $n \in L^{\#}_{4,d2}(v_2)$ and hence $L^{\#}_{d2}(v_2) \subseteq L^{\#}_{4,d2}(v_2)$. We have two disjoint cases: $n \in B$ and $n \neq B$.

Case 1: $n \in B$. By the definition of B, $\exists n_1 \in L^{\#}_{d1}(v_1) \setminus \mathcal{N}$, such that $\langle n_1, f, n\rangle \in I^{\#}_{d1}$. By the induction hypothesis, $L^{\#}_{d1}(v_1) \subseteq L^{\#}_{4,d1}(v_1) = \mu'_{d1}(L^{\#}_{3,d1}(v_1)) = \mu'_{d1}(\gamma(L^{\#}_{2,d1}(v_1)))$, which implies $\exists n_3 \in L^{\#}_{2,d2}(v_1)$ such that $n_1 \in \mu'_{d1}(\gamma(n_3))$. As $\mu'_1(n) = \{n\}, \forall n \in \mathcal{N}$, and $n_1 \notin \mathcal{N}$, $n_3 \notin \mathcal{N}$ either. Therefore, $n_3 \in L^{\#}_{2,d2}(v_1) \setminus \mathcal{N}$.

As $I^{\#}_{d1} \subseteq I^{\#}_{4,d1} = I^{\#}_{ip_{2i}} \cup I^{\#}_{3,d1}[\mu'_{d1}]$ (we used the induction hypothesis again), the relation $\langle n_1, f, n\rangle \in I^{\#}_{d1}$ generates two subcases:

Case 1.1: $\langle n_1, f, n\rangle \in I^{\#}_{3,d1}[\mu'_{d1}] = (\gamma(I^{\#}_{2,d1}))[\mu'_{d1}]$. In this case, we have two nodes n_4, n_5, such that

$$\begin{gathered} \langle n_4, f, n_5\rangle \in I^{\#}_{2,d1} \text{ (equivalent to } \langle \gamma(n_4), f, \gamma(n_5)\rangle \in I^{\#}_{3,d1}) \\ (\ n_1 \in \mu'_{d1}(\gamma(n_4)) \setminus \mathcal{N}\ ) \quad \wedge \quad (\ n \in \mu'_{d1}(\gamma(n_5))\ ) \end{gathered}$$

We have two sub-subcases:

Case 1.1.1: $n_3 = n_4$. In this case, in the abstract semantics for $A(callee)$, $n_3 \in L^{\#}_{2,d1}(v_1)$ and $\langle n_3, f, n_5\rangle \in I^{\#}_{2,d1}$. By the processing of LOAD, $n_5 \in L^{\#}_{2,d2}(v_2)$, which implies $\gamma(n_5) \in L^{a}_{3,d2}(v_2)$. We have:

$$n \in \mu'_{d1}(\gamma(n_5)) \subseteq \mu'_{d1}(L^{\#}_{3,d2}(v_2)) \subseteq \mu'_{d2}(L^{\#}_{3,d2}(v_2)) = L^{\#}_{4,d2}(v_2)$$

To prove the second inclusion, we have used Lemma 19: $\mu'_{d1} \subseteq \mu'_{d2}$.

Case 1.1.2: $n_3 \neq n_4$. To save some space, we denote $n'_3 = \gamma(n_3)$ and $n'_4 = \gamma(n_4)$. As $n_3 \neq n_4$, obviously $n'_3 = \gamma(n_3) \neq \gamma(n_4) = n'_4$.

First, we prove that $e^{\#}(\Xi^{\#}_{2,d1})(n_3)$. Suppose for the sake of contradiction that $\neg e^{\#}(\Xi^{\#}_{2,d1})(n_3)$. By Corollary 23, $\mu'_{d1}(n'_3) = \{n'_3\}$ and by consequence, $n_1 = n'_3$. So, μ'_{d1} contains a mapping from n'_4 to $n_1 = n'_3$. As $n'_3 \neq n'_4$, $n'_3 \in \mu_{d1}(n'_4)$. By Lemma 20, $e^{\#}(\Xi^{\#}_{3,d1})(n'_3)$. This contradicts our assumption that $\neg e^{\#}(\Xi^{\#}_{2,d1})(n_3)$.

The abstract semantics for $A(callee)$ introduces the outside edge $\langle n_3, f, n^L_{lb,c}\rangle \in O^{\#}_{2,d2}$ and puts v_2 to point to the load node $n^L_{lb,c}$ (and possibly to some other nodes):

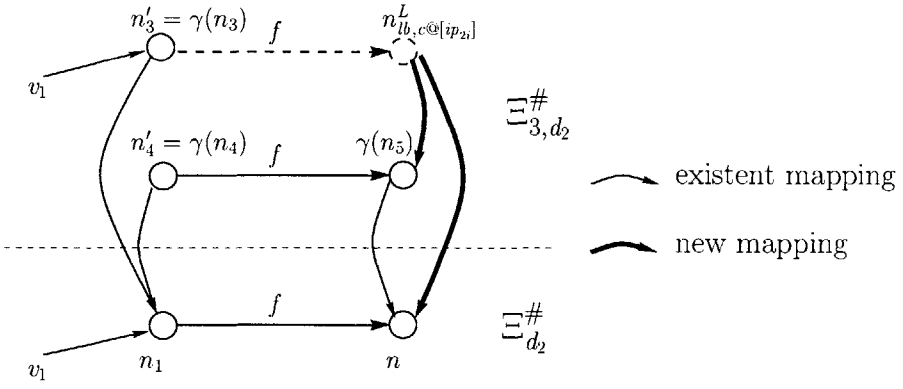


Figure 4-16: Graphic representation of Case 1.1.2 for a LOAD

$n^L_{lb,c} \in L^{\#}_{d2}(v_2)$. Accordingly, $\langle n'_3, f, n^L_{lb,c@[ip_{2i}]}\rangle \in O^{\#}_{3,d2}$, and $n^L_{lb,c@[ip_{2i}]} \in L^{\#}_{3,d2}(v_2)$. Figure 4-16 presents a graphical representation of this case.

Both $\mu'_{d1}(n'_3)$ and $\mu'_{d1}(n'_4)$ contain the node $n_1 \notin \mathcal{N}$. As $\mu'_{d1}(n'_3) \subseteq \mu_{d1}(n'_3) \cup \{n'_3\}$ and a similar relation is true for n'_4, we obtain

$$(\mu_{d1}(n'_3) \cup \{n'_3\}) \cap (\mu_{d1}(n'_4) \cup \{n'_4\}) \setminus \mathcal{N} \supseteq \{n_1\} \neq \emptyset$$

As $\mu_{d1} \subseteq \mu_{d2}$, the previous relation is also true for the mapping μ_{d2}. Furthermore, we already know that $n'_3 \neq n'_4$. Hence, by Constraint 4.20, $n \in \mu_{d2}(n^L_{lb,c@[ip_{2i}]}) \subseteq \mu'_{d2}(n^L_{lb,c@[ip_{2i}]})$ (note that as it appears in $\Xi^{\#}_{d2}$, $n \notin \mathit{ParamNodes}^{\#}(\mathit{callee}, [ip_{2i}])$). We can extend this as follows:

$$n \in \mu'_{d2}(n^L_{lb,c@[ip_{2i}]}) \subseteq \mu'_{d2}(L^{\#}_{2,d2}(v_2)) = L^{\#}_{4,d2}(v_2)$$

Observation: We cannot apply Constraint 4.19 because $\langle n_1, f, n\rangle$ is not necessarily an edge from $I^{\#}_{ip_{2i}}$; all we know is that it exists in $I^{\#}_{3,d1}[\mu'_{d1}]$.

Case 1.2: $\langle n_1, f, n\rangle \notin I^{\#}_{3,d1}[\mu'_{d1}]$. In this case, $\langle n_1, f, n\rangle \in I^{\#}_{ip_{2i}}$. To save some space, we denote $n'_3 = \gamma(n_3)$. As $\Xi^{\#}_{ip_{2i}}$ contains only nodes that are created before ip_{2i}, the context of n_1 does not contain ip_{2i}. Hence, it is different from n'_3. As $n_1 \in \mu'_{d1}(n'_3)$ and $n_1 \neq n'_3$, $n_1 \in \mu_{d1}(n'_3)$. By Lemma 20, $e^{\#}(\Xi^{\#}_{3,d1})(n'_3)$, which is equivalent to $e^{\#}(\Xi^{\#}_{2,d1})(n_3)$.

In these circumstances, the abstract semantics for $A(\mathit{callee})$ creates the outside edge $\langle n_3, f, n^L_{lb,c}\rangle$ and puts v_2 to point to the load node $n^L_{lb,c}$. As a consequence, $\langle n'_3, f, n^L_{lb,c@[ip_{2i}]}\rangle \in O^{\#}_{3,d2}$ and $n^L_{lb,c@[ip_{2i}]} \in L^{\#}_{3,d2}(v_2)$. Figure 4-17 presents a graphical representation of this case. By Constraint 4.19, $n \in \mu_{d2}(n^L_{lb,c@[ip_{2i}]})$. We can develop this as follows:

$$n \in \mu_{d2}(n^L_{lb,c@[ip_{2i}]}) \subseteq \mu'_{d2}(n^L_{lb,c@[ip_{2i}]}) \subseteq \mu'_{d2}(L^{\#}_{3,d2}(v_2)) = L^{\#}_{4,d2}(v_2)$$

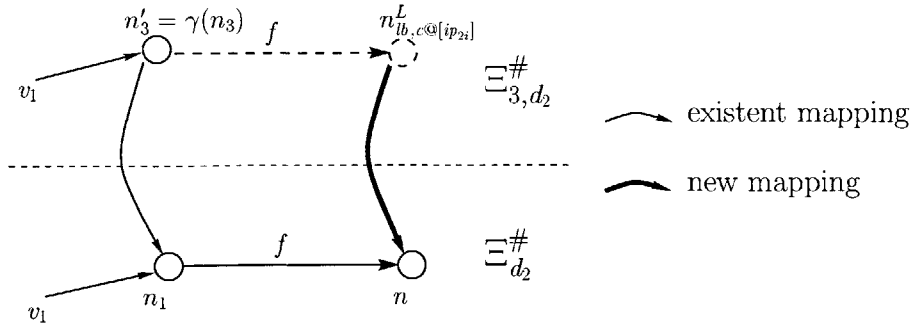


Figure 4-17: Graphic representation of Case 1.2 for a LOAD

Case 2: $n \notin B$. In this case, $n \in L$. Therefore, n is the only element of the set L, i.e., the load node $n^L_{lb,c@[ip_{2i}]}$. As $L \neq \emptyset$, there is a node $n_1 \in L^{\#}_{d1}(v_1) \setminus \mathcal{N}$ such that $e^{\#}(\Xi^{\#}_{d1})(n_1)$. As in Case 1, by the induction hypothesis, $L^{\#}_{d1}(v_1) \subseteq L^{\#}_{4,d1}(v_1)$, and we finally obtain $\exists n_3 \in L^{\#}_{2,d1}(v_1) \setminus \mathcal{N}$, such that $n_1 \in \mu'_{d1}(\gamma(n_3))$. To save some space, we use the notation $n'_3 = \gamma(n_3)$.

For the time being, suppose we have a proof that $e^{\#}(\Xi^{\#}_{2,d1})(n_3)$ is true; we give the actual proof later. Then, the abstract semantics of $A(callee)$ puts v_2 to point to the load node $n^L_{lb,c}$. As a consequence, $n = n^L_{lb,c@[ip_{2i}]} \in L^{\#}_{3,d2}(v_2)$. As $n \in \mu'_{d2}(n)$ (Lemma 21), we have that $n \in \mu'_{d2}(L^{\#}_{3,d2}(v_2)) = L^{\#}_{4,d2}(v_2)$.

In all cases, $n \in L^a_{4,d2}(v_2)$. As n is an arbitrary element of $L^{\#}_{d2}(v_2)$, we have proved that $L^{\#}_{d2}(v_2) \subseteq L^a_{4,d2}(v_2)$.

A LOAD instruction also modifies the set of outside edges. More precisely, for every node $n_1 \in L^{\#}_{d1}(v_1) \setminus \mathcal{N}$ such that $e^{\#}(\Xi^{\#}_{d1})(n_1)$, the abstract semantics of $A(m)$ introduces an outside edge $\langle n_1, f, n\rangle$ from n_1 to the load node $n = n^L_{lb,c@[ip_{2i}]}$.

As in Case 2 above, $\exists n_3 \in L^{\#}_{2,d1}(v_1) \setminus \mathcal{N}$ such that $n_1 \in \mu'_{d1}(n'_3)$ where $n'_3 = \gamma(n_3)$ and $e^{\#}(\Xi^{\#}_{2,d1})(n_3)$ is true. Due to the processing done by the abstract semantics of $A(callee)$ in the case of a LOAD, $\langle n'_3, f, n\rangle \in O^{\#}_{3,d2}$. As $n_1 \in \mu'_{d1}(n'_3) \subseteq \mu'_{d2}(n'_3)$, and $n_1 \notin \mathcal{N}$, we obtain

$$\langle n_1, f, n\rangle \in (\mu'_{d2}(n'_3) \setminus \mathcal{N}) \times \{f\} \times \{n\} \subseteq O^{\#}_{3,d2}[\mu'_{d2}] \subseteq O^{\#}_{4,d2}$$

This shows that the new outside edges are present in $O^{\#}_{4,d2}$. As a consequence, $O^{\#}_{d2} \subseteq O^{\#}_{4,d2}$ and finally, $\Xi^{\#}_{d2} \sqsubseteq \Xi^{\#}_{4,d2}$.

In all cases, $\Xi^{\#}_{d2} \sqsubseteq \Xi^{\#}_{4,d2}$. This ends our proof by induction of Equation 4.21, provided that we are able to fill in the missing part from Case 2 of a LOAD instruction.

□

Proof of the missing part: As a quick reminder, we have to prove that, in the conditions of Case 2, $e^{\#}(\Xi^{\#}_{2,d1})(n_3)$ is true. This is equivalent to proving that $e^{\#}(\Xi^{\#}_{3,d1})(n'_3)$ is true, where $n'_3 = \gamma(n_3)$. Suppose for the sake of contradiction that $\neg e^{\#}(\Xi^{\#}_{3,d1})(n'_3)$.

In Case 2, $e^{\#}(\Xi^{\#}_{d1})(n_1)$, is true. Therefore, there is a path of edges from $I^{\#}_{d1} \cup O^{\#}_{d1}$ that reaches n_1 from a node from the following set:

$$N = ParamNodes^{\#}(m, []) \cup S^{\#}_{d1} \cup U^{\#}_{d1} \cup L^{\#}_{d1}(v_{ret}) \cup RNode^{\#} \tag{4.25}$$

The meticulous reader might protest that the definition of abstract escape predicate (Definition 12) uses all the parameter nodes with context. However, the nodes $ParamNodes^{\#}(m, [])$ are the only parameter nodes that might appear in $\Xi^{\#}_{d1}$. Suppose the nodes from the path are $n_{p,1}, n_{p,2}, \ldots, n_{p,l} = n_1$ (the index p stands for *path*), and the edges from the path are $\langle n_{p,j}, f_j, n_{p,j+1}\rangle \in I^{\#}_{d1} \cup O^{\#}_{d1}$.

Idea: In this proof, we use a technique that we used in the proofs of the invariants that described the relation between the concrete and the abstract semantics. In those proofs, using some "good" properties of an abstraction relation, we translated a path from an abstract state into a path through the concrete heap. In this case, we use the properties of μ'_{d1} to show that the aforementioned path exists inside $\Xi^{\#}_{3,d1}$, too. This allows us to obtain a contradiction and conclude the proof. Instead of using the fact that there is a single node that models a captured object, we use similar properties of the node mappings: Corollary 23 and Corollary 24.

We prove by reverse induction on j that

$$\forall j,\ \neg e^{\#}(\Xi^{\#}_{3,d1})(n_{p,j}) \wedge \exists\ n^{(2)}_{p,j} \text{ such that } n_{p,j} = \gamma(n^{(2)}_{p,j})$$

Intuitively, the relation we want to prove tells that all the edges from the path are in fact captured nodes from $\Xi^{\#}_{3,d1}$ (and so, the path exists in $\Xi^{\#}_{3,d}$).

Initial case $j = l$. In this case, $n_{p,l} = n_1 \in \mu'_{d1}(n'_3)$. As $\neg e^{\#}(\Xi^{\#}_{3,d1})(n'_3)$, by Corollary 23, $n_1 = n'_3 = \gamma(n_3)$ and we obtain $\neg e^{\#}(\Xi^{\#}_{3,d1})(n_{p,l})$ and $n^{(2)}_{p,l} = n_3$.

Induction step $j \to j-1$. The edge $\langle n_{p,j-1}, f_{j-1}, n_{p,j}\rangle$ is an edge from

$$I^{\#}_{d1} \cup O^{\#}_{d1} \subseteq I^{\#}_{ip_{2i}} \cup I^{\#}_{3,d1}[\mu'_{d1}] \cup O^{\#}_{ip_{2i}} \cup O^{\#}_{3,d1}[\mu'_{d1}]$$

By the induction hypothesis $\exists n^{(2)}_{p,j}$ such that $n_{p,j} = \gamma(n^{(2)}_{p,j})$; as a consequence, $n_{p,j}$ is a node with ip_{2i} in its context and it cannot appear in an edge from $I^{\#}_{ip_{2i}} \cup O^{\#}_{ip_{2i}}$. If $\langle n_{p,j-1}, f_{j-1}, n_{p,j}\rangle \in O^{\#}_{3,d1}[\mu'_{d1}]$, by the definition of $O^{\#}_{3,d1}[\mu'_{d1}]$[15]

[15]For the outside edges, we project just the starting node, the target load node remains unchanged.

$n_{p,j}$ is a load node, which contradicts the induction hypothesis $\neg e^{\#}(\Xi^{\#}_{3,d1})(n_{p,j})$.

So, it remains that $\langle n_{p,j-1}, f_{j-1}, n_{p,j}\rangle \in I^{\#}_{3,d1}[\mu'_{d1}]$. This implies the existence of the nodes n_a, n_b such that

- $\langle n_a, f, n_b\rangle \in I^{\#}_{3,d1}$,
- $n_{p,j-1} \in \mu'_{d1}(n_a) \setminus \mathcal{N}$, and
- $n_{p,j} \in \mu'_{d1}(n_b)$.

As $\neg e^{\#}(\Xi^{\#}_{3,d1})(n_{p,j})$, by Corollary 24, we have that $n_b = n_{p,j}$. This implies $\neg e^{\#}(\Xi^{\#}_{3,d1})(n_b)$ and hence, $\neg e^{\#}(\Xi^{\#}_{3,d1})(n_a)$. By Corollary 23, $n_a = n_{p,j-1}$; therefore, $\neg e^{\#}(\Xi^{\#}_{3,d1})(n_{p,j-1})$. Furthermore, as $n_a = n_{p,j-1}$ appears in $I^{\#}_{3,d1} = \gamma(I^{\#}_{2,d1})$, it is of the type $\gamma(n^{(2)}_{p,j-1})$. This completes our induction proof.

Now, remember that $n_{p,1}$ is a node from the set N defined in Equation 4.25. As it is captured in $\Xi^{\#}_{3,d1}$, it cannot be a return node. Also, as we have not encountered the final RETURN of $A(m)$, $L^{\#}_{d1}(v_{ret}) = \emptyset$. Furthermore, as $n_{p,1} = \gamma(n^{(2)}_{p,1})$, the context of $n_{p,1}$ contains ip_{2i} and thus, $n_{p,1} \notin \mathit{ParamNodes}^{\#}(m, [])$. It remains that

$$n_{p,1} \in S^{\#}_{d1} \cup U^{\#}_{d1}$$

Suppose $n_{p,1} \in S^{\#}_{d1}$; the other case is similar. By the induction hypothesis of the proof of Equation 4.21[16], $S^{\#}_{d1} \subseteq S^{\#}_{ip_{2i}} \cup S^{\#}_{3,d1}[\mu'_{d1}]$. $n_{p,1}$ cannot be an element of $S^{\#}_{ip_{2i}}$ because its context contains ip_{2i}. It remains that $n_{p,1} \in S^{\#}_{3,d1}[\mu'_{d1}]$, i.e., $\exists n_x \in S^{\#}_{3,d1}$ such that $n_{p,1} \in \mu'_{d1}(n_x)$. As $\neg e^{\#}(\Xi^{\#}_{3,d1})(n_{p,1})$, by Corollary 24, $n_{p,1} = n_x$. Hence, $n_{p,1} \in S^{\#}_{3,d1}$. Contradiction with $\neg e^{\#}(\Xi^{\#}_{3,d1})(n_{p,1})$!

Therefore, $e^{\#}(\Xi^{\#}_{2,d1})(n_3)$ is indeed true. □

4.5 Correctness of the Optimizations

We can use Corollary 12 and Theorem 13 to obtain a proof for Theorem 5, which we stated at the beginning of this chapter.

Proof of Theorem 5: Consider we have an execution trace T and, inside it, an activation $A(m)$ of method m. Let $G = \circ A(\mathit{exit}_m)$ be the points-to graph that the pointer analysis computes for the exit point of m. Suppose that lb is the label of a NEW instruction executed by $A(m)$; lb is either a label from m or a label from one of m's callees.

We will prove that if $\neg e^{a}(G)(n^{I}_{lb})$, then all the objects which $A(m)$ allocates by executing the NEW instruction from label lb can be allocated in the stack frame of

[16]Remember that all this proof is done in the conditions of Case 2 for a LOAD instruction from the proof of Equation 4.21.

the instance of method m which is the "root" of $A(m)$. Let $\Xi^{\#}_{id_{2r+1}}$ be the abstract state computed by the abstract semantics of $A(m)$ for the end of $A(m)$. By Theorem 13, $\Xi^{\#}_{id_{2r+1}} \sqsubseteq \beta(G)$. From this relation, $e^{\#}(\Xi^{\#}_{id_{2r+1}})(n^I_{lb,[]}) \rightarrow e^a(G)(n^I_{lb})$. As n^I_{lb} is captured in G, we obtain $\neg e^{\#}(\Xi^{\#}_{id_{2r+1}})(n^I_{lb,[]})$. By Corollary 12, we can stack allocate all the objects created by the NEW instruction at label lb. As our trace T was arbitrarily chosen, this completes the proof of Theorem 5. □

The theorem we have just proved states that the stack allocation optimization (Section 3.1) is safe, both when it is applied in its simplest version and when it is enhanced through method inlining.

For the second proposed optimizations, allocation in the thread-local heap (Section 3.2), remember that the condition for it was that on any reverse call path, the inside node coresponding to the optimized NEW instruction, becomes captured at some level. So, any object allocated by an execution of that NEW instruction has the property that its lifetime is included in the lifetime of a method which transitively called the method which allocated the object. As the lifetime of that method is clearly included in the lifetime of its thread — which is the same with the thread which allocates the object because navigating on the reverse call path does not cross any thread boundary — the second optimization is safe too.

The correctness of the third optmizations, the synchronization removal (Section 3.3) infers from the correctness of the allocation in the thread-local heap.

4.6 Analysis Precision

Using the invariants from Section 4.3, and the fact that the pointer analysis is a conservative approximation of the abstract semantics (Section 4.4), we can describe how the points-to graphs that the analysis computes models the heaps from a concrete execution. Intuitively, the analysis has precise information about the objects created at object allocation sites whose corresponding inside nodes are captured in the points-to graph.

More precisely, suppose we have a points-to graph $G = \circ A(lb)$ for the program point right before label lb from a method m. Consider a concrete execution trace that contains an activation $A(m)$ for m, and suppose that at date d, $A(m)$ starts to execute the instruction from label lb. Let o_1 and o_2 be two objects created by $A(m)$ before d, by the NEW instructions from lb_1, respectively lb_2.

If $n^I_{lb_1}$ and $n^I_{lb_2}$ are captured in G, then for each heap reference $\langle o_1, f, o_2\rangle$ from o_1 to o_2, there is an equivalent inside edge $\langle n^I_{lb_1}, f, n^I_{lb_2}\rangle$ in G from the inside node that models o_1 to the inside node that models o_2.

The proof for this statement uses the invariants from Section 4.3. As o_1 is clearly modeled by $n^I_{lb_1}$, and this node is captured, by Invariant 6, we have that o_1 is captured in $A(m)$. Therefore, by Invariant 3, o_1 is modeled only by $n^I_{lb_1}$. Similarly, o_2 is modeled only by $n^I_{lb_2}$, and we can apply Invariant 4 to obtain the previous statement.

Similarly, consider an object o created by $A(m)$ by the NEW instruction from label lb, and suppose such that n^I_{lb} is captured in G. Using the invariants from

Section 4.3, we can prove that if o is pointed to by the local variable v, then n_{lb}^I is pointed to by v in G. This result implies that if o is a started thread object, the set of started threads from G contains n_{lb}^I.

As the analysis has precise information only about the captured nodes, it is important to minimize the number of escaped nodes. In particular, it is important to have a small number of unanalyzable CALLs. In practice, most of the unanalyzable CALLs correspond to native method invocations. We can alleviate the effect of these native calls by providing the manually constructed points-to graphs that model the effects of the most common native methods.

4.7 Final Look Over the Proof

In this chapter, we proved the correctness of the optimizations enabled by the pointer analysis. Our proof idea was to introduce an additional layer, the abstract semantics, between the concrete semantics and the pointer analysis. First, we proved a few invariants which showed how the abstract semantics models the concrete semantics. Next, we proved that the result of the analysis is more conservative than the result of the abstract semantics. Proving this result implied proving that the analysis transfer function for a CALL instruction is more conservative than "stepping into" the called methods (as the abstract semantics does).

Now, after we completed the proof, it is worth looking at how we used each feature of the analysis and the abstract semantics. We used the definition of the analysis transfer function when we proved the invariants which enstablish the link between the concrete and the abstract semantics in Section 4.3. We used the definition of the transfer function for a CALL instruction In Section 4.4. In particular, we used the rules for constructing the node mapping from the inter-procedural analysis in the proof of Equation 4.21: Case 1.1.2 of a LOAD instruction relies on Constraint 2.7 while Case 1.2 relies on Constraint 2.8. The fact that we have actively used all these analysis components is a good mark for the analysis design.

The only thing that remained unexplained yet is the reason for using nodes with contexts in the abstract semantics instead of plain nodes. Now, we can finally explain it. Our proof relied on the existence of three layers — concrete semantics, abstract semantics and pointer analysis — where each layer is more conservative than the previous one. One of the things we had to prove was that the inter-procedural analysis is a safe approximation of the abstract semantics which steps into the called methods. In the first proof attempt, we used plain nodes in the abstract semantics. However, while trying to do the proof, it turned out that in some cases the result of the inter-procedural analysis was not "bigger" than the result of the abstract semantics. In fact, both results were safe models of the concrete state, but there was no hierarchical relation between them. This prevented us from using the invariants proved for the abstract semantics to prove the correctness of Theorem 5. The next example presents such a "problematic" case:

Example 6. Consider the following piece of code, where, for simplicity, we use integer labels:

```
0: void main() {          6: static Object callee() {
1:   v1 = callee();       7:   v4 = new C;
2:   v2 = new C;          8:   v5 = v4.f;
3:   v1.f = v2;           9:   return v4;
4:   v3 = callee();      10: }
5: }
```

As this program has a single thread of execution, the only possible trace is the following one:

date	label	instruction to be executed
0	1	`v1 = callee();`
1	7	`v4 = new C;`
2	8	`v5 = v4.f;`
3	9	`return v4;`
4	2	`v2 = new C;`
5	3	`v1.f = v2;`
6	4	`v3 = callee();`
7	7	`v4 = new C;`
8	8	`v5 = v4.f;`
9	9	`return v4;`
10	5	*end of execution*

In the previous table, the label attached to each date is the label of the instruction that the program starts executing at that date.

The abstract semantics of the only activation of the main method starts with the initial state $\Xi_0^{\#}$, and sequentially processes the instructions from the execution trace. Figure 4-18 offers a graphic representation of the most interesting abstract states. The first instruction is a CALL to `callee`; the abstract semantics creates a stack frame for `callee` and steps into it. The NEW instruction from label 7 makes `v4` point to the inside node n_7^I; in addition, is sets the field `f` of n_7^I to point to the node $n_{\texttt{null}}$ that models the null references (Figure 4-18.a). Next, we load the field `f` from n_7^I, which is currently pointing to $n_{\texttt{null}}$, into the local variable `v3` (Figure 4-18.b).

In the next instructions, `callee` returns n_7^I to `main`, which sets its `f` field to point to the inside node n_2^I which corresponds to the NEW instruction from label 2 (Figure 4-18.c). Notice that as we do not do destructive updates, n_7^I points to both $n_{\texttt{null}}$ and n_2^I. Next, `callee` is called again and it creates a new object, represented by the same node n_7^I that represents the object created by the previous execution of the NEW instruction from label 7 (Figure 4-18.d).

We arrived at the interesting point: after the abstract semantics processes the LOAD instruction from label 8, `v5` points to both $n_{\texttt{null}}$ and n_2^I. (Figure 4-18.e). This is correct, but imprecise, because obviously the only value which can be loaded by the LOAD instruction is `null`. The imprecision is due to the fact that the same node

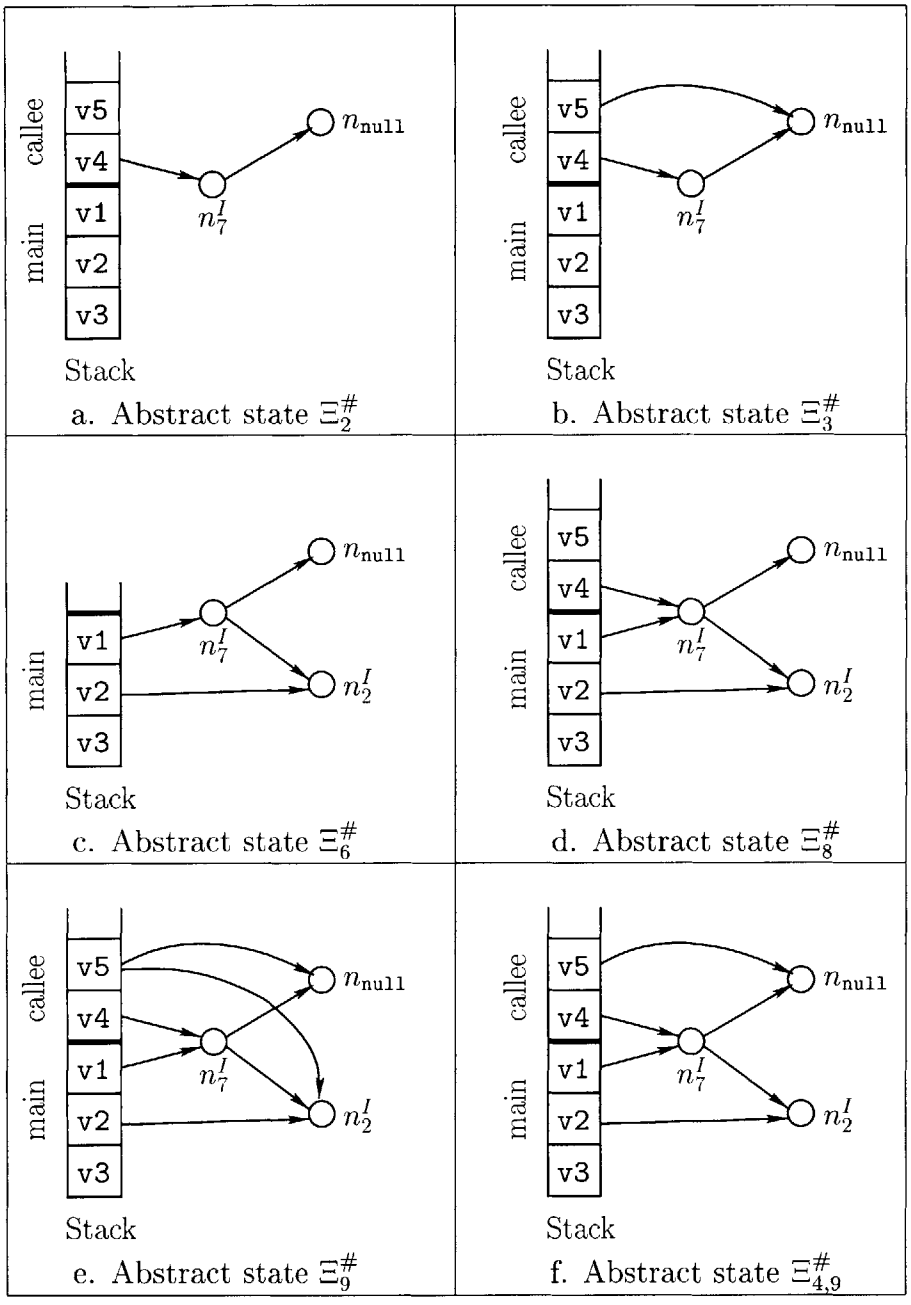


Figure 4-18: Abstract states for Example 6

models the two objects that the NEW instruction from label 7 created in the two invocations of `callee`.

Now let's compute the result of the inter-procedural combination for date 9, i.e., using the notations from Section 4.4:

$$\Xi^{\#}_{4,9} = interproc^{\#}_2(\Xi^{\#}_6, \Xi^{\#}_{2,9}, 4, \texttt{callee})$$

where $interproc^{\#}_2$ is as previously but this time it works with nodes without context and $\Xi^{\#}_{2,9}$ is the state computed by the abstract semantics of the second activation of `callee` for its end, i.e., the date 9. It can easily be checked that $\Xi^{\#}_{2,9}$ is like $\Xi^{\#}_3$ except that its stack does not have the frame for `main`. As there are no placeholders, $\Xi^{\#}_{4,9}$ is simply the merging of the two states. Figure 4-18.f presents a graphic representation of $\Xi^{\#}_{4,9}$. As in $\Xi^{\#}_{2,9}$, `v5` does not point to to n^I_2, it does not point to n^I_2 in $\Xi^{\#}_{4,9}$ either. But `v5` points to n^I_2 in $\Xi^{\#}_9$! So, $\Xi^{\#}_9 \not\sqsubseteq \Xi^{\#}_{4,9}$. △

The problem in the previous example is that when the abstract semantics steps into the second invocation of `callee`, the inside node it creates to model the object allocated in the NEW instruction from label 7, "collides" with the inside node which was created during the first invocation. As a result, the LOAD instruction reads edges which refer to a different object. In the state $\Xi^{\#}_{2,9}$, produced by running the abstract semantics on the second activation of `callee`, this problem does not arise because the abstract semantics is not interested in the objects which were created in the previous activations of `callee`.

This problem would not occur if we separate the nodes introduced for the current instance of a method from the nodes introduced for its previous activations. Pairing a node with the calling context in which it is created is a natural solution for accomplishing this.

Chapter 5

Related Work

This chapter is divided in two parts. First, we present several pointer analyses, and compare them with our analysis. Next, we investigate the much fewer correctness proofs for pointer analyses.

5.1 Pointer Analyses

In the presentation of the related pointer analyses, we focus on three orthogonal design parameters: heap modeling, flow sensitivity, and compositionality.

5.1.1 Heap Modeling

Modeling the heap, a potentially infinite structure, in a static analysis is an interesting problem. Some analyses, including ours, model the heap as a points-to graph of bounded size [6, 11, 26, 17, 7]. Other analysis do not use any graph at all: they either work with a set of pairs of aliased heap paths, i.e., paths that lead to the same object, e.g., [10], or use some other idea, e.g., [4]. In general, storing a points-to graph requires less memory space than storing the set of aliased paths from it: by standard graph theory, the number of paths is asymptotically bigger than the number of edges. Also, the points-to graphs have the advantage of offering an easy to understand view of the memory data structures.

As the number of objects that are created in the concrete execution might be unbounded, it is necessary to *summarize* many objects into a single node. Historically, *k-limiting* [14] is the first class of summarizing techniques. k-limiting uses a bound, k, on the maximum acyclic path length in the heap: objects that are reachable from the program variables along paths of length at most k are individually modeled by distinct nodes, while all the other objects are merged into a single summary node. This technique has several disadvantages:

- The number of nodes contained by a heap structure of depth k is exponential in k. For example, a complete binary tree of depth k contains $2^k - 1$ nodes. To maintain an acceptable number of nodes, k is usually chosen to be very

small, but this choice significantly limits the amount of information that can be retained.

- Summarizing all nodes deeper than k in a data structure ignores hints provided by the program structure. Some researchers [6] believe that some of the information provided by the data structure, which are lost in k-bound techniques, are more relevant than the length of the accessibility path.

In [6], Chase, Wegman and Zadeck proposed the alternative *object allocation site* model. This model is based on the assumption that objects allocated by the same statement in the program are likely to be manipulated in a similar way. The authors assume that this similarity is more important than the similarity that might exist between objects that are below the k depth. Being directly related to the structure of the program, it is expected that the results of the analysis using this model will be easier to understand and use. Our analysis uses an extended form of the *object allocation site* model, which contains, in addition to the standard nodes associated with the object creation sites, special placeholder nodes that allow us to have a compositional analysis.

Deutsch [10] proposed another model for going beyond the k-limiting techniques. His model does not model the heap as a graph of nodes. Instead, he represents the paths in the heap as regular expressions. In some cases, this model improves the accuracy of the analysis of recursive data structures that are accessed in a regular way.

5.1.2 Flow Sensitivity

Some pointer analyses [22, 2, 4] are *flow insensitive*: they do not consider the order of the program statements and compute a single result that is valid for the entire program, or one result for each method. By their nature, the flow insensitive analyses are similar to the type systems: they both compute results that are valid throughout an entire scope. Therefore, it is natural that they are often expressed as sets of typing rules.

The advantage of these analyses is that, similar to the type systems, they have small memory consumption, and are very fast in practice. They are the only pointer analyses that are known to scale well. Steensgaard's analysis is almost linear in practice [22]. Heitze and Tardieu [12] obtained a finely tuned implementation of a version of Andersen's alias analysis [2] that is able to analyze programs of millions of lines of code in seconds.

The disadvantage is the loss of precision. Unlike the type of a variable, which does not change during the execution, the heap is likely to vary significantly as new objects and references are created, and older references are deleted.

More sophisticated pointer analyses [6, 11, 26, 17, 7], are *flow sensitive*: they compute a result for each program point. At least theoretically, they are more precise than the flow insensitive analyses, and are able, in some cases, to handle destructive updates in order to increase their precision even more. This precision comes with a

cost: the flow sensitive analyses use more memory than the flow insensitive ones and are known to have scalability problems. We believe that as machines become more and more powerful, the flow sensitive analyses will be applicable for bigger classes of programs, and the new high-level applications of pointer analysis, e.g., program understanding and versification tools, will appreciate the additional precision of the flow sensitive analyses.

5.1.3 Compositionality

It is possible to obtain an inter-procedural analysis by a simple extension of an intra-procedural one: for each call instruction we introduce a control flow edge toward the beginning of each possible callee and one control flow edge from the exit point of each callee to the instruction immediately after the call instruction [16, Chapter 2]. As a result, the entire program is analyzed as a single big procedure. As this technique does not preserve the matching of the call/return instructions, it creates many false paths, i.e., paths in the control flow graph that do not correspond to any real execution path; the propagation of information along these false paths reduces the analysis precision. Also, analyzing the entire program requires the availability of the entire program code. We consider that this simple technique is unsuited for real programs, and focus instead on analyses that are able to analyze methods separately and have some way of composing the separate results in order to model the effects of the transitively called methods on the pointers.

One possible approach is to analyze the program in a top-down fashion starting from the `main` procedure, reanalyzing each potentially invoked procedure in each new calling context [11, 26, 17]. As the number of possible calling context might be very big, implementations of such analyses require some mechanism of limiting the memory consumption, which might require merging some of the calling contexts.

The other approach, which is used in our analysis, is to analyze the program in a bottom up fashion, starting with the leaves of the call graphs and advancing toward the `main` procedure [7, 4]. For each method, the analysis computes a parameterized result that is later instantiated for the different call sites that might call that method.

Choi, Gupta, Serrano, Sreedhar, and Midkiff present a compositional analysis that uses this idea [7]. Like our analysis, it uses placeholders to abstract over the unknown calling context. It uses a *connection graph* to model the heap. The connection graph abstraction is similar in many aspects to our *points-to graph*. However, there are some differences due to the fact that the their analysis is focused on computing escape information. For instance, their analysis classifies objects as globally escaping, escaping via an argument, and not escaping. Because the primary goal was to compute escape information, the analysis collapses globally escaping subgraphs into a single node instead of maintaining the extracted points-to information. Our analysis retains this information, in part because we anticipated further thread interaction analyses [23] that use this information. To the best of our knowledge, no correctness proof has yet

been presented for the analysis of Choi *et al.*

Vivien and Rinard [24] extended the analysis of Whaley and Rinard [25] to obtain an incremental analysis. Based on profile data, their analysis starts by analyzing only a small part of the program. It then applies a performance/cost strategy to efficiently extend the scope of the analysis to capture more and more nodes, until either the analysis budget was exhausted or the fate of all the interesting nodes[1] has been decided. Sălcianu and Rinard [23] proposed another extension of Whaley and Rinard analysis [25]. The resulting analysis is compositional not only at the method level but also at the thread level. Unfortunately, these last two analysis do not have a correctness proof yet.

5.2 Correctness Proofs for Pointer Analyses

Very few of the published pointer analyses have a correctness proof. For many flow insensitive analyses, this is not a big problem: they are basically type systems and standard type theory can be used to prove their correctness. However, the correctness of the other analyses, especially the flow sensitive ones, is a much more difficult issue.

Among the recently published analyses, only two have been proved correct: the flow insensitive analysis of Blanchet [4, 5], and the flow sensitive analysis of Sagiv, Reps and Wilhelm [21, 20]. The correctness proofs of both of them use the abstract interpretation framework of Cousot and Cousot [8]. In this section, we briefly discuss these two analyses and their correctness proofs.

The analysis proposed by Blanchet [4, 5] is a pure escape analysis aimed at discovering stack allocation opportunities in Java programs. The analysis is flow insensitive; it computes a result for each method. It does not use a graph representation of the heap: instead, it uses integer levels in a type hierarchy to represent the *escaping* parts of the objects. E.g., if an object o does not escape but the object pointed to by its only field f escapes, then the escaping part of o is $o.f$; if the type of the field f is τ, the analysis represents this escaping part by the level of τ in a type hierarchy based on the type declarations. The type hierarchy satisfies the property that the level of a type is at least as big as the level of any of its subtypes and "contained" types, i.e. types of its fields. If the level associated with the escaping part of an object is strictly smaller than the level of the type of the object, then the object does not escape the method, and can be stack allocated. The analysis is inter-procedural and compositional: the result it computes for a method is a function on the calling context. Each method is analyzed once; the obtained function is applied for each calling context. The analysis has polynomial complexity. As it computes additive functions of integers, it is very fast in practice. The correctness proof for this analysis is a perfect application of the abstract interpretation framework.

[1] The nodes that correspond to the allocation sites that allocate most of the objects, as indicated by the profile data.

We consider that Blanchet's analysis and its proof are less complex than ours: his analysis uses a big approximation — the type levels — and is flow insensitive. Due to the heavy use of the type hierarchy, the analysis seems to give a huge importance to the type declarations. For some simple type declarations, the analysis will not be able to do any stack allocation, independent of the specific code of the program. E.g., consider a set of two types where each type has a field of the other type. In this case, there is a single level in the type hierarchy and, no matter what the program code looks like, no stack allocation is performed by the analysis.

The analysis of Sagiv, Reps and Wilhelm [21, 20] is a flow sensitive shape analysis for a toy language. It uses graphs to model the heap but it does not use the object allocation site model. Instead, each object that is directly pointed to by a local variable is modeled by a distinct node; all the other objects are modeled by a summary node. When a node is no longer pointed to by a variable, it is merged with the summary node. When a variable is set to point to the summary node, a node is extracted from the summary node. To allow a significantly precise node extraction, a boolean predicate — "is shared" — is maintained for the summary node. For each program point, the analysis computes a possibly exponential number of graphs. This makes possible a precise processing of destructive updates but results in an exponential complexity. As presented in [21, 20], the analysis is mainly intra-procedural. The authors presents several ideas for extending it to the inter-procedural case but they do not specify any clear inter-procedural analysis.

Unlike the proof for our analysis, the correctness proofs for the two analyses that we presented in this section are both based on the abstract interpretation framework. The abstract interpretation framework regards the analysis of a program as an abstract interpretation of that program in an abstract space. It requires an abstraction function from the concrete states to the abstract ones and a concretization function in the opposite direction. The advantage of the abstract interpretation framework is that it proposes a standard approach to correctness. We tried to apply this framework for our analysis too, but it seems that due to the compexity of the analysis, the abstraction and the concretization functions are difficult to find. We do not know yet if this is the result of our inexperience with the abstract interpretation framework, or is due to some feature of our analysis.

Chapter 6

Conclusions and Future Work

In this thesis we investigated the design and the correctness of a pointer analysis for the Java programming language.

The analysis was presented in the context of the Java programming language [3]. It is a flow sensitive, compositional, inter-procedural pointer analysis. We obtained the analysis by completely redesigning the pointer analysis published by Whaley and Rinard [25]. The final analysis is quite different from the original one. Therefore, we consider it to be a new analysis.

The analysis is based on the abstraction of points-to graphs, which characterize how local variables and fields in objects point to other objects. We use the *object allocation site* model: all objects that are allocated at the same allocation site are modeled by the same node. Each points-to graph also contains escape information that characterizes how objects allocated in the analyzed part of the program can be accessed by other parts. The analysis computes a single parameterized points-to graph for the exit point of each method. This points-to graph uses placeholders to abstract over the calling context. The inter-procedural analysis instantiates such a points-to graph for each calling context, by using a set of rules for disambiguating the placeholders. The algorithm is able to analyze parts of the program, obtaining precise information about the captured nodes, i.e., the nodes which are not reachable from outside the analyzed part of the program.

We presented three of the many possible applications of the analysis: stack allocation, thread-local heap allocation and synchronization removal. However, a compiler for a safe language like Java should not use an analysis that is not formally proved to be correct, no matter how big improvements that analysis might produce. We believe that the the steady advances in hardware speed will shift the focus of the pointer analysis, and program analysis in general, from program optimization to program verification. This will further increase the need of using only analyses that are formally proved to be correct. To address this issue, we allocated the major part of this thesis to the correctness proof of the analysis.

The proof has a multi-layer structure. At the bottom level we have the concrete semantics of Java. The pointer analysis is the top layer. To cope with the big difference in the complexity of these two layers, we introduced a third, intermediate layer:

the abstract semantics. For each relevant point from the execution of a program, the abstract semantics computes an abstract state that models the heap, and an explicit abstraction relation that records how nodes model objects. We proved that in this hierarchy, each layer is a conservative approximation of the layer beneath it. The proof has two parts: the first part relates the abstract semantics to the concrete semantics, while the second part relates the pointer analysis to the abstract semantics. To the best of our knowledge, this is the first correctness proof for a flow sensitive, compositional, inter-procedural analysis.

The correctness proof was long and difficult. To complete it, we had to introduce a considerable amount of notations and auxiliary results. Designing the right formalism, selecting notations, and choosing the right intermediate steps in the proof of the analysis took much longer than the proof itself. Usually, once we decided on the intermediate steps to be accomplished, each step was solved by an easy, but sometimes long, proof by induction.

For this reason, we believe it is interesting to investigate the possibility of simplifying the proof by using the abstract interpretation framework of Cousot and Cousot [8]. Expressing the analysis and the proof in such a standard framework will hopefully allow the use of many existing results. It is possible that the analysis is inherently difficult, or even impossible to express in the abstract semantics framework. In this case, it would be interesting to identify those features of the analysis that are responsible for this situation.

The second observation is more positive than the first one: proofs, in spite of their difficulty, are useful! Designing the analysis was a difficult task, marked by the frustrating discoveries of corner cases that were not handled properly by the initial versions of the analysis. Completing the proofs helped us to clarify our ideas, and detect the missing part. It seems that while intuition remains fundamental for the design of a new analysis, the problems introduced by the aliasing that might appear in the context of pointers are very complex, and require more than just intuition.

A possible direction of future research concerns the various extensions that we can add to the analysis in order to enhance its precision: context sensitivity, strong updates, etc. The correctness proof will have to be modified to cover all of these extensions. As the correctness proof is already very big, special care should be taken to preserve its modularity.

A more immediate continuation of the work presented in this thesis is the completion of the analysis implemention, and the experimental evaluation of the analysis. We are already working on such an implementation and plan to present it in a future publication.

Appendix A

Proof of Equation 4.23

Recall that Equation 4.23 has the form

$$\alpha(ip_{2i})(interproc^{\#}(\Xi^{\#}_{ip_{2i}}, \gamma(\Xi^{\#}_{2,d}), lb_{2i}, callee, [ip_{2i}])) \sqsubseteq \\ interproc^{\#}(\Xi^{\#}_{ip_{2i}}, \Xi^{\#}_{2,d}, lb_{2i}, callee, [])$$

where $d = id_{callee,2u+1}$.

Notations: To increase the readability, we use the notation $\Xi^{\#}_{callee} = \Xi^{\#}_{2,d}$. We also rewrite Equation 4.23 as

$$\alpha(ip_{2i})(simplify^{\#}(\Xi^{\#}_1)) \sqsubseteq simplify^{\#}(\Xi^{\#}_2)$$

where

$$\begin{array}{rcl} \Xi^{\#}_1 & = & \textbf{let} \quad \mu'_1 = mapping^{\#}(\Xi^{\#}_{ip_{2i}}, \gamma(\Xi^{\#}_{callee}), lb_{2i}, callee, [ip_{2i}]) \quad \textbf{in} \\ & & \quad combine^{\#}(\Xi^{\#}_{ip_{2i}}, \gamma(\Xi^{\#}_{callee}), \mu'_1, v_R) \\ & & \qquad \text{and} \\ \Xi^{\#}_2 & = & \textbf{let} \quad \mu'_2 = mapping^{\#}(\Xi^{\#}_{ip_{2i}}, \Xi^{\#}_{callee}, lb_{2i}, callee, []) \quad \textbf{in} \\ & & \quad combine^{\#}(\Xi^{\#}_{ip_{2i}}, \Xi^{\#}_{callee}, \mu'_2, v_R) \end{array} \tag{A.1}$$

Proof outline: First, we formulate three auxiliary results. The first two state that the conversion $\alpha(ip_{2i})$ propagates deep into the definition of $\Xi^{\#}_1$, commuting with $simplify^{\#}$ and $combine^{\#}$. The last result states an interesting relation between the mappings μ'_1 and μ'_2. Next, we use these results to prove Equation 4.23. We conclude by proving the three auxiliary results.

Lemma 25 (α and $simplify^{\#}$ commute). $\forall d \in Date, \forall \Xi^{\#} \in State^{\#}$,

$$\alpha(d)(simplify^{\#}(\Xi^{\#})) \sqsubseteq simplify^{\#}(\alpha(d)(\Xi^{\#})),$$

Lemma 26 ($\alpha(ip_{2i})$ and $combine^{\#}$ commute). *With the notations from Equa-*

tion A.1,

$$\begin{aligned}\alpha(ip_{2i})(\Xi_1^\#) &= \alpha(ip_{2i})(combine^\#(\Xi^\#_{ip_{2i}}, \gamma(\Xi^\#_{callee}), \mu'_1, v_R)) \\ &\sqsubseteq combine^\#(\Xi^\#_{ip_{2i}}, \Xi^\#_{callee}, \alpha(ip_{2i})(\mu'_1), v_R)\end{aligned}$$

where

$$\alpha(ip_{2i})(\mu'_1) = \{\langle \alpha(ip_{2i})(n_1), \alpha(ip_{2i})(n_2)\rangle \mid \langle n_1, n_2\rangle \in \mu'_1\}$$

Lemma 27. *With the notations from Equation A.1,* $\alpha(ip_{2i})(\mu'_1) \subseteq \mu'_2$.

Proof of Equation 4.23: First, by a direct application of Lemma 25, we obtain:

$$\alpha(ip_{2i})(simplify^\#(\Xi_1^\#)) \sqsubseteq simplify^\#(\alpha(ip_{2i})(\Xi_1^\#)) \tag{A.2}$$

Next, we combine Lemma 27, Lemma 26, and the obvious monotonicity of $combine^\#$ in its mapping argument, and obtain:

$$\begin{aligned}\alpha(ip_{2i})(\Xi_1^\#) &\sqsubseteq combine^\#(\Xi^\#_{ip_{2i}}, \Xi^\#_{callee}, \alpha(ip_{2i})(\mu'_1), v_R) \\ &\sqsubseteq combine^\#(\Xi^\#_{ip_{2i}}, \Xi^\#_{callee}, \mu'_2, v_R)\end{aligned}$$

Furthermore, as $simplify^\#$ is monotonic (for the same reasons *simplify* is monotonic), we can extend Equation A.2 as follows:

$$\begin{aligned}\alpha(ip_{2i})(simplify^\#(\Xi_1^\#)) &\sqsubseteq simplify^\#(\alpha(ip_{2i})(\Xi_1^\#)) \\ &\sqsubseteq simplify^\#(combine^\#(\Xi^\#_{ip_{2i}}, \Xi^\#_{callee}, \mu'_1, v_R)) \\ &= interproc^\#(\Xi^\#_{ip_{2i}}, \Xi^\#_{callee}, lb_{2i}, callee, [])\end{aligned}$$

As in our notations, $\alpha(ip_{2i})(simplify^\#(\Xi_1^\#))$ is

$$\alpha(ip_{2i})(interproc^\#(\Xi^\#_{ip_{2i}}, \gamma(\Xi^\#_{2,d}), lb_{2i}, callee, [ip_{2i}]))$$

where $d = id_{callee,2u+1}$, this completes the proof of Equation 4.23 □

Proof of Lemma 25: For any path along the inside and the outside edges of the abstract state $\Xi^\#$ from node n_1 to node n_2, there is a path along the inside and the outside edges of the abstract state $\alpha(d)(\Xi^\#)$ from $\alpha(d)(n_1)$ to $\alpha(d)(n_2)$. This implies that if n escapes in $\Xi^\#$, then $\alpha(d)(n)$ escapes in $\alpha(d)(\Xi^\#)$, too. An immediate consequence of this observation is that for each node or edge which is preserved by $simplify^\#(\Xi^\#)$, its projection through $\alpha(d)$ is preserved by $simplify^\#(\alpha(d)(\Xi^\#))$, too. This ends our proof. □

Proof of Lemma 26: We introduce the following notations:

$$\begin{aligned}\Xi_3^\# &= \alpha(ip_{2i})(\Xi_1^\#)\\ \Xi_4^\# &= combine^\#(\Xi_{ip_{2i}}^\#, \Xi_{callee}^\#, \alpha(ip_{2i})(\mu_1'), v_R)\\ \Xi_5^\# &= \gamma(\Xi_{callee}^\#)\end{aligned}$$

With these notations, we need to prove that each component of $\Xi_3^\#$ is smaller than the corresponding component from $\Xi_4^\#$. We present only the proof for $S_3^\# \subseteq S_4^\#$; the proofs for the other components are similar. Using the definition of $combine^\#$ and the fact that none of the nodes from $S_{id_{2i}}^\#$ contains ip_{2i} in its context, we have:

$$\begin{aligned}S_3^\# &= \alpha(ip_{2i})(S_{ip_{2i}}^\# \cup S_5^\#[\mu_1']) = S_{ip_{2i}}^\# \cup \alpha(ip_{2i})(S_5^\#[\mu_1'])\\ S_4^\# &= S_{ip_{2i}}^\# \cup S_{callee}^\#[\mu_3']\end{aligned}$$

where $\mu_3' = \alpha(ip_{2i})(\mu_1')$. Consider $n \in S_3^\#$. If $n \in S_{ip_{2i}}^\#$, then clearly $n \in S_4^\#$ too. If $n \in \alpha(ip_{2i})(S_5^\#[\mu_1'])$ then, as $S_5^\# = \gamma(S_{callee}^\#)$, there exist nodes n_1, n_2 such that

- $n_1 \in S_{callee}^\#$ (i.e. $\gamma(n_1) \in S_5^\#$)
- $\langle \gamma(n_1),\ n_2 \rangle \in \mu_1'$ and
- $n = \alpha(ip_{2i})(n_2)$.

By the definition of μ_3', $\langle \gamma(n_1),\ n_2 \rangle \in \mu_1'$ implies $\langle n_1,\ \alpha(ip_{2i})(n_2) \rangle \in \mu_3'$ (because $\alpha(ip_{2i})(\gamma(n_1)) = n_1$). Hence, $n = \alpha(ip_{2i})(n_2) \in S_{callee}^\#[\mu_3'] \subseteq S_4^\#$. As n was arbitrarily chosen, $S_3^\# \subseteq S_4^\#$ as desired. This completes the proof of Lemma 26. □

Proof of Lemma 27: Application of Lemma 18 for $d = ip_{2i}$, followed by the following observations:

1. $\alpha(ip_{2i})(\Xi_{ip_{2i}}^\#) = \Xi_{ip_{2i}}^\#$ because none of the nodes from $\Xi_{ip_{2i}}^\#$ contains ip_{2i} in its context (they were all created by instructions executed before ip_{2i}).
2. $\alpha(ip_{2i})(\gamma(\Xi_{callee}^\#)) = \Xi_{callee}^\#$. Trivial: γ changes the context of each node that appear in $\Xi_{callee}^\#$ from [] to $[ip_{2i}]$; next, $\alpha(ip_{2i})$ changes it back from $[ip_{2i}]$ to [].
3. $\alpha(ip_{2i})([ip_{2i}]) = []$. Trivial.

□

Bibliography

[1] Ole Agesen. The cartesian product algorithm. In *Proceedings of the 9th European Conference on Object-Oriented Programming*. Springer-Verlag LNCS, 1995.

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

[3] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, Mass., 1996.

[4] B. 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*, Denver, CO, November 1999.

[5] Bruno Blanchet. *Escape Analysis. Applications to ML and Java*^{TM}. PhD thesis, École Polytechnique, December 2000.

[6] D. Chase, M. Wegman, and F. Zadek. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, pages 296–310, White Plains, NY, June 1990. ACM, New York.

[7] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[8] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM Symposium on the Principles of Programming Languages*, pages 238–252, 1977.

[9] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, Aarhus, Denmark, August 1995.

[10] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994.

[11] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*, pages 242–256, Orlando, FL, June 1994. ACM, New York.

[12] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the SIGPLAN '01 Conference on Programming Language Design and Implementation*, 2001.

[13] 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.

[14] 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.

[15] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufman Publishers, Inc., 1996.

[16] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[17] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the SIGPLAN '99 Conference on Program Language Design and Implementation*, Atlanta, GA, May 1999.

[18] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indexes, and accessed memory regions. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada, June 2000.

[19] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, pages 16–31, January 1996.

[20] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[21] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, 1996.

[22] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, January 1996.

[23] Alexandru Sălcianu and Martin C. Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2001.

[24] Frederic Vivien and Martin C. Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the SIGPLAN '01 Conference on Programming Language Design and Implementation*, 2001.

[25] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[26] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995. ACM, New York.