Flexible Goal-Directed Abstraction

by

Sam Blackshear

B.A., Williams College, 2010

M.S., University of Colorado Boulder, 2012

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2015

This thesis entitled:
Flexible Goal-Directed Abstraction
written by Sam Blackshear
has been approved for the Department of Computer Science

Prof. Bor-Yuh Evan Chang

Prof. Pavol Černý

Prof. Mayur Naik

Prof. Sriram Sankaranarayanan

Dr. Manu Sridharan

Date ____________________

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Blackshear, Sam (Ph.D., Computer Science)

Flexible Goal-Directed Abstraction

Thesis directed by Prof. Bor-Yuh Evan Chang

Static program analysis is a powerful technique for bug-finding, verification, and program understanding. Yet static analyses remain conspicuously absent from the toolbox of the average developer because they must abstract away details about concrete program behavior. Designing effective abstractions is tricky business: imprecise abstractions are inexpensive, but can be useless because they lose too much information about the program, whereas conventional wisdom states that precise abstractions are too expensive to scale.

This dissertation considers the problem of **goal-directed** static analysis, an intriguing domain where there is hope for defying the conventional wisdom about precise and scalable abstractions. In contrast to more traditional whole-program approaches that must apply precise abstractions to the entire program, goal-directed approaches have the potential to be much more tractable because they can focus the effort of the analysis on a single goal **query**.

There are two fundamental challenges in goal-directed analysis: (A) designing abstractions that are as **flexible** as possible so they can be specialized to the needs of the query and (B) using this flexibility wisely to achieve a better precision/scalability tradeoff in practice. This dissertation addresses these challenges by introducing **goal-directed abstraction coarsening**, a new approach to goal-directed analysis. Our approach works backward from the goal query using an abstraction that is as precise as possible by default, but can be coarsened in order to narrow the focus of the analysis and improve scalability. We meet Challenge A by introducing flexible coarsening-based techniques for store abstraction and control-flow abstraction. Our store abstraction precisely represents a small view of the store relevant to the query by combining a separation logic-based representation with a flow-insensitive points-to analysis. We present a general framework for flexible control-flow abstraction that allows the analysis to coarsen the control-flow abstraction by **jumping** over irrelevant code. Finally, we meet Challenge B by presenting goal-directed analyses incorporating our flexible abstractions. Our analyses use programmers' invariant-based reasoning and the structure of event-driven programs to recognize opportunities for coarsening that are likely to enhance scalability without losing precision. We have implemented these analyses and shown that they can achieve precise and scalable results for a variety of client analyses than cannot be handled effectively by previous techniques.

Dedication

To my parents, who have unconditionally supported me in all of my undertakings.

Contents

Chapter 1

Introduction

Software quality is becoming increasingly important. Recent software vulnerabilities like the Heartbleed[1] , Shellshock[2] , and FREAK[3] bugs demonstrate that even a small mistake in a single piece of software can have far-reaching consequences. As software becomes both more pervasive and more complex, we need to support overburdened developers with better automated tools for helping them understand their code and ensure its correctness.

Static program analysis presents a promising approach to improving the state of developer tooling. Static analysis is the only technique capable of automatically **proving** the complete absence of important classes of errors, yet it is underutilized compared to techniques such as testing, profiling, and manual code review. The difficulty of static analysis is that proving nontrivial properties about programs is undecidable in general [Rice, 1953], which means that analyses must resort to **abstraction** to ensure termination. The foundational theory of abstract interpretation [Cousot and Cousot, 1977] lays out a general framework for using **sound** abstractions (abstractions that overapproximate all possible concrete behaviors of a program) to perform static analysis of programs. In the first phase, the analysis uses a sound abstraction to compute an overapproximation of the reachable concrete states of the program. In the second phase, this overapproximation can then be used to resolve **queries** about the concrete behavior program. For example, if the analysis can prove that a query representing some undesirable behavior (e.g., a crash from dereferencing a null pointer) never occurs in an overapproximation of the program's concrete behaviors, we can conclude

[1] `http://heartbleed.com`
[2] `https://shellshocker.net`
[3] `https://freakattack.com`

that it never occurs in the program's actual concrete behaviors. However, an abstraction that is too imprecise may prevent the analysis from proving the absence of the buggy behavior even if the actual program is safe. The result of this abstraction mistake is a **false alarm** that is reported alongside the true bugs found by the analysis. The noise caused by high false alarm rates is a major hindrance to mainstream adoption of static analysis tools [Bessey et al., 2010; Johnson et al., 2013; McPeak et al., 2013].

Given the problems caused by imprecise abstractions, the obvious suggestion is to use abstractions that are as precise as possible in our static analyses. This is easier said than done because analyses using precise abstractions tend to incur scalability issues. For example, understanding which branch of a conditional that the program may take is frequently important for precision, but the cost of a **path-sensitive** abstraction that can distinguish one program path from another can be exponential in the number of conditional branches in the program. In practical terms, these scalability issues make analysis tools too slow to integrate into a developer's workflow [Calcagno et al., 2015; Layman et al., 2007; Sadowski et al., 2015], or (in the worst case) cause the analysis to take so much time or memory that it effectively does not terminate.

Thus, designing practically effective abstractions is tricky business. The fundamental tension is selecting an abstraction that makes a reasonable tradeoff between precision and scalability. Conventional wisdom says that we cannot make precise abstractions scale to large programs. However, **goal-directed** (also called "demand-driven" [Heintze and Tardieu, 2001; Horwitz et al., 1995; Sridharan and Bodík, 2006; Sridharan et al., 2005], "client-driven" [Guyer and Lin, 2003, 2005], "selective" [Oh et al., 2014], "property-directed" [Itzhaky et al., 2014], and "counterexample-guided" [Ball et al., 2011; Clarke et al., 2000; Henzinger et al., 2002; Lal et al., 2012; Zhang et al., 2013, 2014]) analyses present a promising approach for defying this conventional wisdom A goal-directed analysis seeks only to prove the safety of a goal **query** rather than performing a precise analysis of the entire program. Although it may not be possible to scale a very precise abstraction to a large program, it is frequently the case that the safety of a goal query only depends on a few small fragments of the program. If the analysis can identify the fragments relevant to the query and focus its effort on these fragments, then there is hope to scale very precise abstractions regardless of how large the program is.

Focusing the effort of the analysis on a few fragments relevant to the goal query sounds simple, but poses several challenges. Since the analysis may be asked to answer hundreds or thousands of queries, it is not practical to manually design an abstraction for each query. Instead, the analysis must be **flexible** so that it can change to suit the needs of a particular goal query, and this specialization must be performed automatically.

The more flexibility we can give our abstractions, the more closely they can be specialized to the needs of each query. However, with great power comes great responsibility: more ways to change the abstraction means more opportunities to make a choice that will lose valuable precision, cause scalability problems, or both! We must harness the power of flexibility wisely in order to make good choices that yield better precision/scalability tradeoffs in practice.

1.1 Problem statement

Thus, the problems faced by any goal-directed analysis are:

(A) Making abstractions as flexible as possible so they can be specialized to the needs of the query, and

(B) Automatically using this flexibility wisely to enable better precision/scalability tradeoffs in practice.

In Chapter 2, we will explain each of these challenges in detail. We define a spectrum for the flexibility of abstractions in order to evaluate the success of different approaches to goal-directed abstraction in meeting this challenge. In relation to Challenge A, we introduce the concepts of **dimensionality** (Section 2.1 and **granularity** (Section 2.2). Dimensionality describes **how** the abstraction can be changed to meet the needs of a query during automated analysis. Our discussion of dimensionality considers both **store abstraction** (i.e., how the analysis abstracts locals, globals, and the heap) and the **control-flow abstraction** (i.e., how the analysis chooses context-, path-, and flow-sensitivity policies). A maximally flexible goal-directed abstraction should be able to vary its abstraction along each of these dimensions; for example, it should be able to precisely abstract a few local variables while ignoring the rest, and it should be able to switch path-sensitive and flow-insensitive modes.

The concept of **granularity** (Section 2.2) describes **when** or **how often** the abstraction can be changed during automated analysis. In particular, we consider whether the abstraction can be changed in a coarse-grained way (for example, only between distinct analysis stages) or in a more fine-grained way (for example, after each procedure call). A maximally flexible goal-directed abstraction should be able to change **on-the-fly** at any time during analysis.

We characterize the abstractions of previous work using both of these concepts and explain how these approaches meet Challenge B by wisely using the flexibility that they are capable of. However, we argue that previous approaches fall short of fully meeting Challenge (A): no single approach allows flexibility along all of the dimensions defined in Section 2.1 at the fine-grained level described in Section 2.2.

1.2 Thesis statement and contributions

This dissertation presents **goal-directed abstraction coarsening**, a novel approach to goal-directed static analysis that meets both challenges outlined in Section 1.1. Our analysis works backward from the goal query using an abstraction that is as precise as possible by default, but we identify opportunities to improve scalability by **coarsening**, or intentionally losing precision. As we will explain in Section 3.1, this is a very natural approach because coarsening is a fundamental part of any abstract interpretation. We will show that coarsening the abstraction in various ways allows us to vary the abstraction along all of the dimensions defined in Section 2.1. This approach is extremely fine-grained because the decision to change the abstraction by coarsening can be made on-the-fly at any time during analysis.

The more dimensionality and granularity an abstraction has, the more choices the analysis has for changing the abstraction and thus the more challenging it is for the analysis to use its flexibility wisely. This means that meeting Challenge B is especially difficult and important for our approach. Our strategy for meeting this challenge is to make our analysis parametric with respect to a coarsening strategy in order to allow defining appropriate strategies for different problem domains. We demonstrate that two specific strategies based on (a) understanding the reasoning patterns of programmers and (b) leveraging the structure of event-driven programs are effective for precise and scalable analysis for a variety of clients.

The thesis of this dissertation is our approach does indeed meet Challenges (A) and (B):

On-the-fly abstraction coarsening is a flexible and practical approach to goal-directed static analysis.

By **flexible**, we mean that coarsening allows the abstraction to be changed along each of the dimensions we have described at any time during analysis. By **practical**, we mean that we can choose effective strategies for coarsening that enable better precision/scalability tradeoffs in practice for static analysis of real-world object-oriented programs. To defend this thesis, we make the following contributions:

(1) We introduce a precise separation logic-based store abstraction that meets our dimensionality and granularity requirements (Chapter 4) . Our store abstraction can be coarsened each time a transfer function is applied. The heap, local, and global portions of the query are all represented in a uniform way using separation logic. Our store abstraction overcomes the scalability problems that aliasing causes for many previous approaches (see Section 2.1.1, "Abstracting the heap: previous approaches" paragraph) via a novel integration of points-to analysis and separation logic (Section 4.3). This combination enables a tractable backward analysis that only represents constraints relevant to the query precisely.

(2) We present a general framework for control-flow abstraction via **jumping** (Chapter 5). Jumping allows the analysis to jump directly to relevant code, skipping irrelevant code in between. We state soundness conditions for relevance based on **data-relevance** (detecting which commands may write to the abstract store) and **control-feasibility** (detecting which commands are reachable from the current program location) criteria. We demonstrate that adjusting the precision of these criteria allows us to vary the path-, flow-, and context-sensitivity of the analysis in a uniform and fine-grained fashion (in contrast to the previous approaches described in Section 2.1.2).

(3) We show that our store and control-flow abstractions can be combined with effective strategies for harnessing flexibility to create precise and scalable analyses. In particular, we specialize our jumping framework to leverage programmers' invariant-based reasoning (Chapter 6) and to take advantage of the structure of event-driven programs (Chapter 7). In both cases, we obtain significant scalability benefits from using the flexibility of our abstractions wisely.

1.3 Outline of dissertation

The remainder of this dissertation is organized as follows. In Chapter 2, we present a spectrum for characterizing the flexibility of abstractions and use spectrum to place previous work on goal-directed abstraction in context. Chapter 3 presents a challenge example whose verification requires all of the machinery that will be introduced in this thesis: an effective store abstraction for the heap, a flexible form of control-flow abstraction, and a practical approach to harnessing the flexibility of our abstractions to enhance scalability without losing precision. Chapter 4 introduces a store abstraction specialized for precisely representing heap constraints by combining separation logic with points-to analysis. Chapter 5 lays out a general framework for flexible control-flow abstraction based on **jumping** from one code region to another. The next two chapters demonstrate how to leverage the flexibility offered by jumping to achieve better precision/scalability tradeoffs in practice. Chapter 6 shows how to combine jumping with common forms of invariant-based reasoning used by real programmers to improve scalability during precise analysis of programs with deep call chains. Our key idea is to recognize when the programmer is relying on a flow-insensitive invariant and jump directly to the code where the invariant is established, improving scalability by skipping the irrelevant code in between the establishment and use of the invariant. Chapter 7 describes how to use jumping to scale analysis of event-driven Android programs when the analysis must consider an exponential number of possible event orderings in order to be both sound and precise. Our approach is to identify a small set of events that are both data-relevant to the current query and control-feasible with respect to the current event and jump directly to these events, improving scalability by avoiding consideration of large numbers of irrelevant event orderings.

Chapter 2

Prologue: a spectrum for flexible abstraction

In this chapter, we elaborate on the problem statement of flexible abstraction presented in Section 1.1 by presenting a spectrum for describing flexible abstractions. As previously explained, we divide flexibility into the concept of **dimensionality**, or how an abstraction can be changed (to be discussed in Section 2.1) and the concept of **granularity**, or when/how often an abstraction can be changed. For each sub-topic of the two main concepts, we first show what the specific abstraction problem is using examples, then discuss how previous work has addressed the problem. We explain how each piece of work has uses its flexibility wisely (i.e., meets Challenge B) as part of this discussion.

2.1 Dimensionality: how abstractions can be flexible

In the design of any static analysis, the two key decisions are how to abstract the program **store** and how to abstract the **control-flow** of a program. These are two distinct abstraction challenges that can be tackled independently from one another. We discuss each of them in turn.

2.1.1 Store abstraction: abstracting locals, globals, and the heap

The program store holds the data written by the program during concrete execution. In modern programming languages, there are three parts of the store that are important to consider: local variables, global variables, and the heap. From the perspective of defining flexible abstractions specialized to a query, the question is which variables and heap cells the abstraction should track precisely at each point in the program. An ideal goal-directed analysis tracks as few variable and heap cells as possible in order to maintain the level of precision required to proved the safety of the query.

Abstracting local variables: the problem. Consider the simple program in Figure 2.2. The goal query in this example is to prove that the asserted expression `c != null` can never evaluate to false. The program has four local variables (a, b, c, and d). Tracking the data flow into all of the variables is potentially expensive. The `bar` function must be analyzed to determine what values may flow into d, and all of the call sites of `foo` must be analyzed to determine what values may flow into the a and b parameters. However, the abstraction only needs to track the value of the c variable precisely in order to prove the safety of the query. The value of all other local variables can be soundly represented with $\top$, an abstract value representing all possible concrete values. This avoids the cost of tracking irrelevant variables precisely, but without sacrificing the precision required to prove the query safe.

```
void foo(Object a, Object b) {
  Object c = new Object();
  Object d = bar(a, b);
  assert(c != null);
}
```

Figure 2.1: An abstraction that tracks only the relevant variable c precisely can expend less effort in proving the query without sacrificing precision.

Abstracting local variables: previous approaches. Existing techniques are effective at finding abstractions that track only the relevant locals. Counterexample-guided abstraction refinement (CEGAR)-based analyses (e.g., [Ball et al., 2011; Clarke et al., 2000; Henzinger et al., 2002; Lal et al., 2012; Zhang et al., 2013, 2014]) are probably the most popular form of goal-directed analysis and have an effective solution to this problem. The CEGAR approach is to use an coarse abstraction describing the set of predicates that should be tracked precisely (initially, no predicates), then iteratively refine the abstraction in response to counterexamples (i.e., abstract execution sequences that falsify the goal query). For the example in Figure 2.2, a CEGAR-based analysis would initially abstract all variables using $\top$, but would quickly find a counterexample to the query on the code path where the variable c holds the value null. The analysis would block this counterexample by choosing a new abstraction where the variable c is tracked precisely. Under this more precise abstraction, the analysis can conclude that c is assigned to a non-null value at line 2 and thus so the assertion at line 4 cannot fail.

The example in Figure 2.2 is quite simple and only required tracking a single predicate to prove the query, but CEGAR-based analyses can easy handle more complicated examples involving hundreds of relevant predicates via additional iterations of the abstraction refinement loop. In addition, more modern CEGAR-based analyses (e.g., [Lal et al., 2012; Zhang et al., 2013, 2014]) use sophisticated counterexample analysis techniques to learn many important predicates to track from a single counterexample.

Abstracting global variables: the problem. The problem of abstracting global variables in a goal-directed fashion is similar to the problem of abstracting locals, but the mutability of global variables presents an additional complication: for mutable variables, the analysis must think about the **interval** for which the variable should be precisely tracked (that is, when should the analysis track the variable?). Though local variables are also mutable in many languages (that is, the same local variable can be assigned to multiple values), we can transform the program to static single assignment (SSA) [Cytron et al., 1991; Rosen et al., 1988] form, a representation where each local variable is immutable in that it is assigned exactly once in the program syntax. Converting to SSA form is inexpensive and can be performed in a modular fashion for each procedure individually. However, SSA cannot solve the problem of reassignment for global variables because (as their name indicates), globals do not belong to a single procedure and can be reassigned anywhere.

When variables are immutable, it is trivial for the analysis to determine the interval in which the value of a variable must be tracked precisely. Since the value of the variable is assigned once and never changes, the variable can be tracked precisely for the entire program without worrying about incurring superfluous expense from tracking irrelevant updates to the variable. If a variable is mutable, it can be updated multiple times, but a goal-directed analysis only needs to abstract the variable precisely in the interval between the last write to the variable before the query and the use of the variable in (or before) the query. Abstracting the variable precisely for a longer interval adds cost to the analysis without improving precision.

To see the problem presented by mutability, consider the program in Figure 2.2. The program in this example has two global variables, `global1` and `global2`. Precisely abstracting the value of `global1` is

```
 1  static int global1, global2;
 2
 3  void writeGlobal1() { global1 = 1; }
 4
 5  void foo() {
 6    global1 = complicated();
 7    global2 = complicated();
 8    writeGlobal1();
 9    assert (global1 == 1);
10    global1 = complicated();
11  }
```

Figure 2.2: An abstraction that tracks the relevant variable `global1` precisely only during the interval between the write to `global1` at line 3 and the query at line 9 can expend less effort in proving the query without sacrificing precision.

important for proving the safety of the query at line 9, but `global2` is irrelevant to the query. However, precisely tracking `global1` for the entire program is wasteful.The write to `global1` in the `writeGlobal1` is the last write that occurs before the query—the writes at lines 6 and 10 are irrelevant to the query. Line 6 is irrelevant because a different write always occurs before the read in the query, and line 10 is irrelevant because the write always occurs after the query. Tracking these irrelevant writes is potentially expensive because determining the value written requires analyzing the suggestively named `complicated` method.

Abstracting global variables: previous approaches. Most CEGAR-based analyses (e.g., [Ball et al., 2011; Clarke et al., 2000; Lal et al., 2012]) uses **eager** abstractions that remain constant for a single iteration of the main CEGAR loop. Though the abstraction changes on each iteration of the loop, techniques that use eager abstraction do not change the abstraction within the loop. This lack of flexibility leads to the problem described above: an CEGAR technique using eager abstraction will correctly determine the need to track `global1` and not `global2`, but it will do more work than necessary by tracking `global1` for the duration of the analysis rather than just the interval between the last write and the query.

The **lazy abstraction** technique introduced in the BLAST tool [Henzinger et al., 2002] offers a clever solution to this problem. With lazy abstraction, the counterexample analysis works backward from the point of failure (in this case, the query at line 9) until it finds a predicate that can block the path in the counterexample (in this case, the write to `global1` at line 3). The abstraction is then refined to track this predicate, but (crucially) is refined only for the interval where the predicate is determined to be important up to the end of the counterexample trace. For this example, lazy abstraction would correctly discover that the global `global1` only needs to be tracked precisely for a small part of the analysis and will thus avoiding doing extra work by precisely analyzing the irrelevant writes at lines 6 and 10.

Abstracting the heap: the problem. Precisely, yet soundly abstracting the heap is perhaps the most challenging problem in all of static program analysis. The problem is so challenging that that many industrial-strength tools that are highly sophisticated in other respects do not even attempt to tackle this problem. For example, Microsoft's Clousot [Fähndrich and Logozzo, 2010] verifier for C# CodeContracts resorts to unsound assumptions[1] about aliasing configurations to preserve precision and tractability, and

[1] Though it should be noted that these assumptions are often reasonable in practice [Christakis et al., 2015].

the pioneering Astrée [Blanchet et al., 2003] analyzer cannot handle C programs with dynamic memory allocation.

The heap adds complexity by stacking the **aliasing** problem on top of the previously described mutability problem. The core of the aliasing problem is that a write to a single heap cell can update storage referred to by multiple local variables. To demonstrate, the program in Figure 2.3 writes to the `f` field of `Obj` instances stored in two different local variable `a` and `b`. If these local variables are not aliased (e.g., if we replace the ... at line 3 with **new** `Obj()`), then the write at line 4 will update only the `Obj` instance stored in `b`. However, if these local variables are aliased (e.g., if we replace the ... at line 3 with `a`), then the write at line 4 will update the `Obj` instance referred to by both `a` and `b`.

The aliasing problem complicates goal-directed analysis by making it difficult to restrict reasoning to a small portion of the heap relevant to the query. Although the assertion at line 5 of Figure 2.3 only reads from `a.f`, this query cannot be proven safe by reasoning about `a.f` in isolation—the possibility that `a` and `b` may alias forces us to track writes to `b.f` as well. Note that if we replaced reads/writes of `a.f` with reads/writes of a local or global variable, this problem would disappear because a local/global variable can only be updated by writing to the variable of that name. Aliasing allows the programmer to give multiple names to the same storage, which challenges a sound analysis by forcing it to consider many possible associations between names and storage locations.

```
1 Obj a = ...
2 a.f = 1
3 Obj b = ...
4 b.f = 0
5 assert(a.f != 0)
```

Figure 2.3: The aliasing problem. If `a` and `b` are not aliased, the assertion at line 5 can be proven safe. However, if `a` and `b` may alias, then the assertion cannot be proven safe.

Abstracting the heap: previous approaches. Predicate abstraction-based CEGAR analyses like SLAM [Ball et al., 2011] and BLAST [Henzinger et al., 2002] have focused on analyzing C device driver programs that manipulate local and global variables in complex ways, but make minimal use of the heap.

These approaches resolve aliasing using coarse may-alias analyses and are known to struggle with scalability of object-oriented programs that use the heap heavily. The problem is that predicate abstraction is not well-suited for concisely expressing aliasing constraints: the analysis must learn and track explicit inequality predicates between each pair of local variables whose dis-aliasing is important for proving safety of the query. The number of inequality predicates that must learned can be exponential in the number of input pointer variables, hampering scalability (see [Beckman et al., 2008], Figure 4 for a detailed example).

Recent Datalog-based CEGAR approaches [Zhang et al., 2013, 2014] focus specifically on the problem of efficiently finding minimal abstractions just precise enough to prove the safety of a query. The state-of-the-art strategy of [Zhang et al., 2014] generalizes a single counterexample found using a particular abstraction to eliminate all abstractions that will fail for the same reason from consideration, then selects the cheapest remaining abstraction in the family to try next using MAXSAT. This approach handles aliasing much more efficiently than classical predicate abstraction-based approaches. The authors demonstrate that their technique enables a goal-directed variant of the groundbreaking typestate analysis of [Fink et al., 2008] that requires careful reasoning about aliasing for precision.

However, this more tractable handling of the heap comes at the cost of being less precise than classical approaches (as we will discuss in the next subsection): predicate abstraction-based approaches to CEGAR (e.g., [Ball et al., 2011; Henzinger et al., 2002; Lal et al., 2012]) are almost always path-sensitive, whereas Datalog-based approaches have not yet been applied to the problem of selecting minimal path-sensitive abstractions.

Summary. In summary, all CEGAR-based techniques implement effective flexible abstractions for local variables. Lazy abstraction [Henzinger et al., 2002] addresses the mutability problem introduced by abstracting global variables, but is not flexible enough to abstract the heap in a way that handles aliasing effectively. Newer Datalog-based approaches to CEGAR handle aliasing in a scalable fashion, but sacrifice path-sensitivity that older CEGAR techniques can provide. No existing goal-directed approach has the flexibility to both abstract the heap in a scalable fashion and maintain path-sensitivity.

2.1.2 Control-flow abstraction: selecting path-, flow-, and context-sensitivity policies

The control-flow of a program dictates what commands are executed and in what order. To avoid tying ourselves to any particular programming language, let us think of the control-flow of a program as being representing by a **control-flow** graph where nodes are commands and a directed edge from node c_1 to node c_2 means that command c_2 may execute after command c_2. The analysis task is to find abstract stores labeling each edge that overapproximate the concrete stores reachable at that program point. From the perspective of defining flexible abstractions specialized to a query, the question whether the analysis should allow a disjunction of abstract store at each program point and (correspondingly) how many disjuncts to allow. An ideal goal-directed analysis will allow as few stores as possible at each program point while still retaining sufficient precision to prove the safety of the query.

There are essentially three program constructs that might necessitate allowing multiple abstract stores at a given program point: procedure calls, command ordering, and conditionals. The commonly used static analysis terms for introducing additional abstract stores in response to each of these constructs (respectively) are context-sensitivity, flow-sensitivity, and path-sensitivity. We explain the precision benefits of each *-sensitivity and discuss how flexible existing goal-directed approaches are with respect to adding *-sensitivity for a particular query.

Context-sensitivity: the problem of abstracting procedure calls. Context-sensitivity refers to how the analysis abstracts the call stack of the concrete program under analysis. In concrete execution, the call stack allows the program to remember which call site that it entered the current procedure from so it can return to that same call site once it finishes executing the current procedure. In static analysis, we can make the analysis less expensive by abstracting the call stack less precisely. A fully context-insensitive analysis only needs to analyze each procedure once, whereas a fully context-sensitive analysis needs to analyze a procedure in each of its calling contexts.

Context-sensitivity comes in different varieties depending on how distinct calls to the same procedure are distinguished (that is, what the “contexts” are). The two most commonly used varieties are call site or call string-sensitivity [Sharir and Pnueli, 1981] (in which calls to the same procedure are distinguished if they occur at different syntactic call sites) and object or allocation site-sensitivity [Milanova et al., 2002, 2005a] (in which calls to the same procedure are distinguished if they have different receiver objects). We will focus primarily on call site-sensitivity in our discussion, but we refer the reader to two excellent surveys on pointer analysis [Smaragdakis and Balatsouras, 2015; Sridharan et al., 2013] for further discussion of allocation site-sensitivity.

To understand the importance of call site-sensitivity for precision, consider performing a flow-insensitive pointer analysis on the example in Figure 2.4. Proving the safety of the assertion at line 8 requires call-site sensitivity to distinguish the calls to `id` at lines 6 and 7. Without using call site-sensitivity, the analysis will imprecisely conclude that any call to `id` can return either allocation site A1 or allocation site A2, which means that the assertion at line 8 reduces to proving $\{$ A1, A2 $\} \neq \{$ A1, A2 $\}$, which cannot be done.

If the analysis creates separates contexts for each call to `id` (one for the call site at line 6 and one for the call site and line 7), then the analysis can conclude that only allocation site A1 flows to `a` and only allocation site A2 flows to `b`. Using this more precise abstraction forces the analysis to analyze `id` twice instead of just once, but it allows it to prove the assertion at line 8.

```
1  static Object id(Object o) { return o; }
2
3  void foo() {
4    Object x = new Object(); // allocation site A1
5    Object y = new Object(); // allocation site A2
6    Object a = id(x)
7    Object b = id(y)
8    assert(a != b)
9    Object c = new Object()
10   assert(a != c)
11 }
```

Figure 2.4: Call site-sensitivity on the calls to `id` is required to prove the assertion at line 10, but no call site-sensitivity is required to prove the assertion at line 8.

Context-sensitivity: previous approaches. For most analysis clients, a fully context-insensitive abstraction is too imprecise (as we saw in the example above), whereas full context-sensitivity is completely intractable. Because of this problem, significant research effort has focused on identifying limited forms of context-sensitivity that make a reasonable tradeoff between precision and scalability. The most common way to accomplish this is to parameterize the analysis by a constant k and perform k-context-sensitive[2] or k-object-sensitive analysis [Milanova et al., 2002, 2005a]. Here, the parameter k roughly corresponds to the number of concrete call stack frames (note that frames may be abstracted as call sites with call site-sensitivity, receiver objects with object-sensitivity, or even receiver types with type-sensitivity [Smaragdakis et al., 2011]) the analysis abstracts, or how "deep" the context-sensitivity goes.

Higher values for k tend to increase precision, but hurt scalability as the cost of the analysis can grow exponentially with k. In many cases, analysis becomes intractable for reasonably sized programs with $k \geq 2$. Conventional wisdom states that some call sites need more than 2-context-sensitivity for precise analysis, but many do not. Parameterized context-sensitivity (originally introduced by Milanova et al. [Milanova et al., 2002]) introduces a framework to take advantage of this understanding by using different k values at different call sites. This increase in flexibility provides a more fine-grained precision/scalability tradeoff than simply varying k. Liang et al. [Liang et al., 2011] demonstrate that this conventional wisdom does indeed hold (and as a corollary, that an analysis uses this flexibility cleverly can reap significant scalability benefits): their results indicate that applying a precise context-sensitivity policy at 0.4-2.3% of call sites gave equal precision to applying the policy uniformly across all call sites, but takes as little time as using k=0 at all call sites.

In a goal-directed setting, we can be more flexible still by only using precise context-sensitivity call sites that matter for a particular query. For the example in Figure 2.4, the assertion at line 8 requires call site-sensitivity on the calls to `id` (as we have already explained), but the assertion at line 10 does not require any context-sensitivity. Recent work in goal-directed analysis has focused on automatically recognizing such facts and refining the abstraction to add context-sensitivity exactly where it is required. The demand-driven points-to analysis of Sridharan et al. [Sridharan and Bodík, 2006] adds context-sensitivity useful for proving the safety of downcasts in Java programs. The previously mentioned Datalog-based CEGAR technique of Zhang et al. [Zhang et al., 2014] is quite effective at adding both call site-sensitivity and object-sensitivity

[2] This is frequently called k-CFA [Shivers, 1991] in the functional static analysis community. [Might et al., 2010] gives a very clear explanation of the connection between notions of context-sensitivity in the object-oriented and functional static analysis communities.

required to prove a flow-insensitive points-to query (such as the safety of a downcast). As we have explained, their approach converges on the cheapest abstraction in the chosen space of abstractions (in this case, the space of *k*-context- or *k*-object-sensitive abstractions for all *k*) precise enough to prove the query.

The approaches we have mentioned thus far refine flow-insensitive pointer analyses with the call site- or object-sensitivity required to prove a query. Impact pre-analysis [Oh et al., 2014] is unique in that it enhances a flow-sensitive abstract interpretation (in their case, an interval analysis for proving the absence of buffer overflows in C) with the call site-sensitivity required to prove a set of queries.

Flow/path-sensitivity: the problem of abstracting instruction ordering and conditionals. A flow-sensitive analysis visits instructions in the order dictated by the transition relation of the program. By contrast, a flow-insensitive analysis abstracts away all control flow and assumes that statements can execute any number of times and in any order. Flow-insensitive abstraction tends to be less expensive than flow-sensitivity because if the ordering of commands does not matter, a single abstract store can overapproximate the concrete state at all program points. A flow-sensitive abstraction needs a different abstract store at each program point in order to precisely represent the changes resulting from each instruction.

To see the difference in precision between a flow-insensitive abstraction and a flow-sensitive abstraction, consider the example in Figure 2.5. The assertion at line 4 can be proven safe using flow-insensitive reasoning because the fact `x > 0` is a flow-insensitive invariant that holds at **every** program point. Even though this abstraction cannot capture the fact that the assignment of the global variable `x` at line 5 occurs after the assertion at line 4, this imprecision does not stop the analysis from proving the assertion because the assignment preserves the key invariant `x > 0`.

On the other hand, the assertion at line 4 cannot be proven-safe with flow-insensitive reasoning. Though fact `y > 0` holds at the program point of the assertion, it does not hold at every program point (in particular, it does not hold after the assignment to `y` at line 6). Thus, flow-sensitive reasoning is required to prove the safety of this assertion.

Path-sensitive abstraction is a straightforward extension to flow-sensitive abstraction to add additional precision with respect to conditionals. In order to keep a single store at the program point following a conditional branch, a flow-sensitive and path-insensitive analysis must join the values from the true branch of

```
1 static int x = 3, y = 3;
2
3 assert (x > 0)
4 assert (y > 0)
5 x = 1
6 y = 0
```

Figure 2.5: An example demonstrating the differences between flow-insensitive and flow-sensitive abstraction. The assertion at 4 can be proven safe using flow-insensitive reasoning, whereas the assertion at line 4 requires flow-sensitivity to prove safe.

the conditional and the false branch of the conditional. This can lose precision in the case that remembering which branch of the conditional is associated with which abstract state is important. A path-sensitive abstraction can maintain this precision by keeping a disjunction of abstract stores to represent the results of each conditional branch.

For example, consider the program in Figure 2.6. In order to prove this program safe, the analysis must understand that on every program path where the assert command at line 4 is executed, the false branch of the first conditional at line 3 is also executed. If the analysis joins the paths from the then and else branch of the conditional, it will be uncertain whether the value of y is 2 or 3 at the assertion. A path-sensitive abstraction can keep two stores after the first conditional and understand that only the store where y holds the value 3 will enter the true branch of the second conditional and execute the assert command.

```
1 x = * // nondeterministic choice
2 if (x == 0) y = 2
3 else y = 3
4 if (x != 0) assert (y == 3)
```

Figure 2.6: An example demonstrating the need for path-sensitive abstraction. The assertion at line 4 can only be proven safe if the analysis understands the relationship between the values of x and y established by the conditional structure of the program.

A flow-sensitive analysis can easily become a more precise path-sensitive analysis by choosing to keep a case split after each conditional branch rather than performing a join. However, this additional precision comes at a significant cost—the number of case splits than the analysis must perform grows exponentially with the number of conditionals in the program. The analysis can try to curb this explosion by keeping case splits for some conditionals and performing joins for others based on heuristics (for example, see Section 6.2 of [Mauborgne and Rival, 2005]), but in general it is difficult to predict **a priori** which conditionals will be important for precision in a whole-program setting.

Path/flow-sensitivity: previous approaches In a goal-directed setting, we would ideally apply expensive flow- and path-sensitive abstraction only where the precision of these more expensive kinds of reasoning is required to prove the query. The predicate abstraction-based CEGAR approaches that we have already discussed [Ball et al., 2011; Henzinger et al., 2002; Lal et al., 2012] are quite effective in deciding where to apply path-sensitive reasoning. When the analysis decides to track a predicate that is used in a conditional expression, the analysis will keep a case split for that conditional but will join paths from conditional branches whose expressions are not tracked by the analysis. If the counterexample analysis is clever enough to identify a small number of relevance conditional predicates, then this is approach works well for adding valuable precision while maintaining tractability.

Early CEGAR tools [Ball et al., 2011; Clarke et al., 2000; Henzinger et al., 2002] handled procedure calls by inlining the body of each procedure call and thus were fully flow- and context-sensitive. This approach works well enough for small device driver programs like the ones considered by these tools, but does not scale well to larger programs with thousands of procedure calls. Modern CEGAR tools like CORRAL [Lal et al., 2012] deal with this problem by making the policy for inlining procedures a part of the abstraction. By default, CORRAL inlines no procedures and instead replaces each procedure call with a coarse flow-insensitive summaries describing the variables and heap cells that the procedure may modify. As part of abstraction refinement in response to counterexamples, CORRAL uses **stratified inlining** to select procedures to inline that may be relevant to the query and will not cost the analysis too much to analyze precisely. This allows the analysis to vary flow-sensitivity as well as path-sensitivity because all inlined procedures are treated flow-sensitively, all non-inlined procedures are treated flow-insensitively (using their summaries), and conditionals in the inlined procedures are treated path-sensitively if their conditional expressions involve predicates tracked by the analysis.

Though CORRAL's inlining strategies provide additional flexibility that increases scalability over previous CEGAR approaches, it still struggles to scale to large programs (the results from [Lal et al., 2012] only consider programs up to 2000 lines of code). One problem is that the varying flow-sensitivity of CORRAL is not flexible enough to handling deep call chains in a tractable way. For example, say that procedure `main` calls procedure `foo`, `foo` calls another procedure `bar`, and `bar` must be analyzed flow-sensitively to prove the query but `foo` need not be analyzed flow-sensitively. CORRAL's inlining-based approach will need to inline `foo` in order to have to choice to inline `bar`, so it cannot choose to analyze `bar` precisely without also analyzing `foo` precisely. In real-world programs with call chains hundreds of calls long, this forces lots of unnecessary (and expensive) flow-sensitive analysis to perform flow-sensitive analysis of a relevant callee deep in the chain, hampering scalability. Recent work on checking **deep assertions** [Lal and Qadeer, 2014] outfits CORRAL with a program transformation that adds extra control-flow edges from the program entrypoint to each method containing an assertion that is to be posed as a query to the analysis. Though this makes the inlining approach slightly more flexible and improves the results of CORRAL, it does not solve all of the problems with inlining. For example, the transformation will not help in the case that two "sibling" procedures (that is, procedures without a caller/callee relationship) in the separate deep call chain both need to be analyzed precisely.

Finally, FISSILE type analysis [Coughlin and Chang, 2014] presents an intriguing approach to varying coarse flow-insensitive analysis and precise path/flow-sensitive analysis. The idea of FISSILE is to check **almost-everywhere** invariants that hold at almost every program point by performing flow-insensitive type checking by default, switching to a precise flow/path-sensitive separation logic-based analysis when a flow-insensitive type invariant is violated, then switching back to type checking once the precise analysis proves that the invariant has been restored. Though FISSILE is not a goal-directed analysis in the sense it only tries to resolve specific queries (like most of the analyses we have discussed thus far), it is goal-directed in the sense that the switch to a more precise analysis is prompted by the violation of a type invariant. When a type invariant is violated, FISSILE starts a precise flow-sensitive analysis at the beginning of the basic block or procedure where the type invariant was violated.

Like CORRAL, the FISSILE approach to control-flow abstraction is not fully flexible. FISSILE only performs intraprocedural path/flow-sensitive analysis and thus must summarize callees using their types.

In addition, invariants beyond the global type invariant cannot be maintained across multiple sessions of path/flow-sensitive analysis. If a fact not captured in the type invariant is established in one code region and used in another, FISSILE will not be able to preserve the fact and will lose precision while analyzing the second region.

Summary. Most work on flexible control-flow abstraction has focused on varying the context-sensitivity policy used at different call sites (e.g., [Oh et al., 2014; Sridharan and Bodík, 2006; Zhang et al., 2014]). With the exception of the work of Oh et al. [Zhang et al., 2014], these works focus on refining flow-insensitive analyses by adding selective context-sensitivity. However, the work of Oh et al. uses a fixed flow/path-sensitivity policy and thus is inflexible along this dimension (they mention selective flow-sensitivity as a topic for future work (Section 9)).

Classical predicate abstraction-based approaches to CEGAR (e.g. [Ball et al., 2011; Clarke et al., 2000; Henzinger et al., 2002]) are effective at varying path-sensitivity to meet client needs, but not flow-sensitivity. The exceptions are CORRAL [Lal and Qadeer, 2014; Lal et al., 2012] and FISSILE [Coughlin and Chang, 2014], which can vary flow-sensitivity by using inlining and interleaved symbolic analysis/type-checking (respectively), but neither approach can vary its flow-sensitivity in a fine-grained or fully general way. CORRAL's ability to vary call site-sensitivity suffers from the same weaknesses as its ability to vary flow-sensitivity, as it is also implemented via inlining.

No existing approach can vary both context-sensitivity and flow/path-sensitivity in a fine-grained (e.g., command-level rather than procedure-level) fashion.

2.2 Granularity: when abstractions can change

The previous section (Section 2.1) focused on how abstractions can vary in order to be flexible. In this section, we discuss another aspect of flexibility: **when** or **how often** the abstraction can change. We refer to an abstraction that can be changed frequently as fine-grained an abstraction that can be changed infrequently as coarse-grained. Fine-grained abstractions are more flexible (and thus more desirable from our point of view) because they give the analysis more leeway to specialize to the needs of a query.

Most previous works implement a coarse-grained form of flexible abstraction by expanding a single-pass, single abstraction analysis to a **staged** analysis with a different abstraction at each stage. Rather than fixing an abstraction and running the analysis a single time, a staged analysis runs the analysis repeatedly and uses a more precise abstraction in each pass. The analysis decides how to refine the abstraction based on the failure of the analysis to prove queries in the previous pass. In what follows, we will explain how specific analyses have utilized staging and discuss the strengths and weaknesses of each approach.

Staged analyses with fixed abstractions. The most basic kind of staging uses a fixed number of stages with a fixed abstraction at each stage (e.g., [Bodden et al., 2007; Fink et al., 2008; Sinha and Wang, 2010]). The idea is to use cheaper, less precise abstractions in earlier stages to prove the safety of "easy" queries. Queries that cannot be proven safe by early stages are passed on to progressively more precise and more expensive stages. In some cases, the last phase of the analysis emits runtime checks and the earlier static analysis stages serve to prune the number of runtime checks that must be emitted (e.g., [Chugh et al., 2009; Guarnieri, 2010]).

Though staged analysis with fixed abstractions is more flexible than a fixed single-stage analysis, the approach leaves much to be desired in terms of flexibility. The abstraction used in each stage is fixed and thus cannot change to meet the needs of a particular query.

Staged analyses with abstraction refinement. An improvement on basic staged analysis is to refine the abstraction used at stage $n+1$ based on the failure of the abstraction to prove queries at stage n (as opposed to fixing the abstraction used at each stage). Successful goal-directed analysis strategies such as CEGAR, [Ball et al., 2011; Clarke et al., 2000; Henzinger et al., 2002; Lal et al., 2012; Zhang et al., 2014], impact pre-analysis [Oh et al., 2014], and introspective analysis [Smaragdakis et al., 2014a] can be viewed in this way. CEGAR is a staged analysis in which the number of stages and the way the abstraction is refined at each stage is driven by a **counterexample** consisting of the failure to prove a query in the previous stage. Both impact pre-analysis and introspective analysis perform two-stage analyses whose first stage is an imprecise analysis designed to assess where context-sensitivity will be important for precisions [Oh et al., 2014] or will be intractably expensive [Smaragdakis et al., 2014a]. This pre-analysis is used to select a context-sensitivity policy to be used in abstraction of the more precise second phase.

The problem with staged abstraction refinement are that the abstraction only changes between stages, so it is not flexible enough to exploit opportunities where changing the abstraction **during** a stage leads to a better tradeoff. Lazy abstraction [Henzinger et al., 2002] comes the closest to addressing this problem by refining its abstraction for a restricted interval of the analysis (rather than the whole thing), but is not a general solution. Lazy abstraction works well for simple goal-directed store abstraction (locals and globals), but has not been shown to work well for the heap. Lazy abstraction is effective for goal-directed control-flow abstraction for path-sensitivity in the same way as all CEGAR techniques, but cannot vary its flow- or context-sensitivity.

Chapter 3

Overview: flexible abstraction via goal-directed coarsening

In this chapter, we present an overview of goal-directed abstraction coarsening and demonstrate how the flexibility of this approach allows us to create precise and scalable analyses. We first explain our technique and its philosophy at a high level (Section 3.1). We then present a challenging real-world example (Section 3.2) and show how it can be effectively verified using goal-directed coarsening (Sections 3.2.2 and 3.2.3). In the process of explaining the example, we highlight three key contributions of this dissertation:

- A framework for goal-directed control-flow abstraction via **jumping**. The framework is parametric in (a) a sound store abstraction and (b) a **relevance relation** that determines when the analysis performs jumps. We will explain the theory behind this framework in Chapter 5, but in this chapter we focus on showcasing an effective instantiation of the framework in action.

- A **mixed symbolic-explicit** store abstraction that precisely and efficiently represents heap dependencies by combining backward separation logic analysis with points-to analysis (as formalized in Chapter 4). The example in this chapter instantiates (a) with this store abstraction and demonstrates why such a store abstraction is needed to analyze object-oriented programming languages that make heavy use of the mutable heap.

- A strategy for using the flexibility of jumping wisely in order to enable tractable analysis of event-driven Android programs (to be fully described in Chapter 7). This chapter instantiates (b) with this strategy and shows that jumping using this strategy significantly increases the scalability of the analysis without losing precision.

Portions of Sections 3.2.1 and 3.2.3 previously appeared in a paper draft currently under submission entitled "Selective Control-Flow Abstraction via Jumping", which was co-authored by Bor-Yuh Evan Chang and Manu Sridharan.

3.1 Philosophy of goal-directed coarsening

We claim building an analysis that is goal-directed by virtue of coarsening is a very natural approach because:

Proposition 3.1.1 Overapproximate coarsening is a fundamental part of any abstract interpretation.

Sound overapproximation is the cornerstone of static analysis via abstract interpretation and is thus very well understood. In order to overapproximate, all static analyses must coarsen (intentionally lose precision) in order to handle constructs like loops and recursion. Abstract interpretation-based analyses coarsen by using join and widening operators [Cousot and Cousot, 1977]. These operators are coarsenings that are extremely important because they preserve the soundness and termination of the analysis. Our goal-directed coarsening approach expands the usage of coarsening beyond the typical joining at conditions and widening at loop heads to give the analysis the flexibility to intentionally lose precision that is (a) not likely to prove useful in resolving a particular query and/or (b) likely to lead to scalability issues.

The basic idea of our approach is that given a query (Q, ℓ) representing an overapproximation of some undesirable concrete program stores at a program point ℓ, our analysis works backward from the query maintaining an abstract state that is a **necessary** [Cousot et al., 2013] precondition for the bad concrete stores to occur at ℓ. If our analysis can prove that (false, ℓ_i) is a necessary precondition for Q at a set of program points ℓ_i that (together) control-dominate the initial program point ℓ, then no concrete trace can end in a state in the concretization of Q (and thus the bad concrete states are never reachable at runtime). Thus, the goal of our analysis is to **refute** the query (prove its safety) by showing that false is a necessary precondition at each ℓ_i in the manner described above. This analysis is a form of proof by contradiction: it computes an over-approximation of the backward reachable states from the query (Q, ℓ) and refutes the query when it has proven the unreachability of (Q, ℓ). At any time during backward propagation of the query, the analysis can coarsen the abstraction by **weakening** the current query, as we will explain (and justify) shortly.

3.1.1 The importance of necessary preconditions

The use of necessary preconditions rather than **sufficient** preconditions is crucial for enabling our coarsening approach. Other backward analyses for proving the safety of a query or procedure (e.g., [Barnett and Leino, 2005; Blackshear and Lahiri, 2013; Dijkstra, 1976; Flanagan and Saxe, 2001; Flanagan et al., 2002a]) commonly work by computing the **weakest sufficient precondition** [Dijkstra, 1976] for the safety of query, a superficially similar, but fundamentally different process. Given a query representing an overapproximation of bad concrete program states, these approaches typically negate the query (yielding an abstract state whose concretization contains **good** concrete states) and propagate a sufficient precondition for reaching the good states. Thus, these approaches compute a **sufficient** precondition for the **unreachability** of the bad concrete states, whereas our approach computes a **necessary** precondition for the **reachability** of the bad concrete states.

To intuitively see the difference between sufficient and necessary preconditions, consider the incomplete Hoare triple [Hoare, 1969] "{ **?** } `x := nondet()` { $x = 5$ }", where `nondet()` represents a nondeterministic choice, and **?** represents a precondition to be filled in. If we want **?** to be a sufficient precondition, our only choice is false because no concrete state will definitely transition to a state in the concretization of $x = 5$ by executing the command `x := nondet()` (because the nondeterminism may choose any value for `x`). On the other hand, if we want **?** to be a necessary precondition, we can only choose true. The reason is that any concrete state can possibly transition to a state in the concretization of $x = 5$ by executing the command `x := nondet()` (again, because the nondeterminism may choose any value for `x`).

As a final illustration of the difference between sufficient and necessary preconditions, let us consider the statements of soundness for each approach. Let σ represent concrete states, $\hat{\sigma}$ represent abstract states, and γ represent a concretization operator mapping an abstract state to a set of concrete states. Assume that we have a judgment $\langle \sigma, c \rangle \Downarrow \sigma'$ for concrete execution of a command c and abstract predicate transformers (that is, functions that take an (command, abstract state) pair as input and produce an abstract state as output) $sp(c, \hat{\sigma})$ and $np(c, \hat{\sigma})$ for computing the sufficient and necessary (respectively) preconditions for an abstract state $\hat{\sigma}$ with respect to the command c. Think of the $\hat{\sigma'}$ as the initial query for the analysis. Now, the soundness conditions for these abstract transformers can be stated as:

"If $\langle \sigma, c \rangle \Downarrow \sigma'$ and $\sigma \in \gamma(sp(c, \hat{\sigma'}))$, then $\sigma' \in \gamma(\hat{\sigma'})$". (Sufficient precondition soundness)

"If $\langle \sigma, c \rangle \Downarrow \sigma'$ and $\sigma' \in \gamma(\hat{\sigma'})$, then $\sigma \in \gamma(np(c, \hat{\sigma'}))$". (Necessary precondition soundness)

Though both predicate transformers work backward from the original query $\hat{\sigma'}$, the soundness condition satisfied by the sufficient precondition transformer is a "forward" guarantee about the concrete post-state σ', whereas the soundness condition satisfied by necessary preconditions is a backward guarantee about the concrete pre-state σ. Note that when $np(c, \hat{\sigma'}) = \mathsf{false}$ (that is, false is a necessary precondition for the initial query), we can conclude that every concrete state in $\gamma(\hat{\sigma'})$ is unreachable because $\gamma(\mathsf{false}) = \emptyset$. As explained in the beginning of Section 3.1, this fact is the logical basis for our approach to goal-directed analysis.

The advantage of using necessary preconditions rather than sufficient preconditions is that overapproximate coarsening preserves the necessity of a necessary failure condition (that is, it is always sound to overapproximate a necessary condition), whereas underapproximation preserves the sufficiency of a sufficient safety condition. It is easy to see this by consulting the soundness conditions above: logically weakening $np(c, \hat{\sigma'})$ via overapproximation makes it easier to satisfy the soundness condition for np, whereas logically strengthening $sp(c, \hat{\sigma'})$ via underapproximation makes it easier to satisfy the soundness condition for sp (we also refer the reader to Section 7 of [Logozzo et al., 2014] for further discussion of this issue). Proposition (3.1.1) says that overapproximating necessary conditions is easy because coarsening is a part of every analysis, whereas underapproximation is an intriguing but less well-understood topic. For weakest precondition-based analyses, the problem of underapproximation is so challenging that most analyzers handle loops by bounded unrolling of the loop or by requiring manually annotated loop invariants. By contrast, our technique can use standard overapproximation techniques to implement coarsening. Logically **weakening** a necessary precondition (e.g., by dropping conjuncts ϕ_i of a CNF formula $\phi_1 \wedge \phi_2 \wedge \mathsf{true}$ or dropping sub-heaps M_i of a separation logic formula $M_0 * M_1 * \mathsf{true}$) always produces another necessary precondition and thus provides a very natural basis for coarsening.

3.1.2 Coarsening versus refinement

Most approaches to goal-directed analysis are based on abstraction **refinement** rather than coarsening. These approaches typically cannot change their abstractions **on-the-fly** during analysis; as previously explained in Section 2.2, they are typically staged analyses that can only change abstractions between stages (and a stage is usually a full run of the analysis on the entire program).

Clearly, changing the abstraction on-the-fly is a desirable goal because it allows the analysis to adjust the abstraction to the needs of the query in a more fine-grained way. So why do refinement-based approaches choose not to do it? There is obviously no definitive answer to this question, but we believe that the core reason is that it is difficult to justify the soundness of making an abstraction more precise during analysis (rather than between stages). Approaches that do switch to a more precise abstraction during analysis (e.g., [Coughlin and Chang, 2014; Khoo et al., 2010]) require intricate and abstraction-specific "handoff" proofs that show how soundness preserved when switching to a more precise abstraction. Since these proofs must be done manually for each abstraction and are difficult to generalize, we posit that it is frequently easier to start an entirely new analysis with the refined abstraction, as CEGAR and other staged abstraction refinement approaches do. However, doing so sacrifices the benefits that changing the abstraction on-the-fly can bring.

By contrast, justifying the soundness of changing to a **less** precise abstraction during analysis is a fundamental part of every analysis, as stated in Proposition 3.1.1. Changing an abstraction on-the-fly via coarsening is obviously sound and is simple to do with machinery already required for analysis. In fact, the coarsening and on-the-fly aspects of our approach naturally complement one another. It would be foolish to mirror the structure of staged abstraction refinement by using a precise abstraction for an entire stage, then coarsening the abstraction after it "failed to scale"—the first stage would either already be precise enough to prove the query safe (if it can be proven), or the first stage would never finish! The ability to make the decision to coarsen on-the-fly is necessary in order for our approach to be practical in the first place.

3.1.3 Why coarsen when we don't need to?

As we claimed in our discussion of Proposition 3.1.1, the undecidability of program analysis in general forces analyses to coarsen in the presence of constructs like loops and recursion in order to ensure termination and maintain soundness. Coarsening is typically seen as a necessary evil of analysis that forces the analysis into unwanted false alarms. Losing precision when the analysis is not strictly required to thus seems like a bad idea.

For analyses that used fixed abstractions, this view is quite reasonable. The analysis designer typically picks an abstraction just precise enough to reason about the property of interest, so losing precision for any non-essential reason risks negating the effectiveness of the analysis tool. However, our philosophy in goal-directed coarsening is to start off with an abstraction that is as precise as possible—one that we expect is more than precise enough to refute most queries of interest. Since we work backward from the query (rather than forward from the beginning of the program to the query), it is often the case that we find the information we need to refute the query by exploring just a few instructions backward from the query or a few callers backward from the start procedure. In such cases, we do not need to coarsen at all because we do not even have to analyze enough code to reveal the expensive nature of our precise abstraction. However, in the cases that we cannot refute the query quickly, we view coarsening as a useful (indeed, essential) tool that allows us to trade off precision for scalability[1] .

As a concrete example of the kind of precision/scalability tradeoffs that we wish to make with coarsening, consider the simple Java example in Figure 3.1. The goal here is to prove the safety of the assertion at line 4. Observe that there are many different reasons why this assertion might be safe (for conciseness, let $p(n)$ represent the property "n.`tag = 1` $\Longrightarrow$ n.`value` **instanceof** `T`" for some Node instance n):

(1) The assertion is safe if the first element of list `lst` satisfies $p(n)$.

(2) The assertion is safe if `lst` is always null at line 2 and throws a NullPointerException that prevents the assertion from being reached.

[1] Note that coarsening can cause the analysis to be both less precise and less scalable in general if too much precision is lost, but our focus here is on carefully choosing coarsenings that allow us to make the desired precision-for-scalability tradeoff.

```
1 void foo(List lst) {
2   Node x = lst.getFirst();
3   if (x.tag == 1) {
4     assert (x.val instanceof T);
5   }
6 }
```

Figure 3.1: A simple example that can be proven safe using many different strategies.

(3) The assertion is safe if `x` is always null at line 3 and throws a NullPointerException that prevents the assertion from being reached.

(4) The assertion is safe if no instance of class Node has a `tag` field that holds the value 1.

(5) The assertion is safe if every element n in list `lst` satisfies $p(n)$.

(6) The assertion is safe if every instance n of class Node satisfies $p(n)$.

Each of these reasons represents a strategy that an analysis could take in attempting to prove the query safe. A fully precise analysis would need to account for each possibility or else risk failing to prove the query safe.

Though an analysis would ideally be able to quickly explore any and all proof strategies, in reality we may be forced to abandon some strategies in order to scale. Our approach is designed to deal with these harsh realities: we choose a maximally precise initial abstraction that enables as many proof strategies as possible, but also specify when the analysis should abandon strategies by coarsening the abstraction in a way that rules out the strategy. This allows us to leverage the fact that in practice, some proof strategies are much more likely to be (a) **expensive** in that they incur scalability problems or (b) **impractical** in that they fail to prove the query safe.

For example, strategy (1) falls into category (a) because it requires a very precise abstraction capable of reasoning about individual elements in a container of unbounded size (e.g., abstractions used in shape analysis [Chang and Rival, 2008; Sagiv et al., 2002]). Such abstractions are typically quite expensive and can lead to scalability problems. Strategies like (2)–(3) fall into category (b) because real programmers do not frequently try to prevent a crash by guarding it with another crash that is certain to occur! In addition, this kind of proof strategy is likely to be expensive (i.e., fall into category (a) as well) because every dereference that occurs before the query can potentially block the reachability of the query by throwing an exception. Strategy 4 also falls into category (b) because if the fact that this strategy attempts to prove were true, the true branch of the conditional (line 4) would be dead code.

In contrast, strategies (5)–(6) are much more reasonable from both an expense and practicality perspective. Strategy (5) requires reasoning about universal **container invariants**, which is much less expensive than reasoning about individual elements of a container (for example, generics in languages like C++ and Java allow programmers to express type invariants satisfied by each element of a container). Strategy 6 is reasonable because real programmers frequently use **class invariants** [Parkinson] or **object invariants** [Chang and Leino, 2005; Leino and Müller, 2004] that establish relationships like $p(n)$: "n.`tag = 1` $\Longrightarrow$ n.`value` **instanceof** `T`". In Chapter 6, we will present a goal-directed analysis that focuses on using strategies like (5)–(6) to prove the safety of queries while ignoring other expensive and impractical strategies. Chapter 7 presents a different analysis that limits the set of strategies it pursues based on the structure of event-driven programs, and we will demonstrate this strategy in the example of Section 3.2.

Reasoning about the viability of proof strategies in this way is clearly somewhat domain- and query-specific, but there are general principles in play here: strategies that real-world programmers use to reason about safety are more likely to be useful, and simpler explanations for safety are frequently both less expensive to reason about and more likely to be correct (i.e., Occam's razor).

We can rule out bad strategies by coarsening the abstraction in a number of ways. For example, we can rule out strategies like (1) by handling container writes using weak updates, and we can rule out strategies (2)–(3) by dropping constraints on nullness of references (or by not adding such constraints in the first place). Coarsening the abstraction in this way yields less abstract state for the analysis to track precisely and can significantly increase scalability.

The need to coarsen to rule out bad strategies and isolate good ones reinforces the importance of the two challenges of goal-directed analysis defined in Section 1.1: Challenge A (flexibility), and Challenge B (practicality). If our abstraction is not flexible enough, we may not be able to coarsen in a way that rules out a bad strategy without also compromising a good strategy. On the other hand, even if our analysis is maximally flexible in theory, we still need to be able to make good coarsening choices in order to rule out bad strategies and achieve reasonable precision and scalability in practice.

3.2 Example: verifying dereference safety in event-driven Android applications

In this section, we present an example demonstrating the utility of goal-directed coarsening for verifying the absence of null dereferences in event-driven Android programs. We begin by motivating the difficulty and importance of verifying dereference safety in Android (Section 3.2.1). We then walk through the process of our analysis in trying to prove the safety of the dereferences at lines 7, 8, and 9 in Figure 3.2. Only the dereferences at lines 7 and 9 can be proven safe; the dereference at line 8 is buggy (inspired by a real-life bug[2] that we found in the ConnectBot[3] app). Proving the safety of the query at line 7 shows how our mixed-symbolic explicit store abstraction can perform precise backward reasoning about the heap without performing excessive case splits due to aliasing (Section 3.2.2). Proving the safety of the query at line 9 demonstrates that control-flow abstraction via jumping allows us to soundly analyze event-driven systems without reasoning about an intractable number of event orderings (Section 3.2.3).

3.2.1 Motivation: verifying dereference safety in Android

Null dereference errors (a Java `NullPointerException`, or NPE) are a major cause of failures in Android applications. In a search of the commit logs of the ten open-source Android apps that we will later analyze in Section 7.5, we found 738 distinct commits containing the string “NPE” or “null,” roughly 3% of all commits. Further, a recent paper on Facebook’s INFER static analyzer reported that their internal database of production Android app crashes contained many null dereference errors [Calcagno et al., 2015]. Such errors cause crashes that stop the app and degrade user experience. Unlike crashes in web application code that can be fixed on the backend and pushed to the user when the page is refreshed, an app crash cannot be fixed until (1) an update fixing the bug is released and approved by the app store, and (2) the user elects

[2] `https://github.com/connectbot/connectbot/pull/60`
[3] `https://github.com/connectbot/connectbot`

```
     class HostActivity extends onClickListener {
       // in method HostActivity.<init>
1      ManagerService mService = null;
2      HostDatabase mHostDb = null;

       MenuItem mLastItem;
       MenuItem mSettingsItem;

       void onCreate() {
         ServiceConnection cxn =
3          new ServiceConnection() {
             void onConnected(Service s) {
4              mService = (ManagerService) s;
             }
         }
         bindService(..., cxn);
5        findViewById(...).setOnClickListener(this);
6        mHostDb = new Database();
       }

       void onCreateOptionsMenu(Menu menu) {
         Icon last = Cache.getLastIcon();
         if (last != null) {
           mLastItem.icon = last;
         } else {
           mLastItem.icon = new DefaultIcon();
         }
         Icon settingsIcon = menu.settingsIcon;
         mSettingsItem = complicated();
         mSettingsItem.icon = settingsItem;
         Icon lastIcon = mLastItem.icon;
7        lastIcon.setColor(...) // possible NPE?
       }

       void onClick(View v) {
8        Host host = mService.getHost(); // possible NPE?
9        mHostDb.saveHost(host); // possible NPE?
       }

       void onDestroy() {
10       mHostDb = null;
11       mService = null;
       }

     }
```

Figure 3.2: A simple Android app with two components (HostActivity and ServiceConnection) whose lifecycles are shown in Figure 3.3. The programmer uses correct local reasoning to show that the dereference at line 7 is safe and correct reasoning about the lifecycles to ensure that the dereference at line 9 is safe, but mistaken assumptions about the lifecycle make the dereference at line 8 unsafe.

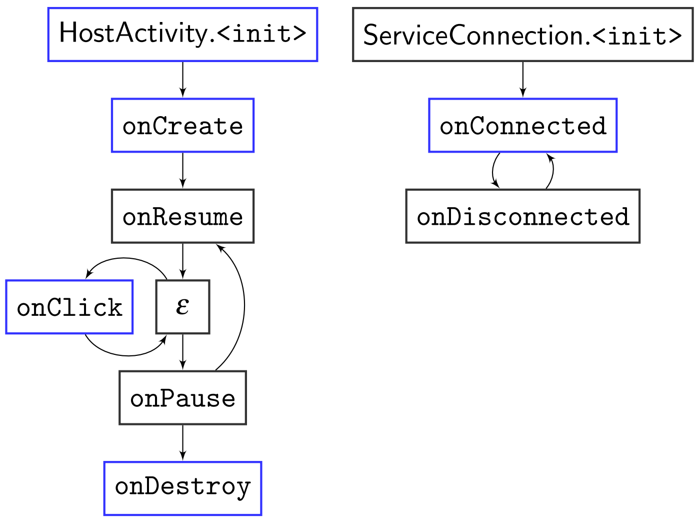


Figure 3.3: Lifecycle graphs for the HostActivity and ServiceConnection classes of Figure 3.2.

to download the update, a process that can take weeks or even months [Calcagno et al., 2015]. Below, we discuss how subtleties of the Android app lifecycle can lead to null deference errors.

Bugs, safety, and lifecycles One reason that null dereferences in Android apps are easy to create and difficult to reason about is the complicated Android lifecycle. Though most events within the lifecycle of a single component are ordered, the lifecycles of different components can interleave arbitrarily and cause unexpected behavior. As a concrete example, Figure 3.3 shows the lifecycle graphs of the HostActivity and ServiceConnection classes from Figure 3.2. A directed edge from event e_1 to e_2 means that e_1 must execute before e_2 is allowed to execute. The ε event represents a special "skip" event to soundly model the fact that the user might not trigger user interaction events such as `onClick`. The edges between `onClick` and ε model the fact that the user may click it an arbitrary number of times.

The HostActivity and ServiceConnection components have independent lifecycles, but (as we can see from the code in Figure 3.2) they share the `mService` object. This leads to a null dereference at line 8 in the case that the `onClick` event fires before the `onConnected` event (since `mService` will still be null). This bug is due to faulty reasoning about the event-driven lifecycle of Android—the developer does not account for all possible interleavings between the HostActivity and ServiceConnection lifecycles.

On the other hand, the dereference at line 9 is safe because of ordering constraints in the HostActivity lifecycle. The developer delays initializing the `mHostDb` field to the expensive Database object until line 6 of the `onCreate` callback to avoid incurring the memory footprint of this object until it is needed. In addition, the developer assigns null to `mHostDb` at line 10 of the `onDestroy` event in order to relieve memory pressure as soon as possible (the enclosing HostActivity object may not become unreachable for some time after this event). These optimizations are safe because the lifecycle for HostActivity dictates not only that the `onCreate` event always executes after the constructor and before the `onClick` event, but also that the `onDestroy` event can only execute after all invocations of the `onClick` event.

Finding lifecycle sensitive bugs via testing is difficult given that (a) real apps have hundreds or thousands of events, (b) the developer must find the right combination of events that lead to a bug, and (c) exercising the app in a way that triggers the right events in the proper order is a tedious process. Thus, an effective static approach to this problem has the potential to significantly improve the state of affairs for

Android app developers.

Challenges of analyzing event-driven Android programs Though numerous static approaches to proving the absence of null dereferences have been proposed (e.g., [Dillig et al., 2008; Loginov et al., 2008; Margoor and Komondoor, 2015; Nanda and Sinha, 2009]), the key challenge in analyzing our motivating example does not concern the client of null dereferences specifically. Even a type-based approach with programmer-written nullness annotations would likely not work well. In Android, the nullness or non-nullness of a reference is frequently not a flow-insensitive invariant that holds at every program point or even an almost-everywhere invariant [Coughlin and Chang, 2014] that holds at nearly every program point. Instead, non-nullness (along with many other properties) holds during some phases of the lifecycle.

Thus, the challenge for analyses is reason precisely, but tractably at both the intra-event and the inter-event level. Cost-effective intra-event reasoning is difficult for all the usual reasons for static analysis—each event is essentially a program that may call thousands of procedures, perform extensive heap manipulation, and contain loops/recursion. The challenge of inter-event reasoning is to perform precise reasoning about event orderings within a lifecycle without incurring the cost of reasoning about **all** event orderings (since this cost is exponential in the number of lifecycle components, as we explain further in Section 7.1).

In what follows, we use the example in Figure 3.2 to demonstrate how goal-directed coarsening meets the intra-event challenge using a mixed symbolic-explicit store abstraction (Section 3.2.2) and the inter-event challenge using control-flow abstraction via jumping (Section 3.2.3). This example has been simplified to contain only the events and instructions relevant to the three queries, but a real app would have many other lifecycle components whose methods and event orderings the analysis might need to consider. In essence, the power of coarsening is that it allows the analysis to soundly reduce a complex real-world program into a simple program containing only the relevant instructions and events like the one in Figure 3.2.

3.2.2 Taming aliasing path explosion with mixed symbolic-explicit store abstraction

We now demonstrate how our mixed symbolic-explicit store abstraction enables precise and efficient analysis of heap-manipulating programs in proving the safety of the query at line 7 of Figure 3.2. We reproduce the code for the `onCreateOptionsMenu` method containing line 7 in Figure 3.4 in order to show the results from the analysis at each program point. In addition, we give a partial points-to graph for the heap objects manipulated by this method in Figure 3.5. Nodes in this graph are either program variables (like **this**) or syntactic **allocation sites** (like $\texttt{act}_1$). We assume that each allocation site in the program is annotated with a unique allocation site name (such as $\mathsf{defaultIcon}$ for the allocation at line 9 of Figure 3.4). A directed edge from a node n_1 to node n_2 in the graph means that an address in the concretization of n_1 may point to an address in the concretization of n_2 at some point during concrete execution.

As we explained in Section 3.1, our coarsening-based analysis takes an initial query Q that is a necessary precondition [Cousot et al., 2013] for the bug to occur and attempts to prove safety (**refute** the query) by propagating the query backward from its initial program point ℓ in an attempt to derive a contradiction. To prove the safety of the dereference at line 22 in Figure 3.4, the initial query is $\mathsf{lastIcon} \mapsto \mathsf{null}$ at line 21 (the program point immediately before the dereference). This query expresses a necessary condition for the dereference at line 22 to fail: the query says that in order for the dereference to fail, the program variable `lastIcon` must hold the value null at line 21.

The analysis begins by propagating the initial query backward across the assignment to `lastIcon` at line 20. This produces a new query that remains a necessary precondition for the null dereference: the analysis reasons that for a null dereference to occur, the `mLastItem` field must point to some object whose `icon` field holds the value null[4] .

This reasoning is expressed by the query at line 19. In the query, we use hatted variables such as $\widehat{this}$ and $\widehat{last}$ to denote existentially qualified **symbolic variables** whose concretization is a single concrete object instance[5] . Each symbolic variable is associated with an instance-from constraint specifying the set of syntactic **allocation sites** that the instance may have been allocated from[6] . For example, the constraint $\widehat{last}$ from $\mathrm{pt}(\mathsf{this}{\cdot}\mathsf{mLastItem})$ says that the symbolic variable must have been allocated from an allocation site in the points-to set of $\mathsf{this}{\cdot}\mathsf{mLastItem}$. It is these instance-from constraints that make our store representation **mixed** symbolic-explicit—they connect symbolic variables involved in separation logic points-to constraints to the explicit allocation sites in the points-to graph. As we will see shortly, instance-from constraints are

4 We interpret any memory M as $M \ast \mathsf{true}$ in separation logic so that a query always describes the entire heap.

5 We write constraints such as $\mathsf{lastIcon} \mapsto \mathsf{null}$ as shorthand for $\mathsf{lastIcon} \mapsto \widehat{v} \wedge \widehat{v} = \mathsf{null}$ (for some $\widehat{v}$).

6 For conciseness, we omit from constraints on some symbolic variables in Figure 3.4 (e.g., $\widehat{this}$).

`void onCreateOptionsMenu(Menu menu) {`

1 `Icon last = Cache.getLastIcon();`

2 $\boxed{\mathsf{this} \mapsto \widehat{this} * \widehat{this}\cdot\mathsf{mLastItem} \mapsto \widehat{last} * \mathsf{last} \mapsto \hat{v} \wedge \underline{\hat{v} = \mathsf{null}} \wedge \underline{\hat{v} \neq \mathsf{null}} \wedge \widehat{last}\ \mathsf{from}\ \mathsf{pt}(\mathsf{this}\cdot\mathsf{mLastItem})}$ $\dagger$

3 `if (last != null) {`

4 $\boxed{\mathsf{this} \mapsto \widehat{this} * \widehat{this}\cdot\mathsf{mLastItem} \mapsto \widehat{last} * \mathsf{last} \mapsto \mathsf{null} \wedge \widehat{last}\ \mathsf{from}\ \mathsf{pt}(\mathsf{this}\cdot\mathsf{mLastItem})}$

5 `mLastItem.icon = last;`

6 $\boxed{\mathsf{this} \mapsto \widehat{this} * \widehat{this} * \widehat{act}\cdot\mathsf{mLastItem} \mapsto \widehat{last} * \widehat{last}\cdot\mathsf{icon} \mapsto \mathsf{null} \wedge \widehat{last}\ \mathsf{from}\ \mathsf{pt}(\mathsf{this}\cdot\mathsf{mLastItem})}$ —

7 `} else {`

8 $\boxed{\mathsf{this} \mapsto \widehat{this} * \widehat{this}\cdot\mathsf{mLastItem} \mapsto \widehat{last} * \widehat{last}\cdot\mathsf{icon} \mapsto \hat{v} \wedge \underline{\hat{v} = \mathsf{null}} \wedge \underline{\hat{v} \neq \mathsf{null}} \wedge \widehat{last}\ \mathsf{from}\ \mathsf{pt}(\mathsf{this}\cdot\mathsf{mLastItem})}$ $\dagger$

9 `mLastItem.icon =` **new**$_{\mathsf{defaultIcon}}$ `DefaultIcon();`

10 $\boxed{\mathsf{this} \mapsto \widehat{this} * \widehat{this}\cdot\mathsf{mLastItem} \mapsto \widehat{last} * \widehat{last}\cdot\mathsf{icon} \mapsto \mathsf{null} \wedge \widehat{last}\ \mathsf{from}\ \mathsf{pt}(\mathsf{this}\cdot\mathsf{mLastItem})}$ —

11 `}`

12 $\boxed{\mathsf{this} \mapsto \widehat{this} * \widehat{this}\cdot\mathsf{mLastItem} \mapsto \widehat{last} * \widehat{last}\cdot\mathsf{icon} \mapsto \mathsf{null} \wedge \widehat{last}\ \mathsf{from}\ \mathsf{pt}(\mathsf{this}\cdot\mathsf{mLastItem})}$

13 `Icon settingsIcon = menu.settingsIcon;`

14 `mSettingsItem = complicated();`

15 $\boxed{\mathsf{this} \mapsto \widehat{this} * \widehat{this}\cdot\mathsf{mLastItem} \mapsto \widehat{last} * \widehat{last}\cdot\mathsf{icon} \mapsto \mathsf{null} \wedge \widehat{last}\ \mathsf{from}\ \mathsf{pt}(\mathsf{this}\cdot\mathsf{mLastItem})}$

16 $\boxed{\begin{array}{l}\mathsf{this} \mapsto \widehat{this} * \widehat{this}\cdot\mathsf{mLastItem} \mapsto \widehat{last} * \widehat{this}\cdot\mathsf{mSettingsItem} \mapsto \widehat{item} * \widehat{last}\cdot\mathsf{icon} \mapsto \mathsf{null} \wedge \\ \quad \widehat{last}\ \mathsf{from}\ \mathsf{pt}(\mathsf{this}\cdot\mathsf{mLastItem}) \wedge \widehat{item}\ \mathsf{from}\ \mathsf{pt}(\mathsf{this}\cdot\mathsf{mSettingsItem}) \wedge \widehat{last} \neq \widehat{item}\end{array}}$ —

17 $\boxed{\begin{array}{r}\mathsf{this} \mapsto \widehat{this} * \widehat{this}\cdot\mathsf{mLastItem} \mapsto \widehat{last} * \widehat{this}\cdot\mathsf{mSettingsItem} \mapsto \widehat{last} * \widehat{last}\cdot\mathsf{icon} \mapsto \mathsf{null} \wedge \\ \widehat{last}\ \mathsf{from}\ \underline{(\mathsf{pt}(\mathsf{this}\cdot\mathsf{mLastItem}) \cap \mathsf{pt}(\mathsf{this}\cdot\mathsf{mSettingsItem}))}\end{array}}$ $\dagger$ —

18 `mSettingsItem.icon = settingsIcon;`

19 $\boxed{\mathsf{this} \mapsto \widehat{this} * \widehat{this}\cdot\mathsf{mLastItem} \mapsto \widehat{last} * \widehat{last}\cdot\mathsf{icon} \mapsto \mathsf{null} \wedge \widehat{last}\ \mathsf{from}\ \mathsf{pt}(\mathsf{this}\cdot\mathsf{mLastItem})}$

20 `Icon lastIcon = mLastItem.icon;`

21 $\boxed{\mathsf{lastIcon} \mapsto \mathsf{null}}$

22 `lastIcon.setColor(...)` *// possible NPE?*

23 `}`

Figure 3.4: Proving the safety of the query at line 22 with mixed symbolic-explicit store abstraction.

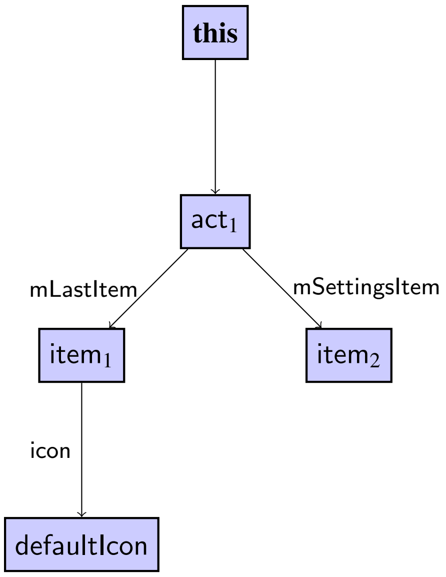


Figure 3.5: A partial points-to graph for Figure 3.4.

the key feature of our store abstraction that allow us to prevent unnecessary aliasing case splits.

Handling an aliasing case split with instance-from **constraints** Returning to the example, the analysis proceeds by pushing the query at line 19 across the write at line 18. At this point, the analysis must perform a case split[7] to account for the possibility that `mLastItem` and `mSettingsItem` may or may not be aliased. In general, any backward symbolic analysis will be forced to fork an aliasing case split at each field write relevant to the current query. This causes a case explosion that is worst-case exponential in the number of relevant field writes. This case explosion is independent of (but compounded by) the well-known scalability problems caused by conditional branching in a path-sensitive analysis. In order to scale, our store abstraction must be able to aggressively prune these aliasing case splits whenever possible. Our solution to this problem is to use instance-from constraints to combine global, up-front points-to information with local information in the current query.

The case split at line 17 represents the case where `mLastItem` and `mSettingsItem` are aliased. This is reflected in the query by the constraints $\widehat{this}\cdot\mathsf{mLastItem} \mapsto \widehat{last}$ and $\widehat{this}\cdot\mathsf{mSettingsItem} \mapsto \widehat{last}$; that is, the two fields hold the same symbolic variable. However, the aliasing fact is also represented in the instance-from constraints. The analysis determines that if `mLastItem` and `mSettingsItem` are aliased, then the $\widehat{last}$ symbolic variable must have been allocated from an allocation site in the **intersection** of the points-to sets of the two fields. This is represented by the instance-from constraint $\widehat{last}$ from $(\mathsf{pt}(\mathsf{this}\cdot\mathsf{mLastItem}) \cap \mathsf{pt}(\mathsf{this}\cdot\mathsf{mSettingsItem}))$. The analysis can use the points-to graph in Figure 3.5 to determine which allocation sites are in each points-to set and actually compute this intersection. Consulting this graph, the analysis determines that $\mathsf{pt}(\mathsf{this}\cdot\mathsf{mLastItem}) = \{\ \mathsf{item}_1\ \}$ and $\mathsf{pt}(\mathsf{this}\cdot\mathsf{mSettingsItem}) = \{\ \mathsf{item}_2\ \}$, so $\mathsf{pt}(\mathsf{this}\cdot\mathsf{mLastItem}) \cap \mathsf{pt}(\mathsf{this}\cdot\mathsf{mSettingsItem}) = \emptyset$. The analysis has thus shown that the symbolic variable $\widehat{last}$ could not have been allocated from any allocation site in the program, and so the query is infeasible. The analysis then refutes the case at line 17 (represented by †) and continues analysis of the other cases.

We note that in this case, the result is the same as using an up-front may-alias analysis as an oracle to prove that `mLastItem` and `mSettingsItem` are not aliased, as many previous tools have done (e.g., [Ball et al., 2011; Henzinger et al., 2002; Manevich et al., 2004]). However, we generalize this kind

[7] We indicate case splits by writing a __ next to each query box.

of aliasing check by **incrementally** restricting the from set for each symbolic variable as we observe it flowing into and out of local variables and heap locations. If it were not the case that $(\mathsf{pt}(\mathsf{this}{\cdot}\mathsf{mLastItem}) \cap \mathsf{pt}(\mathsf{this}{\cdot}\mathsf{mSettingsItem})) = \emptyset$, we would still maintain the more constrained from set for $\widehat{last}$ and would continue restricting the set as analysis continues. Approaches based on may-alias oracles cannot do this: they can prune case splits if the alias analysis is precise enough to prove that there is no aliasing, but they cannot leverage the precision of the symbolic analysis and the points-to analysis together as we can with from constraints. These constraints provide additional precision at a negligible cost.

Using coarsening to maintain a small store abstraction Having refuted the case at line 17, the analysis considers the case split at line 16 representing the case where `mLastItem` and `mSettingsItem` are not aliased. This query reflects the dis-aliasing fact through an explicit inequality constraint between $\widehat{last}$ and $\widehat{item}$. However, in order to represent this fact, the analysis needs the additional points-to constraint $\widehat{this}{\cdot}\mathsf{mSettingsItem} \mapsto \widehat{item}$. The presence of this points-to constraint means that the analysis must track all writes to `mSettingsItem` (as well as any other fields that it may be aliased with).

Though keeping this disaliasing constraint is necessary for full precision (e.g., if the analysis later discovers that the current program point is guarded by an `if`-guard checking `mSettingsItem == mLastItem`), it can also cause scalability issues by forcing the analysis to track more state. This is exactly the kind of conundrum that we have previously described in Section 3.1.3. Here, the analysis decides that the disaliasing fact is not likely to lead to a refutation and thus chooses to coarsen by weakening the query at line 16 to the one at line 15 (i.e, dropping the constraints $\widehat{this}{\cdot}\mathsf{mSettingsItem} \mapsto \widehat{item}$, $\widehat{item}$ from $\mathsf{pt}(\mathsf{this}{\cdot}\mathsf{mSettingsItem})$, and $\widehat{last} \neq \widehat{item}$). Note that this weakened query at line 15 is identical to the the query at line 19—the coarsening makes it as if the analysis never considered an aliasing case split at all.

After coarsening, the instructions at lines 14 and 13 are not relevant to the query[8] . The analysis can simply push the query backward to line 12 without any changes. Note that if the analysis had not chosen to coarsen, it would have needed to analyze the call to `complicated` because it would still be tracking the value of the `mSettings` field. Though analyzing this method could possibly have led to a refutation (for

[8] Here, we assume that a mod analysis computed alongside the points-to analysis is able to show that no instruction in the call to `complicated()` writes to fields in the query.

example, if the body of `complicated` simply returned `mLastItem`), the analysis saves effort by skipping analysis of this method and will still find a refutation even with the weakened query (as we will see).

Refuting the query with path-sensitive reasoning In order to move the query backward from line 12, the analysis must perform a case split to enter both branches of the conditional beginning at line 3. From the else branch case at line 10, the analysis must move backward across the assignment to `mLastItem.icon` at line 9. However, this assignment produces a refutation because the query requires that the value held by `mLastItem.icon` is null, but the assignment writes a non-null DefaultIcon object into the field. The field cannot simultaneously hold both a null and a non-null value, so the analysis refutes the case at line 8.

The analysis continues on the remaining then branch case at line 6. This branch also writes to `mLastItem.icon`, but the write at line 5 assigns the field to the `last` local variable. The analysis processes this assignment by updating the query as shown on line 4. Notice that the constraint $\widehat{last}\cdot\mathsf{icon} \mapsto \mathsf{null}$ has been removed. This is also a form of coarsening—tracking the heap cell is no longer relevant to the query, so the analysis drops the constraint in order to maintain a small store abstraction. Unlike the previously described coarsening, this coarsening is lossless: it cannot cause the analysis to lose precision. If the `icon` field becomes relevant to the query (e.g., due to a guard involving `mLastItem.icon`, the analysis will begin tracking the heap cell again. However, it avoids wasted effort by not tracking the cell when it is not relevant. This is different than other separation logic-based static analysis approaches (e.g., using bi-abduction [Calcagno et al., 2009, 2011]), which never stop tracking a heap cell once they begin tracking it.

Finally, the analysis pushes the query at line 4 across the guard at line 3. In doing so, it conjoins the guard condition `last != `**null** to the query to reflect the fact that the then branch can only be entered if the guard condition evaluates to true. However, this produces a refutation because the query requires that the `last` local variable holds a null value. As before, a variable or heap cell cannot hold both a null value and a non-null value simultaneously, so the analysis refutes the query. Since this was the last case that the analysis needed to consider, it has refuted the query along all paths and thus shown that the dereference at line 22 is safe.

Finally, we note that our store abstraction is effective for representing a small portion of the program store and ignoring commands that are not relevant to the abstracted portion. Our example does not emphasize this aspect of our abstraction because we have minimized the example so that almost every command is relevant. In a more realistic example when numerous irrelevant instructions are interspersed with the relevant instructions in Figure 3.4, our store abstraction would still only need to perform the reasoning shown.

3.2.3 Tractable inter-event analysis with jumping

Though the store abstraction we presented in Section 3.2.2 is quite effective for intra-event reasoning (including interprocedural intra-event reasoning), it does not solve the problem of of tractable inter-event reasoning. The problem (as we explained in Section 3.2.1) is how to soundly and precisely account for lifecycle event orderings without paying the cost of considering all possible event orderings. This is a problem of **control-flow abstraction** that cannot be solved by any approach to store abstraction, no matter how effective. In what follows, we show how we use jumping to address this problem by demonstrating how a jumping analysis can efficiently prove the safety of the dereference at line 9.

Anatomy of a jumping analysis We consider extending an analysis that uses the mixed symbolic-explicit store abstraction described in Section 3.2.2 with the ability to jump, but jumping can just as easily be combined with a store abstraction from backward abstract interpretation (e.g., [Bourdoncle, 1993b; Cousot, 1981; Rival, 2005]) At the intra-event level, the analysis behaves as shown in Section 3.2.2. When the backward analysis reaches an event boundary (that is, the entry block of an application method that is invoked by the Android framework), the analysis chooses to compute a set of relevant events to jump to rather than continuing to follow backward control flow into the complex Android framework code.

From the entry block of the current event e_{cur}, the analysis executes a jump by performing the following steps:

(1) **Identify important commands using data-relevance** For each constraint in the query, the analysis computes the set of **data-relevant** program commands whose concrete execution may produce a configuration in the concretization of the constraint. This process is similar to computing a partial slice that only considers immediately relevant commands (see Section 7.6 for a full discussion of

the differences between our technique and slicing). Finding the set of relevant commands makes use of a global view of the program from a points-to graph computed by an up-front analysis.

(2) **Associate relevant commands to events** The analysis walks backward in the program's call graph from the calling method of each relevant command and stops each time it hits an event boundary. This yields the set of events that may lead to the execution of relevant commands.

(3) **Order relevant events using control-feasibility information.** Though it would be sound to jump to all relevant events or even to jump directly to each relevant command, doing so loses information about the ordering of relevant events/commands, which is bad for analysis precision. In order to be precise, the analysis must account for the fact that only certain events are **control-feasible** with respect to the current event e_{cur} according to the Android lifecycle.

    Our analysis computes control-feasibility information using the lifecycle documentation for Android components that specifies the order in which lifecycle events may occur. The analysis computes a **specialized** lifecycle graph for the declaring class of e_{cur} that explicates ordering constraints between e_{cur} and the other relevant events (see Section 7.2 for more details). It then uses this graph to rule out relevant events that cannot occur **between** the current method e_{cur} and some other relevant event.

(4) **Jump to each control-feasible event.** The analysis forks a case split for each event that is both data-relevant and control-feasible, jumps to the exit block of the event, and continues backward analysis for each case.

Using jumping to refute a tough query We now demonstrate how our jumping analysis uses the process described above to prove the safety of the dereference at line 9 and identify the bug at line 8 in Figure 3.2. As in Section 3.2.2, the analysis proves safety by trying to refute a necessary precondition for the bug to occur. For the dereference at line 9 to fail, the initial query is $\widehat{act} \cdot \mathsf{mHostDb} \mapsto \mathsf{null}$ at program point 9. The diagram in Figure 3.6 visualizes the process of proving the safety of this dereference by refuting the query. The analysis first uses the mixed symbolic-explicit transfer functions described in Section 3.2.2 to propagate this precondition backward to the beginning of the `onClick` event, yielding the necessary bug

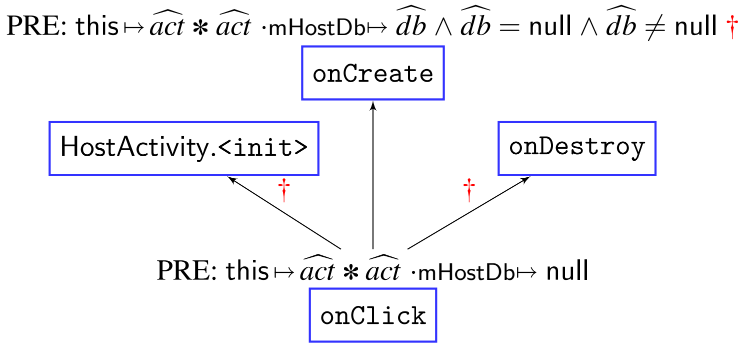

PRE: this $\mapsto \widehat{act} * \widehat{act}$ ·mHostDb $\mapsto \widehat{db} \wedge \widehat{db} =$ null $\wedge \widehat{db} \neq$ null †
onCreate
HostActivity.<init>
onDestroy
†
†
PRE: this $\mapsto \widehat{act} * \widehat{act}$ ·mHostDb $\mapsto$ null
onClick

Figure 3.6: Proving safety of the dereference at line 9 of Figure 6.1 using jumping analysis.

precondition this $\mapsto \widehat{act} * \widehat{act}$ ·mHostDb $\mapsto$ null shown in the figure.

At this point, the analysis chooses to perform a jump because it has reached an event boundary. The analysis decides which events to visit next using the four steps outlined above: first, it computes the data-relevant commands that may change the current bug precondition. This step yields the commands at lines 2, 6, and 10 of Figure 3.2. Second, it uses the call graph to associate these relevant commands with their calling events, which yields the set of events HostActivity.<init>, onCreate, and onDestroy.

Third, the analysis uses the lifecycle graph for HostActivity in Figure 3.3 to perform control-feasibility filtering. The analysis determines that onDestroy is not control-feasible with respect to the current event onClick because onDestroy is not backward-reachable from onClick in the lifecycle graph for HostActivity. The analysis also determines that HostActivity.<init> is not control-feasible because it is **postdominated** by the relevant event onCreate—every feasible concrete execution reaching onClick visits onCreate between HostActivity.<init> and onClick.

Thus, the analysis concludes that it only needs to jump to onCreate. Figure 3.6 represents the decision not to jump to the relevant events HostActivity.<init> and onDestroy by marking the edges to these events with †'s. The directed edge from onClick to onCreate indicates that the analysis performs a jump from the entry block of onClick to the exit block of onCreate with the precondition shown for onClick as the abstract state.

When the analysis encounters the assignment at line 6 of the onCreate event, it refutes the query because there is an inconsistency between this command and the current abstract state: the points-to con-

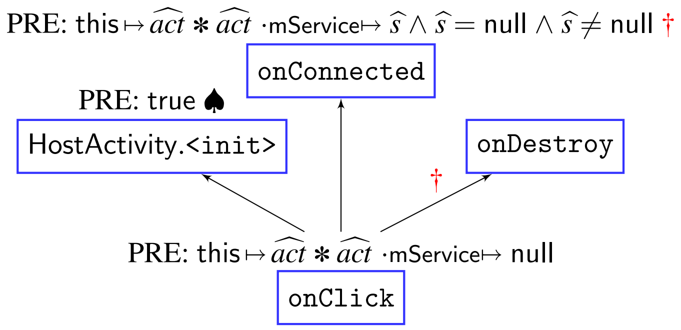

PRE: this $\mapsto \widehat{act} * \widehat{act} \cdot$mService$\mapsto \widehat{s} \wedge \widehat{s} =$ null $\wedge \widehat{s} \neq$ null †
onConnected
PRE: true ♠
HostActivity.<init>
onDestroy
†
PRE: this $\mapsto \widehat{act} * \widehat{act} \cdot$mService$\mapsto$ null
onClick

Figure 3.7: Failed safety proof for buggy dereference at line 8 of Figure 6.1 using jumping analysis.

straint $\widehat{act}$ ·mHostDb$\mapsto$ null says that the `mHostDb` field must hold the value null, but the command assigns a non-null Database value to this field. The analysis has therefore shown the safety of the dereference at line 9.

Identifying the bug Figure 3.7 shows how the same analysis process (correctly) fails to prove the safety of the dereference at line 8, revealing a true bug. The analysis determines that the dereference is safe if the `onConnected` event executes before `onClick` and that the relevant event `onDestroy` is not control-feasible with respect to `onClick`, so it marks these paths as refuted (†). However, in the case that HostActivity.`<init>` is the last relevant event to fire before `onClick`, the command `mService = `**null** at line 1 discharges the precondition for `onClick`, leaving a necessary bug precondition of true. The analysis cannot hope to find a refutation given this precondition, so it gives up and reports the dereference at line 8 as a possible bug (as indicated by the ♠ symbol).

We note that an analysis that (unsoundly) does not consider the interleaving of events of different lifecycle components would not find this bug. In order to see this bug, the analysis must account for the fact that the HostActivity and ServiceConnection lifecycles can interleave.

After the analyzer identified the real-life version of this bug, we submitted two pull requests fixing the bug along with another similar bug[9] to the developers of ConnectBot. The developers accepted both of our pull requests less than a week after they were submitted.

[9] `https://github.com/connectbot/connectbot/pull/61`

Chapter 4

Mixed symbolic-explicit store abstraction

In this chaper, we present a effective goal-directed store abstraction to form the backbone of the backward analysis strategy described in Section 3.1. We first state the requirements for an ideal ideal goal-directed store abstraction, present the challenges of meeting these requirements, and explain our proposed solution (Section 4.1). We then llustrate the mechanics of our goal-directed store abstraction by showing how it can refute **heap reachability** queries to precisely detect a nasty class of Android memory leaks (Section 4.2). Section 4.3 formalizes our store abstraction and explains how we connect separation logic to points to analysis with instance-from constraints (Section 4.3.2) and handle loops via **on-the-fly loop invariant inference** (Section 4.3.4). Finally, Section 4.4 presents a case study demonstrating that our store abstraction is effective in precisely detecting memory leaks in real-world Android applications.

Portions of this chapter previously appeared in the PLDI 2013 paper "Thresher: Precise Refutations for Heap Reachability" [Blackshear et al., 2013], which was co-authored by Bor-Yuh Evan Chang and Manu Sridharan.

4.1 Requirements for effective goal-directed store abstraction

We want our abstraction to be goal-directed by virtue of coarsening the abstraction along the two dimensions defined in Section 2.1.1: locals/globals and heap cells. We also desire the capability to coarsen our abstraction **on-the-fly** as described in Section 2.2. Finally, we want our store abstraction to be **precise** (capable of strong updates on locals, globals, and the heap) and **concise** (capable of describing a small portion of the store relevant to the query without being forced to track irrelevant details).

4.1.1 Meeting the requirements by combining separation logic and points-to analysis

As explained in the “Abstracting the heap” subsections of Section 2.1.1, the primary challenge in constructing a store abstraction with these requirements is effectively handling the mutable heap. Specifically, reasoning precisely about strong updates in the presence of aliasing is difficult to achieve in a tractable fashion.

We can mitigate some of the difficulties of precise heap reasoning by choosing a representation of the program store that is based on **separation logic** [O’Hearn et al., 2001; Reynolds, 2002]. However, building our store abstraction using separation logic does not meet all of our desired requirements on its own. Complications arise because the analysis that uses our store abstraction works **backward** from an initial query (as explained in Section 3.1). There are signficant advantages to working backward: for example, we can focus only on the state relevant to the query (as we will demonstrate when we explain the example in Figure 4.1), and we can examine code closest to the query first (since programmers frequently ensure safety within a small scope of local reasoning [Coughlin et al., 2012]) rather than working forward from the beginning of the program. Unfortunately, backward analysis and separation logic are difficult to mix because a purely backward analysis (initially) does not have any information about pointer aliasing and may be forced to perform case splits to maintain separation.

For example, consider the problem of proving the safety of the query at line 9 of the simple program in Figure 4.1. A backward separation logic-based analysis can avoid tracking the irrelevant variable `z` and consequently, performing a case split based on the conditional at line 4. However, a backward analysis that lacks information about pointer aliasing would be forced to perform a case split at line 8 to soundly account for the fact that `x` and `y` may be aliased. This case split would not be pruned until the assignment to `y` at line 2. The number of case splits that a backward analysis must perform is exponential in the number of writes through a variable that may-alias with a variable in the query, which can cause significant scalability problems if the case splits cannot be pruned quickly. This **aliasing case split explosion** problem is one of the reasons why existing separation logic-based analyses (e.g., [Berdine et al., 2005; Calcagno et al., 2011]) typically work forward rather than backward.

```
1 x = new T()
2 y = new T()
3 z = new V()
4 if (e) z.g = 1
5 else z.g = 2
6 ...
7 x.f = 1
8 y.f = 2
9 assert (x.f != 2)
```

Figure 4.1: An example comparing and contrasting the capabilities of forward and backward separation logic-based analysis.

By contrast, a forward separation logic-based analysis would establish separation between the values pointed to by x and y after processing the allocations at lines 1 and 2. This fact would allow the analysis to understand that the writes at lines 7 and 8 write to different objects and thus could avoid doing a case split. However, a forward separation logic-based approach is not very suitable for goal-directed analysis. The analysis would also need to track the irrelevant local variable z (because it cannot know in advance what parts of the store may be relevant to the query), which would force it to perform a case split for both branches of the conditional statement at line 4. As explained, the backward analysis would not need to perform this case split.

How can we avoid performing aliasing case splits like the backward analysis, yet avoid tracking irrelevant variables such as z (they may lead to other irrelevant case splits) like the forward analysis? Our solution is to combine a separation logic representation with facts from a flow-insensitive points-to analysis. Flow-insensitive points-to analysis (e.g., [Andersen, 1994; Steensgaard, 1996]) is excellent for performing inexpensive global reasoning about aliasing, but is not very precise (no flow-sensitivity and thus no strong updates). We combine the two analysis paradigms using special **instance-**from **constraints** that connect symbolic variables from the separation logic representation to allocation sites in the points-to representation. This gives us the best of both worlds: the backward separation logic-based representation allows us to precisely reason about only the dependencies of the query, whereas integrating points-to facts frequently allows us to avoid aliasing case splits.

4.2 Example: goal-directed store abstraction for precise Android memory leak detection

Here we present a detailed example to illustrate our goal-directed store abstraction. The example is based on real code from Android applications and libraries.

4.2.1 Motivation: refuting heap reachability queries

Static reasoning about nearly any non-trivial property of modern programs requires effective analysis of heap properties. In particular, a heap analysis can be used to reason about **heap reachability**—whether one heap object is reachable from another via pointer dereferences at some program point. Precise heap reachability information improves heap-intensive static analyses, such as escape analysis, taint analysis, and cast checking. A heap reachability checker would also enable a developer to write statically checkable assertions about, for example, object lifetimes, encapsulation of fields, or immutability of objects.

Our interest in heap-reachability analysis arose while developing a tool for detecting an important class of memory leaks in Android applications. Every Android application has least one associated Activity object to control the user interface. All Activity objects go through an operational lifecycle; they are created, run, and are eventually destroyed. Android development guidelines state that application code should not maintain long-lived pointers to Activity objects, as such pointers prevent the objects from being garbage collected at the end of their lifetimes, causing significant memory leaks (we discuss this issue further in Section 4.4). To detect such leaks in practice, it is sufficient to verify that Activity objects are **never** reachable from a static field via object pointers.

For this client, we found that highly precise reasoning about heap reachability, including flow-, context-, **and** path-sensitivity **with** materialization [Sagiv et al., 1998], was required to avoid emitting too many spurious warnings. We are unaware of an existing analysis that can provide such precision for heap-reachability queries while scaling to our target applications (40K SLOC with up to 1.1M SLOC of libraries). Our goal-directed store abstraction allows us to focus the precision and effort of the analysis on the heap reachability query at hand without encountering scalability issues.

4.2.2 Refuting a false leak alarm reported by points-to analysis

Figure 4.2 is a simple application that illustrates the difficulties of precisely checking this heap reachability property (ignore the boxed assertions for now). The `Main` class initializes and starts the application's Activity, the `Act` class. The `Vec` class captures the essence of a list data structure, as implemented in Android. This example is free of the leak described above, as the `Act` object allocated on line 2 is never made reachable from a static field.

For this example program, flow-insensitive points-to analysis techniques cannot prove the desired heap (un)reachability property due to the manner in which the `Vec` class is implemented. In the inset, we show a heap graph obtained by applying Andersen's analysis [Andersen, 1994] with one level of object sensitivity [Milanova et al., 2005b] to the example. Graph nodes represent classes or abstract locations (whose names are shown at the corresponding allocation site in Figure 4.2), and edges represent possible values of field pointers. Object-sensitive abstract locations are named appropriately, for example, $\mathsf{vec}_0.\mathsf{arr}_1$ for arr_1 instances allocated when `Vec.push(−)` is invoked on instances of vec_0. Each edge indicates a may points-to relationship, written as $a_1 \cdot f \mapsto a_2$, meaning there may be an execution where field f of abstract location a_1 contains the address of location a_2. The graph imprecisely shows that Activity object act_0 is reachable from both static fields $\mathsf{Act} \cdot \mathsf{objs}$ and $\mathsf{Vec} \cdot \mathsf{EMPTY}$, hence falsely indicating that a leak is possible.

The root cause of the imprecision in Figure 4.3 is the edge $\mathsf{arr}_0 \cdot \mathsf{contents} \mapsto \mathsf{act}_0$, which indicates that the array assigned to $\mathsf{Vec} \cdot \mathsf{EMPTY}$ may contain the Activity object. `Vec` is implemented using the **null object pattern** [Woolf, 1997]: rather than allocating a separate internal array for each empty `Vec`, all `Vec` objects initially use $\mathsf{Vec} \cdot \mathsf{EMPTY}$ as their internal array. The code in `Vec` is carefully written to avoid ever adding objects to $\mathsf{Vec} \cdot \mathsf{EMPTY}$ while also avoiding additional branches to check for the empty case.

But, a flow-insensitive points-to analysis is incapable of reasoning precisely about this code, and hence it models the statement **this**`.tbl[`**this**`.sz] = val` on line 23 as possibly writing to $\mathsf{Vec} \cdot \mathsf{EMPTY}$, polluting the points-to graph. Real Android collections are implemented in this way.[1]

Note that a more precise heap abstraction would not help in this case—because the (concrete) null

[1] In fact, we discovered buggy logic in the actual Android libraries that allowed writing to a null object, thereby polluting all empty containers! The bug was acknowledged and fixed by Google (`https://code.google.com/p/android/issues/detail?id=48055`).

```
   public class Main {
     public static void main(String[] args) {
2      Act a = new_act0 Act(); a.onCreate();
3    }
   }

   public class Act extends Activity {
     private static final Vec objs = new_vec0 Vec();
     public void onCreate() {
4      Vec acts = new_vec1 Vec();
```

5 $\boxed{\mathsf{this} \mapsto \widehat{this} * \mathsf{acts} \mapsto \widehat{acts} * \widehat{acts}\cdot\mathsf{tbl} \mapsto \mathsf{arr}_0 * \widehat{this} \mapsto \mathsf{act}_0 * \widehat{acts}\cdot\mathsf{sz} \mapsto \hat{v}^{\mathsf{int}}_{\mathsf{sz}} * \widehat{acts}\cdot\mathsf{cap} \mapsto \hat{v}^{\mathsf{int}}_{\mathsf{cap}} \wedge \hat{v}^{\mathsf{int}}_{\mathsf{sz}} < \hat{v}^{\mathsf{int}}_{\mathsf{cap}}}$

```
       acts.push(this
```

6 $\boxed{\widehat{this}\cdot\mathsf{sz} \mapsto \hat{v}^{\mathsf{int}}_{\mathsf{sz}} * \widehat{this}\cdot\mathsf{cap} \mapsto \hat{v}^{\mathsf{int}}_{\mathsf{cap}} * \mathsf{this} \mapsto \mathsf{vec}_1 * \mathsf{vec}_1\cdot\mathsf{tbl} \mapsto \mathsf{arr}_0 * \underline{\mathsf{val} \mapsto \mathsf{act}_0} \wedge \hat{v}^{\mathsf{int}}_{\mathsf{sz}} < \hat{v}^{\mathsf{int}}_{\mathsf{cap}}}$

```
       );
7      ...
8      objs.push("hello"
```

9 $\boxed{\mathsf{this} \mapsto \mathsf{vec}_0 * \mathsf{vec}_0\cdot\mathsf{tbl} \mapsto \mathsf{arr}_0 * \underline{\mathsf{val} \mapsto \mathsf{act}_0} * \widehat{this}\cdot\mathsf{sz} \mapsto \hat{v}^{\mathsf{int}}_{\mathsf{sz}} * \widehat{this}\cdot\mathsf{cap} \mapsto \hat{v}^{\mathsf{int}}_{\mathsf{cap}} \wedge \hat{v}^{\mathsf{int}}_{\mathsf{sz}} < \hat{v}^{\mathsf{int}}_{\mathsf{cap}}}$ †

```
       );
10   }
   }

   public class Vec {
     private static final Object[] EMPTY = new_arr0 Object[1];
     private int sz; private int cap; private Object[] tbl;

     public Vec() {
11     this.sz = 0;
12     this.cap = -1;
13     this.tbl = EMPTY;
```

14 $\boxed{\mathsf{this} \mapsto \widehat{this} * \widehat{this}\cdot\mathsf{tbl} \mapsto \mathsf{arr}_0 * \widehat{this}\cdot\mathsf{sz} \mapsto \hat{v}^{\mathsf{int}}_{\mathsf{sz}} * \widehat{this}\cdot\mathsf{cap} \mapsto \hat{v}^{\mathsf{int}}_{\mathsf{cap}} \wedge \hat{v}^{\mathsf{int}}_{\mathsf{sz}} < \hat{v}^{\mathsf{int}}_{\mathsf{cap}}}$ †

```
     }

     public void push(Object val) {
15     Object[] oldtbl = this.tbl;
```

16 $\boxed{\mathsf{this} \mapsto \widehat{this} * \widehat{this}\cdot\mathsf{sz} \mapsto \hat{v}^{\mathsf{int}}_{\mathsf{sz}} * \widehat{this}\cdot\mathsf{cap} \mapsto \hat{v}^{\mathsf{int}}_{\mathsf{cap}} * \widehat{this}\cdot\mathsf{tbl} \mapsto \mathsf{arr}_0 * \mathsf{val} \mapsto \mathsf{act}_0 \wedge \hat{v}^{\mathsf{int}}_{\mathsf{sz}} < \hat{v}^{\mathsf{int}}_{\mathsf{cap}}}$

```
       if (this.sz >= this.cap) {
17       this.cap = this.tbl.length * 2;
18       this.tbl = new_arr1 Object[this.cap];
```

19 $\boxed{\mathsf{this} \mapsto \widehat{this} * \widehat{this}\cdot\mathsf{tbl} \mapsto \underline{\mathsf{arr}_0} * \mathsf{val} \mapsto \mathsf{act}_0}$ †

```
         for (int i = 0; i < this.sz; i++) {
20         this.tbl[i] = oldtbl[i]; // copy from oldtbl
21       }
22     }
```

23 $\boxed{\mathsf{this} \mapsto \widehat{this} * \widehat{this}\cdot\mathsf{tbl} \mapsto \mathsf{arr}_0 * \mathsf{val} \mapsto \mathsf{act}_0}$

```
       this.tbl[this.sz] = val;
```

24 $\boxed{\underline{\mathsf{arr}_0\cdot\mathsf{contents} \mapsto \mathsf{act}_0}}$

```
       this.sz = this.sz + 1;
25   }
   }
```

Figure 4.2: Refuting a false alarm with context-sensitive, path-sensitive witness search. We show witness queries in the boxes. The † indicate refuted branches of the witness search.

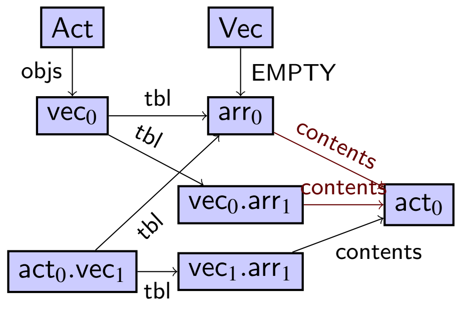


Figure 4.3: Points-to graph for the program in Figure 4.2. The red edges in the graph are infeasible points-to edges that can be refuted by our analysis.

object is shared among **all** instances the `Vec` class, **no refinement on the heap abstraction** alone would be sufficient to rule out this false alarm.

More precise analysis of this example requires reasoning about the relationship between the `sz` and `cap` fields of each `Vec`. This relationship is established in the `Vec`'s constructor and must be preserved until its `push` method is called. Though there is a large body of work focused on the important problem of refining heap abstractions (e.g., [Liang and Naik, 2011; Sridharan and Bodík, 2006]), this example shows that doing so alone is sometimes not sufficient for precise results. An analysis that lacks path sensitivity and strong updates will be unable to prove that `Vec`'s never write into the shared array and must therefore conflate the contents of all `Vec` objects. The **witness-refutation** technique that we detail in this paper enables after-the-fact, on-demand refinement to address this class of **control-precision** issues.

Refinement by Witness Refutation. We refine the results of the flow-insensitive points-to analysis by attempting to refute all executions that could possibly witness an edge involved in a leak alarm. The term "witness" is highly overloaded in the program analysis literature, so we begin by carefully defining its use in our context. We first define a **path** as a sequence of program transitions. A **path witness** for a **query** Q is a path that ends in a state that satisfies Q. Such a path witness may be concrete/under-approximate/must in that it describes a sequence of program transitions according to a concrete semantics that results in a state where Q holds (e.g., a test case execution). Analogously, an abstract/over-approximate/may path witness is such a sequence over an abstract semantics (e.g., a trace in an abstract interpreter).

Building on the definition of a path program [Beyer et al., 2007], we define a **path program witness** for a query Q as a path program that ends in a state satisfying Q. A **path program** is a program projected onto the transitions of a given execution trace (essentially, paths augmented with loops). Note that a path program witness may be under- or over-approximate in its handling of loops. In this paper, we use the term "witness" to refer to over-approximate path program witnesses unless otherwise stated, as our focus is on sound refutation of queries.

Our analysis performs a goal-directed, backwards search for a path program witness ending in a state that satisfies a query Q. We **witness** a query by giving a witness that produces it, or we **refute** a query by proving that no such witness can exist. Our technique proceeds in three phases.

Obtain a Conservative Analysis Result. First, we perform a standard points-to analysis to compute an over-approximation of the set of reachable heaps, such as the points-to graph in Figure 4.3.

Formulate queries. Second, we formulate queries to refute alarms generated using the points-to analysis result. For the Activity leak detection client, an alarm is a points-to path between a **static** field and an Activity object. For example, the following points-to path from the graph in Figure 4.3 is a (false) leak alarm:

$$\mathsf{Act}{\cdot}\mathsf{objs} \mapsto \mathsf{vec}_0, \mathsf{vec}_0{\cdot}\mathsf{tbl} \mapsto \mathsf{arr}_0, \mathsf{arr}_0{\cdot}\mathsf{contents} \mapsto \mathsf{act}_0$$

To refute an alarm, we attempt to refute each individual edge in the corresponding points-to path. If we witness all edges in the path, we report a leak alarm. If we refute some edge e in the path, we delete e from the points-to graph and attempt to find another path between the source node and the sink node. If we find such a path, we restart the process with the new path. If we refute enough edges to disconnect the source and sink in the points-to graph, we have shown that the alarm raised by the flow-insensitive points-to analysis is false.

For our client, we wish to show the flow-insensitive property that a particular points-to constraint cannot hold at any program point. Thus, for each points-to edge e to witness, we consider a query for e at every program statement that could produce e. This information can be obtained by simple post-processing or instrumentation of the up-front points-to analysis [Blackshear et al., 2011].

Search for witnesses. Finally, given a query Q at a particular program point, we search for path program witnesses on demand.

In Figure 4.2, we illustrate a witness search that produces a refutation for the points-to constraint $\mathsf{arr}_0{\cdot}\mathsf{contents} \mapsto \mathsf{act}_0$ holding at program point 24. That is, we prove that the points-to constraint is unrealizable at that program point. By starting a witness search from each statement that potentially produces the edge, we will see that $\mathsf{arr}_0{\cdot}\mathsf{contents} \mapsto \mathsf{act}_0$ is in fact unrealizable at any program point.

Notationally, we use a single arrow $\mapsto$ to denote an **exact points-to constraint**, whose source and sink are symbolic values typically denoting addresses of memory cells, and a double arrow $\Mapsto$ to denote a may points-to edge between abstract locations (cf., Section 4.3.2). For example, the exact points-to constraint $\mathsf{arr}_0{\cdot}\mathsf{contents} \mapsto \mathsf{act}_0$ describes a single memory cell whose address is some instance in the concretization of arr_0 and contents is some instance in the concretization of act_0. This distinction is critical for enabling strong updates in a backwards analysis.

4.2.3 Mixed symbolic-explicit queries

We illustrate the witness search by showing the sub-queries (boxed) that arise as the search progresses. Moving backwards from our starting point at line 24, the sub-query at program point 23 says that we need the following heap state at that point:

$$\mathsf{this} \mapsto \widehat{this} * \widehat{this}{\cdot}\mathsf{tbl} \mapsto \mathsf{arr}_0 * \mathsf{val} \mapsto \mathsf{act}_0 \qquad (\dagger)$$

where $\widehat{this}$ is a **symbolic variable** that represents the receiver of the method. A symbolic variable (written as a hatted letter $\widehat{v}$) is an existential standing for an arbitrary instance drawn from some definite set of abstract locations. Here, $\widehat{this}$ represents some instance drawn from the points-to set of local variable this, which is $\{\mathsf{vec}_0, \mathsf{vec}_1\}$. We represent this fact with an **instance constraint**:

$$\widehat{this}\ \mathsf{from}\ \{\mathsf{vec}_0, \mathsf{vec}_1\} \qquad (\ddagger)$$

that we track as part of the query at program point 23. For the moment, we elide such ‘from’ constraints and discuss them further in Section 4.2.4 and Section 4.3. This sub-query conjoining the heap state from (†) and the instance constraint from (‡) is an example of a **mixed** symbolic-explicit state because we introduce a fresh symbolic variable for the contents of **this**, but also have named abstract locations arr_0 and act_0. We say that a query is fully **explicit** if all of its points-to constraints are between named abstract locations from the points-to abstraction. A named abstract location can be seen as a symbolic variable that is constrained to be from a singleton abstract location set. This connection to the points-to abstraction in an explicit query enables our witness search to prune paths that are inconsistent with the up-front points-to analysis result, as we demonstrate in Section 4.2.4.

Backward path-by-path analysis. Returning to the example, the path splits into two prior to program point 23, one path entering the **if** control-flow branch at point 22, the other bypassing the branch to point 16. We consider both possibilities and indicate the fork in Figure 4.3 by indenting from the right margin. For the path into the branch, the loop between program points 19 and 22 has no effect on the query in question from point 23, so it simply continues backward to program point 19. Observe that because we are only interested in answering a specific query, this irrelevant loop poses no difficulty. At program point 19, we encounter a refutation for this path: the preceding assignment statement writes an instance of arr_1 to the $\widehat{this}\cdot\mathsf{tbl}$ field, which contradicts the requirement that $\widehat{this}\cdot\mathsf{tbl}$ hold an instance of arr_0 (underlined). Thus, we have discovered that no concrete program execution can assign a newly allocated array to **this**`.tbl` at line 18, that is, an instance of arr_1 and then place an Activity object in the `EMPTY` array at line 23 because **this**`.tbl` will point to that newly allocated array by then.

Resuming the path that bypasses the **if** branch, the analysis at program point 16 determines that entering the **if** branch changed the query and thus adds a **control-path constraint** to the abstract state indicating that the value of the **this**`.sz` field (i.e., $\widehat{v}^{\mathsf{int}}_{\mathsf{sz}}$) must be less than the value of the **this**`.cap` field (i.e., $\widehat{v}^{\mathsf{int}}_{\mathsf{cap}}$). As we will see, tracking the path constraint above is critical to obtaining a refutation for the example.

From here, this path reaches the method boundary, leading the analysis to process the possible call sites at program points 9 and 6. The path at program point 9 can be refuted at this point, as the query requires that the val parameter be bound to an instance of act_0 (underlined), but the actual argument is the string `"hello"`. Thus, we have identified that this call to push cannot be the reason that an Activity object is placed into the `EMPTY` array because it is pushing a string, not an Activity.

The other path from the `acts.push` call site (i.e., program point 6) can continue. The query at program point 5 before the call simply changes the program variables of the callee to those of the caller. Continuing this path, we enter the constructor of `Vec` at program point 14. Here, we discover that the values of the `sz` and `cap` fields as initialized in the constructor contradict the control constraint $\widehat{v}^{\mathsf{int}}_{\mathsf{sz}} < \widehat{v}^{\mathsf{int}}_{\mathsf{cap}}$. Intuitively, the witness search has observed the invariant that a `Vec`'s `tbl` field cannot point to the `EMPTY` array after a call to its `push` method. We have refuted the last path standing, and so we have shown that the statement at line 24 cannot produced the edge $\mathsf{arr}_0{\cdot}\mathsf{contents} \mapsto \mathsf{act}_0$.

In the above, we have been rather informal in describing why certain points-to facts can be propagated back unaffected and why producing certain facts can be done with a strong update-style transfer function. These properties come from details of our query states that relate to the notions of separation and frame in separation logic [Reynolds, 2002].

Furthermore, in the example program from Figure 4.2, there is one (and only one) more statement that could produce the constraint $\mathsf{arr}_0{\cdot}\mathsf{contents} \mapsto \mathsf{act}_0$, which is the assignment at line 20 inside the copy loop. A witness search for this query starting at line 20 leads to a refutation similar to the one described above, but to discover it we must first infer a non-trivial loop invariant. Because we are interested in an over-approximate path program witness-refutation search, we have to obtain loop invariants. This is in contrast to typical backwards symbolic executors, which need to unroll loops. We consider these issues further in Section **??**.

4.2.4 Taming path explosion from aliasing

In the previous subsection, we focused on how refutations can occur. For example, backwards paths were pruned at line 18 and at program point 9 because we reach an allocation site that conflicts with our instance constraints in the query (e.g., we need an arr_0 not an arr_1 at line 18). In this section, we present a simple example to explain how our mixed symbolic-explicit representation enables our analysis to derive such contradictions earlier and thus mitigates the **aliasing path explosion problem**.

An over-approximate backwards symbolic executor that lacks information about aliasing will be forced to fork a number of cases to account for aliasing at every field write, quickly causing a case explosion that is worst-case exponential in the number of field writes. This case explosion is independent of (but compounded by) the well-known scalability problems caused by conditional branching in a path-sensitive

1 $\boxed{\mathsf{x} \mapsto \widehat{y}, \mathsf{y} \mapsto \widehat{y}, \mathsf{p} \mapsto \widehat{z} \wedge \widehat{z} \text{ from } \mathrm{pt}_{\mathring{G}}(\mathsf{p}) \cap \mathrm{pt}_{\mathring{G}}(\mathsf{y.f}) \cap \mathring{r}, \widehat{y} \text{ from } \mathrm{pt}_{\mathring{G}}(\mathsf{x}) \cap \mathrm{pt}_{\mathring{G}}(\mathsf{y})}$

$\boxed{\mathsf{x} \mapsto \widehat{x}, \mathsf{y} \mapsto \widehat{y}, \widehat{y}{\cdot}\mathsf{f} \mapsto \widehat{z} \wedge \widehat{z} \text{ from } \mathrm{pt}_{\mathring{G}}(\mathsf{y.f}) \cap \mathring{r}, \widehat{y} \text{ from } \mathrm{pt}_{\mathring{G}}(\mathsf{y}) \wedge \widehat{x} \neq \widehat{y}}$

```
x.f = p;
```

2 $\boxed{\mathsf{y} \mapsto \widehat{y}, \widehat{y}{\cdot}\mathsf{f} \mapsto \widehat{z} \wedge \widehat{z} \text{ from } \mathrm{pt}_{\mathring{G}}(\mathsf{y.f}) \cap \mathring{r}, \widehat{y} \text{ from } \mathrm{pt}_{\mathring{G}}(\mathsf{y})}$

```
z = y.f;
```

3 $\boxed{\mathsf{z} \mapsto \widehat{z} \wedge \widehat{z} \text{ from } \mathring{r}}$

Figure 4.4: An example illustrating how ‘from’ constraints help tame path explosion from aliasing. Note that the set of abstract locations to which symbolic variable $\widehat{z}$ might belong is restricted each time we observe $\widehat{z}$ flow through a variable or field.

analysis.

To address this aliasing path explosion problem, the key observation that we make is that contradictions from instance constraints can be derived **before** the allocation site by exploiting information from the up-front points-to analysis. In particular, the set of possible allocation sites for any instance can be restricted as we reason about how they flow into and out of program variables and heap locations. This observation motivates our mixed symbolic-explicit representation, which we demonstrate with a simple example shown in Figure 4.4. Our initial query at point 3 asks if program variable `z` can point to an instance $\widehat{z}$ from some set of abstract locations $\mathring{r}$—we call this a **points-to region**. Moving backwards across the statement `z = y.f`, we derive a pre-query at point 2 that says our original query can be witnessed if `y` points to some instance $\widehat{y}$ and that instance points to $\widehat{z}$ through its `f` field (i.e., $\mathsf{y} \mapsto \widehat{y}, \widehat{y}{\cdot}\mathsf{f} \mapsto \widehat{z}$). Additionally, we now know that the instance $\widehat{z}$ must be drawn from the **intersection** of $\mathring{r}$ and the abstract locations in the points-to set of `y.f`, which we write as $\mathrm{pt}_{\mathring{G}}(\mathsf{y.f})$. If we can use the points-to graph $\mathring{G}$ to determine that no such abstract location exists (e.g., if we had $\mathring{r} = \{a_0, a_2\}$ and $\mathrm{pt}_{\mathring{G}}(\widehat{y}.\mathsf{f}) = \{a_1\}$), then we have refuted this query and can prune this path immediately.

Assuming $\mathring{r}$ and $\mathrm{pt}_{\mathring{G}}(\mathsf{y.f})$ are not disjoint (i.e., $\mathring{r} \cap \mathrm{pt}_{\mathring{G}}(\mathsf{y.f}) \neq \emptyset$), we proceed with our backwards analysis to the field write `x.f = p`. We must consider two cases at program point 1: one where `x` and `y` are aliased (the top query) and one where they are not (the bottom query). In the aliased case, we can further constrain the instance $\widehat{u}$ to be from $\mathrm{pt}_{\mathring{G}}(\mathsf{x}) \cap \mathrm{pt}_{\mathring{G}}(\mathsf{y})$. Some previous tools have used an up-front, over-approximate points-to analysis as an aliasing oracle to rule out aliased cases like this one (e.g., PSE [Manevich et al., 2004])—if `x` and `y` cannot possibly point to the same abstract location (i.e., $\mathrm{pt}_{\mathring{G}}(\mathsf{x}) \cap \mathrm{pt}_{\mathring{G}}(\mathsf{y}) = \emptyset$), this aliased case is ruled out. Our approach generalizes this kind of aliasing check by explicitly introducing 'from' constraints that are incrementally restricted. For example, we also constrain $\widehat{z}$ to be from $\mathrm{pt}_{\mathring{G}}(\mathsf{p}) \cap \mathrm{pt}_{\mathring{G}}(\mathsf{y.f}) \cap \mathring{r}$, where the additional restriction is $\mathrm{pt}_{\mathring{G}}(\mathsf{p})$. This constraint says that the field write `x.f = p` produced the query in question only if the instance $\widehat{z}$ is drawn from some abstract location shared by these three sets.

Finally, we consider the case where `x` and `y` are not aliased (i.e., $\widehat{x} \neq \widehat{y}$). Here, the only change to the query is the addition of the constraints $\mathsf{x} \mapsto \widehat{x}$ and $\widehat{x} \neq \widehat{y}$. This disequality further constrains the query so that if we later discover that `x` and `y` are in fact aliased, we can refute this query. Accumulation of this kind of disaliasing constraint is common (e.g., [Chandra et al., 2009]), but expensive (cf., Section 4.3.4).

We remark that the instance 'from' constraints can be viewed as a generalization of a fully explicit representation. To represent 'from' constraints explicitly, instead of a symbolic points-to constraint $\mathsf{x} \mapsto \widehat{x}$, we disjunctively consider all cases where we replace the symbolic variable $\widehat{x}$ with an abstract location from $\mathrm{pt}_{\mathring{G}}(\mathsf{x})$ (the points-to set of x) For example, suppose $\mathrm{pt}_{\mathring{G}}(\mathsf{x}) = \{a_1, a_2\}$; then we case split and consider two heap states: (1) $\mathsf{x} \mapsto a_1$ and (2) $\mathsf{x} \mapsto a_2$. This representation corresponds roughly to a backwards extension of **lazy initialization** [Khurshid et al., 2003] over abstract locations instead of types. Note that while PSE-style path pruning only applies to ruling out the aliased case in field writes, the explicit representation of 'from' constraints permits the same kind of flow-based restriction shown in Figure 4.4. However, the cost is case splitting a separate query for each possible abstract location from which each symbolic variable is drawn (e.g., $|\mathrm{pt}_{\mathring{G}}(\mathsf{y.f}) \cap \mathring{r}| \cdot |\mathrm{pt}_{\mathring{G}}(\mathsf{y})|$ queries at program point 2).

4.3 A store abstraction and analysis combining points-to analysis and separation logic

In this section, we formalize our goal-directed store abstraction that combines points-to analysis and separation logic and argue that an analysis build on this abstraction is **refutation sound**—that we only declare a query refuted when no concrete path producing that query can exist. We introduce a simple object-oriented language (Section 4.3.1), explain how we connect points-to analysis to separation logic queries with instance-from constraints (Section 4.3.2), define a goal-directed backward analysis using our

statements	$s ::= c \mid s_1 \,;\, s_2 \mid s_1 \,[]\, s_2 \mid \texttt{loop}\, s$
commands	$c ::= \texttt{skip} \mid x \texttt{ := } y \mid x \texttt{ := } y.f \mid x.f \texttt{ := } y \mid x \texttt{ := } \texttt{new}_a\, \tau\texttt{()} \mid \texttt{assume}\, e$
expressions	$e ::= x \mid \cdots$
types	$\tau ::= \{f_1, \ldots, f_n\} \mid \cdots$

program variables x, y object fields f abstract locations a

Figure 4.5: A simple imperative language with objects and fields.

store abstraction (Section 4.3.3), and show how our analysis handles loops using on-the-fly loop invariant inference (Section 4.3.4).

4.3.1 A simple imperative language with objects

The language providing the basis for our formalization is defined in Figure 4.5. This language is a standard imperative programming language with object fields and dynamic memory allocation. Atomic commands c include assignment, field read, field write, object allocation, and guards.

For the purposes of our discussion, it is sufficient if an object type is just a list of field names. We leave unspecified a sub-language of pure expressions, except that it allows reading of program variables. The label a on allocation names the allocation site so that we can tie it to the points-to analysis. Compound statements include a do-nothing statement `skip`, sequencing $s_1 \,;\, s_2$, non-deterministic choice $s_1 \,[]\, s_2$, conditional branching $\texttt{if}\,(e)\, s_1\, \texttt{else}\, s_2$, and looping $\texttt{loop}\, s$. Standard `if` and `while` statements can be defined in the usual way (i.e., $\texttt{if}\,(e)\, s_1\, \texttt{else}\, s_2 \overset{\text{def}}{=} (\texttt{assume}\, e \,;\, s_1) \,[]\, (\texttt{assume}\, !e \,;\, s_2)$ and $\texttt{while}\,(e)\, s \overset{\text{def}}{=} \texttt{loop}\,(\texttt{assume}\, e \,;\, s) \,;\, \texttt{assume}\, !e$). The concrete semantics of this language are given in Figure 4.6.

For ease of presentation, our formal language is intraprocedural. However, our implementation is fully interprocedural. We handle procedure calls by modeling parameter binding using assignment and keeping an explicit abstraction of the call stack. The call stack is initially empty representing an arbitrary calling context but grows as we encounter call instructions during symbolic execution. If we reach the entry block of a function with an empty call stack, we propagate symbolic state backwards to all possible callers of the current function. We determine the set of possible callers using the call graph constructed alongside the points-to analysis. In the following chapter, Section 5.1 formalizes an unstructured language with procedure calls that can be combined with the commands from Figure 4.5.

$$\boxed{\sigma \vdash s \downarrow \sigma'}$$

$$\textsc{EvalSkip}\quad \frac{}{\sigma \vdash \texttt{skip} \downarrow \sigma}
\qquad
\textsc{EvalAssign}\quad \frac{\sigma \vdash e \downarrow v}{\sigma \vdash x := e \downarrow \sigma[x \to v]}
\qquad
\textsc{EvalRead}\quad \frac{}{\sigma \vdash x := y.f \downarrow \sigma[x \to \sigma(\sigma(y)\cdot f)]}$$

$$\textsc{EvalWrite}\quad \frac{}{\sigma \vdash x.f := y \downarrow \sigma[\sigma(y)\cdot f \to \sigma(x)]}
\qquad
\textsc{EvalNew}\quad \frac{o \notin \sigma \qquad \tau = \{f_1, \ldots, f_n\}}{\sigma \vdash x := \texttt{new}\,\tau() \downarrow \sigma[o\cdot f_1 \to \mathsf{null}, \ldots, o\cdot f_n \to \mathsf{null}]}$$

$$\textsc{EvalSeq}\quad \frac{\sigma \vdash s_1 \downarrow \sigma' \qquad \sigma' \vdash s_2 \downarrow \sigma''}{\sigma \vdash s_1 ; s_2 \downarrow \sigma}
\qquad
\textsc{EvalIfTrue}\quad \frac{\sigma \vdash e \downarrow \texttt{true} \qquad \sigma \vdash s_1 \downarrow \sigma_1}{\sigma \vdash \texttt{if}\,(e)\, s_1\, \texttt{else}\, s_2 \downarrow \sigma_1}
\qquad
\textsc{EvalIfFalse}\quad \frac{\sigma \vdash e \downarrow \texttt{false} \qquad \sigma \vdash s_2 \downarrow \sigma_2}{\sigma \vdash \texttt{if}\,(e)\, s_1\, \texttt{else}\, s_2 \downarrow \sigma_2}$$

$$\textsc{EvalLoopTrue}\quad \frac{\sigma \vdash e \downarrow \texttt{true} \qquad \sigma \vdash s \downarrow \sigma' \qquad \sigma' \vdash \texttt{loop}\, s \downarrow \sigma''}{\sigma \vdash \texttt{loop}\, s \downarrow \sigma''}
\qquad
\textsc{EvalLoopFalse}\quad \frac{\sigma \vdash e \downarrow \texttt{false}}{\sigma \vdash \texttt{loop}\, s \downarrow \sigma}$$

Figure 4.6: A standard big-step operational semantics for the primitive imperative language given in Figure 4.5.

4.3.2 Instance-from constraints

Given a program, we first do a standard points-to analysis to obtain a points-to graph $\mathring{G}$: $\langle \mathring{V}, \mathring{E} \rangle$ consisting of a set of vertices $\mathring{V}$ and a set of edges $\mathring{E}$ (e.g., Figure 4.3). A vertex represents a set of memory addresses, which include program variables $x \in \mathbf{Var}$ and abstract locations $a \in \mathbf{AbsLoc}$ (i.e., $\mathring{V} \supseteq \mathbf{Var} \cup \mathbf{AbsLoc}$). An abstract location a abstracts non–program-variable locations (e.g., from dynamic memory allocation). We do not fix the heap abstraction, such as the level of context sensitivity, but we do assume that we are given the abstract location to which any `new` allocation belongs (via the subscript annotation).

A points-to edge from $\mathring{E}$ is either of the form $x \Mapsto a$ or $a_0 \cdot f \Mapsto a_1$. The form $x \Mapsto a$ means a concrete memory address represented by the program variable x may contain a value represented by abstract location a. We write $a_0 \cdot f \Mapsto a_1$ to denote that f is the label for the edge between nodes a_0 and a_1. This edge form means that $a_0 \cdot f$ is a field of an object in the concretization of a_0 that may contain a value represented by abstract location a_1. A **static** field in Java can be modeled as a global program variable.

Our analysis permits formulating a query Q over a finite number of heap locations along with constraints over data fields, as shown in Figure 4.7. We give a heap location via an **exact points-to edge constraint** having one of two forms: $x \mapsto \widehat{v}$ or $\widehat{v} \cdot f \mapsto \widehat{u}$. Recall that in contrast to points-to edges that summarize a **set** of concrete memory cells, an exact points-to constraint expresses a single memory cell. The first form $x \mapsto \widehat{v}$ means a memory address represented by the program variable x contains a value represented by a symbolic variable $\widehat{v}$ (and similarly for the second form for a field). Since we are mostly concerned with memory addresses for concrete object instances, we often refer to symbolic variables as **instances**. The memory any stands for an arbitrary memory. While it is standard in separation logic to write any as true, here we simply wish to make explicit the memory and pure components and thus reserve true for pure formulæ. The pure part P are first-order formulæ over symbolic variables that we leave mostly unspecified.

We introduce one non-standard pure constraint form: the **instance constraint** $\widehat{v}$ from $\mathring{r}$ says the symbolic variable $\widehat{v}$ is an instance of a points-to region $\mathring{r}$ (i.e., is in the set of values described by region $\mathring{r}$). A **points-to region** is a set of abstract locations a or the special region data. For uniformity, the region data is used to represent the set of values that are not memory addresses, such as integer values. As we have seen

queries	$Q ::= M \wedge P \mid \mathsf{false}$
memories	$M ::= \mathsf{any} \mid x \mapsto \widehat{v} \mid \widehat{v}\cdot f \mapsto \widehat{u} \mid M_1 * M_2$
pure formulæ	$P ::= \mathsf{true} \mid P_1 \wedge P_2 \mid \widehat{v}\,\mathsf{from}\,\mathring{r} \mid \cdots$
points-to regions	$\mathring{r}, \mathring{s} ::= a \mid \mathsf{data} \mid \mathring{r}_1 \Cup \mathring{r}_2$
instances	$\widehat{v}, \widehat{u}$
refutation states	$R ::= Q \mid R_1 \vee R_2 \mid \exists \widehat{v}. R$

Figure 4.7: Queries and analysis state.

in Section 4.2.4, instance constraints enable us to use information from the up-front points-to analysis in our witness-refutation analysis. As an example, the informal query $\mathsf{arr}_0\cdot\mathsf{contents} \mapsto \mathsf{act}_0$ from Section **??** is expressed as follows:

$$\widehat{v}_1\cdot\mathsf{contents}_{[\widehat{v}_3]} \mapsto \widehat{v}_2 \wedge \widehat{v}_1\ \mathsf{from}\ \{\mathsf{arr}_0\} \wedge \widehat{v}_2\ \mathsf{from}\ \{\mathsf{act}_0\} \wedge \widehat{v}_3\ \mathsf{from}\ \mathsf{data}$$

where $\widehat{v}_3$ stands for the index of the array. This query considers existentially an instance of each abstract location arr_0 and act_0.

When writing down queries, we assume the usual commutativity, associativity, and unit laws from separation logic [Reynolds, 2002]. Since we are interested in witnessing or refuting a subset of edges corresponding to part of the memory, we interpret any memory M as $M * \mathsf{any}$ (or intuitionistically instead of classically [Ishtiaq and O'Hearn, 2001; Reynolds, 2002]).

4.3.3 Goal-directed backward analysis with instance-from constraints

As described in Section 4.2, we perform a path-program–by–path-program, backwards symbolic analysis to find a witness for a given query Q. A refutation state R is simply a disjunction of queries (as shown in Figure 4.7), which we often view as a set of candidate witnesses. We include an existential binding of instances $\exists \widehat{v}.R$ to make explicit when we introduce fresh instances, but we implicitly view instances as renamed so that they are all bound at the top-level (i.e., all formulæ are in prenex normal form).

Informally, a path program witness is a query Q_{wit} bundled with a sub-program examined so far s_{wit} and a sub-program left to be explored s_{pre}.

Definition 4.3.1 (Path Program Witness) A **path program witness** for an input statement-query pair $\langle s, Q \rangle$

is a triple $\langle s_{\text{pre}}, Q_{\text{wit}}, s_{\text{wit}} \rangle$ where (1) $s \equiv s_{\text{pre}} \; ; s_{\text{post}}$, and (2) s_{wit} is a sub-statement of s_{post} such that (a) if an execution of s_{post} leads to a store σ_{post} satisfying the input query Q, then it must be from a store σ_{wit} satisfying Q_{wit} and (b) executing s_{wit} from σ_{wit} also leads to σ_{post}.

Any intermediate state in our backwards analysis is such a path program witness. Intuitively, the statement s_{wit} captures the path sub-program identified by the backwards analysis that is relevant to producing the input query Q (so far).

A refutation occurs when Q_{wit} is false, that is, we have discovered that it is not possible to end up in a state satisfying Q. A "full" witness is when Q_{wit} is any; that is, we can no longer find a refutation. A "partial" witness is a witness where Q_{wit} is a query other than any or false.

We formalize a backwards path program enumeration transforming queries into sub-queries to eventually produce an any or a false refutation state. To describe the analysis, we define the judgment form $\vdash \{R'\} \; s \; \{R\}$ in Figure 4.8. This judgment form is a standard Hoare triple, but because our analysis is backward, we read this judgment form from right-to-left. It says, "Given a post-formula R, we find a pre-formula R' such that executing statement s from a state satisfying R' yields a state satisfying R (up to termination)." Conceptually, the post-formula R is a (disjunctive) set of queries, and the pre-formula R' is the set of sub-queries. we focus on describing the backwards path program enumeration transforming queries into sub-queries to eventually produce an any or a false witness query Q_{wit}. The path program s_{wit} can be obtained by a simple instrumentation of the rules similar to our prior work [Blackshear et al., 2011].

Deriving refutations. At any point, we can extend the witness-refutation search for some disjunct. Here, we express this step with WITCASES, which says a disjunctive refutation state $R_1 \vee R_2$ can be derived by finding a witness for R_1 and R_2. We make this system algorithmic for an implementation by representing cases as a disjunctive set of pending queries $\cdots \vee Q_i \vee \cdots$ that we extend individually. Rule WITREFUTED simply says if we have derived false in the post-state for a statement, then we have false in the pre-state as well. To scale beyond the tiniest of programs, we need to be able to refute queries quickly so that the number of queries to consider, that is, the number of symbolic execution paths to explore, remains small. We have three tools for refuting queries: (1) separation (i.e., a query where a single memory cell would need to point

$$\boxed{\vdash \{R\}\; s\; \{Q\}}$$

$$\textsc{WitSkip}\quad \frac{}{\vdash \{\,\mathsf{any} \wedge \mathsf{true}\,\}\; s\; \{\,\mathsf{any} \wedge \mathsf{true}\,\}}$$

$$\textsc{WitFrame}\quad \frac{\vdash \{\,\bigvee_i M_i' \wedge P_i'\,\}\; s\; \{\,M \wedge P\,\} \qquad s \text{ must not modify } M_{\mathrm{fr}}}{\vdash \{\,\bigvee_i (M_{\mathrm{fr}} * M_i') \wedge P_i'\,\}\; s\; \{\,(M_{\mathrm{fr}} * M) \wedge P\,\}}$$

$$\boxed{\vdash \{R'\}\; s\; \{R\}}$$

$$\textsc{WitRefuted}\quad \frac{}{\vdash \{\,\mathsf{false}\,\}\; s\; \{\,\mathsf{false}\,\}}$$

$$\textsc{WitCases}\quad \frac{\vdash \{\,R_1'\,\}\; s\; \{\,R_1\,\} \qquad \vdash \{\,R_2'\,\}\; s\; \{\,R_2\,\}}{\vdash \{\,R_1' \vee R_2'\,\}\; s\; \{\,R_1 \vee R_2\,\}}$$

$$\boxed{\vdash \{R\}\; c\; \{Q\}}$$

$$\textsc{WitNew}\quad \frac{}{\vdash \{\,\mathsf{any} \wedge \boxed{\widehat{v}\ \mathsf{from}\ a \cap \mathring{r}} \wedge P\,\}\; x := \mathtt{new}_a\, \tau\mathtt{()}\; \{\,x \mapsto \widehat{v} \wedge \widehat{v}\ \mathsf{from}\ \mathring{r} \wedge P\,\}}$$

$$\textsc{WitAssign}\quad \frac{}{\vdash \{\,y \mapsto \widehat{v} \wedge \boxed{\widehat{v}\ \mathsf{from}\ \mathsf{pt}_{\mathring{G}}(y) \cap \mathring{r}} \wedge P\,\}\; x := y\; \{\,x \mapsto \widehat{v} \wedge \widehat{v}\ \mathsf{from}\ \mathring{r} \wedge P\,\}}$$

$$\textsc{WitRead}\quad \frac{P' = \widehat{u}\ \mathsf{from}\ \mathsf{pt}_{\mathring{G}}(y) \wedge \boxed{\widehat{v}\ \mathsf{from}\ \mathsf{pt}_{\mathring{G}}(y.f) \cap \mathring{r}} \wedge P}{\vdash \{\,\exists \widehat{u}.\, y \mapsto \widehat{u} * \widehat{u}\cdot f \mapsto \widehat{v} \wedge P'\,\}\; x := y.f\; \{\,x \mapsto \widehat{v} \wedge \widehat{v}\ \mathsf{from}\ \mathring{r} \wedge P\,\}}$$

$$\textsc{WitWrite}\quad \frac{\begin{array}{c} M_i = x \mapsto \widehat{v}_i * y \mapsto \widehat{u}_i * (\bigast_{j \neq i} \widehat{v}_j\cdot f \mapsto \widehat{u}_j \wedge \widehat{v}_j\ \mathsf{from}\ \mathring{r}_j \wedge \widehat{u}_j\ \mathsf{from}\ \mathring{s}_j) \\ Q_i = M_i \wedge \boxed{\widehat{v}_i\ \mathsf{from}\ \mathsf{pt}_{\mathring{G}}(x) \cap \mathring{r}_i} \wedge \boxed{\widehat{u}_i\ \mathsf{from}\ \mathsf{pt}_{\mathring{G}}(y) \cap \mathring{s}_i} \wedge (\bigwedge_{j \neq i} \widehat{v}_j \neq \widehat{v}_i) \wedge P \\ Q = \exists \widehat{x}.\, x \mapsto \widehat{x} * (\bigast_i \widehat{v}_i\cdot f \mapsto \widehat{u}_i \wedge \widehat{v}_i\ \mathsf{from}\ \mathring{r}_i \wedge \widehat{u}_i\ \mathsf{from}\ \mathring{s}_i \wedge \widehat{v}_i \neq \widehat{x}) \wedge \widehat{x}\ \mathsf{from}\ \mathsf{pt}_{\mathring{G}}(x) \wedge P \end{array}}{\vdash \{\,Q \vee \bigvee_i Q_i\,\}\; x.f := y\; \{\,(\bigast_i \widehat{v}_i\cdot f \mapsto \widehat{u}_i \wedge \widehat{v}_i\ \mathsf{from}\ \mathring{r}_i \wedge \widehat{u}_i\ \mathsf{from}\ \mathring{s}_i) \wedge P\,\}}$$

Figure 4.8: Witness-refutation search is a path-program–by–path-program backwards analysis. Boxed terms emphasize opportunities for refuting paths using instance-from constraints.

to two locations simultaneously), (2) instance constraints (i.e., a query where an instance cannot be in any points-to region), and (3) other pure constraints (i.e., a query with pure constraints that are unsatisfiable, such as, from detecting an infeasible control-flow path). In our inference rules, we assume a refutation state R is always normalized to false if the formula is unsatisfiable. Since we are interested in sound refutations and over-approximate witnesses, for scaling, we can also weaken queries at the cost of potentially losing precision. We revisit this notion in Section 4.3.4.

Instance constraints are pure constraints that tie the exact points-to constraints in the query to the accumulated information from the up-front points-to analysis and the flow of the symbolic variables as discussed in Section 4.2.4. They can axiomatized as follows:

$$\widehat{v}\ \mathsf{from}\ \emptyset \iff \mathsf{false} \tag{1}$$

$$\widehat{v}\ \mathsf{from}\ \mathring{r}_1 \wedge \widehat{v}\ \mathsf{from}\ \mathring{r}_2 \iff \widehat{v}\ \mathsf{from}\ \mathring{r}_1 \cap \mathring{r}_2 \tag{2}$$

$$\mathsf{true} \iff \widehat{v}\ \mathsf{from}\ \mathbf{AbsLoc} \cup \{\mathsf{data}\} \tag{3}$$

In particular, we derive a contradiction when we discover an instance that can be drawn from any abstract location (axiom 1). Though our formalism groups instance constraints and other pure constraints together, our implementation keeps them separate for simplicity in checking. Instance constraints are checked using basic set operations and other pure constraints are checked with an off-the-shelf SMT solver, though it should be possible to encode instance constraints into the solver using this axiomatization.

We include a frame rule on M, WITFRAME, to simplify our presentation, which together with WITSKIP allow us to isolate the parts of the query that a block of code may affect and to ignore irrelevant statements. In other words, any statements that cannot affect the memory state in the query can be skipped. We can thus focus our discussion on an auxiliary judgment form $\vdash \{R\}\ c\ \{Q\}$ that describes how the assignment commands affect a query and its point-to and instance constraints. Informally, it says, "We can witness a query Q after assignment command c by executing c if we can also witness one of the sub-queries R before c." From an algorithmic perspective, we can view $\vdash \{R\}\ c\ \{Q\}$ as generating sub-queries that when combined with a frame may yield additional contradictions. We assume this judgment implicitly has access to the points-to graph $\mathring{G}\colon \langle \mathring{V}, \mathring{E} \rangle$ computed by the up-front analysis. We use the $\mathrm{pt}_{\mathring{G}}(\cdot)$ function to get the points-to set of a program variable via $\mathrm{pt}_{\mathring{G}}(x) \overset{\mathrm{def}}{=} \{a \mid (x \mapsto a) \in \mathring{E}\}$ or a field of a program variable via $\mathrm{pt}_{\mathring{G}}(y.f) \overset{\mathrm{def}}{=} \{a_j \mid a_i \in \mathrm{pt}_{\mathring{G}}(y) \text{ and } (a_i \cdot \mathsf{f} \mapsto a_j) \in \mathring{E}\}$.

Backward transfer functions and instance-from **constraints.** Rule WITNEW says that the exact points-to constraint $x \mapsto \widehat{v}$ can be witnessed, or **produced**, by the allocation command x := $\mathtt{new}_a\, \tau()$ if instance $\widehat{v}$ may have been created at allocation site a. Following our axiomatization, the instance constraint $\widehat{v}$ from $a \cap \mathring{r}$ may immediately reduce to false if $a \notin \mathring{r}$. Or, we can drop it (without loss of precision) because this instance cannot exist before its allocation at this statement. Such a contradiction is precisely the reason for refuting the path at the new arr_1 allocation (program point 19) in our motivating example (Figure 4.2).

Now, consider rule WITASSIGN: it says that the exact points-to constraint $x \mapsto \widehat{v}$ can be produced by the assignment command $x := y$ if $y \mapsto \widehat{v}$ can be witnessed before this assignment **and** the instance $\widehat{v}$ can come from a region common to the points-to set of y and the region $\mathring{r}$. If the points-to set of y and the region $\mathring{r}$ are disjoint, we can derive a contradiction because the instance $\widehat{v}$ cannot come from any allocation site. Observe that WITASSIGN leverages the 'from' constraint and the up-front points-to analysis result to eagerly discover that **no** allocation site satisfies the conditions required for a witness (rather than observing that a **particular** allocation site does not satisfy the conditions required for a witness, as in WITNEW). To get a sense for why these eager refutations are critical for scaling, consider the path refutation due to the binding of val to `"hello"` at the `objs.push` call site (program point 9). In our example, this refutation is via WITNEW because `"hello"` is a `String` allocation, but we can easily imagine a variant where `objs.push` is called with a program variable y that can conservatively point-to a large set of non-Activity objects. For such a program, WITASSIGN would allow us to discover a path refutation at the assignment corresponding to the binding rather than requiring us to continue exploration across the potentially exponential number of paths to the allocation sites that flow to y.

The WITREAD rule is quite similar to WITASSIGN except that we existentially quantify over the instance to which y points (i.e., $\widehat{u}$). We set up an initial points-to region constraint for the fresh symbolic variable $\widehat{u}$ based on the points-to set of y and narrow the points-to region of $\widehat{v}$ using the points-to set of $y.f$. As in WITASSIGN, we can derive a contradiction based on this narrowing if the region $\mathring{r}$ and the points-to set of $y.f$ are disjoint. Here, we have taken some liberty with notation placing, for example, 'from' constraints

$$\textsc{WitAssume}\quad \frac{}{\vdash \{M \wedge P \wedge e[M]\}\ \texttt{assume}\, e\ \{M \wedge P\}}$$

$$\textsc{WitSeq}\quad \frac{\vdash \{R''\}\, s_1\, \{R'\} \quad \vdash \{R'\}\, s_2\, \{R\}}{\vdash \{R''\}\, s_1\, ;\, s_2\, \{R\}}$$

$$\textsc{WitChoice}\quad \frac{\vdash \{R_1\}\, s_1\, \{R\} \quad \vdash \{R_2\}\, s_2\, \{R\}}{\vdash \{R_1 \vee R_2\}\, s_1 \,[]\, s_2\, \{R\}}$$

Figure 4.9: Analysis rules for conditionals, sequencing, and nondeterministic choice.

under the iterated separating conjunctions; we recall that $*$ collapses to $\wedge$ for pure constraints.

In the WITWRITE rule, the post-formula consists of two cases for each edge $\widehat{v}_i \cdot f \mapsto \widehat{u}_i$ in the pre-formula: (1) the field write $x.f := y$ did not produce the edge $\widehat{v}_i \cdot f \mapsto \widehat{u}_i$ (the first disjunct Q), or (2) the field write did produce the edge (the second set of disjuncts over all Q_i). If the write $x.f := y$ did produce the points-to edge $\widehat{v}_i \cdot f \mapsto \widehat{u}_i$, then the points-to regions of $\widehat{v}_i$ and $\widehat{u}_i$ are restricted based on the points-to sets of x and y, respectively. The "not produced" case represents the possibility that this write updates an instance other than a $\widehat{v}_i$ (as reflected by the $x \mapsto \widehat{x}$ and $\widehat{v}_i \neq \widehat{x}$ conditions).

While WITWRITE can theoretically generate a huge case split, we have observed that the combination of instance constraints and separation typically allow us to find refutations quickly in practice (see Section 4.4). In particular, the "not written" case can often be immediately refuted by separation. For example, we end up with a contradictory query where a local variable `x` has to point to two different instances simultaneously (i.e., $\texttt{x} \mapsto \widehat{v} * \texttt{x} \mapsto \widehat{u} \wedge \widehat{v} \neq \widehat{u}$).

Guards and control-flow. Except for loops (see Section 4.3.4), the remaining rules mostly relate to control flow and are quite standard. These rules are given in Figure 4.9. To discover a contradiction on pure constraints, we state that the guard condition of an `assume` must hold in the pre-query. We write $e[M]$ for interpreting the program expression e in the memory state M.

The WITCHOICE rule analyzes each branch independently. Our implementation avoids path explosion due to irrelevant path sensitivity by adding pure constraints from `if`-guards only when the queries on each side of the branch are different (as in previous work [Das et al., 2002; Manevich et al., 2004]), though our rules do not express this.

$$\textsc{WitLoop}\quad \frac{\vdash \{R\}\, s\, \{R\}}{\vdash \{R\}\, \texttt{loop} s\, \{R\}} \qquad\qquad \textsc{WitCoarsen}\quad \frac{R_2' \models R_1' \quad \vdash \{R_2'\}\, s\, \{R_2\} \quad R_1 \models R_2}{\vdash \{R_1'\}\, s\, \{R_1\}}$$

Figure 4.10: Analysis rules for handling loops and coarsening.

4.3.4 Loop invariant inference and query simplification

In this section, we finish our description of witness-refutation search by discussing our loop invariant inference scheme. Roughly speaking, we infer loop invariants by repeatedly performing backwards symbolic execution over the loop body until we reach a fixed point over the domain of points-to constraints. To ensure termination, we drop all pure constraints affected by the loop body and fix a static bound on the number of instances of each abstract location to materialize (unlike modern shape analyzers [Distefano et al., 2006; Sagiv et al., 2002]). In our experiments, a static bound of one has been sufficient for precise results.

We now proceed to our discussion of per-path loop invariant inference using the WitLoop and WitAbstraction rules defined in Figure 4.10. Because we are interested in an over-approximate path program witness-refutation search, we have to obtain loop invariants. The WitLoop rule simply states that if the loop body has no effect on the query, the loop has no effect on it.

By itself, this rule only handles the degenerate case where a loop can be treated as `skip` with respect to the query. For this case, the disjunctive set of queries R is trivially a loop invariant. For more interesting cases, we use WitCases to consider each query in the refutation state individually so that we can infer an over-approximate loop invariant for each one. Thus, we infer a loop invariant on-the-fly for each **path program** rather than joining all queries at the loop exit and then inferring an invariant for all backwards paths into the loop (similar to [Leino and Logozzo, 2005] but with heap constraints).

To preserve refutation soundness, we want to ensure that a contradiction false is only derived when there does not exist a concrete path witnessing the given query and thus must over-approximate loops. A sound, backwards over-approximation can be obtained by weakening the post-loop query Q_i. Since an individual query is purely conjunctive, we can weaken it quite easily by "dropping" constraints (i.e., removing conjuncts). Intuitively, dropping constraints is refutation-sound because it can only make it more difficult to derive a contradiction. This over-approximation is captured by the WITCOARSEN rule. The rule says that at any program point, we can drop constraints, and doing so preserves refutation soundness (Theorem 4.3.1).

We write $R_1 \models R_2$ for the semantic entailment and correspondingly rely on a sound decision procedure in our implementation (used in WITABSTRACTION). Entailment between a finite separating conjunction exact points-to constraints can be resolved in a standard way by subtraction [Berdine et al., 2005]. Without inductive predicates, the procedure is a straightforward matching. Entailment between the ‘from’ instance constraints can be defined as follows:

$$(\widehat{v}_1 \text{ from } \mathring{r}_1) \models (\widehat{v}_2 \text{ from } \mathring{r}_2) \quad \text{iff} \quad \widehat{v}_1 = \widehat{v}_2 \text{ and } \mathring{r}_1 \subseteq \mathring{r}_2 \qquad (\S)$$

As previously mentioned, ‘from’ constraints are represented as sets associated with a symbolic variable and solved with ordinary set operations. We discharge other pure constraints using an off-the-shelf SMT solver, so precision of reasoning about those constraints is with respect to the capabilities of the solver.

With these tools, our loop invariant inference is a rather straightforward fixed-point computation. For a loop statement `loop`s and a post-loop query Q, we iteratively apply the backwards predicate transformer for the loop body s to saturate a set of sub-queries at the loop head. Let R_0 be some refutation state such that $\vdash \{R_0\}\ s\ \{Q\}$, and let $R_{i+1} = R_i \vee R'$ where $\vdash \{R'\}\ s\ \{R_i\}$. We ensure that the chain of $R_0 \models R_1 \models \cdots$ converges by bounding the number of instances or materializations from the abstract locations. Since there are a finite number of abstract locations, the number of points-to constraints in any particular query is bounded by the number of edges in the points-to graph (i.e., $|\mathring{E}|$). For the base domain of pure constraints, widening [Cousot and Cousot, 1977] can be used to ensure convergence. Our implementation uses a trivial widening that drops pure constraints that may be modified by the loop. We use a simpler approach for recursive methods. The instance bounding yields a very simplistic widening that has been surprisingly effective.

In our implementation, we also use a mod/ref analysis derived from the points-to analysis to see if we can quickly skip a loop or method call because the loop or method call cannot affect the current query.

Query simplification with disaliasing. The WITABSTRACTION rule captures backwards over-approximation by saying that at any point, we can weaken a refutation state without losing refutation soundness. Conceptually, we can weaken by replacing any symbolic join $\vee$ with an over-approximate join $\sqcup$. We perform one such join by replacing the refutation state $Q_1 \vee Q_2$ with Q_2 if $Q_1 \models Q_2$. Note that this join does not lose precision. Intuitively, for a refutation state R, we are interested in witnessing **any** query in R or refuting **all** queries in R. Here, a refutation of query Q_2 implies a refutation of Q_1, so we only need to consider Q_2.

To enable this join to apply often, we enforce a normal form for our queries by dropping certain kinds of constraints. As formalized in Figure 4.8, the backwards transfer functions for assignment commands c are as precise as possible, including the generation of disequality constraints in WITWRITE. These disequality constraints are needed locally to check for refutations due to separation, as detailed in Section 4.3.3. However, if this check passes, we drop them before proceeding and instead keep only the disaliasing information implied by separation and the instance from constraints. While this weakening could lose precision (e.g., if the backwards analysis would later encounter an `if`-guard for the aliasing condition), we hypothesize that this situation is rare and that the most useful disaliasing information is captured by separation and instance constraints.

In our implementation, we are much closer to a path-by-path analysis than our formalization would indicate. Refutation states are represented as a worklist of pending (non-disjunctive) queries to explore. To apply the simplification described above, we must keep a history of queries seen at a given program point: if we have previously seen a weaker query at this program point, then we can drop the current query. We keep a query history only at procedure boundaries and loop heads. This simplification has been especially critical for procedures.

Soundness. We define a concrete store σ be a finite mapping from variables or address-field pairs to concrete values (i.e., $\sigma : \textbf{Var} \uplus (\textbf{Addr} \times \textbf{Field}) \rightharpoonup_{\text{fin}} \textbf{Val}$) and give a standard big-step operational semantics to our basic imperative language The judgment form $\sigma \vdash s \Downarrow \sigma'$ says, "In store σ, statement s evaluates to store σ'." Furthermore, we write $\sigma \models R$ to say that the store σ is in the concretization of the refutation state R. The definition of $\sigma \models R$ is as would be expected in comparison to separation logic. We need to utilize two other concrete semantic domains: a valuation η that maps instances to values (i.e., $\eta : \textbf{Instance} \rightarrow \textbf{Val}$) and a

regionalization ρ that maps abstract locations to sets of concrete addresses (i.e., $\rho : \mathbf{AbsLoc} \to \wp(\mathbf{Addr})$). The regionalization gives meaning to the 'from' instance constraint. With these definitions, we precisely state the soundness theorem.

Theorem 4.3.1 (Refutation Soundness) If $\vdash \{R_{\text{pre}}\}\ s\ \{R_{\text{post}}\}$ and $\sigma_{\text{pre}} \vdash s \Downarrow \sigma_{\text{post}}$ such that $\sigma_{\text{post}} \models R_{\text{post}}$, then $\sigma_{\text{pre}} \models R_{\text{pre}}$. As a corollary, refutations (i.e., when R_{pre} is false) are sound.

Interestingly, the standard consequence rule from Hoare logic states the opposite in comparison to WITABSTRACTION by permitting the strengthening of queries. Doing so would instead preserve witness precision; that is, any path program witness exhibits some witness path (up to termination).

4.4 Case study: precisely identifying Android Activity leaks

We evaluated our witness-refutation analysis by using it to find Activity leaks, a common class of memory leaks in open-source Android applications. We explain this client in more detail below.

Our experiments were designed to test two hypotheses. The first and most important concerns the **precision** of our approach: we hypothesized that witness-refutation analysis reports many fewer false alarms than a flow-insensitive points-to analysis. We tried using a flow-insensitive analysis to find leaks, but found that the number of alarms reported was too large to examine manually. To be useful, our technique needs to prune this number enough for a user to effectively triage the results and identify real leaks.

Our second hypothesis concerns the **utility** of our techniques: we posited that (1) our mixed symbolic-explicit is an improvement over both a fully explicit and a fully symbolic representation, (2) our query simplification significantly speeds up analysis, and (3) our on-the-fly loop invariant inference is needed to preserve precision in the presence of loops.

Client: a class of pernicious Android memory leaks Activity leaks occur when a pointer to an Activity object can outlive the Activity. The operating system frequently destroys Activity's when configuration changes occur (e.g., rotating the phone). Once an Activity is destroyed, it can never be displayed to the user again and thus represents unused memory that should be reclaimed by the garbage collector. However, if an application developer maintains a pointer to an Activity after it is destroyed, the garbage collector will be unable to reclaim it. In our experiments, we check if **any** Activity instance is ever reachable from a static field, a flow-insensitive property. Though a developer could safely keep a reference to an Activity object via a static field that is cleared each time the Activity is destroyed, this is recognized as bad practice.

Activity leaking is a serious problem. It is well-documented that keeping persistent references to Activity's is bad practice; we refer the reader to an article[2] in the Android Developers Blog as evidence. The true problem is that it is quite easy for developers to inadvertently keep persistent references to an Activity. Sub-components of Activity's (such as Adapter's, Cursor's, and View's) typically keep pointers to their parent Activity, meaning that any persistent reference to an element in the Activity's hierarchy can potentially create a leak.

Precision of our techniques: threshing alarms. We implemented our witness-refutation analysis in the THRESHER tool, which is publicly available.[3] Additional details on our implementation are included at the end of this section. All of our experiments were performed on a machine running Ubuntu 12.04.2 with a 2.93 GHz Intel Xeon processor and 32GB of memory. Though our analysis is quite amenable to parallelization in theory, our current implementation is purely sequential.

To evaluate the precision of our approach, we computed a flow-insensitive points-to graph for each application and the Android library (version 2.3.3) using WALA's 0-1-Container-CFA pointer analysis (a variation of Andersen's analysis with unlimited context sensitivity for container classes). For each heap path from a static field f to an Activity instance A in the points-to graph, we asked THRESHER to witness or refute each edge in the path from source to sink. If we refuted an edge in the heap path, we searched for a new path. We repeated this until THRESHER either witnessed each edge in the heap path (i.e., confirmed the flow-insensitive alarm) or refuted enough edges to prove that no heap path from f to A can exist (i.e., filtered out the leak report). We allowed an exploration budget of 10,000 path programs for each edge; if the tool exceeded the budget, we declared a timeout for that edge and considered it to be not refuted. On paths with call stacks of depth greater than three, we soundly skipped callees by dropping constraints that executing the call might produce (according to a mod/ref analysis computed alongside the points-to analysis). We limited

[2] `http://developer.android.com/resources/articles/avoiding-memory-leaks.html`
[3] `https://github.com/cuplv/thresher`

the size of the path constraint set to at most two. Allowing larger path constraint sets slowed down the symbolic executor without increasing precision. We ran in two configurations: one with the Android library as-is (first row for each benchmark), and one where we added a single annotation to the HashMap class to indicate that the shared `EMPTY_TABLE` field can never point to anything (second row for each benchmark). We did this because we observed that the use of the null object pattern in the HashMap class was a major source of imprecision for the flow-insensitive analysis (cf. Figure 4.2), but we wanted to make sure that it was not the only one our tool was able to handle.

Table 4.1 shows the results of this experiment. We first comment on the most interesting part of the experiment: the *filtering effectiveness* of our analysis. As we hoped, our analysis is able to refute many of the false alarms produced by the flow-insensitive points-to analysis. Overall, we refute 129/457 = 28% of these false alarms in the un-annotated configuration and 172/196 = 87% of these false alarms in the annotated configuration. Contrary to our expectations, we found many **more** refutations in the annotated configuration, confirming that our technique can indeed remedy imprecision other than the pollution caused by HashMaps.

Unfortunately, this also means that our analysis is not always able to remedy the imprecision caused by HashMaps. The major problem is that the unannotated configuration fails to refute many of the HashMap-related edges due to timeouts. In fact, most of the false alarms that are not common to both configurations stem from (soundly) not considering timed-out edges to be refuted. We observed that a timeout commonly corresponds to a refutation that the analysis was unable to find within the path program budget. This is not surprising; finding a witness for an edge only requires finding a single path program that produces the edge (which we can usually do quickly), but to find a refutation we must refute **all** path programs that might produce an edge (which is slow and sometimes times out, potentially causing precision loss).

For example, the single timeout in the unannotated run of K9Mail occurs on a HashMap-related edge that is refutable, but quite challenging to refute. As it turns out, refuting this edge is extremely important for precision—upon further investigation, we discovered that the analysis would have reduced the number of false alarms reported by over 100 if it had been able to refute it! In the annotated configuration, this edge disappears from the flow-insensitive points-to graph because it stems from HashMap pollution. We can see

Benchmark *Size*		*Filtering* Effectiveness						Computational *Effort*			
Benchmark	**SLOC**	**A**	**RA(%)**	**TruA(%)**	**FalA(%)**	**Fld**	**RFld**	**REdg**	**WEdg**	**TO**	**T (s)**
PulsePoint♠	no src	24	16 (67)	8 (33)	0 (0)	3	2	47	40	1	750
		16	8 (50)	8 (50)	0 (0)	2	1	40	31	0	95
StandupTimer♣	2K	25	15 (60)	0 (0)	10 (40)	5	3	18	26	0	1199
		25	15 (60)	0 (0)	10 (40)	5	3	18	26	0	1068
DroidLife♠	3K	3	0 (0)	3 (100)	0 (0)	1	0	0	4	0	1
		3	0 (0)	3 (100)	0 (0)	1	0	0	4	0	1
OpenSudoku	6K	7	1 (14)	0 (0)	6 (86)	1	0	2	21	1	1596
		0	0 (0)	0 (0)	0 (0)	0	0	0	0	0	0
SMSPopUp♠	7K	5	1 (20)	4 (80)	0 (0)	1	0	10	24	0	49
		5	1 (20)	4 (80)	0 (0)	1	0	10	24	0	46
aMetro♠	20K	144	18 (12)	36 (25)	90 (63)	8	1	62	66	3	4226
		54	18 (33)	36 (67)	0 (0)	3	1	55	24	0	18
K9Mail♠	40K	364	78 (21)	64 (18)	222 (61)	14	3	141	106	1	1130
		208	130 (63)	64 (49)	14 (7)	8	5	124	80	0	374
Total	78K	572	129 (22)	115 (20)	332 (58)	33	9	280	287	6	8991
		311	172 (55)	115 (37)	24 (8)	20	10	247	189	0	1602

Table 4.1: This table characterizes the size of our benchmarks, highlights our success in distinguishing false alarms from real leaks, and quantifies the effort required to find refutations. ♠'s indicate a benchmark in which we found an observable leak, and ♣ indicates a latent leak. The *Size* column grouping gives the number of source lines of code **SLOC** and the number of bytecodes in the call graph **CGB** for each app. The first row for each app gives the results for the annotated configuration, and the second row gives the results for the unannotated configuration. The *Filtering* column grouping characterizes the effectiveness of our approach for filtering false alarms. The first four columns list the number of (static field, Activity) alarm pairs reported by the points-to analysis **A**, number of alarms refuted by our approach **RA**, number of true alarms **TruA**, and the number of false alarms **FalA**. The final two columns of this group give the number of leaky fields reported by the points-to analysis **Fld** and the number of these fields **RFld** that we can refute (i.e., prove that the field in question cannot point to **any** Activity). The *Effort* columns describe the amount of work required by our filtering approach. We list the number of edges refuted **REdg**, edges witnessed **WEdg**, edge timeouts **TO**, and the time **T (s)** taken by the symbolic execution phase in seconds. This number does not include points-to analysis time, which ranged from 8–46 seconds on all benchmarks.

that this increases the number of alarms we are able refute from 78 to 130 even though the number of alarms reported by the flow-insensitive points-to analysis falls from 364 to 208.

We now comment on the *computational effort* required to refute/witness edges. We first observe that the number of edges refuted is almost always greater then the number of alarms refuted, indicating that it is frequently necessary to refute several edges in order to refute a single (source, sink) alarm pair. For example, in the un-annotated run of aMetro, we refute 62 edges in order to refute 18 alarms. This demonstrates that the flow-insensitive points-to analysis is imprecise enough to find many different ways to produce the same false alarm.

We note that the running times are quite reasonable for an analysis at this level of precision, especially in the annotated configuration. No benchmark other than aMetro takes more than a half hour. Our tool would be fast enough to be used in a heavyweight cloud service or as part of an overnight build process.

Real Activity **Leaks.** As hypothesized, our tool's precision enabled us to ignore most false alarms and focus on likely leaks. We found genuine leaks in PulsePoint, DroidLife, SMSPopUp, aMetro, and K9Mail. Many of the leaks we found would only manifest under specialized and complex circumstances, but a few of the nastiest leaks we found would almost always manifest and are due to the same simple problem: an inappropriate use of the singleton pattern. We briefly explain one such leak from the K9Mail app.

In the code in Figure 4.11, the developer uses the singleton pattern to ensure that only one instance of EmailAddressAdapter is ever created. The leak arises when `getInstance()` is called with an Activity instance passed as the `context` parameter (which happens in several places in K9Mail). The Activity instance is passed backwards through the constructors of two superclasses via the `context` parameter until it is finally stored in the `mContext` instance field of the CursorAdapter superclass. For **every** Activity act_0 that calls `getInstance()`, the flow-insensitive points-to analysis reports a heap path $\mathsf{EmailAddressAdapter}\cdot\mathsf{sInstance} \mapsto \mathsf{adr}_0, \mathsf{adr}_0\cdot\mathsf{mContext} \mapsto \mathsf{act}_0$. When the Activity instance is destroyed, the garbage collector will never be able to reclaim it because none of the pointers involved in the leak are ever cleared.

We found this leak in a version of K9Mail that was downloaded in September 2011 (all versions of the benchmarks we used are available in project's GitHub repository). We looked at the current version and

```
public class EmailAddressAdapter extends ResourceCursorAdapter {

  private static EmailAddressAdapter sInstance;

  public static EmailAddressAdapter getInstance(Context context) {
    if (sInstance == null) {
      sInstance = new$_{\mathsf{adr}_0}$ EmailAddressAdapter(context);
    }
    return sInstance;
  }

  private EmailAddressAdapter(Context context) {
    super(context);
  }
}
```

Figure 4.11: A confirmed Activity leak discovered in K9Mail.

noticed that the EmailAddressAdapter class had been refactored to remove the singleton pattern. We found the commit that performed this refactoring and asked the developers of K9Mail if the purpose of this commit was to address a leak issue; they confirmed that it was.[4]

We also discovered a very simple latent leak in StandupTimer that was also due to a bad use of the singleton pattern. We noticed that several of the path programs THRESHER produced for a field in this app would be a full witness for a leak if a single boolean flag `cacheDAOInstances` were enabled. Our tool correctly recognizes that this flag cannot ever be set and refutes the alarm report, but a modification to the program that enabled this flag would result in a leak. The path program witnesses our tool produces are always helpful in triaging reported leak alarms, but in this case even the **refuted** path program witness provided useful information that allowed us to identify an almost-leak. With a less constructive refutation technique, we might have missed this detail.

Utility of our techniques. To test our second set of hypotheses, we ran THRESHER on the benchmarks from Table 4.1 without using each of three key features of our analysis: mixed symbolic-explicit query representation, query simplification, and loop invariant inference. We hypothesized that: (1) using an alternative query representation would negatively affect scalability and/or performance, (2) not simplifying queries would negatively affect scalability and/or performance, and (3) the absence of loop invariant inference would negatively affect precision.

To test hypothesis (1), we implemented a fully symbolic query representation. In a fully symbolic representation, we do not track the set of allocation sites that a variable might belong to. We have up-front points-to information, but use it only to confirm that two symbolic variables are **not** equal (i.e., to prevent aliasing case splits in the style of [Manevich et al., 2004]) and to confirm that a symbolic variable was allocated at a given site (as in WITNEW). This precludes both pruning paths based on the boxed 'from' constraints in Figure 4.8 and performing the entailment check between symbolic variables defined in § in Section 4.3.4.

Using this fully symbolic representation, our analysis ran slower and timed out more often, but did not refute any fewer alarms than the run with the mixed representation. We observed several cases where a

[4] `https://groups.google.com/forum/?fromgroups=#!topic/k-9-mail/JhoXL2c4UfU`

Benchmark	**Ann?**	**T (slowdown)**	**TO (Δ)**
PulsePoint	N	1237 (1.6X)	7 (+6)
	Y	220 (1.9X)	3 (+3)
StandupTimer	N	4946 (4.1X)	4 (+4)
	Y	4104 (3.8X)	4 (+4)
OpenSudoku	N	2984 (1.9X)	4 (+3)
	Y	-	-
SMSPopUp	N	95 (1.9X)	0 (+0)
	Y	76 (1.7X)	0 (+0)
aMetro	N	6863 (1.6X)	5 (+2)
	Y	18 (1X)	0 (+0)
K9Mail	N	990 (0.9X)	2 (+1)
	Y	454 (1.2X)	0 (+0)

Table 4.2: Performance of the fully symbolic representation as compared to the mixed symbolic-explicit representation.

timeout caused the fully symbolic representation to miss refuting an edge that the mixed representation was able to refute, but in each case the edge turned out not to be important for precision (that is, it was one of many edges that needed to be refuted in order to refute an alarm, but both representations failed to refute all of these edges).

The results of this experiment are shown in Table 4.2. We omit the results for DroidLife since they were unaffected by the choice of representation. For every other benchmark, we give the time taken with a fully symbolic representation, the number of times slower than the mixed representation this was (**T (slowdown)**), the number of edges that timed out, and how many timeouts were added over the mixed representation (**TO (Δ)**).

We can see that in both the annotated and un-annotated configurations, most benchmarks run at least 1.6X slower and time out on at least one more edge than they did with the mixed representation. The anomalous behavior of K9Mail in the un-annotated configuration occurs because the mixed representation is able to refute an edge that the fully symbolic representation times out on. Ultimately, this leads the mixed representation to make more progress towards (but ultimately fail in) refuting a particular alarm. The fully symbolic representation declares this particular alarm witnessed after the edge in question times out, which allows it to skip this effort and finish faster.

Thus, hypothesis (1) seems to hold: using a fully symbolic representation negatively affected both performance and scalability as predicted, but choosing a fully symbolic representation did not ultimately affect the precision of the analysis in terms of alarms filtered.

To test hypothesis (2), we re-ran THRESHER on our benchmarks using the annotated Android library without performing any query simplification at all. This significantly hurt the performance of THRESHER on PulsePoint (102.4X slower), K9Mail (3.2X slower), and SMSPopUp (4.3X slower), but did not change the number of alarms refuted or witnessed for these benchmarks. On StandupTimer, not performing simplification caused the tool to run out of memory before completing the analysis, thus affecting both precision and performance. The performance of the tool on other apps was not significantly affected. Thus, hypothesis (2) seems to hold for the benchmarks that require significant computational effort.

Finally, to test hypothesis (3), we implemented a simple loop invariant inference that simply drops all possibly-affected constraints at any loop. With only this simple inference, the analysis was unable to refute some critical HashMap-related edges (using the un-annotated library). This meant that the analysis could never distinguish the contents of different HashMap objects. This imprecision prevented the analysis from refuting leak reports involving multiple HashMap's even on small, hand-written test cases. Our full loop invariant inference (Section 4.3.4) handled the hand-written cases precisely, but due to unrelated analysis limitations, it did not achieve any fewer overall refutations on our real benchmarks. Nevertheless, our testing confirmed hypothesis (3): our loop invariant inference was clearly necessary to properly handle Android HashMap's and similar data structures.

Implementation. THRESHER is built on top of the on the WALA[5] program analysis framework for Java and uses the Z3 [de Moura and Bjørner, 2008] SMT solver with JVM support via ScalaZ3 [Köksal et al., 2011] to determine when path constraints are unsatisfiable.

Like most static analysis tools that handle real-world programs, our tool has a few known sources of unsoundness. We do not reason about reflection or handle concurrency. We have source code for most (but not all) non-native Java library methods. In particular, the Android library custom implementations of core Java library classes (including collections) that we analyze. To focus our reasoning on Android library and application code, we exclude classes from Apache libraries, `java/nio/Charset`, and `java/util/concurrent` from the call graph. Though we track control flow due to thrown exceptions, we do not handle the **catch**`()` construct; instead, we assume that thrown exceptions are never caught.

Android apps are event-driven and (in general) Android event handlers can be called any number of times and in (almost) any order. We use a top-level harness that invokes every event handler defined for an application. Our harness allows event handlers to be invoked in any order, but insists that each handler is called only once in order to prevent termination issues. In our experiments, we did not observe any unsound refutations due to these limitations. We will consider a more accurate model for analysis of event-driven Android programs in Chapter 7.

[5] `http://wala.sf.net`

4.5 Related work

Dillig et al. present precise heap analyses for programs manipulating arrays and containers [Dillig et al., 2010, 2011a], with path and context sensitivity [Dillig et al., 2008]. Our analysis introduces path and context sensitivity via on-demand refinement, in contrast to their exhaustive, summary-based approach. Our symbolic variables are similar to their index variables [Dillig et al., 2010, 2011a] in that both symbolically represent concrete locations and enable lazy case splits. Unlike index variables, our symbolic variables do not distinguish specific array indices or loop iterations, since this was not required for our memory leak client. Also, our analysis does not require container specifications [Dillig et al., 2011a]; instead, we analyze container implementations directly. Hackett and Aiken [Hackett and Aiken, 2006] present a points-to analysis with intra-procedural path sensitivity, which is insufficient for our needs.

Several previous systems focused on performing effective backward symbolic analysis. The pioneering ESC/Java system [Flanagan et al., 2002b] performed intra-procedural backward analysis, generating a polyno-mially-sized verification condition and checking its validity with a theorem prover. Snugglebug [Chandra et al., 2009] performed inter-procedural backward symbolic analysis, employing directed call graph construction and custom simplifiers to improve scalability. Cousot et al. [Cousot et al., 2011] present backward symbolic analysis as one of a suite of techniques for transforming intermittent assertions in a method into executable pre-condition checks. PSE [Manevich et al., 2004] used backward symbolic analysis to help explain program failures, but for greater scalability, it did not represent full path conditions. Our work is distinguished from these previous systems by the integration of points-to analysis information, which enables key optimizations like mixed symbolic-explicit states and abstraction for loop handling.

Our analysis can be seen as refining the initial flow-insensitive abstraction of the points-to analysis based on a "counterexample" reachability query deemed feasible by that analysis. However, instead of gradually refining this abstraction as in, for example, counterexample-based abstraction refinement (CEGAR) [Clarke et al., 2003] and related techniques [Cousot et al., 2007] , our technique immediately employs concrete reasoning about the program, and then re-introduces abstraction as needed (e.g., to handle loops). In general, the above predicate-abstraction-based approaches have not been shown to work well for proving properties of object-oriented programs, which present additional challenges due to intensive heap usage, frequent virtual dispatch, etc. Architecturally, our system is more similar to recent staged analyses for typestate verification [Dor et al., 2004; Fink et al., 2008], but our system employs greater path sensitivity and more deeply integrates points-to facts from the initial analysis stage. A path program [Beyer et al., 2007] was originally defined in the context of improving CEGAR by pruning multiple counterexample traces through a loop at once. SMPP [Harris et al., 2010] performs SMT-based verification by exhaustively enumerating path programs in a forward-chaining manner (in contrast to our goal-directed search). The recent DASH system [Beckman et al., 2008] refines its abstraction based on information from dynamic runs and employs dynamic information to reduce explosion due to aliasing.

Our witness-refutation search uses the "bounded" fragment of separation logic [Reynolds, 2002] and thus has a peripheral connection to recent separation-logic–based shape analyzers [Berdine et al., 2007; Chang and Rival, 2008]. In contrast to such analyzers, we do not use inductive summaries and instead use materializations from a static points-to analysis abstraction. Shape analysis using bi-abductive inference [Calcagno et al., 2009] enables a compositional analysis by deriving pre- and post-conditions for methods in a bottom-up manner and making a best effort to reach top-level entry points. The derivation of heap pre-conditions is somewhat similar to our witness-refutation search over points-to constraints, but our backwards analysis is applied on demand from a flow-insensitive query and is refined by incorporating information from an up-front, whole program points-to analysis. Recent work [Distefano and Filipović, 2010] has applied bi-abduction to detect real Java memory leaks in the sense of an object that is allocated but never used again. In contrast, our client is a flow-insensitive heap reachability property that over-approximates a leak that is not explicit in the code, but is realized in the Android run-time.

Similar to our path program witnesses, other techniques have aimed to either produce a concrete path witness for some program error or help the user to discover one. Bourdoncle [Bourdoncle, 1993a] presents a system for "abstract debugging" of program assertions, in which the compiler aims to discover inputs leading to violations statically. Rival [Rival, 2005] presents a system based on combined forward and backward analysis for elucidating and validating error reports from the Astrée system [Cousot et al., 2005]. Work by Ball et al. [Ball et al., 2005] observes that for showing the **existence** of program errors (as opposed to verifying their absence), a non-standard notion of abstraction suffices in which one only requires the existence of a concrete state satisfying any particular property of the corresponding abstract state (as opposed to all corresponding concrete states satisfying the property). We observe an analogous difference between refutation and witness discovery in Section **??**. Similar notions underlie the "proof obligation queries" and "failure witness queries" in recent work on error diagnosis [Dillig et al., 2012].

Previous points-to analyses have included refinement to improve precision. Guyer and Lin's client-driven pointer analysis [Guyer and Lin, 2005] introduced context and flow sensitivity at possibly-polluting program points based on client needs. Sridharan and Bodik [Sridharan and Bodík, 2006] presented an approach for adding field and context sensitivity to a Java points-to analysis via refinement. Recently, Liang et al. [Liang and Naik, 2011; Liang et al., 2010, 2011] have shown that highly-targeted refinements of a heap abstraction can yield sufficient precision for certain clients. Unlike our work, none of the aforementioned techniques can introduce path sensitivity via refinement. A recent study on the precision of Andersen's analysis [Blackshear et al., 2011] used dependency rules akin to a fully-explicit analog of our mixed symbolic-explicit transfer functions in a flow-insensitive context.

Finally, more recent work has addressed the problem of discovering Android Context leaks using dynamic analysis. The LEAKCANARY[6] tool from Square watches instances of Activity's and (optionally) user-defined types to ensure that all references to such types are destroyed at the end of certain methods (e.g., the Activity.`onDestroy()` method). Square developers report that the tool has helped them identify and eliminate numerous memory leaks, reducing runtime crashes due to `OutOfMemoryError`'s in the Square Android app by 94%[7] .

[6] `https://github.com/square/leakcanary`
[7] `https://corner.squareup.com/2015/05/leak-canary.html`

Chapter 5

Jumping: a framework for goal-directed control-flow abstraction

Chapter 4 addressed the problem of goal-directed store abstraction. We now turn our attention to the problem of control-flow abstraction, the second dimension of abstraction defined in Section 2.1. Our approach to this problem is the primary theoretical contribution of this thesis: a framework for goal-directed control-flow abstraction via **jumping**. The framework provides a mechanism by which any overapproximate backward abstract interpretation can soundly perform jumps over irrelevant code given a **relevance relation** that meets certain soundness conditions. This framework reflects our philosophical separation between the problems of store abstraction and control-flow abstraction. Roughly, we view store abstraction is the problem of how we soundly process the core commands c of the language (our framework for jumping is parametric in both these commands and their corresponding transfer functions) and we view control-flow abstraction as the problem of soundly deciding what commands the analysis should visit next (our framework is parametric in a relevance relation that chooses such commands up to soundness criteria given in Section 5.3.2).

In this chapter, we formalize our framework for jumping analyses, prove its soundness, and argue for its generality in expressing flexible abstractions. Section 5.1 defines a simple unstructured language that allows uniform reasoning about control-flow. We give inference rules for a jumping analysis parameterized by a store abstraction and a relevance relation (Section 5.2). Section 5.3 introduces the notions of **data-relevance** (what commands are relevant to the current query?) and **control-feasibility** (what commands can be executed before the current program point?) as building blocks for thinking about control-flow abstraction (Section 5.3.1) and use these concepts to define soundness conditions for a relevance relation in

Section 5.3.2. Section 5.4 uses this condition to state and prove the soundness of jumping based on a sound relevance relation.

Portions of this chapter previously appeared in a paper draft currently under submission entitled “Selective Control-Flow Abstraction via Jumping”, which was co-authored by Bor-Yuh Evan Chang and Manu Sridharan.

5.1 A simple unstructured programming language

We consider the imperative programming language of commands and unstructured control-flow presented in Figure 5.1. We choose an unstructured representation because it “flattens” the program by lowering all structured control-flow constructs (e.g. `if`’s, `while`’s, `switch`’s, function calls, etc.) into a uniform representation: transitions between atomic commands (assignment, allocation, etc.). Since this language represents all control flow in the same way, reasoning about control-flow and control-flow abstraction is simple and cleanly separated from reasoning about commands.

A program in our language consists of a finite set of transitions t. We use P for the program of interest and T for a set of transitions in P. A transition $\ell_1 \dashv\! [c] \!\rightarrow \ell_2$ consists of a pre-label ℓ_1, a command c, and a post-label ℓ_2. A well-formed program must contain a no-op initialization transition $t_{\mathsf{init}} : \ell_{\mathsf{dummy}} \dashv\! [\mathtt{skip}] \!\rightarrow \ell_{\mathsf{entry}}$.

5.1.1 Concrete semantics

Our framework for jumping is parametric in a command language for manipulating stores and an abstraction of concrete states. We assume that the concrete semantics of commands are provided via a judgment form $\langle \sigma, c \rangle \Downarrow \sigma'$ that specifies how c transforms a concrete state σ to another state σ'. For example, we could extend the c production of Figure 5.1 with the commands from Figure 4.5, and use the concrete semantics for the commands of this language from Figure 4.6. However, we will define the concrete semantics for the special control-manipulating commands in Figure 5.1 to clarify how conditionals, loops, and procedure calls are represented in our framework.

Concrete semantics for control-manipulating commands. In Figure 5.2, we define the concrete semantics for a few special commands (`assume`, `call`, `return`, and `skip`) that manipulate the program’s

programs	$P, T ::= \{t_1, \ldots, t_n\}$
transitions	$t ::= \ell_1 \dashv\!\![c]\!\!\rightarrow \ell_2$
commands	$c ::= \texttt{skip} \mid \texttt{assume}\, e \mid \texttt{call}\ \ell \mid \texttt{return}\ \ell \mid \ldots$
program labels	$\ell \in \mathbf{Label}$
call strings	$L \in \mathbf{Strings} ::= [\,] \mid \ell :: L$
abstract call strings	$\hat{L} \in \widehat{\mathbf{Strings}}$
concrete stores	$\rho \in \mathbf{Store}$
concrete states	$\sigma \in \Sigma \quad ::= (\rho, L)$
abstract stores	$\hat{\rho} \in \widehat{\mathbf{Store}}$
abstract states	$R \in \hat{\Sigma} \quad ::= \top \mid \bot \mid (\hat{\rho}, \hat{L}) \mid R_1 \vee R_2$
command semantics	$\langle \sigma, c \rangle \Downarrow \sigma'$
abstract semantics	$\vdash \{R_{\text{pre}}\}\ c\ \{R_{\text{post}}\}$
concretization	$\gamma : \hat{\Sigma} \to \wp(\Sigma)$
invariant map	$I : \mathbf{Label} \to \hat{\Sigma}$

Figure 5.1: A language composed of atomic commands connected by unstructured control flow.

$$\boxed{\langle \sigma, c\rangle \Downarrow \sigma'}$$

C-ASSUME
$$\frac{\langle \sigma, e\rangle \Downarrow \mathsf{true}}{\langle \sigma, \texttt{assume}\, e\rangle \Downarrow \sigma}$$

C-CALL
$$\frac{}{\langle (\rho, L), \texttt{call}\ \ell\rangle \Downarrow (\rho, \ell :: L)}$$

C-RETURN
$$\frac{\ell_1 = \ell_2}{\langle (\rho, \ell_1 :: L), \texttt{return}\ \ell_2\rangle \Downarrow (\rho, L)}$$

$$\boxed{\langle \sigma, \ell\rangle \xrightarrow[t]{} \langle \sigma', \ell'\rangle}$$

C-TRANS
$$\frac{t = \ell \dashv\!\! [c] \!\!\rightarrow \ell' \quad \langle \sigma, c\rangle \Downarrow \sigma'}{\langle \sigma, \ell\rangle \xrightarrow[t]{} \langle \sigma', \ell'\rangle}$$

$$\boxed{\langle \sigma_{\mathrm{pre}}, \ell_{\mathrm{pre}}\rangle \xrightarrow[T]{}^{*} \langle \sigma_{\mathrm{post}}, \ell_{\mathrm{post}}\rangle}$$

C-STEP
$$\frac{\langle \sigma_{\mathrm{pre}}, \ell_{\mathrm{pre}}\rangle \xrightarrow[T]{}^{*} \langle \sigma', \ell'\rangle \quad \langle \sigma', \ell'\rangle \xrightarrow[t]{} \langle \sigma_{\mathrm{post}}, \ell_{\mathrm{post}}\rangle}{\langle \sigma_{\mathrm{pre}}, \ell_{\mathrm{pre}}\rangle \xrightarrow[T \,\hat{}\, t]{}^{*} \langle \sigma_{\mathrm{post}}, \ell_{\mathrm{post}}\rangle}$$

C-STOP
$$\frac{}{\langle \sigma, \ell\rangle \xrightarrow[\emptyset]{}^{*} \langle \sigma, \ell\rangle}$$

Figure 5.2: Small-step concrete semantics for the simple unstructured language of Figure 5.1.

control flow rather than the concrete store. These commands are part of the $\langle \sigma, c\rangle \Downarrow \sigma'$ judgement form, which asserts "The concrete state σ steps to the concrete state σ' by evaluating command c".

Programs in our language encode conditional branching using an $\texttt{assume}\,e$ command that blocks unless e evaluates to `true` (as specified by the C-ASSUME rule). We encode looping using `assume` along with back edges in the transition relation.

For example, a structured conditional statement `if (`e`)` c_1 `else` c_2 would be represented by the set of transitions $\{\ \ell_1 \dashv\![\texttt{assume}\,e]\!\mapsto \ell_2,\ \ell_2 \dashv\![c_1]\!\mapsto \ell_4,\ \ell_1 \dashv\![\texttt{assume}\,\neg e]\!\mapsto \ell_3,\ \ell_3 \dashv\![c_2]\!\mapsto \ell_4\ \}$. Similarly, the structured loop statement `while (`e`)` c would be represented by the set of transitions $\{\ \ell_1 \dashv\![\texttt{assume}\,e]\!\mapsto \ell_2,$ $\ell_2 \dashv\![c]\!\mapsto \ell_1,\ \ell_1 \dashv\![\texttt{assume}\,\neg e]\!\mapsto \ell_3\ \}$.

We represent procedure calls using `call` and `return` commands that are linked to (respectively) callee procedures and caller sites in the program transitions. Both commands manipulate a call string L composed of program labels. The command $\texttt{call}\ \ell$ (C-CALL) prepends the return label ℓ onto the call string, while the $\texttt{return}\ \ell$ command blocks unless ℓ matches the first label in the call string (ℓ_1). (C-RETURN). Transitions involving these commands have special well-formedness conditions: a transition $\ell_1 \dashv\![\texttt{call}\ \ell]\!\mapsto \ell_2$ is well-formed when ℓ is the pre-label of the instruction to be executed after the call returns, and transition $\ell_1 \dashv\![\texttt{return}\ \ell]\!\mapsto \ell_2$ is well-formed when $\ell = \ell_2$ (i.e., ℓ is the return site). For example, the two-procedure program `foo() { bar(); bar(); skip } bar() { skip; }` would be represented by the sets of transitions $\{\ \ell_1 \dashv\![\texttt{call}\ \ell_2]\!\mapsto \ell_5,\ \ell_2 \dashv\![\texttt{call}\ \ell_3]\!\mapsto \ell_5,\ \ell_3 \dashv\![\texttt{skip}]\!\mapsto \ell_4\ \}$ (for `foo`), and $\{\ \ell_5 \dashv\![\texttt{skip}]\!\mapsto \ell_6,$ $\ell_6 \dashv\![\texttt{return}\ \ell_2]\!\mapsto \ell_2,\ \ell_6 \dashv\![\texttt{return}\ \ell_3]\!\mapsto \ell_3\ \}$ (for `bar`).

For simplicity in presentation, we assume that all variables in the program are globally scoped and that parameter binding is accomplished via assignment of caller-owned globals to callee-owned globals.

Small-step concrete semantics for transition systems. Figure 5.2 also defines a transition relation for a small-step operational semantics of transition systems given by the judgment form $\langle \sigma, \ell\rangle \xrightarrow[t]{} \langle \sigma', \ell'\rangle$. This judgment form is defined by applying the concrete command semantics $\langle \sigma, c\rangle \Downarrow \sigma'$ to transition $t\colon \ell \dashv\![c]\!\mapsto \ell'$. A judgment $\langle \sigma, \ell\rangle \xrightarrow[t]{} \langle \sigma', \ell'\rangle$ is well-formed only if the pre- and post-labels of t are ℓ and ℓ', respectively.

The judgment form $\langle \sigma, \ell\rangle \xrightarrow[t]{} \langle \sigma', \ell'\rangle$ asserts "The concrete state σ at label ℓ steps to a concrete state

σ' at label ℓ' by executing transition t". The judgment form is defined by the single rule C-TRANS, which simply delegates to the concrete semantics for commands to execute the command attached to the transition t.

The judgment form $\langle \sigma, \ell \rangle \xrightarrow[T]{}^* \langle \sigma', \ell' \rangle$ is the multi-step transition relation; that is, the reflexive-transitive closure of the single-step transition relation. This judgment form asserts "The concrete state σ at label ℓ steps to a concrete state σ' at label ℓ' by executing each of the transitions in T". We overload the meta-variable T to be sequence of transitions (i.e., a trace) rather than a set, and we write $T \mathbin{\hat{}} t$ for adding the transition t to the end of he trace T. We also sometimes abuse the set operator $\in$ by lifting it to an operation on traces with the expected semantics.

The $\langle \sigma, \ell \rangle \xrightarrow[T]{}^* \langle \sigma', \ell' \rangle$ judgment form is defined by the C-STOP and C-STEP rules. The C-STOP rule expresses the reflexive property of the judgment form; a concrete state σ at ℓ steps to the same state and label by executing no transitions. The C-STEP rule says that if the state σ_{pre} at label ℓ_{pre} steps to the concrete state σ' and label ℓ' by executing the transitions in T and then takes a single step across transition t to state σ_{post} and label σ_{post}, then the state σ_{pre} at label ℓ_{pre} steps to σ_{post} and label σ_{post} by executing the transitions in $T \mathbin{\hat{}} \{t\}$.

5.2 Formalizing jumping analysis

In this section, we explain the (parameterized) abstract state used by our analysis (Section 5.2.1) and formalize an abstract semantics for a backward abstract interpretation augmented with the ability to jump.

5.2.1 Abstract state

Continuing our discussion of Figure 5.1, we write R for an abstract state that over-approximates a set of concrete states defined by a concretization γ. Notationally, we use a semantic entailment relation $R_1 \models R_2$ defined over concretization as $\gamma(R_1) \subseteq \gamma(R_2)$. We write $\top$ for the state such that $\gamma(\top) \overset{\text{def}}{=} \Sigma$ and $\bot$ for the state such that $\gamma(\bot) \overset{\text{def}}{=} \emptyset$. Otherwise, a state is a finite disjunction of pairs of a store abstraction $\hat{\rho}$ and a call string abstraction $\hat{L}$. We leave the particular store and call string abstractions of interest unspecified.

Our framework takes a sound abstract semantics for commands represented by the judgment form

$\vdash \{R_{\text{pre}}\}\ c\ \{R_{\text{post}}\}$ as a parameter (for example, we could use the abstract semantics from Figure 4.8). This judgment form is a backward Hoare triple stating that for all concrete post-states in R_{post} in which c terminates for some concrete pre-state, then that concrete pre-state is in R_{pre}. More formally, the abstract semantics must satisfy the following soundness condition:

$$\text{If } \vdash \{R_{\text{pre}}\}\ c\ \{R_{\text{post}}\} \text{ and } \langle \sigma_{\text{pre}}, c\rangle \Downarrow \sigma_{\text{post}} \text{ such that } \sigma_{\text{post}} \in \gamma(R_{\text{post}}), \text{ then } \sigma_{\text{pre}} \in \gamma(R_{\text{pre}}). \qquad (*)$$

As an informal shorthand, we say R_{post} is **may-witnessed** by executing the command c from R_{pre}. As a corollary of this soundness condition, if the analysis **refutes** an input query R_{post} (i.e., derives $\bot$ on all backward paths originating from R_{post}), then R_{post} represents a set of unreachable concrete states. However, the analysis may over-approximate by failing to refute R_{post} even if R_{post} is not concretely reachable.

5.2.2 Abstract semantics: control-flow abstraction via jumping

To describe static analysis of transition systems, we define an invariant map $I : \textbf{Label} \to \hat{\Sigma}$ that maps each program label ℓ to candidate invariants at ℓ given by an abstract state R. Our jumping refutation analysis is defined by the judgment form $I \vdash \ell$ that asserts, "I over-approximates the concrete states from which ℓ can be reached in a state satisfying $I(\ell)$." As a shorthand, when the judgment $I \vdash \ell$ holds, we say that I **may-witnesses** $I(\ell)$, or simply I **may-witnesses** ℓ.

This judgment form relies on an auxiliary judgment form $I \vdash t$ that asserts, "For a transition $t : \ell_1 \dashv\!\! [c] \!\!\mapsto \ell_2$, $I(\ell_1)$ overapproximates the concrete states from which executing c yields a state satisfying $I(\ell_2)$." As above, we say that I **may-witnesses** transition t when the judgment $I \vdash t$ holds.

In Figure 5.3, we define these two judgment forms. The A-TRANSITION rule defines $I \vdash t$, which is analogous to the consequence rule of standard Floyd-Hoare logic. The rule says that I may-witnesses $\ell_1 \dashv\!\! [c] \!\!\mapsto \ell_2$ if there is a triple $\vdash \{R'\}\ c\ \{R\}$ that satisfies soundness condition $(*)$, such that $I(\ell_2)$ is stronger than R and $I(\ell_1)$ is weaker than R'. This rule is essentially just a wrapper that lifts an abstract semantics for commands to an abstract semantics for transitions that is constrained by our invariant map I.

The key rule for our jumping analysis is A-JUMP, which decides the transitions that the analysis should visit next. This rule relies on a relevance relation written using the judgment form $\langle R, \ell_{\text{post}}\rangle \rightsquigarrow T_{\text{rel}}$

$$\boxed{I \vdash \ell_1 \mathrel{-\!\![c]\!\!\rightarrow} \ell_2}$$

A-TRANSITION
$$\frac{I(\ell_2) \models R \quad \vdash \{R'\}\, c\, \{R\} \quad R' \models I(\ell_1)}{I \vdash \ell_1 \mathrel{-\!\![c]\!\!\rightarrow} \ell_2}$$

$$\boxed{I \vdash \ell}$$

A-JUMP
$$\frac{I(\ell_{\text{post}}) \models R \quad \langle R, \ell_{\text{post}} \rangle \rightsquigarrow T_{\text{rel}} \quad I \vdash t \text{ for all } t\colon \ell_i \mathrel{-\!\![c_{ij}]\!\!\rightarrow} \ell_j \in T_{\text{rel}} \qquad R \models I(\ell_j) \text{ for all } \ell_j \quad I \vdash \ell_i \text{ for all } \ell_i}{I \vdash \ell_{\text{post}}}$$

Figure 5.3: Jumping analysis. The key A-JUMP rule expresses the ability to skip code based on a relevance relation.

that asserts, "Given an abstract state R at program label ℓ, the set of relevant program transitions is T_{rel}." Intuitively, the rule says to perform a backward jump from the current label ℓ_{post} to the post-label of each relevant transition in T_{rel}, skipping all transitions in between.

The rule's first two premises $I(\ell_{\text{post}}) \models R$ and $\langle R, \ell_{\text{post}} \rangle \rightsquigarrow T_{\text{rel}}$ state that we compute a set of relevant transitions T_{rel} using some weakening of the query $I(\ell_{\text{post}})$. Allowing this weakening of the state abstraction is crucial, as weakening of the state may make the set of relevant transitions T_{rel} smaller. The "$R \models I(\ell_j)$ for all ℓ_j" premise constrains the post-state of each relevant transition to be weaker than the current state R. Together, these two premises can be seen as consequence for the transitions skipped by the jump.

The premise "$I \vdash t$ for all $t\colon \ell_i \mathrel{-\!\![c_{ij}]\!\!\rightarrow} \ell_j \in T_{\text{rel}}$" checks that I may-witnesses each relevant transition $t \in T_{\text{rel}}$—that is, it uses the auxiliary judgment form to abstractly execute each relevant transition that was jumped to. Finally, the remaining premise "$I \vdash \ell_i$ for all ℓ_i" recursively continues the backward analysis by checking that I may-witness the pre-label ℓ_i of each relevant transition that was jumped to.

5.2.3 Inference, loops, and recursion

While the judgment form $I \vdash \ell$ is most easily read as a checking system for judging when I may-witnesses ℓ for a given I, we can obtain an inference system that computes an invariant map I with a standard post-fixed-point computation via abstract interpretation. We begin the abstract interpretation from a map I_0 initialized with the initial query at the start label ℓ and all other labels mapping to $\perp$. The analysis applies the A-JUMP rule to that start label and updates the invariant map with the inferred values for R. A weakening (as in premise $R' \models I(\ell_1)$ of A-TRANSITION) corresponds to an update to the invariant map with a join (i.e., $I_{i+1}(\ell_1) = I_i(\ell_1) \sqcup R'$) or widen $\triangledown$ as appropriate to break loops in the abstract interpretation. This process continues with additional labels via the recursive invocation "$I \vdash \ell_i$ for all ℓ_i" in the A-JUMP rule until the invariant map computation reaches a fixed point.

In the analysis, an arbitrary context- and object-sensitivity policy can be implemented by the choice of the call string abstraction $\hat{L} \in \hat{\textbf{Strings}}$ and the state abstraction $R \in \hat{\Sigma}$.For example, a simple k-callstring context-sensitivity policy could keep a disjunct for distinct call strings up to length k (joining or widening abstract stores $\hat{\rho}$ as needed).

In our implementation, we uniformly handle all sources of looping and recursion by widening at targets of back edges. Our widening operator bounds the length of the materialized prefix of the abstract call string $\hat{L}$ (i.e., program labels $\ell_1 :: \cdots :: \ell_k :: \mathsf{anystring}$) and the number of materialized heap locations (i.e., $\mapsto$ constraints) in the abstract store $\hat{\rho}$.

5.3 Data-relevance, control feasibility, and soundness of a relevance relation

The A-JUMP rule of Figure 5.3 is an extremely general rule that allows a wide variety of strategies for choosing the transitions that the analysis should jump to. All transitions not jumped to are skipped. But what transitions can the analysis soundly skip? A-JUMP allows the analysis to skip any transitions except for the set of transitions T_{rel} returned by the relevance relation, so the burden of ensuring soundness falls squarely upon this relation. In this section, we will first build intuition for what transitions can and cannot be skipped (Section 5.3.1) before formally defining the soundness conditions that must imposed on a relevance relation in order to ensure sound jumping (Section 5.3.2).

5.3.1 Building intuition for data-relevance and control-feasibility

For a relevance relation to be sound, it must ensure that it does not omit any important transitions that could be involved in may-witnessing the query of interest. There are many different strategies that the relevance relation can use to ensure this soundness property. We will show that each of these strategies can be thought of as (1) choosing a set of **data-relevant** transitions that would be sound to return on their own, then (2) soundly filtering this set using **control-feasibility** information. As a first consideration, consider a relevance relation that returns all transitions in the program as data-relevant and performs no filtering. This relevance relation is trivially sound: it cannot skip any important transitions because it does not skip any transitions at all. This corresponds to a fully flow- and context-insensitive view of the program, as every transition will be a jump target from every other transition.

Simple control-feasibility: choose the predecessor transitions of the current program label. The above strategy of taking all program transitions can be improved by considering a simple form of control-feasibility: postdominance in the control-flow graph. Intuitively, if transition t' postdominates transition t, there is no need to consider both t and t' as relevant, as all backward paths to t must go through t'. Hence, it is sufficient to only consider t' as relevant. Via this reasoning, we can conclude that instead of treating all transitions in the program as relevant, one can instead use just the immediate predecessor transitions of the current program label while remaining sound. That is, let us define $\langle R, \ell_{\text{cur}} \rangle \rightsquigarrow \mathbf{preds}(\ell_{\text{cur}})$. We have simply recovered the standard approach taken by flow/path-sensitive analyses—visit all of the predecessor (for a backward analysis) or successor (for a forward analysis) transitions of the current program label ℓ. This is the approach taken by the THRESHER tool described in Chapter 4.

Simple data-relevance strategy: choose the transitions that may affect the current query. Treating just the immediate predecessors as relevant is quite precise, but it does not utilize the key strength of jumping: the ability to skip irrelevant transitions entirely. We can make better use of jumping by refining the set of transitions returned by the data-relevance step using information about the program's data-flow. We can leverage data-flow information by modifying the data-relevance step to return all transitions that may affect the query state as data-relevant, then remain sound by performing no filtering during the control-feasibility step. For example, if the abstract state R is $\mathtt{x} \geq 0$ for a program variable `x`, then writes to any variable other than `x` clearly cannot affect the query state and can be soundly skipped. That is, we can define $\langle R, \ell_{\text{cur}} \rangle \rightsquigarrow \mathbf{mods}(R)$, where **mods** denotes the commands that may write to the current abstract state R.

An interesting aspect of our framework is that we can actually be a bit more restrictive than the mod-set by considering only transitions that **weaken** the abstract state to be relevant. The informal intuition is that if a transition moves a query closer to being may-witnessed (i.e., weakens the query), then it cannot be skipped, whereas if a transition moves a query closer to being refuted (i.e., strengthens the query) or does not change the query, it may be soundly skipped. Consider a simple example to develop intuition for this fact:

Example 5.3.1 (Relevance) What commands are safe to skip for the query $\mathtt{x} = 5$ at some program point in a program containing the commands `y := 5`, `x := 3`, `x := y`, and `assume y != 5`? Informally, we cannot skip the command `x := y`, as it may move the query closer to being may-witnessed depending on the value of `y`. Skipping the command `y := 5` is clearly safe since it will not affect the query. Skipping `x := 3` is also safe because although this command affects the query, it would refute the query rather than moving it closer to being may-witnessed. Finally, skipping `assume y != 5` is safe as well because although it affects the query, it moves it closer to being refuted. Considering this transition would yield the stronger sub-query $\mathtt{x} = 5 \wedge \mathtt{y} \neq 5$.

Thus, it is sound to choose the subset of modifier commands that weaken the query as relevant; that is, we can define $\langle R, \ell_{\text{cur}} \rangle \rightsquigarrow \{\, t : \ell_1 \dashv c \mapsto \ell_2 \mid\ \vdash \{R'\}\, c\, \{R\} \text{ and } R' \models R \,\}$.

This strategy of taking the set of data-relevant transitions (either modifiers or weakeners) is less precise than taking the set of the immediate predecessors. An analysis that uses the data-relevance strategy will lose flow-sensitivity while jumping because it does not take the program's control-flow into account. On the other hand, the data-relevance strategy is likely to be more efficient because it considers only the (typically small) set of transitions that may affect the query without reasoning about any of the other transitions in the program or the control-flow between them.

Combining data-relevance and control-feasibility A very powerful strategy is combining the previous two: first identify a set of data-relevant transitions for the current query, then use control-feasibility information such as postdominance in the control-flow graph to filter this set as much as possible. Roughly, we can define $\langle R, \ell_{\text{cur}} \rangle \rightsquigarrow \{\ t \mid t \in \textbf{dataRel}(R) \text{ and } \textbf{controlFeas}(t, \ell_{\text{cur}})\ \}$. Doing this allows us to get the best of both worlds while jumping: we can skip vast swaths of irrelevant code by limiting our consideration to the data-relevant transitions, and we can maintain flow-, path-, and context-sensitivity while jumping by filtering away control-infeasible transitions using information about the control-flow between data-relevant transitions.

5.3.2 Defining relevance soundness

Motivated by the preceding discussion of sound strategies for selecting relevant transitions to explore, we define our soundness condition for relevance relations in a way that permits all strategies to be thought of as computing data-relevant transitions, then filtering these conditions using control-feasibility:

Condition 5.3.1 (Relevance soundness)
If $\langle R, \ell_{\text{post}} \rangle \rightsquigarrow T_{\text{rel}}$, $\langle \sigma, \ell_{\text{pre}} \rangle \xrightarrow[T]{}^{*} \langle \sigma', \ell_{\text{post}} \rangle$, $t_{\text{irrel}}\colon\ \ell_1 \dashv\!\! [c] \!\!\mapsto \ell_2 \in P - T_{\text{rel}}$, and $\vdash \{R'\}\ c\ \{R\}$, then either (a) $R' \models R$, or (b) $\exists\ T_1, T_2$ s.t. $T = T_1 \mathbin{\hat{}} T_2$, $t_{\text{irrel}} \notin T_2$ and $T_{\text{rel}} \cap T_2 \neq \emptyset$.

We write $\langle \sigma, \ell \rangle \xrightarrow[T]{}^{*} \langle \sigma', \ell' \rangle$ for the judgment form of multi-step concrete evaluation, that is, the reflexive-transitive closure of single-step concrete evaluation. This multi-step concrete evaluation records each transition it visits between ℓ_{pre} and ℓ_{post} in the trace T. We denote trace concatenation with $T_1 \mathbin{\hat{}} T_2$.

Data-relevance condition Condition 5.3.1(a) captures the soundness of returning data-relevant transitions by imposing restrictions on a transition t_{irrel} that is **not** returned by the relevance relation. It states that for any program transition t_{irrel} not included in the set of relevant transitions for state R, the pre-state R' with respect to the transition command c is at least as strong as R. From the analysis perspective, this means that it is sound to exclude t_{irrel} from the jump targets because it cannot possibly move R closer to being witnessed. This condition ensures that the returned relevant transitions T_{rel} must be a superset of the transitions that either weaken the input abstract state do not satisfy Condition 5.3.1(b) R.

This is a very general notion of data-relevance, as it captures relevance based on both modification and weakening. Changing Condition 5.3.1(a) to $R_{\text{pre}} = R$ would require that all skipped transitions do not change the query. As previously mentioned, this is also sound, but is less general because does not allow us to skip transitions such as `assume`'s and `return`'s that strengthen the query (by adding path constraints and constraining the call string, respectively). Skipping these transitions is sometimes desirable: doing so may lose precision, but enhance scalability by yielding fewer relevant transitions for the analysis to jump to. Using the more general weakening condition in Condition 5.3.1(a) allows a particular relevance relation to choose to exclude such transitions or not as desired for the requirements of the analysis.

Control-feasibility condition Condition 5.3.1(b) captures the soundness of filtering data-relevant transitions based on control-feasibility information. It says that if a transition t_{irrel} is not included in the set of relevant transitions for program point ℓ_{post}, then we can decompose the trace of visited transitions T into a pre-trace T_1 and a post-trace T_2 such that the post-trace contains a relevant transition, but does not contain t_{irrel}. This means then some relevant transition $t_{\text{rel}} \in T_{\text{rel}}$ must always happen **between** t_{irrel} and ℓ_{post}. From the perspective of our backward analysis, this means that it is sound to exclude t_{irrel} from the set of jump targets because some relevant transition t_{rel} that will be jumped to postdominates t_{irrel} in the program control-flow.

Our relevance soundness condition is quite general: it permits all of the transition selection strategies we have discussed so far and opens the door for any strategy whose structure can be described as computing data-relevant transitions, then filtering using control-feasibility.

5.4 Soundness of jumping analysis

Next, we state and prove a soundness theorem demonstrating that any relevance relation satisfying this soundness condition can be used to define a sound jumping analysis.

Theorem 5.4.1 (Soundness of jumping analysis)

If $\langle \sigma_{\mathsf{dummy}}, \ell_{\mathsf{dummy}} \rangle \xrightarrow[T]{}^{*} \langle \sigma_{\text{post}}, \ell_{\text{post}} \rangle$ and $I \vdash \ell_{\text{post}}$ such that $\sigma_{\text{post}} \in \gamma(I(\ell_{\text{post}}))$, then $\sigma_{\mathsf{dummy}} \in \gamma(I(\ell_{\mathsf{dummy}}))$.

The theorem says that if the concrete state σ_{post} at program point ℓ_{post} is in the concretization of the abstract state stored at label ℓ_{post} of the invariant map, then σ_{dummy} is in the concretization of the abstract state stored at the pre-entry label ℓ_{dummy}. In the theorem, we write concrete state σ_{dummy} for a distinguished element of Σ that represents the uninitialized, "junk" state before beginning the execution of the program. Using this concrete junk state as a technical device, our instantiation defines an abstraction such that the only abstract state that includes σ_{dummy} is $\top$. Thus, we obtain a refutation (i.e., we have discovered that the program configuration $\langle \sigma_{\text{post}}, \ell_{\text{post}} \rangle$ is unreachable) whenever $I(\ell_{\mathsf{dummy}})$ maps to a non-$\top$ state.

We present the proof of this theorem in Appendix A.

Chapter 6

Combining jumping with invariant-based reasoning

Chapters 4 and 5 present approaches to flexible store and control-flow abstraction (respectively) based on the idea of abstraction coarsening. These chapters explain how we meet the challenge of making abstractions flexible (Challenge A), but the challenge of using flexibility wisely to design practically effective analyses (Challenge B) remains. In this chapter, we show how to instantiate the control-flow abstraction framework from Chapter 5 with the store abstraction from Chapter 4 and a relevance relation specialized for refuting reachability queries in modern object-oriented programs. Our goal is to use the framework to design an analysis that leverages common forms of invariant-based reasoning used by real-world programmers for better scalability. From the perspective of our goal-directed analysis via abstraction coarsening, we seek to outfit the THRESHER analysis described in Chapter 4 with the ability to coarsen its control-flow abstraction on-the-fly. The analysis will decide to coarsen the control-flow abstraction and improve scalability (hopefully without losing precision) by choosing to explore **data-relevant** commands whenever the analysis guesses that the safety of the query may depend on a flow-insensitive invariant. The hope is that the analysis will retain the precision to quickly refute the query based on local information if possible, but will also identify cases where safety relies on an invariant established far away from the program point of the initial query and improve scalability by **jumping** directly to the relevant code.

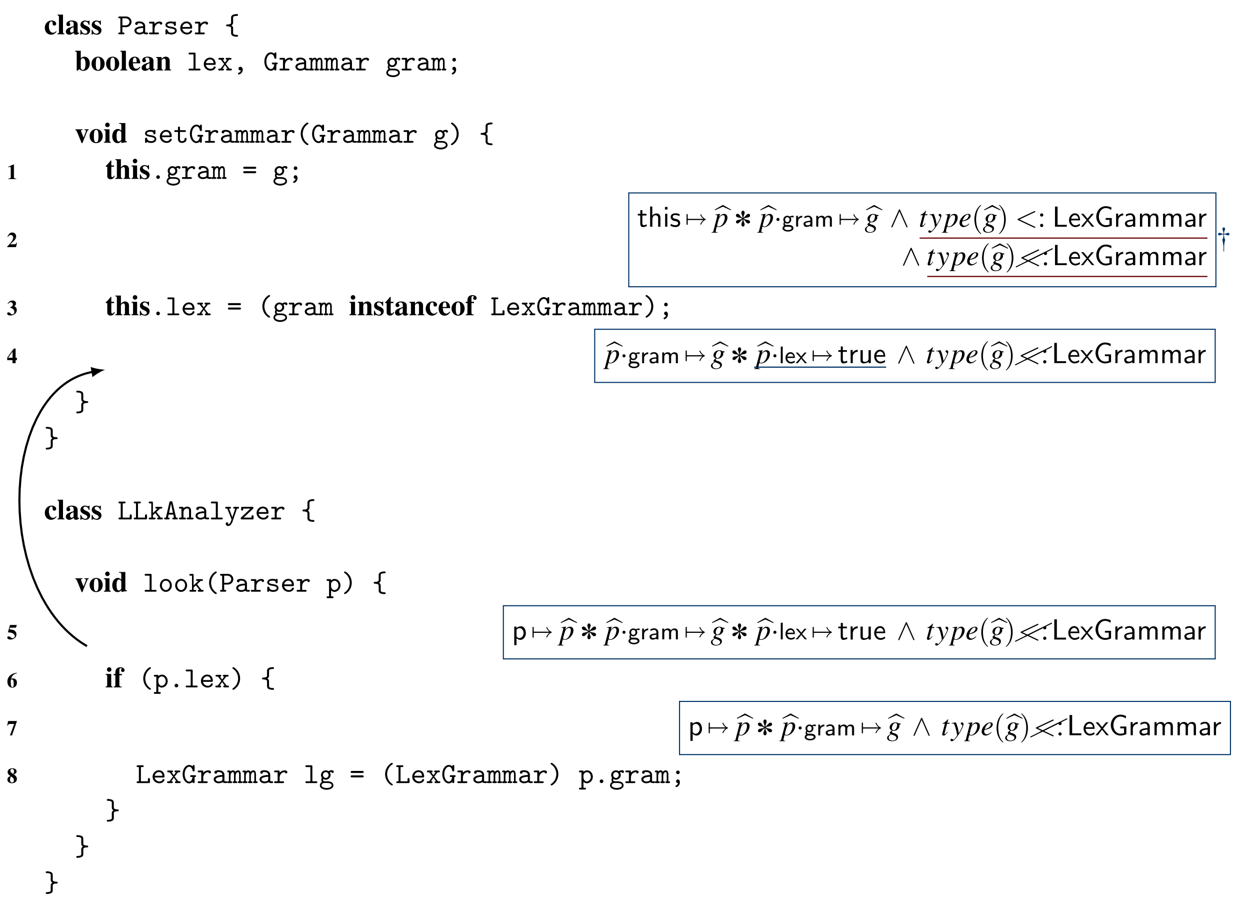
We begin by presenting an example with a query whose safety relies on the kind of programmer invariant-based reasoning we wish take advantage of (Section 6.1). We explain why programs that use this sort of reasoning present a scalability challenge for existing approaches and demonstrate how our approach is able to quickly refute the query (Sections 6.1.1 and 6.1.2). Section 6.2 presents strategies for deciding when to coarsen the control-flow abstraction by exploring data-relevant transitions rather than control-feasible transitions. The **eager relevance checking** strategy presented in Section 6.2.1 identifies coarsening opportunities that are certain to lead immediately to a refutation. The **invariant schema** strategy presented in Section 6.2.2 allows the analysis to identify commonly used global invariants (such as object invariants and container invariants) and directly jump to the code that establishes the invariant in order to quickly find a refutation. In Section 6.3, we show how to precisely compute data-relevance information for heap constraints used in our goal-directed store abstraction (Section 6.3.3) and combine this algorithm with the strategies from the previous section to define a relevance relation (Section 6.3.4). Finally, Section 6.4 demonstrates that using this relevance relation to outfit THRESHER with flexible control-flow abstraction provides a significant scalability benefit over flexible store-abstraction alone. Our HOPPER tool was able to refute many tough queries requiring scalable flow-, path, and context-sensitive reasoning that THRESHER could not handle.

Portions of this chapter appeared in an unpublished draft entitled "Goal-Directed Coarsening of the Control-Flow Abstraction via Jumping", which was co-authored by Bor-Yuh Evan Chang and Manu Sridharan.

6.1 Example: leveraging object invariants with jumping and invariant schemas

In this section, we explain the challenges of the proving the safety of the downcast at line 8 in Figure 6.1 and demonstrate how we can use control-flow abstraction via jumping to efficiently refute the query The code in this figure is a snippet manually and carefully extracted from the **antlr**[1] Dacapo benchmark [Blackburn et al., 2006] that isolates exactly the code required for proving the safety of his downcast. The goal of our jumping analysis is to explore **only** this relevant code by automatically identifying these relevant code regions and jumping between them without either **exploring** the tens of thousands of lines of irrelevant code that can execute in between or expecting a programmer-written specification that modularizes this verification.

[1] http://www.antlr.org/



```
class Parser {
  boolean lex, Grammar gram;

  void setGrammar(Grammar g) {
    this.gram = g;
```

$$\boxed{\begin{array}{r}\mathsf{this} \mapsto \widehat{p} * \widehat{p}\cdot\mathsf{gram} \mapsto \widehat{g} \wedge \underline{\mathit{type}(\widehat{g}) <: \mathsf{LexGrammar}} \\ \wedge\, \underline{\mathit{type}(\widehat{g}) \not<: \mathsf{LexGrammar}}\end{array}}\,\dagger$$

```
    this.lex = (gram instanceof LexGrammar);
```

$$\boxed{\widehat{p}\cdot\mathsf{gram} \mapsto \widehat{g} * \underline{\widehat{p}\cdot\mathsf{lex} \mapsto \mathsf{true}} \wedge \mathit{type}(\widehat{g}) \not<: \mathsf{LexGrammar}}$$

```
  }
}

class LLkAnalyzer {

  void look(Parser p) {
```

$$\boxed{\mathsf{p} \mapsto \widehat{p} * \widehat{p}\cdot\mathsf{gram} \mapsto \widehat{g} * \widehat{p}\cdot\mathsf{lex} \mapsto \mathsf{true} \wedge \mathit{type}(\widehat{g}) \not<: \mathsf{LexGrammar}}$$

```
    if (p.lex) {
```

$$\boxed{\mathsf{p} \mapsto \widehat{p} * \widehat{p}\cdot\mathsf{gram} \mapsto \widehat{g} \wedge \mathit{type}(\widehat{g}) \not<: \mathsf{LexGrammar}}$$

```
      LexGrammar lg = (LexGrammar) p.gram;
    }
  }
}
```

Figure 6.1: Jumping analysis avoids path explosion by performing control-flow abstraction. The analysis collects the key path constraint $\widehat{p}\cdot\mathsf{lex} \mapsto \mathsf{true}$ and jumps directly to the code that allows the analysis to find a refutation using this path constraint. In this example, hatted variables represent a single instance of an object, $\mapsto$ arrows denote exact points-to, and $<:$ denotes Java subtyping.

The initial query $\hat{\rho}: \mathsf{p} \mapsto \widehat{p} * \widehat{p}\cdot\mathsf{gram} \mapsto \widehat{g} \wedge \mathit{type}(\widehat{g}) \not<: \mathsf{LexGrammar}$[2] is shown at line 7. This separation logic formula expresses the conditions under which the cast will fail—local p is a memory cell containing the object address $\widehat{p}$ whose gram field contains an object address $\widehat{g}$, and $\widehat{g}$ is not a LexGrammar. As previously described in Section 3.1, our analysis attempts to prove the safety of the cast by trying to **refute** this query. This analysis is a form of proof by contradiction: it computes an over-approximation of the backward reachable states from $\hat{\rho}$ on line 7 and refutes the query when it has derived unreachability of $(\hat{\rho}, \mathit{line}\ 7)$ (e.g., $\bot$—no possible concrete states) at a set of locations that together control-dominate line 7.

At a high level, proving that the cast on line 8 is safe requires showing that the disjunctive invariant `p.lex` = true $\implies$ `p.gram` **instanceof** `LexGrammar` holds for any `p` passed to the `look` method. This invariant is established by the `setGrammar` method of the `Parser` class (assume that it contains the only writes to the `gram` and `lex` fields). Thus, the verification challenge (without a specification of this invariant) lies in (a) understanding that the predicates involved in this invariant are the important ones to track and (b) proving that this invariant holds on all control-flow paths reaching `look`.

Challenge (b) is particularly difficult for existing techniques. In the case where the control flow paths between calls to `setGrammar` and `look` are numerous and complex and the analysis cannot benefit from pruning paths based on a specification, the scalability of a typical path-sensitive analysis will suffer due to path explosion. In the original **antlr** code, there are more than 2^{63} such paths. Though no practical analysis tool would attempt to verify the example by exploring **all** of these paths, the complexity of the control flow between the two methods exacts a toll on the scalability of existing tools. A CEGAR tool like CORRAL [Lal et al., 2012] needs to inline at least 194 methods just to reach the calls to `look` and `setGrammar` from the entry point of **antlr**. A backward path-sensitive analysis like THRESHER [Blackshear et al., 2013] needs to visit at least 161 methods to explore a complete path from `look` to `setGrammar`. Proving that the required invariant holds on all control-flow paths through these methods hurts scalability even when tools are effective at collapsing redundant paths.

This scalability bottleneck is especially frustrating because path-sensitive analysis of the control-flow between `setGrammar` and `look` is unnecessary—`setGrammar` contains the key code for proving the cast

[2] Our queries should be interpreted as constraining a sub-heap, that is, a formula M means $M * \mathsf{true}$ for the whole heap.

safe (i.e., refuting the query), and other path conditions constraining calls to `look` are irrelevant. However, no analysis has **a priori** knowledge of the property that all of this code is irrelevant, and so it must laboriously discover this property during analysis.

Our jumping analysis avoids this scalability bottleneck by identifying important code regions using data-relevance information and jumping directly between them. Jumping is a form of control-flow abstraction that mitigates the scalability issues described here by avoiding precise analysis of irrelevant control-flow altogether. This means that the cost of proving the safety of this example will not grow as the complexity of the control-flow between `setGrammar` and `look` grows so long as no relevant instructions occur in the control-flow. Our analysis addresses challenge (a) by using **invariant schemas** to decide when a query has collected enough important path conditions and challenge (b) by performing control-flow abstraction via jumping between relevant code regions, as we will see.

6.1.1 Invariant-based jumping algorithm

Each time our jumping analysis applies a transfer function on a command that changes the current query, it performs the following steps:

(1) *Match and abstract using invariant schemas.* The analysis tries to match the query to an **invariant schema** describing the structure of commonly used invariants. For this example, the analysis employs an object invariant schema that captures relational object invariants by matching queries that have constraints on two distinct fields of the same object instance. If an invariant schema matches a query, the analysis abstracts the query by dropping the points-to constraints that do not match the schema (i.e., soundly summarizing those constraints with `true`).

(2) *Find important code using a relevance relation.* For each points-to constraint in the query that remains after abstraction, the analysis computes the set of **data-relevant** program commands whose execution may produce a heap configuration in the concretization of the constraint (see Condition 5.3.1(a)). This process is similar to computing a partial slice that only considers immediately data-relevant commands rather than a transitive closure (see Section 6.5 for a full discussion of the differences between our

technique and slicing). Finding the set of data-relevant commands makes use of a global view of the program from a points-to graph computed by a separate, up-front analysis.

(3) *Jump to relevant code.* The analysis forks a case split for each relevant command, "jumps" to the command, and continues analysis for each case. This coarsens the control-flow abstraction used by the analysis by assuming that each relevant command is backward-reachable from the current program point. If in step (1), the query did not match an invariant schema, the analysis pushes the query backward to its predecessor program points rather than jumping (i.e., follows the program transitions backward in the normal manner).

6.1.2 Refuting the example query

The analysis begins by trying to match the initial query at line 7 against the object invariant schema. The query does not match the schema, so the analysis pushes the query backward across the **if** `(p.lex)` guard at line 6. This augments the query with the constraint $\widehat{p} \cdot \mathsf{lex} \mapsto \mathsf{true}$ and produces the new query at line 5. This new query matches the object invariant schema described above, as it constrains both the gram and lex fields of the abstract object instance $\widehat{p}$. The analysis then abstracts the query by dropping points-to constraints not matched by the schema (pure constraints such as $type(\widehat{g}) \not<: \mathsf{LexGrammar}$ are always retained). In this case, only the local constraint $\mathsf{p} \mapsto \widehat{p}$ is dropped.

After abstraction, the analysis computes the relevant constraints for the query, determining that line 1 is relevant for the constraint on gram and line 3 is relevant for the constraint on lex. In this case, since line 3 post-dominates line 1, our analysis reasons that it only needs to jump to line 3, as indicated by the arrow to line 4 in the figure.

After the jump, the analysis applies the transfer function to move the query across line 3 and reasons that in order to produce the constraint $\widehat{p} \cdot \mathsf{lex} \mapsto \mathsf{true}$, it must be the case that $type(\widehat{g}) <: \mathsf{LexGrammar}$. But this contradicts the extant query constraint $type(\widehat{g}) \not<: \mathsf{LexGrammar}$, and thus the analysis has found a contradiction and refuted the query (as indicated by the † symbol).

6.1.3 Challenges of designing a jumping analysis

Our motivating example above illustrates a case where jumping analysis seamlessly mitigates the previously described scalability issues while preserving precision, but in general crafting an effective jumping analysis presents several challenges:

(1) *Computing sound, yet precise data-relevance information.* Imprecise data-relevance information that includes many irrelevant commands will not enhance scalability because the analysis will have to perform too many case splits each time it jumps. However, data-relevance information that leaves out an important command is unsound because it may cause the analysis to falsely refute the query.

(2) *Deciding when to coarsen the control-flow abstraction.* Coarsening at the wrong time can lead to decreased precision in the analysis if the analysis chooses to jump over code that is important for finding a refutation. On the other hand, choosing to coarsen at an opportune time can significantly improve scalability. Thus, choosing when to coarsen in a way that enhances scalability without losing the precision necessary to find a refutation is crucially important.

6.2 Deciding when to coarsen the control-flow abstraction

Coarsening the control-flow abstraction via jumping fundamentally causes the analysis to lose precision. We only want coarsen the control-flow abstraction when we think that we can trade off superfluous precision (that is, precision not required to prove the safety of the query) for better scalability by jumping over irrelevant code. Coarsening at other times may cause the analysis to lose too much precision and fail to refute the goal query. In this section, we present strategies for deciding when to coarsen based on **eager relevance checking** (Section 6.2.1) and **invariant schemas** (Section 6.2.2). We now describe each of these strategies and give example use-cases demonstrating why each strategy identifies jumps that are likely to improve scalability without sacrificing important precision.

6.2.1 Identifying precision-preserving jumps with eager relevance checking

As discussed previously, choosing when to jump is of the utmost importance for enhancing scalability without sacrificing precision. We craft the first part of our jumping strategy based on the following observation: if a jump is certain to lead to a refutation, then performing the jump cannot lose precision, and hence we should always perform such jumps. Eager relevance checking makes use of this observation by consulting the relevance relation to identify cases when a jump is certain to lead to a refutation. It exploits the situation when the relevance relation is precise enough to show that some part of the query cannot be witnessed by any command in the program.

For a query R, eager relevance checking involves two steps. We first compute the relevant transitions for each constraint in R individually. If a constraint has no relevant transitions except for t_{init}, then we weaken R to an abstract state R' containing only that constraint and apply A-JUMP. Since the only relevant transition is t_{init}, the analysis will jump to ℓ_{entry} and find a refutation based on the fact that $R \neq \top$ at ℓ_{dummy} (recall from Section 5.4 that the only abstract state whose concretization includes the initial state σ_{dummy} is $\top$).

Use case for eager relevance checking. We found that eager relevance checking improved the performance of our analysis by quickly identifying unsatisfiable constraints on pure values. For example, consider a query containing a points-to constraint $\widehat{v}\cdot\mathsf{opcode} \mapsto 1$, stating that the opcode of instance $\widehat{v}$ must hold the value 1. These kinds of constraints arise frequently when the programmer defines a small set of constant integers to tag different kinds of values (for example, **final int** `ADD = 1, SUB = 2;`) and then uses comparisons to these constants as guards. If our relevance relation can determine that all possible writes to $\widehat{v}\cdot\mathsf{opcode}$ write a value other than 1, then it will report that only t_{init} is relevant for this constraint. Eager relevance checking will then cause the analysis to weaken the query to $\widehat{v}\cdot\mathsf{opcode} \mapsto 1$ before jumping and immediately finding a refutation.

6.2.2 Identifying invariant-based reasoning with invariant schemas

When eager relevance checking cannot show that a jump is certain to lead to a refutation, we must decide when we think the analysis has collected enough constraints to find a refutation after jumping. **Invariant schemas** are based on the idea that jumps can be fruitful when the current query seems to rely on a global invariant. When programmers rely on local reasoning, jumping is likely to lose precision without enhancing scalability, as the facts needed to establish safety are available locally. However, programmers also frequently rely on global reasoning by establishing invariants that hold throughout the program. In cases where safety relies on a global invariant, it may make sense to abandon local reasoning and jump to the location where the invariant is established. In doing so, we hope to avoid scalability bottlenecks that result from continuing precise local analysis while retaining precision by focusing the analysis on the invariant of interest.

We developed our invariant schemas by manually inspecting Java code and determining where and why it would be useful for the analysis to switch from local reasoning to global reasoning when trying to refute a query. Each invariant schema is function that identifies forms of commonly used invariants. We tried to choose invariant schemas that capture very general kinds of invariants, but we imagine that more client-specific invariant schemas could also be useful.

We define each invariant schema S as a function that takes an abstract state R as input and (if it matches) produces a weakened version of the state. The purpose of an invariant schema is twofold: (1) matching the current query to tell the analysis when it is wise to perform a jump, and (2) abstracting the query with respect to the schema, reducing the number of possible jump targets. Our schemas only match the abstract store $\hat{\rho}$ and choose to lose all information about the abstract call string $\hat{L}$. In this way, we eliminate the need to consider any `call` commands as relevant for our backward analysis, enabling inter-procedural jumps. This decision is consistent with our design principle of minimizing the number of relevant jump targets, which increases scalability in practice.

Our analysis currently uses two invariant schemas, an **object invariant** schema S_{obj} and a **container invariant** schema $S_{\text{container}}$. We describe each schema via an existentially-quantified separation logic formula that we attempt to match against the query, underlining **retained constraints** in green. All non-retained constraints are removed by abstraction.

Object invariant schema. Our object invariant schema S_{obj} seeks to identify cases where the property of interest relies on some relationship between fields of the same object. S_{obj} searches the query for two points-to constraints $\underline{\widehat{v}{\cdot}f_0 \mapsto \widehat{u}_0} * \underline{\widehat{v}{\cdot}f_1 \mapsto \widehat{u}_1}$ where $f_0 \neq f_1$ and then **weakens** the query by dropping all other points-to constraints. This schema is quite general: it captures any object invariant that involves a relationship between two fields of the same object instance. Crucially, we do not have to understand the nature of the relationship between the two fields in order to utilize this schema—we only need to hypothesize that **some relationship** exists and preserve this relationship while jumping.

Use case for the object invariant schema. Jumping when S_{obj} is matched is effective because object invariants are often established early in program execution (e.g., in a class initializer or constructor) and then used much later on. In our motivating example in Section **??**, the object invariant relating the lex and gram fields is established when the fields are mutated in `setGrammar`, but then is used much later in the program to ensure the safety of the cast. Note that in the example, our analysis never needs to explicitly infer the actual invariant $\texttt{p.lex} = \text{true} \implies \texttt{p.gram}$ **instanceof** `LexGrammar`. It is sufficient for the analysis to recognize that there is some relationship between `p.lex` and `p.gram` and perform a jump while preserving this relationship.

Container invariant schema. Our container invariant schema $S_{\text{container}}$ seeks to identify invariants that hold for all elements of a particular container (e.g., "this container contains only integers that are less than 7"). $S_{\text{container}}$ searches the query for an access path of length 2 or greater $\widehat{v}_0{\cdot}f_0 \mapsto \widehat{v}_1 * \underline{\widehat{v}_1{\cdot}f_1 \mapsto \widehat{v}_2}$, and then weakens the query by dropping all points-to constraints except for the last one in the access path. This schema abstracts away all constraints on the access path to the container and allows the analysis to focus only on the contents of the container. In the common case that elements satisfying the invariant are inserted into a container at one point in the program and accessed via the container later on, this invariant schema allows the analysis to jump directly from code performing the accesses to the insertion code.

Use case for the container invariant schema. Programmers often ensure that all elements of a container have a certain property at insertion time via code like

if `(x != `**null**`) lst.add(x)`. Later in the program, the programmer will read an element from the list and dereference it without checking for null (e.g., `lst.get(i).foo()`). Without knowing the container invariant on `lst`, this dereference is difficult to prove safe. However, our analysis will use the container invariant schema to jump directly to all locations where elements are inserted into `lst` and check that the invariant is maintained for all inserted elements, avoiding analysis of all irrelevant code between the read and write(s).

6.3 A relevance relation that leverages programmer reasoning

In this section, we define a relevance relation that explores predecessor commands of the current program point by default, but chooses to coarsen the control-flow abstraction by jumping to data-relevant commands using the strategies defined in Section 6.2. We motivate the design of our data-relevance relation (Section 6.3.1), present abstract semantics for procedure calls in Section 6.3.2 (a required preliminary step), explain how we compute precise data-relevance information for heap constraints (Section 6.3.3), and formally define our relevance relation (Section 6.3.4).

6.3.1 Design principle: coarsen infrequently, but aggressively

Our strategy for deciding when to coarsen the control-flow abstraction is quite conservative in that the analysis only chooses to make a jump that may lose precision when the analysis state matches one of the invariant schemas described in Section 6.2.2. In addition, the analysis coarsens the store abstraction before jumping; it weakens the query by dropping all constraints not matched by the invariant schema (including constraints on the call string, meaning that we lose all context-sensitivity when jumping). This is important because it limits the number of relevant locations that must be jumped to and thereby maximizes the scalability benefits of jumping.

Thus, though the decision to coarsen is made infrequently, the coarsening that we perform is aggressive in the sense that it loses all information about parts of the query not related to the invariant. We want our data-relevance relation to be similarly aggressive: our goal is to minimize the number of locations that need to be jumped to for the maximum scalability benefit. The most important decision point is whether the relevance relation should exclude transitions that can **strengthen** the query and thus make finding a refutation easier. In order to accomplish this, we seek (1) to be as precise as possible in computing the relevant commands for a query (as we will explain in Section 6.3.3), and (2) to be deliberately coarse by not considering **strengthening** transitions as relevant. The second part of this strategy takes advantage of relevance soundness Condition 5.3.1(a), which says that for soundness we only need to consider transitions that **weaken** the current query state to be relevant. Excluding a strengthening transition such as `assume y != 5` in Example 5.3.1 may skip a transition that could have led the analysis to derive a refutation, or it may hamper scalability by forcing the analysis to explore infeasible paths that could have been ruled out by the path condition. But if this path condition turns out not to be important for deriving a refutation (since there are typically far more irrelevant conditionals than relevant ones for a given query), then the analysis wastes time and potentially hurts scalability trying to may-witness an unimportant constraint.

Excluding strengthening transitions that make the query more constrained (such as `assume`'s that constrain by adding path conditions and `return`'s that constrain by adding to the call string) is likely to improve scalability by limiting the relevant locations to be jumped to, but can also lose precision. Choosing to exclude `assume`'s (resp. `return`'s) can cause the analysis to lose path-sensitivity (resp. context-sensitivity) while jumping. However, we found that in the absence of control-feasibility filtering (which the relevance relation defined here does not use while jumping), choosing to include these strengthening transitions significantly increases the number of relevant transitions to be jumped to in practice without increasing precision.

Thus, the data-relevance relation defined Section 6.3.3 in chooses to exclude **all** strengthening transitions. The choice to exclude strengthening transitions from our data-relevance relation precludes path-sensitivity and context-sensitivity, whereas the decision to forego control-feasibility while jumping precludes flow-sensitivity. This means that jumping corresponds to taking a single step of fully flow-insensitive analysis. The analysis is fully precise before and after jumping, but during a jump our goal is to compute the smallest set of relevant transitions allowed by Condition 5.3.1 in hopes that jumping will enhance scalability as much as possible in practice. Stated differently, we rely on underlying store abstraction defined in Chapter 4 for precision, but rely on our jumping strategy to ensure that this expensive precision is applied only to relevant code regions.

$\boxed{\vdash \{R'\}\ c\ \{R\}}$

A-RETURN
$$\frac{}{\vdash \{(\hat{\rho},\ell::\hat{L})\}\ \texttt{return}\ \ell\ \{(\hat{\rho},\hat{L})\}}$$

A-CALL-OK
$$\frac{\ell = \ell_1}{\vdash \{(\hat{\rho},\hat{L})\}\ \texttt{call}\ \ell_1\ \{(\hat{\rho},\ell::\hat{L})\}}$$

A-CALL-REF
$$\frac{\ell \neq \ell_1}{\vdash \{\bot\}\ \texttt{call}\ \ell_1\ \{(\hat{\rho},\ell::\hat{L})\}}$$

A-CALL-ANY
$$\frac{}{\vdash \{(\hat{\rho},\mathsf{anystring})\}\ \texttt{call}\ \ell\ \{(\hat{\rho},\mathsf{anystring})\}}$$

Figure 6.2: Backward abstract semantics for `call` and `return` commands.

6.3.2 Abstract semantics for procedure calls

Before formalizing our data-relevance relation, we must present abstract semantics for the `call` and `return` commands used to represent procedure calls in the jumping analysis framework in Figure 6.2. We gave the corresponding concrete semantics in Figure 5.2 and explained the meaning of the backward Hoare triple $\vdash \{R'\}\ c\ \{R\}$ in Section 5.2.1. We gave the abstract semantics for the `assume` command used to represent conditionals and looping win Figure 4.9.

The A-RETURN rule says that when the analysis moves backward across the statement `return` ℓ, the return label ℓ is prepended to the abstract call string. This constrains the abstract call string to reflect that any concrete execution could only have reached this program point if it previously visited a matching `call` ℓ instruction that pushed ℓ onto the call string. The A-CALL-OK rule expresses the case where the analysis subsequently encounters this matching call. If the label ℓ on top of the call string matches the label ℓ_1 of the call command, the analysis weakens the state by popping the label off of the call string. By contrast, the A-CALL-REF rule expresses the case where the analysis subsequently encounters a non-matching calls. If the label on top of the call string ℓ does not match the label ℓ_1 of the call command, the analysis **refutes** the current path (derives $\bot$) since no concrete state could have a non-ℓ_1 label on top of its call string immediately after executing the command `call` ℓ_1. Finally, the A-CALL-ANY rule says that an unconstrained call string $\mathsf{anystring}$ can be propagated backward across a `call` command without any changes.

6.3.3 Computing precise data-relevance information for heap dependencies

In this subsection, we show how to compute precise data-relevance information for the separation logic-based store representation presented in Chapter 4. The key challenge here is the capability to compute a precise approximation of relevant writes, i.e., it must precisely identify commands that may write to relevant portions of the heap. If the relevance relation is not effective at precisely identifying such commands, it will report too many commands as relevant and lessen the scalability benefits of jumping.

Our data-relevance relation leverages the up-front points-to analysis and instance-from constraints used by the store abstraction introduced in Chapter 4 to precisely identify heap dependencies. Recall that an instance-from constraint $\widehat{v}$ from $\mathring{r}$ states that the instance $\widehat{v}$ must have been allocated from the region $\mathring{r}$, where a region is a set of allocation sites. We define $\widehat{v}\,\mathsf{from}\,\emptyset \iff \mathsf{false}$ since it means that $\widehat{v}$ could not have been allocated from any allocation site in the program. These constraints are useful for precisely computing heap dependence information, as we will explain.

Computing relevance for heap dependencies requires determining what commands might be relevant to a points-to constraint $\widehat{v}\cdot f \mapsto \widehat{u}$ with instance-from constraints $\widehat{v}\,\mathsf{from}\,\mathring{r} \wedge \widehat{u}\,\mathsf{from}\,\mathring{s}$. We can clearly restrict possibly-relevant commands to those updating field f, of the form $x.f := y$. Given the instance-from constraints, we can further restrict relevant commands to those meeting the following condition:

$$\mathrm{pt}_{\mathring{G}}(x) \cap \mathring{r} \neq \emptyset \wedge \mathrm{pt}_{\mathring{G}}(y) \cap \mathring{s} \neq \emptyset$$

The function $\mathrm{pt}_{\mathring{G}}(x)$ denotes the points-to set of x as computed by an up-front points-to analysis. The above condition ensures consistency between the points-to sets of x and y and the corresponding regions $\mathring{r}$ and $\mathring{s}$, rejecting any command that could not possibly produce the $\widehat{v}\cdot f \mapsto \widehat{u}$ points-to constraint.

This reasoning is captured in the R-WRITE rule of Figure 6.3. This rule is one of several comprising the auxiliary judgment form $\hat{\rho} \rightsquigarrow T_{\mathrm{rel}}$, which asserts that the transitions in T_{rel} may be relevant to the abstract store $\hat{\rho}$. Other rules defining relevance for local constraints leverage points-to information and instance-from constraints in a manner similar to the R-WRITE rule, as we will explain. The R-READ, R-NEW, and R-ASSIGN rules compute the data-relevant commands for a local points-to constraint $x \mapsto \widehat{v}$ in a similar way. These rules essentially encode a flow-insensitive variation of reaching definitions extended with from constraints. The

R-Read and R-Assign rules say that any field read $x := y.f$ (resp. $x := y$) such that the intersection of the points-to set of $y.f$ (resp. y) and the allocation region $\mathring{r}$ constraining $\widehat{v}$ is non-empty is relevant to the points-to constraint $x \mapsto \widehat{v}$. The R-New rule says that any allocation command $x := \mathtt{new}_a\, \tau\texttt{()}$ that allocates an allocation site a in the allocation region $\mathring{r}$ constraining $\widehat{v}$ is relevant to the points-to constraint $x \mapsto \widehat{v}$. The R-Any rules defines the base cases of relevance for the separation logic predicate any that is satisfied by any heap.

The R-Sep rule gives structure to the $\hat{\rho} \rightsquigarrow T_{\mathrm{rel}}$ judgment by recursively applying the relevance relation to each sub-memory of the store to find the data-relevant transitions for the entire store. It says that the set of relevant transitions for the store $\hat{\rho}$ is the union of the relevant transitions for each of its sub-memories.

The other auxiliary judgment form $\hat{L} \rightsquigarrow T_{\mathrm{rel}}$ asserts that the transitions in T_{rel} may be relevant to the abstract call string $\hat{L}$. The R-Call rule says that a call command with return label ℓ_1 must be considered data-relevant to a call string with a label $\ell = \ell_1$ as its first label. In our backward analysis, the abstract semantics for `call` can weaken the abstract state by popping a label off of the call string, thereby creating a less constrained call string. Thus, we must consider all `call` instructions with labels matching the top of the call string to be data-relevant in order to be sound.

However, as explained in Section 6.2.2, our invariant schemas always choose to weaken the abstract call string to anystring before computing relevance and jumping, so this rule is never applied in our system. Instead, R-AnyString will always be applied. This rule is simply the analog of R-Any and R-Top rules for call strings.

Finally, the top-level judgment form $R \rightsquigarrow T_{\mathrm{rel}}$ asserts that the transitions in T_{rel} may be relevant to the abstract state R. R-Bot and R-Top rules define the base cases of relevance for an unreachable state and a state representing all concrete states, respectively. In both cases, the only relevant transition is the initial transition t_{init}. The R-Bot rule ensures that the relevance relation satisfies requirement (b) for the abstract state $\bot$.

The R-Cases rule says that for a disjunction of abstract states $R_0 \vee R_1$, the set of data-relevant transitions is the union of the relevant transitions for R_0 and R_1. The R-Split rule decomposes an abstract state into its store component and call string component, computes the relevant transitions for each component

$$\boxed{R \leadsto T_{\text{rel}}}$$

$$\overset{\textsc{R-Cases}}{\dfrac{R_0 \leadsto T_0 \quad R_1 \leadsto T_1}{R_0 \vee R_1 \leadsto T_0 \cup T_1}} \qquad \overset{\textsc{R-Bot}}{\dfrac{}{\bot \leadsto \{t_{\mathsf{init}}\}}} \qquad \overset{\textsc{R-Top}}{\dfrac{}{\top \leadsto \{t_{\mathsf{init}}\}}} \qquad \overset{\textsc{R-Split}}{\dfrac{\hat{\rho} \leadsto T_1 \quad \hat{L} \leadsto T_2}{(\hat{\rho}, \hat{L}) \leadsto T_1 \cup T_2}}$$

$$\boxed{\hat{\rho} \leadsto T_{\text{rel}}}$$

$$\overset{\textsc{R-Sep}}{\dfrac{\hat{\rho} = M_0 * M_1 \wedge P \quad M_0 \wedge P \leadsto T_0 \quad M_1 \wedge P \leadsto T_1}{\hat{\rho} \leadsto T_0 \cup T_1}}$$

$$\overset{\textsc{R-Assign}}{\dfrac{T_{\text{rel}} = \{\, t \mid t \in P \text{ and } t = \ell_i \dashv\![x := y]\!\!\rightarrow \ell_j \text{ and } \mathrm{pt}_{\mathring{G}}(y) \cap \mathring{r} \neq \emptyset \,\}}{x \mapsto \widehat{v} \wedge \widehat{v} \;\mathsf{from}\; \mathring{r} \wedge P \leadsto T_{\text{rel}}}}$$

$$\overset{\textsc{R-New}}{\dfrac{T_{\text{rel}} = \{\, t \mid t \in P \text{ and } t = \ell_i \dashv\![x := \mathtt{new}_a\, \tau\mathtt{()}]\!\!\rightarrow \ell_j \text{ and } a \in \mathring{r} \,\}}{x \mapsto \widehat{v} \wedge \widehat{v} \;\mathsf{from}\; \mathring{r} \wedge P \leadsto T_{\text{rel}}}}$$

$$\overset{\textsc{R-Read}}{\dfrac{T_{\text{rel}} = \{\, t \mid t \in P \text{ and } t = \ell_i \dashv\![x := y.f]\!\!\rightarrow \ell_j \text{ and } \mathrm{pt}_{\mathring{G}}(y.f) \cap \mathring{r} \neq \emptyset \,\}}{x \mapsto \widehat{v} \wedge \widehat{v} \;\mathsf{from}\; \mathring{r} \wedge P \leadsto T_{\text{rel}}}}$$

$$\overset{\textsc{R-Write}}{\dfrac{T_{\text{rel}} = \{\, t \mid t \in P \text{ and } t = \ell_i \dashv\![x.f := y]\!\!\rightarrow \ell_j \text{ and } \mathrm{pt}_{\mathring{G}}(x) \cap \mathring{r} \neq \emptyset \text{ and } \mathrm{pt}_{\mathring{G}}(y) \cap \mathring{s} \neq \emptyset \,\}}{\widehat{v} \cdot f \mapsto \widehat{u} \wedge \widehat{v} \;\mathsf{from}\; \mathring{r} \wedge \widehat{u} \;\mathsf{from}\; \mathring{s} \wedge P \leadsto T_{\text{rel}}}}$$

$$\overset{\textsc{R-Any}}{\dfrac{}{\mathsf{any} \wedge P \leadsto \{t_{\mathsf{init}}\}}}$$

$$\boxed{\hat{L} \leadsto T_{\text{rel}}}$$

$$\overset{\textsc{R-Call}}{\dfrac{T_{\text{rel}} = \{\, t \mid t \in P \text{ and } t = \ell_i \dashv\![\mathtt{call}\ \ell_1]\!\!\rightarrow \ell_j \text{ and } \ell = \ell_1 \,\}}{\ell :: \hat{L} \leadsto T_{\text{rel}}}} \qquad \overset{\textsc{R-AnyString}}{\dfrac{}{\mathsf{anystring} \leadsto \{t_{\mathsf{init}}\}}}$$

Figure 6.3: A data-relevance relation that uses an up-front points-to analysis and instance-from constraints to precisely identify heap dependencies. This relevance relation considers only commands that **weaken** the current query to be relevant.

using the auxiliary relevance judgments, and returns the union of the relevant transitions.

Data-relevance of constraints on pure types. The rules in Figure 6.3 only describe how to compute the data-relevant commands for points-to constraints whose righthand side is an an abstract instance, but in practice, we frequently need to compute relevant commands for exact points-to constraints whose righthand side is a pure type (e.g., the constraint $\widehat{p}\cdot\mathsf{lex} \mapsto \mathsf{true}$ in Figure 6.1). We can add a variation of the R-WRITE rule to compute the data-relevant commands for such a constraint as follows. We first remove the $\widehat{u}$ from $\mathring{s}$ constraint from the rule. We then generalize the clause $c = x.f := y$ that restricts the set of commands considered to the more permissive clause $c = x.f := e$. Finally, we swap the side condition $\mathrm{pt}_{\mathring{G}}(y) \cap \mathring{s} \neq \emptyset$ for a satisfiability check $SAT(\widehat{u} = e \wedge P)$, where SAT invokes a decision procedure for the expressions allowed in the expression language e[3] . The resulting R-WRITE-PURE rule is shown inset:

$$\textsc{R-Write-Pure}\quad \frac{T_{\mathrm{rel}} = \{\ t \mid t \in P \text{ and } t = \ell_i \dashv\!\![x.f := e]\!\!\mapsto \ell_j \text{ and } \mathrm{pt}_{\mathring{G}}(x) \cap \mathring{r} \neq \emptyset \text{ and } SAT(\widehat{u} = e \wedge P)\ \}}{\widehat{v}\cdot f \mapsto \widehat{u} \wedge \widehat{v} \text{ from } \mathring{r} \wedge P \rightsquigarrow T_{\mathrm{rel}}}$$

The R-ASSIGN and R-READ rules can be extended in a similar way.

Design decision: ignoring `assume`'s and `return`'s. We note that Figure 6.3 contains no rules for computing the data-relevance of `assume` and `return` commands. As explained in Section 6.3.1, this is intentional: Condition 5.3.1(a) tells us that any relevance relation can choose to ignore commands that strengthen the abstract store. A command `assume` e can only strengthen the store by conjoining the expression e to the existing pure formulae in the abstract store. The `return` ℓ can only strengthen the store by constraining the call string via prepending the label ℓ.

Cost of computing data-relevance. We briefly comment on the cost of computing the data-relevant commands for a query. Clearly, computing data-relevance information needs to be efficient in order for piecewise refutation analysis to yield its promised scalability benefits. Our data-relevance relation's use of a precomputed points-to analysis typically allows us to compute relevance quite quickly.

One potential scalability concern is than many rules in Figure 6.3 quantify over every command in

[3] Our implementation uses the Z3 SMT solver [de Moura and Bjørner, 2008] as its decision procedure.

the program P. We note that in practice, we can often compute data-relevance much more efficiently by exploiting procedural abstraction and up-front points-to information. The R-ASSIGN, R-NEW, and R-READ rules compute the relevant statements for a constraint on some local variable x, so we only needs to inspect each command in the method that x belongs to.

Shrinking the number of commands that the R-WRITE rule consider is slightly more challenging, but we can do it using the points-to graph. Let $\mathring{E}$ be the edge set of the points-to graph and let $x \mapsto\!\!\Rightarrow a$ denote a may-points to edge from the graph. Further, let the containing method of a local variable x be given by $method(x)$, Assuming that we are interested in a heap constraint $\widehat{v}\cdot f \mapsto \widehat{u} \wedge \widehat{v}\,\mathsf{from}\,\mathring{r} \wedge \widehat{u}\,\mathsf{from}\,\mathring{s}$, we compute the sets $P_{\widehat{v}} = \{\ method(x) \ \mid\ (x \mapsto\!\!\Rightarrow a) \in \mathring{E} \wedge a \in \mathring{r}\ \}$ and $P_{\widehat{u}} = \{\ method(y) \ \mid\ (y \mapsto\!\!\Rightarrow a) \in \mathring{E} \wedge a \in \mathring{s}\ \}$. These are sets of methods containing locals that may point to $\widehat{v}$ and $\widehat{u}$, respectively. Any method containing a command that discharges the constraint $\widehat{v}\cdot f \mapsto \widehat{u}$ must have local variables pointing to both $\widehat{v}$ and $\widehat{u}$, so we only need to look at write commands from methods in the set $P_{\widehat{v}} \cap P_{\widehat{u}}$. In practice, this set is typically small enough to investigate efficiently.

6.3.4 Crafting an invariant-based relevance algorithm

Finally, we can combine the jumping strategies described in Section 6.2.2 with the technique for computing precise data-relevance information for heap dependencies from Section 6.3.3 to define our relevance relation (Figure 6.4). The relevance relation is quite straightword and works just as we have previously described: it acts like THRESHER by choosing to follow the predecessor transitions of the current program point by default (line 7), but chooses to "jump" by exploring the data-relevant commands for the current program state based on eager relevance checking (line 3) or invariant schemas (line 9).

We claim that the algorithm meets the relevance soundness criteria defined in Condition 5.3.1. The algorithm either returns the predecessor transitions of the current program point, which meets Condition 5.3.1(b), or it returns the set of data-relevant transitions for the current query, which meets Condition 5.3.1(a).

```
**Require:** Current abstract state $R_{\text{cur}}$
**Require:** Current label $\ell_{\text{cur}}$
**Require:** Program transition relation $P$
**Require:** Call graph CG
**Ensure:** Returned transition set $T_{\text{rel}}$ satisfies Condition 5.3.1
1: $T_{\text{rel}} \leftarrow$ dataRel($R_{\text{cur}}$) // compute $R_{\text{cur}} \rightsquigarrow T_{\text{rel}}$
2: if $T_{\text{rel}} = \emptyset$ then
3:     return $T_{\text{rel}}$ // eager relevance checking
4: end if
5: // check if store matches invariant schemas
6: if $R$ does not match $S_{\text{obj}}$ or $S_{\text{container}}$ then
7:     return preds($\ell_{\text{cur}}$, $P$) // invariant schema doesn't match, follow predecessors
8: end if
9: return $T_{\text{rel}}$ // invariant schema matched, jump by returning data-relevant transitions
```

Figure 6.4: A relevance relation that follow control-dependencies by default, but jumps by following data dependencies if there are no data-relevant transitions or the query matches an invariant schema.

6.4 Case study: proving the safety of tough casts

We implemented the practical jumping analysis described in Section 6.2.2 and Section 6.3 in the HOPPER tool, an extension of the WALA[4] and Z3 [de Moura and Bjørner, 2008]-based THRESHER tool. The core of THRESHER is an engine for refuting queries written in separation logic. Clients are implemented as lightweight add-ons that take a program as input and emit separation logic queries for the core refuter to process. HOPPER extends THRESHER by adding the ability to perform jumps, but is otherwise identical. Both THRESHER and HOPPER are typically run with a time budget for each query. If the tool cannot refute the query within the allotted budget, the analysis gives up and reports the query as not refuted.

In order to evaluate the effectiveness of goal-directed coarsening of control-flow via jumping, we sought to test the following hypotheses:

(1) HOPPER's use of jumping provides a significant scalability advantage over THRESHER, given a fixed time budget per query.

(2) The effects of jumping cannot be matched by THRESHER even with a much larger time budget.

Experimental setup We tested our hypotheses using the benchmarks[5] and queries from a state-of-the-art Datalog refinement analysis capable of adding unlimited object-sensitivity to a flow-insensitive points-to analysis in order to refute queries [Zhang et al., 2014]. The queries are all downcasts that cannot be proven safe by a flow-insensitive, context-insensitive points-to analysis. We ran all of our experiments using an OpenJDK1.7 JVM on a Linux machine with a 2.93 GHz Intel Xeon processor and 32GB of memory.

First, we ran the Datalog refinement analysis (**Dlog**) on the set of queries from each benchmark. We then ran THRESHER (**Thr**) and HOPPER (**Hop**) in a pipeline (THRESHER-first) on all of the queries with a budget of 90 seconds per query. We chose the 90 second budget through trial and error—both tools refuted fewer queries with a lower budget, and much larger budgets did not result in significantly more refutations for either tool.

[4] `http://wala.sourceforge.net`
[5] `https://code.google.com/p/pjbench/`

Finally, we manually triaged the casts that were not proven safe by the THRESHER-HOPPER pipeline and classified each cast as (1) genuinely unsafe, (2) beyond the capability of THRESHER/HOPPER to prove safe even with an infinite budget due to imprecision, or (3) provable by THRESHER/HOPPER in principle, but not in the time given.

Manual triaging The results of our automated analysis and manual triaging are shown in Figure 6.5. We present the results in terms of the number of **unproven** casts remaining after applying the given tool/triaging. The "Manual Triaging" column grouping presents the number of queries for each benchmark (**Qry**), the number of these that are actually safe casts (**Safe**) and the number of these that THRESHER would be able to prove safe given infinite time (**Prov**). Most genuinely unsafe casts that we found relied on non-validated assumptions about external input. For example, in the case of **toba-s**, a Java bytecode optimization framework, a large number of casts assumed that the bytecode would conform to the structure imposed by the Java bytecode verifier

Of the 30 casts that THRESHER lacked the necessary precision to prove, the primary sources of imprecision were the need to compute complex invariants over loops (18 casts), achieve strong updates on summary locations (6 casts), or analyze strings precisely (4 casts). THRESHER performs on-the-fly loop invariant inference over heap constraints [Blackshear et al., 2013], but the required loop invariants for the 18 casts in question involved both heap and pure constraints.

Automated analysis The results of automated analysis are shown in the "Unproven Casts after Analysis" column grouping of Figure 6.5. For all tools, we computed the number of unproven casts by subtracting the number of casts proven safe by the tool from the number of "provable" casts shown in the **Prov** column. We only include the provable casts in these counts since it is only for these casts that the performance of HOPPER and THRESHER can be meaningfully compared.

Our results show that the THRESHER-HOPPER pipeline proves more casts safe than the Datalog refinement approach on all benchmarks where both tools ran to completion.[6] This result is not surprising, as proving cast safety frequently requires path/flow-sensitivity and strong updates that cannot be provided by

[6] The OOM entries in Figure 6.5 indicate benchmarks where the Datalog analysis ran out of memory due because our machine had only 32GB of memory (Zhang et al. report using a machine with 128GB).

Benchmark Size		Manual Triaging			Unproven Casts after Analysis			
Bench	**KLOC**	**Qry**	**Safe**	**Prov**	**Dlog**	**Thr**	**Hop**	**Hop Impr (%)**
antlr	131	145	145	130	75	12	5	58
hedc	153	22	22	18	OOM	0	0	-
javasrc-p	66	81	79	76	64	24	18	25
luindex	190	144	141	137	113	15	8	47
lusearch	198	155	154	157	107	11	9	18
schroeder-m	334	18	18	18	OOM	2	0	100
toba-s	69	60	31	24	OOM	12	1	92
weblech	194	8	6	6	OOM	0	0	-
Total	1335	633	596	566	359*	76	41	47

Figure 6.5: Proving cast safety with three different tools: Datalog refinement [Zhang et al., 2014] (**Dlog**), THRESHER (**Thr**), and HOPPER (**Hop**). The size of each programs under analysis (including both application and library code) is given by the **KLOC** column. The Manual Triaging columns give the number of queries for each benchmark (**Qry**), the number of queries that are safe (**Safe**), and the number of safe queries that THRESHER is precise enough to prove (**Prov**). The Unproven Casts after Analysis columns subtract the number of casts proven safe by each tool (**Dlog**, **Thr**, and **Hop**) from the number of casts in the **Prov** column to give the number of unsafe casts remaining after running the tool (or OOM if the tool ran out of memory). The final **Hop Impr** column shows the percentage improvement of HOPPER over THRESHER for each benchmark (**Total** is the geometric mean of the percentage improvement for each benchmark).

a flow-insensitive approach.

The last three columns of Figure 6.5 contain the data that support experimental claim (1). HOPPER improves on the results of THRESHER on each of the six benchmarks for which THRESHER can be improved, proving a total of 35 additional casts safe. Of the 41 provable casts that HOPPER still could not prove safe, the primary cause was a need to add both call site-sensitivity and object-sensitivity simultaneously that we could not capture with an invariant schema.[7] On the other hand, we only observed one case in which a jump lost precision that would have led to a refutation, showing the effectiveness of our jumping policy for identifying only precision-preserving jumps.

The final **Hop Impr** column shows the reduction in unproven casts achieved by HOPPER (where 100% represents a perfect result), summarizing the effectiveness of our approach. Overall, HOPPER reduces the number of unproven queries from THRESHER by 47%, validating our first hypothesis that HOPPER significantly improves scalability compared to THRESHER.

Figure 6.5 focuses small set of benchmarks and queries for a single client in order to gain a deep understanding of the strengths and weaknesses of our technique. We have also used HOPPER to check for array out-of-bounds errors, null dereferences, and unsafe downcasts on eight benchmarks from the Dacapo2006 [Blackburn et al., 2006] suite. In these experiments, HOPPER improved on the results of THRESHER for every benchmark/client that we ran. We have not yet performed the significant amount of manual triaging required to quantify HOPPER's improvement when eliminating unsafe queries and queries not provable by THRESHER for these benchmarks/clients. The full results of these experiments are included in the extended version of this paper.

Finally, to demonstrate that the scalability improvement provided by HOPPER cannot be easily replicated with THRESHER by using a larger budget (Hypothesis 2), we re-ran THRESHER on each of the 35 casts that the HOPPER component of the pipeline proved safe with a budget of 9000 seconds per cast instead of 90 seconds. Even with this much larger budget, THRESHER was not able to prove **any** additional casts safe, showing that the jumping ability of HOPPER yields a scalability advantage that THRESHER cannot match.

[7] For example, some queries required object sensitivity on a `Comparator` and container passed to a generic `sort` method, along with call-site sensitivity for the call to `sort` and its transitive callees.

6.5 Related work

Program slicing. Identifying commands that may affect a query using a data-relevance is closely related to program slicing [Tip, 1995]. Our approach is most closely related to semantic slicing [Bourdoncle, 1993b; Rival, 2005], since we perform a slice with respect to an abstract state rather than a seed command. A key difference between our data-relevance relation and semantic slicing is that we only compute the first step of a slice (i.e., the **immediately** relevant commands) rather than computing a transitive closure of relevant commands as a complete slice does. In many cases, taking a complete slice includes the majority of the program and is prohibitively expensive to compute. Our analysis aims to incrementally compute the parts of the slice that are important for finding a refutation while using abstraction to keep the size of the slice under control. Our approach is much more efficient than the obvious approach of taking a full slice with respect to the query and analyzing the sliced program.

CEGAR. As discussed in Chapter 1, counterexample-guided abstraction refinement (CEGAR) [Ball et al., 2011; Clarke et al., 2000; Henzinger et al., 2002; Lal et al., 2012] is a common approach to goal-directed analysis. CEGAR attempts to avoid analysis of irrelevant code by starting with the coarsest possible abstraction and then repeatedly refining the abstraction in response to spurious counterexamples. Our analysis performs goal-directed **coarsening** instead of refinement: it begins with a very precise abstraction and selectively coarsens the control-flow abstraction to limit the application of precise reasoning to small fragments of code. The decision to coarsen is made on-the-fly and can be based on sources of information other than counterexamples.

CEGAR-based analysis of software has primarily been applied to C device driver programs and has not yet been shown to scale to modern object-oriented programs that make heavy use of heap allocation. The CORRAL tool analyzes C# programs, but their published C# results only consider programs up to 2K lines of code [Lal et al., 2012] . Zhang et al.'s Datalog refinement CEGAR algorithm handles large real-world Java programs, but their approach is not path-sensitive and thus cannot refute many queries that our techniques can (see the comparison in Section 6.5).

Impact pre-analysis. Recent work (also discussed in Section 2.1.2) has presented a form of abstraction coarsening using a cheap, context-insensitive "impact" or "introspective" analysis to estimate the precision and scalability impact of using a context-sensitive abstraction at a particular call site [Oh et al., 2014; Smaragdakis et al., 2014a] or tracking the relationship between two variables [Oh et al., 2014]. This pre-analysis is used to select an effective abstraction for a subsequent precise analysis. In both cases, the abstraction used by the precise analysis is fixed, whereas the abstraction we use can be coarsened on-the-fly. Neither approach is capable of selective path-sensitivity like our approach, though Oh et al. [Oh et al., 2014] mention this as a direction for future work.

Skipping irrelevant code. Identifying statements important to a query using a relevance relation is closely related to program slicing [Tip, 1995]. Our approach is most closely related to semantic slicing [Bourdoncle, 1993b; Rival, 2005], since we perform a slice with respect to an abstract state rather than a seed command. Two key differences between the relevance relation and semantic slicing are (1) the relevance relation only computes the first step of a slice and (2) our analysis weakens queries according to an invariant schema before jumping, which reduces the number of relevant commands. In many cases, taking a complete slice will include the majority of the program and will be quite expensive to compute. Our analysis aims to incrementally compute the parts of the slice that are important for finding a refutation while using abstraction to keep the size of the slice under control. This approach is much more efficient than the obvious approach of taking a full slice with respect to the the original query and then performing analysis on the sliced program.

Recent work on checking **deep assertions** [Lal and Qadeer, 2014] introduces a program transformation that adds extra control-flow edges from the program entrypoint to each method containing an assertion. Their experiment shows that this transformation magnifies the effectiveness of the inlining heuristics used in CORRAL, enabling faster proofs and counterexamples for queries. Unlike our jumping technique, this transformation does not leverage a relevance relation to identify other code important for proving the safety of the query. In particular, the transformation does not help the analysis identify methods without assertions that need to be analyzed precisely (such as `setGrammar` in the motivating example of Section **??**).

Previous work on refinement-based points-to analysis [Sridharan and Bodík, 2006] had a similar ability to jump over code irrelevant to a particular query. That system was restricted to selectively adding context- and field-sensitivity to a points-to analysis, whereas our focus is on the more general problem of scaling path-sensitive reachability analysis by using jumping.

Goal-directed backward symbolic analysis. Previous backward symbolic analyses (both over-approximate [Blackshear et al., 2013; Manevich et al., 2004] and under-approximate [Chandra et al., 2009; Sinha et al., 2012]) have enjoyed improved scalability due to their goal-directed exploration, but still suffer performance bottlenecks from path explosion. Techniques have been proposed that mitigate the explosion somewhat, such as directed call-graph construction [Chandra et al., 2009] and alternating forward and backward search [Sinha et al., 2012]. The key advance in the current work is a method for skipping symbolic analysis of code entirely using a relevance relation.

Precise and scalable whole-program analysis. Dillig et al. introduce scalable, non-goal-directed approaches to whole-program analysis that achieve the high level of precision we are targeting: path-sensitivity [Dillig et al., 2008] and container index-sensitivity [Dillig et al., 2011a,b] along with context-sensitivity and strong updates. We see exhaustive whole-program techniques and goal-directed techniques such as our jumping analysis as two distinct approaches with different challenges. In whole-program approaches the challenge is to manually define a fixed abstraction that enables precise and scalable analysis of any program, whereas in goal-directed approaches the challenge is to craft techniques for automatically creating query-specialized abstractions to enable precise and scalable handling of a particular query.

Leveraging invariant-based reasoning of programs. Our object invariant schema captures safety invariants that are similar to those relied on by the tagged union idiom in C programs. Previous work [Jhala et al., 2007] has focused on proving safe usage of tagged unions by inferring and checking dependent types expressing the relationship between the tag and the type of elements in the union. Our system uses invariant schemas to discover and check invariants that are useful for proving a given query in a goal-directed fashion. This approach is more flexible than flow-insensitive dependent types because we can exploit path-specific invariants [Beyer et al., 2007] (e.g., we can discover and use an invariant that holds for a given instance of an object rather than for all instances). But since our approach never performs explicit inference of invariants, we cannot re-use invariants as effectively as a system for inferring dependent types.

Chapter 7

Using jumping for tractable analysis of event-driven Android programs

Chapter 6 presented an instantiation of the jumping analysis framework from Chapter 5 specialized for leveraging common forms of invariant-based reasoning used by real programmers. This approach is effective when the invariants that queries reply upon for safety meet two conditions: (1) they are explicitly established in the program text, and (2) they are flow-insensitive. The first condition is requirement because the invariants must be recognized by an invariant schema like the ones defined in Section 6.2.2. The second condition is a requirement because (as discussed in Section 6.3.1) the jumping analysis does not maintain flow-sensitivity while jumping.

In this chapter, we present a different instantiation of the jumping framework from Chapter 5 designed to handle queries where neither of these conditions holds. Specifically, we consider the problem of selective flow-sensitive static analysis of **event-driven** systems. These systems are becoming increasingly important due to their prevalent use in web and mobile applications. In event-driven systems, control-flow occurs via invocation of event callbacks that may or may not be ordered. **Inter-event** flow-sensitive reasoning is often important for precision, but such reasoning can be prohibitively expensive due to the large number of possible event orderings that must be considered (as we will discuss in Section 7.1).

In event-driven systems, event ordering invariants are maintained not by the application, by the framework code that implements the event dispatch logic. This framework code is frequently not available to the analysis, and even if the code is available the invariants may be too difficult to recognize given the complexity of the code (in either case, the first condition above is violated). Furthermore, the ordering of events is a flow-sensitive property, which violates the second condition defined above.

Clearly, we need a new kind of jumping analysis to deal with the unique structure of even-driven systems. At a high-level, our idea is to precisely follow backward control-flow (i.e., behave like the analysis of Chapter 4) until an event boundary is reached, then perform control-flow abstraction by jumping to relevant events. We use data-relevance information to identify a small set of events relevant to the current query, then use control-feasibility information to filter out events that could not have fired before the current event (and thus maintain flow-sensitivity while jumping).

This chapter is organized as follows. Section 7.1 explains the challenges of analyzing event-driven Android programs and briefly describes our approach to meeting these challenges. We have already presented a motivating example of the jumping analysis we formalize in this chapter in Section 3.2 (specifically, we motivate the importance of our Android client in Section 3.2.1 and show the process of the analysis in Section 3.2.3), which we encourage the reader to revisit in the context provided by this chapter. Section 7.2 shows how we can represent inter-event ordering information from the Android documentation in order to perform control-feasibility filtering. Section 7.3 combines the control-feasibility filtering with a precise data-relevance relation similar to the one described in Section 6.3 to define a relevance algorithm for events (Section 7.3.2). In Section 7.4, we explain how we avoid many difficulties of modeling the complex Android framework by explicating the reflective bridge between the framework and an application. This approach allows us to analyze the framework code directly rather than requiring specifications or an error-prone harness. Finally, we implemented our jumping analysis in the HOPDROID tool and evaluated our tool on the challenging client of checking null dereferences in event-driven Android programs (Section 7.5). Our results showed that augmenting the goal-directed store abstraction described in Chapter 4 with our event-based control-flow abstraction strategy significantly increased scalability, allowing the analysis to decrease the number of unproven dereferences by an average of 54%. In addition, we found 11 real bugs in four different Android applications, nine of which have already been fixed via our submitted patches.

Portions of this chapter previously appeared in a paper draft currently under submission entitled "Selective Control-Flow Abstraction via Jumping", which was co-authored by Bor-Yuh Evan Chang and Manu Sridharan. Portions of Section 7.4 previously appeared in the SOAP 2015 paper "DROIDEL: A General Approach to Android Framework Modeling" [Blackshear et al., 2015], which was co-authored by

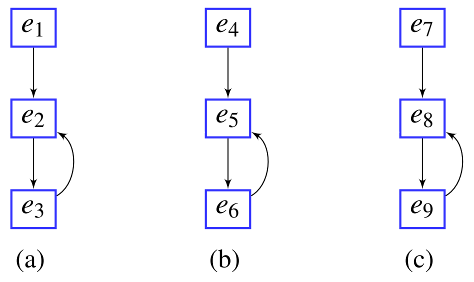


Figure 7.1: A simple event system containing three components with independent lifecycles.

Alexandra Gendreau and Bor-Yuh Evan Chang.

7.1 Challenges of analyzing event-driven programs

To illustrate the problem posed by precise tractable event-driven analysis, consider the simple event system in Figure 7.1. This system has three components (a), (b), and (c) with independent **event lifecycle graphs**. Events within an individual lifecycle graph are ordered by directed edges: $e_1 \to e_2$ specifies that e_1 must execute before event e_2 . Otherwise, the events are unordered with respect to events in other lifecycle graphs. For example, the system specifies that any of events e_2, e_4–e_9 can execute immediately after e_1, but e_3 cannot. Event interleavings across lifecycle graphs are important to consider since all events may access shared mutable state.

The challenge in performing a flow/path-sensitive analysis of such systems is respecting intra-lifecycle ordering constraints while soundly accounting for interleavings of event lifecycles. The obvious approach to achieving this result is to compute and analyze the product graph of all event lifecycle graphs in the event system. However, the number of edges in the product graph will be exponential in the number of components, and all such edges must be considered to perform a flow/path-sensitive analysis (even for the tiny system of Figure 7.1, the product graph contains 27 edges). For practical event systems with tens of components and hundreds of events like the Android applications we consider in Section 7.5, this graph quickly becomes intractable to represent—let alone analyze.

In practice, additional complications arise that make this problem even more difficult. Analyzing an individual event can be quite expensive because each event is essentially a standalone program—it may call thousands of procedures and contain loops and recursion. Component lifecycles can execute more than once, so the analysis may have to visit each edge in the product graph multiple times in order to compute a fixed point. Finally, systems like Android implement lifecycle components and events via objects and instance methods, so the analysis may need to consider an unbounded number of instances of each lifecycle component.

Our insight is that although inter-event flow-sensitive reasoning is required to prove many properties of event-driven systems, most of these properties can be proven without considering all of the possible interleavings across component lifecycles. To leverage this insight to improve scalability, we need a **selective** form of control-flow abstraction flexible enough to apply flow/path-sensitive reasoning within an event lifecycle, but not across all edges in the lifecycle product graph. Though previous approaches to control-flow abstraction have effectively addressed the problem of selective context/object-sensitivity [Oh et al., 2014; Smaragdakis et al., 2014a; Sridharan and Bodík, 2006; Zhang et al., 2014], we are not aware of any previous work that can vary flow/path-sensitivity in the manner desired here. Previous flow-sensitive approaches to analyzing Android applications (e.g., [Fritz et al., 2014]) have avoided this issue by assuming the lifecycles of different components cannot interleave, but this is unsound and (as we have seen in the example of Figure 3.2 discussed in Section 3.2.3) can cause the analysis to miss real bugs.

In the remainder of this chapter, we tackle the challenge of selective flow/path-sensitive abstraction by using the framework for control-flow abstraction via jumping defined in Chapter 5. Our key idea that if we can identify the set of events that may affect the the query at hand, we only need to reason about all possible orderings between these events in order to be sound. If we can coarsen the control-flow abstraction by removing these irrelevant event orderings, we can give the analysis many fewer cases to explore. We have found that since the number of relevant events for a given query is typically small in practice, this approach is tractable even for large event systems. Our jumping framework allows us to limit analysis to relevant events while retaining flow/path-sensitivity for the query at hand.

The core idea behind our approach is an alternation between following data dependences using a **data-relevance relation** and considering control dependences using a notion of **control-feasibility**. The data-relevance relation enables the analysis to identify commands that may affect the current query, while control-feasibility information allows the analysis to consider event lifecycle constraints to preserve flow/path-sensitivity while jumping.

7.2 Representing inter-event control-flow

In this section, we explain how we represent the inter-event control-flow in Android applications in a way that allows a static analysis to address the challenges laid out in Section 7.1. We will make use of this information to check inter-event control-feasibility when we define a practical relevance relation for Android in the next section (Section 7.3).

For Android programs, we must consider two distinct kinds of control-flow information: intra-event control flow and inter-event control flow. Handling intra-event control-flow is the same as handling inter-procedural control flow in an ordinary Java program, which is a well-understood problem. Control-flow between methods can be represented using a call graph and control-flow within a method can be represented using a control-flow graph for the method.

Representing **inter-event** control-flow is more difficult because this information is not directly represented in the call graph. In fact, the logic for maintaining orderings among events lives in native code in the Android framework, so ordering information cannot be inferred by analyzing the Java portion of the framework alone.

Our approach to representing inter-event control-feasibility constraints is to formally define the meaning of the event ordering information that programmers have access to: the lifecycle documentation for Android components (e.g., the Activity lifecycle[1]). This documentation takes the form of **lifecycle graphs** where nodes are lifecycle event methods and directed edges express ordering constraints among the events. We have already seen how such graphs are useful in Section 3.2.3: Figure 3.3 specified the ordering of lifecycle events for the components used in the example and allowed the analysis to filter the set of relevant events to be jumped to.

To go from the documentation to a graph to a representation that can provide control-feasibility information during static analysis, we need the following:

[1] `http://developer.android.com/guide/components/activities.html#Lifecycle`

(1) A well-defined **semantics** for Android lifecycle graphs. Our analysis can then use these graphs to filter out irrelevant transitions based on the control-feasibility condition of relevance soundness (Condition 5.3.1(b)).

(2) A specialization of generic lifecycle graphs of core Android components (e.g., Activity, Service) to a lifecycle graph for a specific **application subclass** of that component. This specialization resolves Java method overriding to make explicit the method code for each **application subclass**, and for precision, it incorporates other callbacks, such as those for handling user interface widgets.

(3) A way for the analysis to resolve lifecycle events on **object instances**. Since events in Android are methods on lifecycle objects, we need to prove that object instance $\widehat{o}_1$ **must-aliases** $\widehat{o}_2$ for two events $\widehat{o}_1.m_1$ and $\widehat{o}_2.m_2$ in order to show that $\widehat{o}_1$ and $\widehat{o}_2$ are constrained by the same lifecycle graph. If we cannot prove this fact, it is unsound to do any control-feasibility filtering because $\widehat{o}_1$ and $\widehat{o}_2$ could be different instances of the same lifecycle class (and therefore have potentially independent lifecycles).

Giving semantics to Android lifecycle graphs Consider the lifecycle graphs in Figure 3.3. These graphs specify the sequence of possible event traces for a particular Android lifecycle component (though the Android documentation never explicitly explains their meaning). If we think of the nodes of a lifecycle graph as labels for their outgoing edges, we can interpret a lifecycle graph as a nondeterministic finite automata (NFA) that accepts the language of all feasible concrete event traces for its lifecycle component. In order to account for partial traces (e.g., traces ending in an exception that interrupts the lifecycle), every node must be an accepting state. For example, the lifecycle graph for the `HostActivity` class from Figure 3.3 (reproduced for convenience) corresponds to the NFA on the left of Figure 7.2.

In order to connect the meaning of a lifecycle graph G to our model of concrete program execution, let us consider labeling a NFA edge not with the name of its corresponding event e, but with the entry transition of the event method, which we write as $\mathrm{entry}(e)$. This means that the strings accepted by the lifecycle NFA (which we write as $\ulcorner G \urcorner$ for a lifecycle graph G) are strings of transitions t (i.e., traces T) rather than strings of events e. We can now state a soundness condition for lifecycle graphs.

Condition 7.2.1 (Lifecycle graph soundness) If concrete execution can reach event $e \in G$, the lifecycle

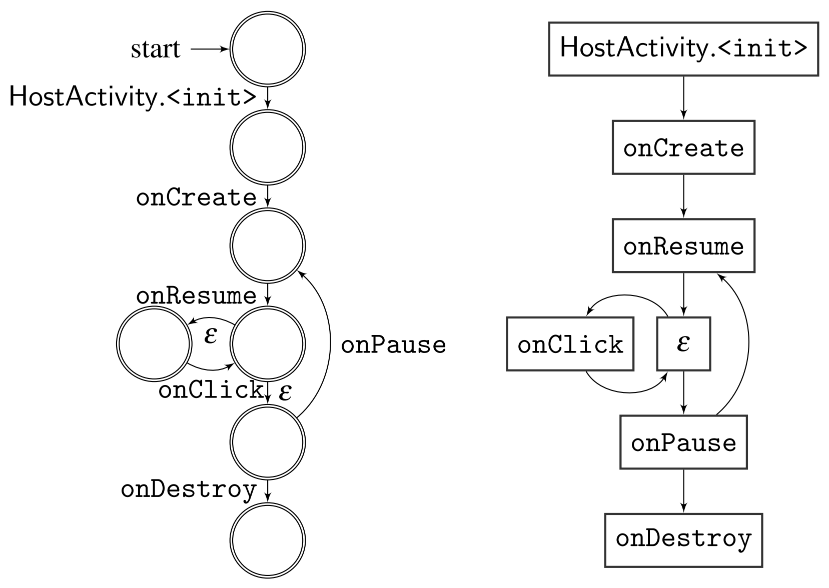


Figure 7.2: Converting the HostActivitylifecycle graph from Figure 3.3 (right) to an event trace-accepting NFA (left).

$$\begin{array}{ll}
\text{event graphs} & G ::= \{\ldots, e_1 \to e_2, \ldots\} \\
\text{events} & e ::= C.m \mid \widehat{o}.m \mid \varepsilon \\
\quad \text{classes} \;\; C & \text{methods} \;\; m \\
\text{memories} & M ::= \mathsf{any} \mid x \mapsto \widehat{v} \mid \widehat{v} \cdot f \mapsto \widehat{u} \mid M_1 * M_2 \mid M \wedge P \\
\text{pure formulæ} & P \\
\quad \text{program variables} \;\; x & \text{object instances} \;\; \widehat{o}, \widehat{v}, \widehat{u}
\end{array}$$

Figure 7.3: Lifecycle graphs and abstract memories.

graph G accepts the concrete trace projected onto the transitions of the lifecycle graph. More formally, if $\langle \sigma, \ell_{\mathsf{dummy}} \rangle \xrightarrow[T\,\hat{}\,t]{}^{*} \langle \sigma', \ell' \rangle$ and event $e \in G$ where $t = \text{entry}(e)$, then $\ulcorner G \urcorner$ accepts events$(T\,\hat{}\,t, G)$.

The function events(T, G) simply projects a concrete trace T onto the transitions of a lifecycle graph G:

$$\text{events}(T, G) \stackrel{\text{def}}{=} \begin{cases} t\,\hat{}\,\text{events}(T_1) & \text{if } T = t\,\hat{}\,T_1 \text{ and } \exists\, e \in G.\ \text{entry}(e) = t \\ \text{events}(T_1) & \text{if } T = t\,\hat{}\,T_1 \text{ and } \nexists\, e \in G.\ \text{entry}(e) = t \\ [] & \text{if } T = [] \end{cases}$$

We assume that the lifecycle graphs specified in the Android documentation are sound.

Specializing lifecycle graphs to application classes Android applications hook into the framework by subclassing special Android core components like Activity. Thus far, we have discussed events rather abstractly, but events in Android correspond to methods on Java objects. We make this explicit by considering events as pairs of the method and the class in which it is defined (i.e., $C.m$) or as pairs of the method and the receiver object on which it is invoked (i.e., $\widehat{o}.m$) as shown in Figure 7.3. A well-formed lifecycle graph can consist of class-method events or object-method events; we call the former version a **static lifecycle graph** and the latter a **dynamic lifecycle graph**. We will explain the special event ε and the structure of abstract memories M shortly.

The Android lifecycle documentation specifies the ordering of methods for core components, but we would like static lifecycle graphs specialized for the application classes. The specialization of lifecycle methods is straightforward by following the method resolution semantics of Java given a class hierarchy. Suppose we wish to specialize a general lifecycle graph G describing an Android core component C_{core} for an application subclass C_{app} (i.e., $C_{\text{app}} <: C_{\text{core}}$). For each event method node $C_{\text{core}}.m$ in G, we replace the node with $C.m$ where C is the class from which C_{app} inherits method m (e.g., $C = C_{\text{app}}$ if C_{app} overrides m).

An application class can also register custom callback methods that are triggered by external events such as user interaction. For example, the HostActivity class from Figure 3.2 extends the `OnClickListener` interface, overrides the `onClick` method, and registers itself as the listener for `onClick` events by calling `setOnClickListener(`**this**`)` at line 5. For soundness, we need to account for all such callback methods, which we could do simply by treating them as independent lifecycle components that have no ordering constraints. However, for precision it is important for the analysis to associate these callback methods with the appropriate component. The analysis should also understand that these user-triggered events can only occur during the "active" phase of the registering lifecycle component when the user can interact with the component. For Activity components, this active phase is the interval between `onResume` and `onStop`.

We incorporate custom callback events into the lifecycle graph with a simple flow-insensitive analysis. For an application class C_{app}, we consider its reachable methods in the call graph to determine what custom callbacks it may register. This analysis considers both callbacks statically registered in the XML configuration files and callbacks dynamically registered with calls to methods like `setOnClickListener`. To represent the active phase of a class $C <:$ Activity, we introduce an ε event between `onResume` and `onClick` events as we saw in Figure 7.2. An ε event is a no-op event that translates to an ε-transition in the NFA formulation.

Once we have identified the set of callbacks $\{\ldots, e_{\text{cb}}, \ldots\}$ that can execute during the active phase of the registering component, we "attach" each custom callback event e_{cb} to the active phase with edges $\varepsilon \to e_{\text{cb}}$ and $e_{\text{cb}} \to \varepsilon$. This models the fact that the user may or may not trigger an interaction event and that interaction events can be triggered an arbitrary number of times. This analysis is flow-insensitive because we do not consider the program point where registering methods like `setOnClickListener` are called. We also do not consider orderings between core lifecycle components (e.g., modeling the launching order of Activity's). Incorporating this information via techniques like those presented in [Yang et al., 2015] could improve the precision of our static lifecycle graphs.

Callback registering methods like `setOnClickListener` may register any object with the appropriate method defined as the callback object (not just the **this** object). A common pattern is to use anonymous inner classes to implement these callbacks, as the anonymous ServiceConnection object created at line 3 of Figure 6.1 does. As a consequence, a lifecycle graph may need to contain methods invoked on multiple object instances (e.g., the **this** object and the anonymous inner class object). We consider this issue next.

Resolving lifecycle events on object instances A significant challenge in leveraging lifecycle information in a flow/path-sensitive analysis is to soundly account for the fact that the lifecycle applies to object instances at runtime. Our approach is to instantiate static lifecycle graphs to object instances during the analysis phase.

To describe this approach more concretely, we consider describing program states using intuitionistic separation logic assertions M that constrain sub-memories structured as shown in the grammar of Figure 7.3. We write any to mean the abstract memory that concretizes to any concrete memory (instead of the standard true, to avoid confusion with the boolean literal). For example, suppose our abstract memory is $(\mathsf{this} \mapsto \widehat{o}_1) * (x \mapsto \widehat{o}_2) * M$ for some memory M while currently analyzing code in some event method $C.m_2$; that is, we are in the lifecycle event $\widehat{o}_1.m_2$ in the corresponding dynamic lifecycle graph with some facts about objects $\widehat{o}_1$ and $\widehat{o}_2$. We would like to leverage an event-ordering constraint $C.m_1 \to C.m_2$ in the static lifecycle graph for C, but for soundness, we have to consider both $\widehat{o}_1.m_1$ and $\widehat{o}_2.m_2$ as possible events.

Our analysis handles this problem by performing an eager case split on aliasing (if we have no existing aliasing information on $\widehat{o}_1$ and $\widehat{o}_2$). That is, just before considering the event-ordering constraint, we case split the abstract state into an aliased case $(\mathsf{this} \mapsto \widehat{o}_1) * (x \mapsto \widehat{o}_2) * M \wedge \widehat{o}_1 = \widehat{o}_2$ and a disaliased case $(\mathsf{this} \mapsto \widehat{o}_1) * (x \mapsto \widehat{o}_2) * M \wedge \widehat{o}_1 \neq \widehat{o}_2$. The aliased case gives us the must-alias fact that we need to soundly leverage the event-ordering constraint for control-feasibility filtering.

The eager case split means that we have a separate proof obligation for the disaliased case where we cannot use the event-ordering constraint in the static lifecycle graph. However, as we will see in more detail in Section 7.3, applying data-relevance often allows us to quickly rule out this case. In the common case that the relevant commands in $C.m_1$ and $C.m_2$ are writes to **this** of the lifecycle object, then data-relevance rules out $C.m_1$. Even if the relevant writes are through non-**this** pointers (e.g., `p.f = ···`), our precise reasoning about aliasing and strong updates typically handles this disaliased case quickly.

7.3 An effective relevance algorithm for Android

In this section, we bridge the gap between theory and practice by combining the jumping framework from Chapter 5 with the formalization of Android lifecycle graphs in Section 7.2 to design HOPDROID, a practical jumping analysis for analyzing event-driven Android programs. We achieve this by devising a sound relevance relation for THRESHER [Blackshear et al., 2013], a precise backward analysis that tightly integrates the results of an up-front points-to analysis to refute separation logic queries (as described in Chapter 4).

We have given the intuition for the jumping strategy that our relevance relation implements in Sections 3.2.3 and 7.1: when the analysis reaches an event boundary, it identifies events that contain data-relevant commands, filters the set of data-relevant events using control-feasibility constraints based on Android lifecycle information, then jumps to the remaining events. To realize this vision, we need to address two issues: (1) precisely computing data-relevance information for the separation logic constraints that THRESHER uses and (2) using the semantics of lifecycle graphs to perform control-feasibility filtering. We show how we solve these issues in Section 7.3.1 before presenting an algorithm for computing relevant transitions that utilizes our solutions in Section 7.3.2.

7.3.1 Heap data-relevance and event control-feasibility

To utilize Android lifecycle information in our relevance relation, we must connect the meaning of lifecycle graphs from Section 7.2 (Condition 7.2.1) to the control-feasibility condition of relevance soundness (Condition 5.3.1(b)). To maintain precision while jumping from one event to another, we must ensure that we only perform jumps that respect the ordering constraints encoded in Android lifecycle graphs (while simultaneously considering the necessary interleavings to be sound). Our solution here is to utilize the reachability and **postdominance** information encoded in the lifecycle graph.

Using lifecycle graphs for control-feasibility filtering Since our analysis is backward, only jumps from the current program point to a preceding transition in a concrete execution trace ending at the current program point are control-feasible. Thus, we can filter a set of possibly relevant transitions using control-feasibility by considering backward reachability in the lifecycle graph.

If some event e is not backward-reachable from the current event e_{cur}, then we know no concrete trace ending in e_{cur} can possibly have visited e first (following the semantics of lifecycle graphs as concrete trace-accepting NFAs in Condition 7.2.1). Thus, we can prune all events that are not backward-reachable from e_{cur} from the lifecycle graph G to produce a pruned lifecycle graph G' where e_{cur} is a leaf node.

For example, we can use this technique to reason that if the analysis is currently in the `onClick` event of Figure 3.3, the `onDestroy` event has not yet occurred in the current lifecycle. If we prune nodes and edges not backward reachable from the current node `onClick`, we can prune the `onDestroy` event. We do not need to consider jumps from e_{cur} to pruned events.

The analysis can further refine the possible jump targets using postdominance on the pruned graph G'. Consider the postdominance tree rooted at e_{cur}, that is, a tree where each node is the immediate post-dominator of its children. For any set of potentially relevant events E, we only need to consider the smallest set $E' \subseteq E$ such that E' postdominates E. As a consequence, for all $e \in E$, there is an $e' \in E'$ such that e' is between e_{cur} and e in the postdominator tree rooted at e_{cur}. The correctness of this reasoning follows directly from the meaning of lifecycle graphs and the definition of postdominance: if e_{cur} postdominates e' and e' postdominates e, we can conclude that every trace accepted by the lifecycle NFA that visits e_{cur} always visits e' beforehand without visiting e in between.

To give a more concrete example using the HostActivity lifecycle graph in Figure 3.3, we would like to able determine that if the analysis is currently in the `onClick` event and we know that only the `onCreate` and HostActivity.`<init>` events are relevant, we only need to jump to `onCreate`. We can derive this fact by demonstrating that `onClick` postdominates `onCreate` and `onCreate` postdominates HostActivity.`<init>` in G'.

Computing data-relevance for heap dependencies We have previously our approach to computing precise data-relevance information for heap dependencies in Section 6.3. This data-relevance relation considers all commands that may **weaken** the abstract store (that is, increase the magnitude of its concretization) to be relevant. Here, we take a very similar approach with one important difference: we want to consider all commands that may **modify** the abstract store to be data-relevant, not just the weakening

```
1 x = 1 // query−weakening command
2 x = 0 // query−strengthening command
3 assert (x == 0)   // query x ≠ 0
```

Figure 7.4: An example demonstrating the importance of considering strengthening commands as data-relevant.

commands. Considering modifying commands rather than just weakening commands to be data-relevant is more precise, but more expensive.

The reason is we want to consider modifying commands to be data-relevant here is that we wish to preserve flow-sensitivity while jumping by using control-feasibility information, whereas the strategy outlined in Chapter 6 leverages flow-insensitive invariants and thus does not attempt to retain flow-sensitivity during jumps (as explained in Section 6.3.1). To understand this in more detail, consider the simple example in Figure 7.4. Our analysis will try to refute the query $x \neq 0$ in order to prove the safety of the assertion at line 3. The command at line 2 strengthens this query, whereas the command at line 1 weakens this query. The data-relevance relation defined in Section 6.3 would soundly report only the weakening command at line 1 as data-relevant, but jumping only to this command loses precision and causes the analysis to miss a refutation by skipping over the refuting command at line 2.

On the other-hand, say that we consider both of the commands to be data-relevant. If we perform no control-feasibility filtering (like the approach in Chapter 6), this will lead to the same result as above because we will jump to both commands and fail to refute the case that jumps to line 1. Thus, we might as well report only weakening commands as data-relevant since this reduce the number of cases for the analysis to consider without affecting precision. This is the approach taken by the data-relevance relation in Section 6.3.

With our current approach, we will perform control-feasibility filtering that will allow us to find refutations in the case that a strengthening transition dominates the current program point (as in Figure 7.4). Thus, we want to consider any command that may modify the abstract store to be relevant. Figure 7.5 defines a data-relevance relation that does exactly this. We refer the reader to Section 6.3 for an explanation of the judgment forms in this figure. This data-relevance relation is quite similar to the one defined in Figure 6.3 except that it does not perform any checks on the righthand-side expression of commands. These checks are what allow that data-relevance relation to report only strengthening commands as relevant.

7.3.2 Defining the relevance relation

Finally, we present our algorithm for computing Android-specialized relevance information (Figure 7.6) and argue that our algorithm is sound with respect to relevance soundness (Condition 5.3.1). The algorithm implements the $\langle R, \ell \rangle \rightsquigarrow T_{\text{rel}}$ judgment form for relevance relations and is executed each time the A-JUMP rule is applied.

In the usual case where the current program label ℓ_{cur} is not the entry label of an event, the algorithm behaves like a standard path-sensitive backward analysis by choosing to visit the predecessor labels of the current program label next (lines 2–4). Clearly, this satisfies relevance soundness by satisfying the control-feasibility condition (Condition 5.3.1(b)), as we have already argued in Section 5.3.

In the case that the current program label is the entry label of an event, we perform jumps to a computed set of relevant transitions using the data-relevance and control-feasibility constraints described in Section 7.3.1. Specifically, the algorithm computes the set of data-relevant events that may write to the current abstract state (lines 5–20) and then filters this set of events using control-feasibility information from the lifecycle graph of the current event (lines 22-37).

First, the algorithm computes the set of data-relevant transitions T_{rel} for R_{cur} using the points-to analysis, as we have explained in Section 7.3.1. In principle, the algorithm could return the set T_{rel} and still be sound by satisfying relevance soundness Condition 5.3.1(a), but this would be imprecise because it would not take the ordering of events in the lifecycle graph into account. The algorithm thus walks backward from the calling method of each relevant transition t_{rel} (given by method(t_{rel})) in the call graph until it reaches an event on each path (loop from lines 8–20). The resulting set of data-relevant events E_{rel} contains the set of all events whose execution might lead to a relevant transition. Returning the exit transition of each of these events E_{rel} would also satisfy relevance soundness via a combination of Condition 5.3.1(b) and (a) because by construction of E_{rel}, these exit transitions collectively postdominate all relevant transitions.

$$\boxed{R \rightsquigarrow T_{\text{rel}}}$$

$$\textsc{R-Cases}\quad \frac{R_0 \rightsquigarrow T_0 \quad R_1 \rightsquigarrow T_1}{R_0 \vee R_1 \rightsquigarrow T_0 \cup T_1} \qquad \textsc{R-Bot}\quad \frac{}{\bot \rightsquigarrow \{t_{\mathsf{init}}\}} \qquad \textsc{R-Top}\quad \frac{}{\top \rightsquigarrow \{t_{\mathsf{init}}\}} \qquad \textsc{R-Split}\quad \frac{\hat{\rho} \rightsquigarrow T_1 \quad \hat{L} \rightsquigarrow T_2}{(\hat{\rho}, \hat{L}) \rightsquigarrow T_1 \cup T_2}$$

$$\boxed{\hat{\rho} \rightsquigarrow T_{\text{rel}}}$$

$$\textsc{R-Sep}\quad \frac{\hat{\rho} = M_0 * M_1 \wedge P \quad M_0 \wedge P \rightsquigarrow T_0 \quad M_1 \wedge P \rightsquigarrow T_1}{\hat{\rho} \rightsquigarrow T_0 \cup T_1} \qquad \textsc{R-Assign}\quad \frac{T_{\text{rel}} = \{\, t \mid t \in P \text{ and } t = \ell_i \mathrel{-\!\![x := y]\!\!\rightarrow} \ell_j \,\}}{x \mapsto \widehat{v} \wedge \widehat{v}\ \mathsf{from}\ \mathring{r} \wedge P \rightsquigarrow T_{\text{rel}}}$$

$$\textsc{R-New}\quad \frac{T_{\text{rel}} = \{\, t \mid t \in P \text{ and } t = \ell_i \mathrel{-\!\![x := \mathtt{new}_a\, \tau\mathtt{()}]\!\!\rightarrow} \ell_j \,\}}{x \mapsto \widehat{v} \wedge \widehat{v}\ \mathsf{from}\ \mathring{r} \wedge P \rightsquigarrow T_{\text{rel}}} \qquad \textsc{R-Read}\quad \frac{T_{\text{rel}} = \{\, t \mid t \in P \text{ and } t = \ell_i \mathrel{-\!\![x := y.f]\!\!\rightarrow} \ell_j \,\}}{x \mapsto \widehat{v} \wedge \widehat{v}\ \mathsf{from}\ \mathring{r} \wedge P \rightsquigarrow T_{\text{rel}}}$$

$$\textsc{R-Write}\quad \frac{T_{\text{rel}} = \{\, t \mid t \in P \text{ and } t = \ell_i \mathrel{-\!\![x.f := y]\!\!\rightarrow} \ell_j \text{ and } \mathrm{pt}_{\mathring{G}}(x) \cap \mathring{r} \neq \emptyset \,\}}{\widehat{v}\cdot f \mapsto \widehat{u} \wedge \widehat{v}\ \mathsf{from}\ \mathring{r} \wedge \widehat{u}\ \mathsf{from}\ \mathring{s} \wedge P \rightsquigarrow T_{\text{rel}}} \qquad \textsc{R-Any}\quad \frac{}{\mathsf{any} \wedge P \rightsquigarrow \{t_{\mathsf{init}}\}}$$

$$\boxed{\hat{L} \rightsquigarrow T_{\text{rel}}}$$

$$\textsc{R-Call}\quad \frac{T_{\text{rel}} = \{\, t \mid t \in P \text{ and } t = \ell_i \mathrel{-\!\![\mathtt{call}\ \ell_1]\!\!\rightarrow} \ell_j \text{ and } \ell = \ell_1 \,\}}{\ell :: \hat{L} \rightsquigarrow T_{\text{rel}}} \qquad \textsc{R-AnyString}\quad \frac{}{\mathsf{anystring} \rightsquigarrow \{t_{\mathsf{init}}\}}$$

Figure 7.5: A data-relevance relation that reports (some) commands that **modify** the query and commands that weaken the current query as relevant (in contrast to the data-relevance relation of Figure 6.3, which considers only weakening commands to be relevant).

```
Require: Current abstract state $R_{\text{cur}}$
Require: Current label $\ell_{\text{cur}}$
Require: Program transition relation $P$
Require: Call graph CG
Ensure: Returned transition set $T_{\text{rel}}$ satisfies Condition 5.3.1
 1: // not at event entry, follow predecessors
 2: if $\ell_{\text{cur}}$ is not the entry label of an event then
 3:    return preds($\ell_{\text{cur}}$, $P$)
 4: end if
 5: // at event entry, get data-relevant events
 6: $T_{\text{rel}} \leftarrow$ dataRel($R_{\text{cur}}$) // compute $R_{\text{cur}} \rightsquigarrow T_{\text{rel}}$
 7: $E_{\text{rel}} \leftarrow \emptyset$ // events leading to a relevant transition
 8: for all $t_{\text{rel}} \in T_{\text{rel}}$ do
 9:    $W \leftarrow$ [ method($t_{\text{rel}}$) ] // method worklist
10:    $V \leftarrow \emptyset$ // track visited methods to handle CG cycles
11:    while $W \neq \emptyset$ do
12:       Remove $m$ from $W$
13:       if $m$ is event then
14:          $E_{\text{rel}} \leftarrow \{ m \} \cup E_{\text{rel}}$
15:       else if $m \notin V$ then
16:          Add preds($m$, CG) to $W$
17:       end if
18:       $V \leftarrow \{ m \} \cup V$
19:    end while
20: end for
21: // filter data-relevant events with a lifecycle graph
22: $e_{\text{cur}} \leftarrow$ event($\ell_{\text{cur}}$)
23: $G \leftarrow$ specializeLifecyleGraph(class($e_{\text{cur}}$), CG)
24: $E_{\text{inG}} \leftarrow \{ e \mid e \in E_{\text{rel}} \wedge e \in G \}$
25: $E_{\text{notinG}} \leftarrow \{ e \mid e \in E_{\text{rel}} \wedge e \notin G \}$
26: $E_{\text{feas}} \leftarrow \emptyset$ // data-relevant/control-feasible events in $G$
27: $W \leftarrow$ [ $e_{\text{cur}}$ ] // lifecycle graph event worklist
28: $V \leftarrow \emptyset$ // track visited events to handle cycles in $G$
29: while $W \neq \emptyset$ do
30:    Remove $e$ from $W$
31:    if $e \in E_{\text{inG}}$ then
32:       $E_{\text{feas}} \leftarrow \{ e \} \cup E_{\text{feas}}$
33:    else if $e \notin V$ then
34:       Add preds($e$, $G$) to $W$
35:    end if
36:    $V \leftarrow \{ e \} \cup V$
37: end while
38: return exitTrans($E_{\text{feas}} \cup E_{\text{notinG}}$)
```

Figure 7.6: An algorithm for selecting relevant transitions to visit in event-driven, heap-manipulating Android programs.

However, we can gain additional precision by removing events from T_{rel} based on control-feasibility information from the lifecycle graph, which is what the algorithm does next. Lines 22 and 23 compute a lifecycle graph specialized for the class of the current event e_{cur}, as we have described in Section 7.2. The algorithm then partitions the set of relevant events E_{rel} based on their presence in the lifecycle graph (lines 24–25). It does this because only the events E_{inG} that are in the lifecycle graph should be filtered in the subsequent step—the events in E_{notinG} are unordered with respect to e_{cur} and the algorithm must return all of them for soundness.

The loop from lines 29–37 performs control-feasibility filtering on nodes in the lifecycle graph. This loop computes a subset E_{feas} of E_{inG} that must be returned for soundness. The loop walks backward from the current event e_{cur} in the lifecycle graph G and stops each time it reaches a relevant event. The construction of E_{feas} ensures that at the end of the loop, relevant events that are not backward reachable from e_{cur} will be excluded from the set E_{feas}, and as will events postdominated by both e_{cur} and some other relevant event. We have argued for the soundness of excluding events based on backward reachability and postdominance in the lifecycle graph in Section 7.3.1.

Finally, the algorithm takes the union of the lifecycle graph control-feasible relevant events E_{feas} and the unordered relevant events E_{notinG} and returns their exit transitions as the set of transitions that must be visited (line 38). The set $E_{\text{feas}} \cup E_{\text{notinG}}$ is a subset of the set E_{rel} whose exit transitions already satisfy relevance soundness. We have only removed events from this set by soundly filtering based on lifecycle control-feasibility information, so returning the exit transitions of $E_{\text{feas}} \cup E_{\text{notinG}}$ also satisfies relevance soundness.

7.4 Explicating reflection in the Android framework

In this section, we present a detailed explanation of how our analysis models the Android framework using the DROIDEL[2] tool. Though these details may seem a bit low-level, we believe that it is important to carefully explain how our model works because models used in static analysis often significantly impact the soundness and precision of the analysis. For models of complex frameworks like Android, failure to fully

[2] `https://github.com/cuplv/droidel`

model the execution context of application code can lead to surprising unsoundnesses even when models seem over-approximate for the analysis of interest.

Our approach in DROIDEL is to prevent soundness concerns by avoiding modeling whenever possible. Instead, we focus on explicating the parts of the Android framework that are polymorphic with respect to apps. By instantiating this polymorphism for a given app, we can eliminate difficult-to-analyze code and sources of unsoundness, such as uses of reflection.

In the remainder of this section, we explain how we designed and implemented DROIDEL using this approach. We first discuss the difficult of modeling Android and explain the problems with existing models (Section 7.4.1). Section 7.4.2 focuses on how an analysis designer would use DROIDEL, while Section 7.4.3 explains how our implementation works to enable analysis designers to adapt and extend our approach.

7.4.1 Problems with existing Android framework models

Reflection is a notoriously thorny issue that most static analyses do not handle soundly [Livshits et al., 2015]. Thus frameworks that make heavy use of reflection, like Android, pose problems for static analysis. Because the Android framework is complex and full of reflection, static analyses for Android typically choose to create models of the Android framework rather than analyzing the framework code itself. Creating these models is both tedious and error-prone, as it requires careful study of the framework's source code, documentation, and dynamic behavior. However, carefully crafted models are extremely important because an incomplete or incorrect model can compromise both the soundness and the precision of an analysis.

Since carefully crafted framework models are so important, we would hope that once a well-tested, authoritative framework model for Android has been created, all static analyses for Android would be able to re-use it. Unfortunately, to our knowledge, no such general framework model exists. The primary reason for this current state of affairs is that framework models tend to be **client-specific**—they summarize only the semantics of the framework with respect to a particular analysis client. This enables the framework model designer to abstract away the complex behavior of the framework code that is not relevant to the client of interest. For example, the FLOWDROID taint analysis tool [Fritz et al., 2014] models calls to framework methods from application code via handwritten **taint wrappers** that summarize the framework's behavior for the taint analysis client. Although the creators of FLOWDROID have spent an immense amount of effort understanding and modeling the Android framework, their models cannot be readily reused by other analyses for Android. To see the problem concretely, consider this response on the Soot mailing list[3] from a FLOWDROID developer to a frustrated analysis designer who wishes to build a new analysis client on top of FLOWDROID:

> Question: "The call graph is missing edges. . . ."
> Response: "Another idea would be to just live with the incomplete call graph. . . . . We know that we don't have call edges for some call sites. . . . . You write that you do not want to perform taint tracking. In that case, the taint wrappers provided by FlowDroid will not be of much help."

Clearly, the Android static analysis community would benefit from a general model of the framework that is independent of any particular client and can be used with any program analysis platform. In this section, we present an approach to fill this void.

Android applications (apps) hook into the framework by extending special framework classes such as Activity or Service and overriding known callback methods such as `onCreate` or `onDestroy`. The framework executes an app by using reflection to look up the application classes that extend these special types and to invoke the appropriate callback methods in response to user interaction. In brief, DROIDEL works by explicating this reflection. That is, our approach to "the modeling problem" is to analyze the Android framework code itself but to de-obfuscate the library's usage of reflection. DROIDEL does this de-obfuscation automatically by replacing reflective method calls with automatically generated app-specific stubs that invoke the appropriate app code.

The key observation underlying our approach is that most uses of reflection are simply to make the Android framework generic for all apps. DROIDEL takes advantage of this observation to create a non-reflective, app-specific version of the Android framework for each application it analyzes. The replacement of reflective calls and the generation of stubs is performed entirely at the Java source code level; the output of DROIDEL is a Java program with a single entry-point that can be processed by any existing Java analysis platform (e.g., Soot, WALA, Chord).

[3] `https://mailman.cs.mcgill.ca/pipermail/soot-list/2015-February/007745.html` and `https://mailman.cs.mcgill.ca/pipermail/soot-list/2015-February/007747.html`.

One of our contributions is the open-source DROIDEL implementation following the approach we have advocated. DROIDEL is already being used by researchers from IBM Research, the University of Texas, the University of Maryland, and the University of Colorado for a wide variety of analyses, including taint analysis, malware detection, permission analysis, and null dereference checking.

7.4.2 Designing DROIDEL for general usability

In designing DROIDEL, we focused on the following two principles.

Model the framework as little as possible. Most existing approaches to analyzing Android applications explicitly seek to avoid analyzing the Android framework, but our approach is exactly the opposite. Each bit of framework code that is not analyzed must be carefully modeled to avoid introducing unsoundness (as we argued in Section 7.4.1). Instead of replacing the framework code with a large model, we choose to augment it with small models that minimally explicate the reflection and native code that the framework uses to interact with applications.

Be as standalone as possible. Modeling Android is hard work. We want others to benefit from our modeling efforts. In practical terms, this means that our model must be usable by any client analysis or program analysis framework in order to be widely adopted. To avoid being client-specific, we avoid abstracting away any Android framework code so that we do not eliminate any behaviors of potential importance. To avoid being analysis framework-specific, we generate all of our stubs and models at the Java source code level so they can be understood by any Java program analysis tool.

Note that our explicating approach is entirely compatible with additional modeling such as client-specific modeling (e.g., for increasing precision or improving the scalability of the analysis). We believe that starting from the framework source code and incrementally modeling key portions of the code is bound to lead to more trustworthy analysis results than beginning with a model that has no direct relationship to the framework source.

In building DROIDEL, we began by studying the source code for the Android framework and identifying uses of reflection that may allocate application objects or call methods on application objects. We

```
public interface DroidelStubs {
  // Reflective allocations of app objects
  Application getApplication(String cname);
  Activity getActivity(String cname);
  Service getService(String cname);
  BroadcastReceiver getBroadcastReceiver(String cname);
  ContentProvider getContentProvider(String cname);
  Fragment getFragment(String cname);
  View inflateViewById(int id, Context c);
  // Reflective method invocations
  void callXMLRegisteredCB(Context c, View v);
}
```

Figure 7.7: DROIDEL generates app-specialized stubs that implement this interface. We manually replace reflective calls in the Android framework with calls to `DroidelStubs` methods.

```
class AppStubs implements DroidelStubs {

  Activity getActivity(String cname) {
    if (cname == "ActivityA") {
      return new ActivityA();
    } else if (cname == "ActivityB") {
      return new ActivityB();
    } else { return new Activity(); }
  }

  View inflateViewById(int id, Context c) {
    switch (id) {
      case R.id.passwordView: return new TextView(c);
      case R.id.tweetView: return new TextView(c);
      default: return null;
    }
  }

  void callXMLRegisteredCB(Context c, View v) {
    if (c instanceof ActivityA) {
      ((ActivityA) c).myOnClick(v);
    } else if (c instanceof ActivityB) {
      ((ActivityB) c).myOnClick(v);
    }
  }
}
```

Figure 7.8: A partial implementation of `DroidelStubs` from Figure 7.7 for an app with `ActivityA` and `ActivityB`, two TextViews, and an XML configuration-registered callback `myOnClick`.

then manually replaced each such use of reflection in the Android framework with a call to an appropriate method from the `DroidelStubs` interface shown in Figure 7.7. This interface acts as a bridge between application and framework code—it allows the framework to obtain pointers to application-space objects. The `DroidelStubs` interface also centralizes and serves to document the instances of framework reflection that it explicates.

After replacing uses of reflection with calls to method stubs from `DroidelStubs`, we made one final change to the Android framework code: we changed `ActivityThread.main`, the "main" method that the Android framework uses to run an application, to take an implementation of the `DroidelStubs` interface as input.

The result is a slightly modified version of the Android framework that calls stubs from `DroidelStubs` rather than using reflection in several key places. This modified framework code can be compiled once and then used to analyze any application. The application-specific part of DROIDEL is generating an implementation of `DroidelStubs`, which we explain further in Section 7.4.3.

As the Android framework changes, future uses of framework reflection can easily be handled by adding new methods to this interface, updating the framework with calls to the new methods, and updating the application-specific part of DROIDEL to generate implementations of these new methods on a per-app basis.

Analyzing an app with DROIDEL. To enable whole-program analysis, DROIDEL creates a special `androidMain` method whose body allocates an instance of the auto-generated `DroidelStubs` implementation and calls the `ActivityThread.main` method with this object as its argument. An Android program analysis that wishes to use a DROIDEL-processed program need only import: (a) the application classes, (b) the DROIDEL-generated stub classes, and (c) the modified Android framework classes. The `androidMain` method can be used as a single entry point for whole-program analysis.

7.4.3 Implementation

There are two parts to DROIDEL: (1) a one-time manual modification of the Android framework sources to replace uses of reflection with calls to the appropriate methods of the `DroidelStubs` interface and (2) a per-app code generation module to automatically create an application-specific implementation of `DroidelStubs`.

Manually explicating reflection in the Android framework. To see a concrete example of replacing uses of reflection with calls to stub methods in `DroidelStubs`, consider the following snippet drawn from the `android.app.Instrumentation` class:

```
// Replace the use of reflection (a call to newInstance) with
// a call to the DROIDEL stub getActivity.
~~Activity a = (Activity) clazz.newInstance();~~
Activity a = droidelStubs.getActivity(clazz.getName());
```

We manually identified that the call to `clazz.newInstance()` might create an Activity object from the application, so we replaced this use of reflection with a call to `droidelStubs.getActivity`, a method of the `DroidelStubs` interface. For a method like `getActivity`, the DROIDEL implementation will generate allocations for each subclass of Activity defined in the application.

Application-specific stub generation. When DROIDEL runs on an app, it synthesizes an application-specific implementation of each of the stub methods of the `DroidelStubs` interface. To generate the getter methods for the core Android components Application, Activity, Service, BroadcastReceiver, ContentProvider, and Fragment, DROIDEL parses the application manifest `AndroidManifest.xml` for the app to determine which components have been declared by the developer and then builds the class hierarchy for the app and ensures that it can find each component.

To give an example of what an app-specific implementation of `DroidelStubs` would look like, we continue the discussion of the stub method `getActivity` from above. In Figure 7.8, we show an implementation of `DroidelStubs` for an app with two subclasses of Activity (named `ActivityA` and `ActivityB`). The generated implementation of `getActivity` simply dispatches based on the `cname` parameter. The documentation for `newInstance` states that this reflective call invokes the default (zero-argument) constructor for the given `Class`, so our stub methods allocate each type by invoking its zero-argument constructor. Generating the getter methods for the other core Android components is similar, though there is some special handling of Fragments because their usage has changed slightly as the Android framework has evolved.

Generating the `inflateViewById` stub is slightly different because Views are components of the Android layout rather core Android components, so they are typically declared in resource files (e.g., `res/layout/filename.xml`) rather than in the application manifest. Additionally, View objects have associated identifiers that the app developer can use to distinguish between different layout components at run time. Understanding the association between View objects and their identifiers is crucial for a static analysis because View objects are frequently retrieved using these identifiers with methods such as `findViewById`. For example, the developer might have one TextView object with identifier `R.id.passwordView` and another TextView object with with identifier `R.id.tweetView`. It is important for the static analysis client to understand that calling `findViewById(R.id.passwordView)` will return a different TextView than calling `findViewById(R.id.tweetView)`, or else the state of the two View objects may be conflated.

Thus, DROIDEL parses all of the layout resource files to identify which View objects the app may use and to associate instances with their identifiers. It then generates stubs that model the reflective instantiation of View objects from the layout XML configuration file (called **layout inflation** in Android). For the simple two-TextView layout objects described above, DROIDEL would generated the stub implementation for `inflateViewById` shown in Figure 7.8.

The functionality to generate these application-specific stubs is the core of DROIDEL. These stubs explicate Android framework's use of reflection to allocate application objects specified in XML configuration files. However, there is another tricky use of reflection in the Android framework. In Android apps, the developer can register callbacks either in the application code itself or (for certain callbacks) in the layout XML configuration file for the application. The first kind of registration is handled easily because its behavior is apparent in the framework code (i.e., does not use reflection), but the second kind of registration requires special treatment.

To give an example of the XML registration construct, suppose the application developer uses the layout XML configuration file to register a callback on a Button using the following snippet:

```
<Button android:onClick="myOnClick" ... />
```

The semantics of this XML snippet are that when the layout hierarchy containing this Button is attached to an Android Context object (e.g., via `Activity.setContentView`), the `myOnClick` method of that Activity will be registered as callback to be invoked when the user clicks the Button. The lookup of the `myOnClick` method for a particular Context object is performed reflectively and thus must be explicated to be understood by the static analysis client. DROIDEL deals with this use of reflection by parsing the layout XML configuration file to identify XML-registered callbacks. It then generates an implementation of the `callXMLRegisteredCB` stub method that invokes each method of a Context subclass whose name matches the method name in the layout XML.

Let us assume that the two Activity classes from our running example, `ActivityA` and `ActivityB`, each have an `myOnClick` method with the proper signature for overriding the interface method `OnClickListener.onClick`. The `callXMLRegisteredCB` stub shown in Figure 7.8 corresponds to the implementation that DROIDEL would generate for this app. Since the layout hierarchy that registers the `myOnClick` method in the layout XML can be used in any `Context` object at run time, this way of generating stubs makes sure that every method matching `myOnClick` gets called.

Limitations of DROIDEL. There are many uses of reflection in the complex Android framework that DROIDEL does not (yet) explicate (for example, reflective allocation of Preferences objects). In addition, DROIDEL does not generate stubs to summarize the behavior of native methods in Android. Both of these issues are not fundamental problems with our approach, but limitations of the current implementation that we plan to address in the future.

Another issue is that we currently need to perform the manual explication of reflection in the Android framework separately for each version of the Android framework. We believe that this process can be automated in the future, as the explication that needs to be performed is almost identical for each version of framework we have considered. We note that the state of affairs is worse for harness-based approaches since the semantics of each version of the framework must be manually scrutinized in order to ensure that the generated harness over-approximates its behaviors.

7.5 Case study: proving null dereference safety in event-driven Android programs

In order to evaluate the effectiveness of the jumping strategy outlined in this chapter, we sought to test the following experimental hypothesis:

Jumping is a scalable approach to flow/path-sensitive inter-event analysis. We hypothesize that augmenting a state-of-the-art path-sensitive analysis with jumping increases precision by allowing the analysis to reason about event orderings, yet limits the number of event orderings that must be considered enough to make analysis tractable.

Experimental setup. In order to test our hypotheses, we chose to evaluate jumping analysis on the client of proving the absence of null-dereferences in event-driven Android programs. We chose this client because null dereferences are a common problem in real-world Android apps, and the event-driven nature of Android makes precisely verifying the absence of null dereferences a significant challenge for analyses (see Section 3.2.1 for a more detailed discussion of this client). We implemented the practical jumping analysis described in Section 7.3 in the HOPPER[4] tool, a variant of the WALA- and Z3 [de Moura and Bjørner, 2008]-based THRESHER [Blackshear et al., 2013] tool. HOPPER extends THRESHER by adding the ability to perform jumps, but the tools are otherwise identical.

The core of THRESHER is an engine for refuting queries written in separation logic. Clients are implemented as lightweight add-ons that take a program as input and emit separation logic queries for the core refuter to process. We extended THRESHER/HOPDROID with a new client for checking null dereferences. The client leverages `@NonNull` annotations inferred by the NIT tool[5] to eliminate easy cases where non-nullness of fields, function return values, or function parameters is a flow-insensitive property. For each non-static field read/write `x.f` or function call `x.m()` in the the program, the client emits the necessary bug precondition $x \mapsto \mathsf{null}$ as a query to refute in order to prove dereference safety.

We give THRESHER and HOPDROID a maximum budget of 10 seconds to refute each query. We chose this budget through trial and error—we found that larger budgets did not allow the analysis to find appreciably more refutations, whereas smaller budgets caused too many timeouts. In the case that the tool

[4] `https://github.com/cuplv/hopper`
[5] `http://nit.gforge.inria.fr/`

cannot refute a query within the budget, a timeout is declared and the dereference is reported as a potential bug. We ran all experiments in single-threaded configuration on a Mac desktop machine running Mac OS 10.10.2 with 64GB of RAM and 3.5GHz Intel Xeon processors.

Android applications can make use of concurrency—events execute atomically, but the execution of two events can interleave if the events execute on separate threads. In addition, app developers can use Java threads for multithreaded execution in the usual way. THRESHER and HOPDROID do not soundly account for either of these features, as both tools assumes that all events execute atomically on a single thread. Both tools also do not soundly handle reflection and native code for which we do not have handwritten stubs—these constructs are treated as no-ops.

Representing Android event dispatch Instead of generating a harness to model the event dispatch performed by the Android framework, THRESHER and HOPDROID analyze the actual logic for event dispatch in the Android framework source code. To allow this, we pre-processed each app we analyzed with DROIDEL (cf. Section 7.4) to explicate the reflection the Android framework uses to call application methods and then used the `ActivityThread.main` method of the framework as a single entrypoint for call graph construction. There are several advantages to analyzing the actual event dispatch code instead of using a harness: (1) we do not have to worry about soundly and precisely modeling the execution context of events, which can be a significant challenge, and (2) generating a harness that precisely represents all ordering constraints is impractical, as we have already argued in Section 7.1 and Section 7.4.

7.5.1 Proving dereferences safe with jumping

We ran both THRESHER and HOPDROID on the corpus of ten open-source Android apps shown in Figure 7.9. The apps range in size from 3K source lines of code to 57K source lines of code. Since the primary challenge of analyzing these apps comes from considering interleavings of their lifecycle components, we also report the number of core lifecycle components (i.e., Application, Activity, Fragment, Service, and ContentProvider subclasses) and the total number of events in each app. Our analysis must consider the possibility of interleavings between events of different components for soundness, but must preserve the ordering of events within the lifecycle of a single component for precision. Recall from Section 7.1 that

	Benchmark Size			Unsafe Derefs				HopDroid Effectiveness
Bench	**KLOC**	**Com**	**Evt**	**Deref**	**Nit**	**Thr**	**Hop (Impr %)**	**Total Hop Proven (%)**
drupaleditor	3	10	127	928	679	179	72 (60)	92
npr	5	14	120	829	617	181	51 (72)	94
lastfm♠	13	34	272	4840	3528	954	477 (50)	90
duckduckgo	11	12	174	1969	1341	518	143 (72)	93
github	19	70	572	3603	2520	601	290 (52)	92
seriesguide♠	32	80	871	8184	5438	986	625 (37)	92
connectbot♠	33	13	201	2190	1562	316	74 (77)	97
textsecure	38	63	588	5921	3643	698	330 (53)	94
k-9	55	52	750	19032	11968	3104	1988 (36)	90
wordpress♠	57	98	1325	15066	9775	2431	1362 (44)	91
Total	266	446	5000	62562	41071	9968	5412 (54)	92

Figure 7.9: Proving dereference safety in event-driven Android apps with HopDroid. The “Benchmark Size” column grouping gives the number of (thousands of) lines of application source code (**KLOC**), lifecycle components (**Com**), and events (**Evt**) for each benchmark. The “Unsafe Derefs” column grouping lists the number of possibly-unsafe dereferences in each app before analysis (**Deref**) followed by the number remaining after running Nit, (**Nit**), Thresher (**Thr**), and HopDroid (**Hop**). The HopDroid column also lists the percentage reduction in unproven derefs of HopDroid over Thresher (**Impr %**). The final column grouping gives the percentage of derefs proven safe by HopDroid (**Total Hop Proven**). The “Total” row gives the sum of all numeric rows and the geometic mean of the **Hop Impr** and **Total Hop Proven** percentages. ♠’s indicate benchmarks where our partial manual triaging revealed a true bug.

the size of a reified harness that considers the interleavings between just a single instance per component is exponential the number of components.

The "Unsafe Derefs" columns of Figure 7.9 summarize the results of proving null dereference safety on our benchmark apps with NIT, THRESHER and HOPDROID. Each column reports the number of unproven dereferences after running the tool (where 0 represents proving all dereferences safe, so lower is better). The results show that although about a third of the dereferences can be proven safe using the flow-insensitive analysis of NIT, the path-, flow-, and context-sensitive THRESHER analysis was significantly more precise (providing evidence that precision beyond flow-insensitivity is necessary for proving dereference safety in Android apps). The **Hop Impr** column gives the percentage reduction in unsafe dereferences achieved by running HOPDROID (where 100% represents proving all remaining dereferences safe, so higher is better). HOPDROID substantially improved on the already-significant precision of THRESHER—on average, HOPDROID reduced the number of dereferences unproven by THRESHER by more than half.

The difference between HOPDROID and THRESHER is that the jumping capability of HOPDROID enabled precise inter-event analysis, as we predicted in our experimental hypothesis. We noticed that when THRESHER reaches an event boundary without proving safety, it continues precise backward analysis of the event dispatch code of the Android framework and (almost always) times out without finding a proof. By contrast, HOPDROID jumps from an event boundary to a (typically) small set of relevant events and is frequently able to prove safety based on precise and tractable inter-event reasoning.

The final **Total Hop Proven** column shows that for every benchmark, HOPDROID proved at least 90% of the dereferences safe (92% safe on average). We note that previous state-of-the-art work in null dereference checking for ordinary, non-event-driven Java programs (e.g., [Loginov et al., 2008; Madhavan and Komondoor, 2011; Margoor and Komondoor, 2015; Nanda and Sinha, 2009]) reports proving 84-91% of dereferences safe on average. Achieving similar precision results in the presence of the formidable scalability challenges introduced by an event-driven setting is a significant advance.

7.5.2 Manual triaging of alarms

To understand why HOPDROID sometimes fails to proof safety, we manually triaged a sample of 20 unproven dereferences from each of our 10 benchmark applications (a total of 200 alarms). We classified the unproven dereferences into three categories (a) true bugs, (b) scalability issues, or (c) precision issues. We placed a dereference into the true bugs category if we found a concretely feasible sequence of events would lead the application to throw a NullPointerException. We classified a dereference as a scalability issue if we determined that HOPDROID possessed the necessary precision to prove the dereference safe, but was not able to do so within the 10 second budget. Finally, we labeled a dereference as a precision issue if HOPDROID did not have the precision required to prove the query correct. This category includes both analysis imprecision (e.g., loop invariant inference, container abstraction) as well as modeling imprecision (e.g., Android UI models, Android/Java reflection and native code).

The results from our manual triaging are shown inset. In the 200 alarms we examined, most dereferences that cannot be proven safe are due to precision issues (172). Of these 172 alarms, 132 would require more precise modeling of the Android framework and 41 are due to more fundamental analysis imprecision. Many of Android modeling issues are additional constraints on the interaction between different lifecycle components that we do not account for. For example, proving safety of some dereferences required understanding details such as the order in which Activity's launch each other or the fact that a callback on a Button cannot be invoked if the `visible` attribute of the Button is set to false. Handling all of the corner cases of the complex Android framework is challenging task that we leave to future work.

(a) Bug	(b) Scalability	(c) Precision
11	17	172

Nearly all of the the analysis imprecision issues stem from imprecise abstraction of containers and strings. Both of these precision problems are orthogonal to HOPDROID's approach to analysis of event-driven programs and could in principle be addressed by enhancing HOPDROID with better abstractions or solvers (e.g., [Dillig et al., 2011a] for containers and [Kiezun et al., 2012] for strings).

The fact that only 17 of the 200 unproven dereferences we examined could not be proven due to scalability issues strengthens our conviction that jumping is an effective approach for tractable analysis of event-driven systems. Though HOPDROID is not perfect, it proves an impressive 92% of the dereferences it encounters. The vast majority of proof failures are due to our incomplete modeling of Android rather than scalability issues.

Bugs found We found eleven bugs in four different apps: lastfm (1), seriesguide (5), connectbot (4) and wordpress (1). The bug in wordpress had already been eliminated by the developers, though in an indirect way (replacing the functionality in the buggy class with an entirely new class). We sent pull requests fixing the bugs in each of the remaining projects. The developer of seriesguide and connectbot accepted all of our pull requests. The developers of lastfm have not yet responded to our pull requests. This project is updated infrequently and has a backlog of pending pull requests.

Of the eleven bugs that we found, five of them involved misunderstanding or misusing the Android lifecycle in some way. This strengthens our belief that the lifecycle is a source of confusion for developers that would be well-served by better analysis tools. We further note that four of the five bugs involved interactions between the lifecycles of different components. These bugs could not be found by an unsound approach that models the lifecycle of each component, but does not consider interleaving lifecycles of different components.

7.6 Related work

Static analysis of Android applications. Numerous techniques have considered static analysis of Android apps, but to the best of our knowledge few have tackled the problem that we address in this chapter: soundly considering the interleaving of different lifecycle components. The harness method generated by the state-of-the-art FLOWDROID [Fritz et al., 2014] tool soundly reflects the sequential execution of component lifecycles, but not their interleaving. This unsound modeling avoids the cost of computing a product graph as described in Section 7.1, but will miss bugs like the five lifecycle-sensitive bugs we found in Section 7.5 along with the bug explained in Section 3.2.3.

ANADROID [Liang et al., 2013] is the only tool we are aware of that explicitly claims to handle interleavings between lifecycles of different components. Their **entry point saturation** technique efficiently computes a fixed point over all possible event ordering. However, this computation does not take intra-lifecycle event orderings into account and thus will lose precision. We found this kind of precision to be crucially important in Section 7.5—HOPDROID's improvement over THRESHER comes entirely from inter-event reasoning.

The GATOR tool presents an intriguing representation of Android control-flow in the form of a **call-back control flow graph** (CCFG). This structure represents dependencies between launched Activity's, callback registrations and invocations, etc. Incorporating information from the CCFG into our analysis would likely improve precision by yielding additional control-feasibility constraints.

Harness generation for Android. Previous work has developed numerous techniques for modeling various features of Android in order to avoid analyzing the framework code like we do (Section 7.4). SCANDROID [Fuchs et al., 2009] and FLOWDROID [Fritz et al., 2014] were the first static analysis tools to consider modeling the event-driven lifecycle of the core components in Android. DROIDSAFE [Gordon et al., 2015] attempts to analyze some of the framework code while replacing other parts with **accurate analysis stubs** that summarize framework behavior with respect to tainting and points-to analysis. These stubs are correct for points-to and taint analysis, but abstract away other behaviors of the framework. This approach is not compatible with our goal of being a general model independent of any particular client. In addition, their stubs do not seem to take account of the application under analysis, which can lead to unsound results. For example, their model[6] of layout inflation in `View.java` allocates a single View instance rather than considering the fact that layout inflation may instantiate any of the View's declared in the layout XML configuration file for the app currently being analyzed.

The GATOR tool [Rountev and Yan, 2014; Yang et al., 2015] of Yang et al. and the SMARTDROID [Zheng et al., 2012] tool focus on precisely modeling the control-flow not only between lifecycle callbacks, but also between callbacks registered to GUI components. COMDROID [Chin et al., 2011], EPICC [Octeau et al., 2013], and APPOSCOPY [Feng et al., 2014] specialize in modeling the Intent mechanism that Android uses to implement **inter-component-communication** between core components of a single app and (in some cases) between core components of different apps on the same device.

[6] `https://github.com/MIT-PAC/droidsafe-src/blob/master/modeling/api/android/view/View.java#L288`

Though not all of these approaches explicitly reify a harness modeling the application callbacks invoked by the Android framework, they are all (to the best of our understanding) **harness-based** in the sense that they model the invocation behavior of the framework by considering a hard-coded set of callback methods to be entry points for analysis. By contrast, DROIDEL works by explicating reflection in the framework and then analyzing the framework code to allow the analysis itself to determine what callbacks may be invoked.

Modular analysis approaches like INFER [Calcagno et al., 2015] avoid modeling Android by ignoring the library and performing modular analysis of each application method independent of its calling context. However, this approach will not be able to identify lifecycle-related bugs like many of the ones we found.

Analysis of asynchronous and event-driven programs. Identifying a small set of commands relevant to the query and their corresponding events using data-relevance exploits the fact that the data dependencies of a program are often less complex than its control dependencies in practice. Recent techniques for concurrent program verification [Farzan et al., 2013] and bug finding [Burckhardt et al., 2010] have used a similar insight: an effective way to prevent the complexity of a concurrent program analysis (static or dynamic) from growing exponentially in the number of threads is to design the analysis around tracking data dependencies rather than control dependencies. This approach works because control dependencies typically explode as additional threads are added, but data dependencies usually do not. Jumping based on a relevance relation allows the analysis to exploit both data-relevance and control-feasibility information to improve scalability, and jumping can be applied in sequential, concurrent, and event-driven settings.

Jhala et al. show that the IFDS framework can be extended to enable analysis of event-driven programs and present a goal-directed algorithm for proving safety properties in their extended framework [Jhala and Majumdar, 2007]. Their focus is on handling unordered events whose execution may interleave, whereas we focus on the problem of preserving the ordering between lifecycle events whose execution is atomic.

Handling Java reflection. As mentioned in Section 7.4, reflection is a challenging feature for static analyses to handle soundly both in Java and in other languages [Livshits et al., 2015]. Several approaches to handling Java reflection more soundly have been proposed. Tamiflex [Bodden et al., 2011] uses dynamic analysis to observe the targets of reflective method calls at run time, then uses this information to generate reflective summaries that are sound with respect to the observed concrete behavior. The solution offered by Tamiflex is much more general than our Android-specific reflection handling, but the instrumentation Tamiflex performs does not work with Android applications.

Recent work by Li et al. [Li et al., 2014] and Smaragdakis et al. [Smaragdakis et al., 2014b] present promising new approaches to fully-static resolution of reflective calls in Java. Both techniques leverage meaningful operations performed on the return value of reflective calls (such as downcasts) to provide a more sound handling of reflection without compromising scalability.

Chapter 8

Conclusion and future directions

In this dissertation, we have presented goal-directed abstraction coarsening. Our approach works backward from an abstraction of the goal query that is as precise as possible by default, but can be coarsened in order to improve the scalability of the analysis and keep the analysis focused on the query. Our thesis was that this is a **flexible** and **practical** approach to goal-directed static analysis. We have supported the claim of flexibility by presenting a goal-directed store abstraction (Chapter 4) and a framework for goal-directed control-flow abstraction based on jumping (Chapter 5). Both of these abstractions can be coarsened on-the-fly at any point during automated analysis along each of the dimensions explained in Chapter 2.

We have supported the claim of practicality by combining our store and control-flow abstractions to design three different goal-directed static analyses. The THRESHER tool uses our goal-directed store abstraction to precisely represent and efficiently refute heap reachability queries. The HOPPER tool extends THRESHER by combining our goal-directed store abstraction with the ability to coarsen the store abstraction by jumping to relevant code based on data-relevance information. HOPPER chooses to jump when it detects that the safety of the goal query may rely on a flow-insensitive invariant established earlier in the program. We show that this strategy allows HOPPER to quickly prove the safety of tough downcasts that could not be handled by the THRESHER approach alone. Finally, HOPDROID uses our framework for control-flow abstraction to perform THRESHER-style analysis at the intra-event level, but jumps based on data-relevance and Android lifecycle control-feasibility information at event boundaries. This combination allows tractable analysis of event-driven Android apps while soundly accounting for all possible event orderings.

8.1 Future directions

This dissertation lays a foundation for thinking about goal-directed analyses that opens many doors for future work. We briefly discuss a few ideas that we consider to be particularly promising.

A relevance relation for concurrent program analysis. Analyzing concurrent programs is extremely challenging due to the state-space explosion that results from considering all possible interleavings of threads. As with analysis of event-driven programs (cf. Section 7.1), the key challenge for concurrent program analysis (both over- and under-approximate) is effective control-flow abstraction to soundly reduce the number of interleavings to be considered (e.g., via **partial order reduction** [Coons et al., 2013; Flanagan and Godefroid, 2005; Peled, 1993; Valmari, 1989]).

We believe that control-flow abstraction via jumping can be applied to this problem using a strategy similar to the one we used for analysis of event-driven programs. For example, we could design an analysis that uses precise control-feasibility information within code regions protected by a lock, then jumps based on data-relevance information once it reaches the boundary of a locked region. In addition, we could construct graphs representing the spawning/joining structure of threads (similar to the lifecycle graphs from Section 7.2) to perform control-feasibility filtering during jumping. Using data-relevance information would hopefully limit the number of interleavings to a small number relevant to the query, whereas using control-feasibility information would further reduce the interleavings to consider and allow us to maintain flow-sensitivity while jumping.

Answering multiple queries simultaneously. This dissertation has focused primarily on the problem of refuting a single query as quickly as possible. However, if a goal-directed analysis has multiple queries to answer, it may be able to achieve better performance by answering multiple queries simultaneously. Many existing goal-directed analyses (e.g., [Oh et al., 2014; Zhang et al., 2014]) take advantage of this fact. Our analysis is capable of trivially handling multiple queries at the same time by answering them in parallel, but we have not tried more interesting strategies for grouping queries together to be handled by a single thread. We could try grouping together queries from the same procedure or basic block, or try to identify queries involving similar parts of program store by using the points-to analysis.

In a similar vein, this dissertation has not considered the problem of effective summaries for goal-directed analysis. We use simple top-down summaries at loop heads, procedure boundaries, and jump targets that prevent the analysis from exploring paths it has already seen (Section 4.3.4, “Query simplification” paragraph and Section 5.2.3). More sophisticated strategies are certainly possible and would likely improve performance.

Backtracking and ensemble solving. In our description of the coarsening-based approach to goal-directed analysis, we have spoken as if the choice to coarsen is final and cannot be undone. This is not the case: at each point the analysis chooses to coarsen, the analysis could backtrack to that point after failing to find a refutation, choose not to coarsen instead, and continue analysis. We did not consider this option due to its (potentially) prohibitive cost, but exploring more clever techniques for backtracking may give us a way to coarsen without having to worry about losing precision.

A similar idea for avoiding precision loss during coarsening is to run multiple coarsening strategies in parallel (for example, we could run THRESHER, HOPPER, and HOPDROID on a query simultaneously) in the style of **portfolio solving** for SAT/SMT solving [Wintersteiger et al., 2009; Xu et al., 2008] or **ensemble methods** for machine learning [Zhou, 2012]. These approaches have been successful in a variety of domains because multiple solvers/classifiers can only yield better results than a single solver/classifier. Using a portfolio of coarsening strategies that make vastly different precision/scalability tradeoffs is a promising approach to getting the best performance and precision for each query.

Bibliography

Lars O. Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, University of Copenhagen, DIKU, 1994.

Thomas Ball, Orna Kupferman, and Greta Yorsh. Abstraction for falsification. pages 67–81, 2005.

Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with SLAM. Commun. ACM, 54(7):68–76, 2011. doi: 10.1145/1965724.1965743. URL `http://doi.acm.org/10.1145/1965724.1965743`.

Michael Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'05, Lisbon, Portugal, September 5-6, 2005, pages 82–87, 2005. doi: 10.1145/1108792.1108813. URL `http://doi.acm.org/10.1145/1108792.1108813`.

Nels Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. Proofs from tests. pages 3–14, 2008.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. volume 3780 of Lecture Notes in Computer Science, pages 52–68. Springer, 2005. ISBN 3-540-29735-9.

Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. volume 4590 of Lecture Notes in Computer Science, pages 178–192. Springer, 2007. ISBN 978-3-540-73367-6.

Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. Commun. ACM, 53(2):66–75, 2010.

Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In PLDI, pages 300–309, 2007.

S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications, pages 169–190, New York, NY, USA, October 2006. ACM Press. doi: http://doi.acm.org/10.1145/1167473.1167488.

Sam Blackshear and Shuvendu K. Lahiri. Almost-correct specifications: a modular semantic framework for assigning confidence to warnings. In ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013, pages 209–218, 2013. doi: 10.1145/2462156.2462188. URL `http://doi.acm.org/10.1145/2462156.2462188`.

Sam Blackshear, Bor-Yuh Evan Chang, Sriram Sankaranarayanan, and Manu Sridharan. The flow-insensitive precision of Andersen's analysis in practice. volume 6887 of Lecture Notes in Computer Science, pages 60–76. Springer, 2011. ISBN 978-3-642-23701-0.

Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Thresher: Precise refutations for heap reachability. pages 275–286. ACM, 2013. ISBN 978-1-4503-2014-6.

Sam Blackshear, Alexandra Gendreau, and Bor-Yuh Evan Chang. Droidel: A general approach to android framework modeling. In Proceedings of the 4th ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis, SOAP '15, New York, NY, USA, 2015. ACM.

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. pages 196–207. ACM, 2003. ISBN 1-58113-662-5.

Eric Bodden, Laurie J. Hendren, and Ondrej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings, pages 525–549, 2007. doi: 10.1007/978-3-540-73589-2_25. URL `http://dx.doi.org/10.1007/978-3-540-73589-2_25`.

Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In ICSE, pages 241–250, 2011.

François Bourdoncle. Abstract debugging of higher-order imperative languages. pages 46–55, 1993a.

François Bourdoncle. Abstract debugging of higher-order imperative languages. pages 46–55, 1993b. doi: 10.1145/155090.155095. URL `http://doi.acm.org/10.1145/155090.155095`.

Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In ASPLOS, pages 167–178, 2010.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. pages 289–300. ACM, 2009. ISBN 978-1-60558-379-2.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. J. ACM, 58(6):26, 2011.

Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter OHearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In NFM, 2015.

Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: a powerful approach to weakest preconditions. In PLDI, pages 363–374, 2009.

Bor-Yuh Evan Chang and K. Rustan M. Leino. Inferring object invariants: Extended abstract. Electr. Notes Theor. Comput. Sci., 131:63–74, 2005.

Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. pages 247–260. ACM, 2008. ISBN 978-1-59593-689-9.

Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In MobiSys, pages 239–252, 2011.

Maria Christakis, Peter Müller, and Valentin Wüstholz. An experimental evaluation of deliberate unsoundness in a static program analyzer. In Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings, pages 336–354, 2015. doi: 10.1007/978-3-662-46081-8_19. URL `http://dx.doi.org/10.1007/978-3-662-46081-8_19`.

Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009, pages 50–62, 2009. doi: 10.1145/1542476.1542483. URL `http://doi.acm.org/10.1145/1542476.1542483`.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings, pages 154–169, 2000. doi: 10.1007/10722167_15. URL `http://dx.doi.org/10.1007/10722167_15`.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. J. ACM, 50(5):752–794, 2003.

Katherine E. Coons, Madan Musuvathi, and Kathryn S. McKinley. Bounded partial-order reduction. In Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013, pages 833–848, 2013. doi: 10.1145/2509136.2509556. URL `http://doi.acm.org/10.1145/2509136.2509556`.

Devin Coughlin and Bor-Yuh Evan Chang. Fissile type analysis: modular checking of almost everywhere invariants. pages 73–86. ACM, 2014. ISBN 978-1-4503-2544-8.

Devin Coughlin, Bor-Yuh Evan Chang, Amer Diwan, and Jeremy G. Siek. Measuring enforcement windows with symbolic trace interpretation: what well-behaved programs say. In International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012, pages 276–286, 2012. doi: 10.1145/2338965.2336786. URL `http://doi.acm.org/10.1145/2338965.2336786`.

Patrick Cousot. Semantic foundations of program analysis. In Steven S. Muchnick and Neil D. Jones, editors, Program Flow Analysis: Theory and Applications, chapter 10, pages 303–342. Prentice-Hall, 1981.

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. pages 238–252, 1977.

Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ analyzer. pages 21–30, 2005.

Patrick Cousot, Pierre Ganty, and Jean-François Raskin. Fixpoint-guided abstraction refinements. volume 4634 of Lecture Notes in Computer Science, pages 333–348. Springer, 2007. ISBN 978-3-540-74060-5.

Patrick Cousot, Radhia Cousot, and Francesco Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. pages 150–168, 2011.

Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. pages 128–148, 2013. doi: 10.1007/978-3-642-35873-9_10. URL `http://dx.doi.org/10.1007/978-3-642-35873-9_10`.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst., 13(4):451–490, October 1991. ISSN 0164-0925. doi: 10.1145/115372.115320.

Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. pages 57–68, 2002.

Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. LNCS, pages 337–340. Springer, 2008.

Edsger W. Dijkstra. A Discipline of Programming. Prentice-Hall, 1976.

Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In Rajiv Gupta and Saman P. Amarasinghe, editors, PLDI, pages 270–280. ACM, 2008. ISBN 978-1-59593-860-2.

Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In ESOP, pages 246–266, 2010.

Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In Thomas Ball and Mooly Sagiv, editors, POPL, pages 187–200. ACM, 2011a. ISBN 978-1-4503-0490-0.

Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In PLDI, 2011b.

Isil Dillig, Thomas Dillig, and Alex Aiken. Automated error diagnosis using abductive inference. In Jan Vitek, Haibo Lin, and Frank Tip, editors, PLDI, pages 181–192. ACM, 2012. ISBN 978-1-4503-1205-9.

Dino Distefano and Ivana Filipović. Memory leaks detection in Java by bi-abductive inference. 2010.

Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. volume 3920 of Lecture Notes in Computer Science, pages 287–302. Springer, 2006. ISBN 3-540-33056-9.

Nurit Dor, Stephen Adams, Manuvir Das, and Zhe Yang. Software validation via scalable path-sensitive value flow analysis. pages 12–22, 2004.

Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers, pages 10–30, 2010. doi: 10.1007/978-3-642-18070-5_2. URL `http://dx.doi.org/10.1007/978-3-642-18070-5_2`.

Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Inductive data flow graphs. In POPL, pages 129–142, 2013.

Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: semantics-based detection of Android malware through static analysis. In FSE, pages 576–587, 2014. doi: 10.1145/2635868.2635869.

Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. ACM Trans. Softw. Eng. Methodol., 17(2), 2008.

Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005, pages 110–121, 2005. doi: 10.1145/1040305.1040315. URL `http://doi.acm.org/10.1145/1040305.1040315`.

Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001, pages 193–205, 2001. doi: 10.1145/360204.360220. URL `http://doi.acm.org/10.1145/360204.360220`.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002, pages 234–245, 2002a. doi: 10.1145/512529.512558. URL `http://doi.acm.org/10.1145/512529.512558`.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. pages 234–245, 2002b.

Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In PLDI, June 2014.

Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. SCanDroid: Automated security certification of Android applications. Technical Report CS-TR-4991, University of Maryland, College Park, November 2009.

Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-flow analysis of Android applications in DroidSafe. In NDSS, 2015.

Salvatore Guarnieri. GULFSTREAM: staged static analysis for streaming javascript applications. In USENIX Conference on Web Application Development, WebApps'10, Boston, Massachusetts, USA, June 23-24, 2010, 2010. URL `https://www.usenix.org/conference/webapps-10/gulfstream-staged-static-analysis-streaming-javascript-applications`.

Samuel Z. Guyer and Calvin Lin. Client-driven pointer analysis. In Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings, pages 214–236, 2003. doi: 10.1007/3-540-44898-5_12. URL `http://dx.doi.org/10.1007/3-540-44898-5_12`.

Samuel Z. Guyer and Calvin Lin. Error checking with client-driven pointer analysis. Sci. Comput. Program., 58(1-2):83–114, 2005.

Brian Hackett and Alex Aiken. How is aliasing used in systems software? pages 69–80. ACM, 2006. ISBN 1-59593-468-5.

William R. Harris, Sriram Sankaranarayanan, Franjo Ivancic, and Aarti Gupta. Program analysis via satisfiability modulo path programs. pages 71–82. ACM, 2010. ISBN 978-1-60558-479-9.

Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. June 2001.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. pages 58–70, 2002.

C. A. R. Hoare. An axiomatic basis for computer programming. Commun. ACM, 12(10):576–580, 1969.

Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. In SIGSOFT'95: Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1995.

Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In Chris Hankin and Dave Schmidt, editors, POPL, pages 14–26. ACM, 2001. ISBN 1-58113-336-7.

Shachar Itzhaky, Nikolaj Bjørner, Thomas W. Reps, Mooly Sagiv, and Aditya V. Thakur. Property-directed shape analysis. In Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, pages 35–51, 2014. doi: 10.1007/978-3-319-08867-9_3. URL `http://dx.doi.org/10.1007/978-3-319-08867-9_3`.

Ranjit Jhala and Rupak Majumdar. Interprocedural analysis of asynchronous programs. In POPL, pages 339–350, 2007. doi: 10.1145/1190216.1190266. URL `http://doi.acm.org/10.1145/1190216.1190266`.

Ranjit Jhala, Rupak Majumdar, and Ru-Gang Xu. State of the union: Type inference via Craig interpolation. In TACAS, pages 553–567, 2007.

Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. Why don't software developers use static analysis tools to find bugs? In 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, pages 672–681, 2013. URL `http://dl.acm.org/citation.cfm?id=2486877`.

Yit Phang Khoo, Bor-Yuh Evan Chang, and Jeffrey S. Foster. Mixing type checking and symbolic execution. pages 436–447. ACM, 2010. ISBN 978-1-4503-0019-3.

Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In Hubert Garavel and John Hatcliff, editors, TACAS, volume 2619 of Lecture Notes in Computer Science, pages 553–568. Springer, 2003. ISBN 3-540-00898-5.

Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. ACM Trans. Softw. Eng. Methodol., 21(4):25, 2012. doi: 10.1145/2377656.2377662. URL `http://doi.acm.org/10.1145/2377656.2377662`.

Ali Sinan Köksal, Philippe Suter, and Viktor Kuncak. Scala to the Power of Z3: Integrating SMT and Programming. 2011.

Akash Lal and Shaz Qadeer. A program transformation for faster goal-directed search. In FMCAD, 2014.

Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. In CAV, CAV'12, pages 427–443, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-31423-0.

Lucas Layman, Laurie Williams, and Robert St. Amant. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In Proceedings of the First International Symposium on Empirical Software Engineering and Measurement, ESEM 2007, September 20-21, 2007, Madrid, Spain, pages 176–185, 2007. doi: 10.1109/ESEM.2007.11. URL `http://dx.doi.org/10.1109/ESEM.2007.11`.

K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. pages 119–134, 2005.

K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings, pages 491–516, 2004. doi: 10.1007/978-3-540-24851-4_22. URL `http://dx.doi.org/10.1007/978-3-540-24851-4_22`.

Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. Self-inferencing reflection resolution for Java. In ECOOP, pages 27–53, 2014. doi: 10.1007/978-3-662-44202-9_2.

Percy Liang and Mayur Naik. Scaling abstraction refinement via pruning. In Mary W. Hall and David A. Padua, editors, PLDI, pages 590–601. ACM, 2011. ISBN 978-1-4503-0663-8.

Percy Liang, Omer Tripp, Mayur Naik, and Mooly Sagiv. A dynamic evaluation of the precision of static heap abstractions. pages 411–427. ACM, 2010. ISBN 978-1-4503-0203-6.

Percy Liang, Omer Tripp, and Mayur Naik. Learning minimal abstractions. In POPL, pages 31–42, 2011.

Shuying Liang, Andrew W. Keep, Matthew Might, Steven Lyde, Thomas Gilray, Petey Aldous, and David Van Horn. Sound and precise malware analysis for android via pushdown reachability and entry-point saturation. In SPSM@CCS, pages 21–32, 2013.

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. Commun. ACM, 58(2):44–46, 2015. doi: 10.1145/2644805.

Alexey Loginov, Eran Yahav, Satish Chandra, Stephen Fink, Noam Rinetzky, and Mangala Gowri Nanda. Verifying dereference safety via expanding-scope analysis. pages 213–224, 2008. doi: 10.1145/1390630.1390657. URL `http://doi.acm.org/10.1145/1390630.1390657`.

Francesco Logozzo, Shuvendu K. Lahiri, Manuel Fähndrich, and Sam Blackshear. Verification modulo versions: towards usable verification. In ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014, page 32, 2014. doi: 10.1145/2594291.2594326. URL `http://doi.acm.org/10.1145/2594291.2594326`.

Ravichandhran Madhavan and Raghavan Komondoor. Null dereference verification via over-approximated weakest pre-conditions analysis. In OOPSLA, pages 1033–1052, 2011. doi: 10.1145/2048066.2048144. URL `http://doi.acm.org/10.1145/2048066.2048144`.

Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. PSE: explaining program failures via postmortem static analysis. 2004.

Amogh Margoor and Raghavan Komondoor. Two techniques to improve the precision of a demand-driven null-dereference verification approach. Sci. Comput. Program., 98:645–679, 2015. doi: 10.1016/j.scico.2014.09.006. URL `http://dx.doi.org/10.1016/j.scico.2014.09.006`.

Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. volume 3444 of Lecture Notes in Computer Science, pages 5–20. Springer, 2005. ISBN 3-540-25435-8.

Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. Scalable and incremental software bug detection. In Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013, pages 554–564, 2013. doi: 10.1145/2491411.2501854. URL `http://doi.acm.org/10.1145/2491411.2501854`.

Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-cfa paradox: illuminating functional vs. object-oriented program analysis. In Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010, pages 305–315, 2010. doi: 10.1145/1806596.1806631. URL `http://doi.acm.org/10.1145/1806596.1806631`.

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In International Symposium on Software Testing and Analysis (ISSTA), Rome, Italy, July 2002.

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. ACM Trans. Softw. Eng. Methodol., 14(1):1–41, 2005a.

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. ACM Trans. Softw. Eng. Methodol., 14(1):1–41, 2005b. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/1044834.1044835.

Mangala Gowri Nanda and Saurabh Sinha. Accurate interprocedural null-dereference analysis for Java. pages 133–143, 2009. doi: 10.1109/ICSE.2009.5070515. URL `http://dx.doi.org/10.1109/ICSE.2009.5070515`.

Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In USENIX Security, SEC'13, pages 543–558, Berkeley, CA, USA, 2013. USENIX Association. ISBN 978-1-931971-03-4.

Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In PLDI, page 49, 2014. doi: 10.1145/2594291.2594318.

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. volume 2142 of Lecture Notes in Computer Science, pages 1–19. Springer, 2001. ISBN 3-540-42554-3.

Matthew Parkinson. Class invariants: The end of the road? Aliasing, Confinement and Ownership in Object-oriented Programming (IWACO), page 9.

Doron Peled. All from one, one for all: on model checking using representatives. In Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings, pages 409–423, 1993. doi: 10.1007/3-540-56922-7_34. URL `http://dx.doi.org/10.1007/3-540-56922-7_34`.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. pages 55–74. IEEE Computer Society, 2002. ISBN 0-7695-1483-9.

H. G. Rice. Classes of recursively enumerable sets and their decision problems. Trans. Amer. Math. Soc., 74:358–366, 1953.

Xavier Rival. Understanding the origin of alarms in Astrée. volume 3672 of Lecture Notes in Computer Science, pages 303–319. Springer, 2005. ISBN 3-540-28584-9.

B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: 10.1145/73560.73562. URL `http://doi.acm.org/10.1145/73560.73562`.

Atanas Rountev and Dacong Yan. Static reference analysis for GUI objects in Android software. page 143, 2014.

Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Soederberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In International Conference on Software Engineering (ICSE), 2015.

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. ACM Trans. Program. Lang. Syst., 20(1):1–50, 1998.

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst., 24(3):217–298, 2002.

M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis, chapter 7, pages 189–233. Prentice-Hall, 1981.

Olin Grigsby Shivers. Control-Flow Analysis of Higher-Order Languages or Taming Lambda. PhD thesis, Carnige-Mellon Univeristy, May 1991.

Nishant Sinha and Chao Wang. Staged concurrent program analysis. In Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010, pages 47–56, 2010. doi: 10.1145/1882291.1882301. URL `http://doi.acm.org/10.1145/1882291.1882301`.

Nishant Sinha, Nimit Singhania, Satish Chandra, and Manu Sridharan. Alternate and learn: Finding witnesses without looking all over. In CAV, pages 599–615, 2012.

Yannis Smaragdakis and George Balatsouras. Pointer analysis. Foundations and Trends in Programming Languages, 2(1):1–69, 2015.

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011, pages 17–30, 2011. doi: 10.1145/1926385.1926390. URL `http://doi.acm.org/10.1145/1926385.1926390`.

Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: context-sensitivity, across the board. In PLDI, page 50, 2014a.

Yannis Smaragdakis, George Kastrinis, George Balatsouras, and Martin Bravenboer. More sound static handling of Java reflection, November 2014b.

Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In Michael I. Schwartzbach and Thomas Ball, editors, PLDI, pages 387–400. ACM, 2006. ISBN 1-59593-320-4.

Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. 2006.

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for Java. 2005.

Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. Alias analysis for object-oriented programs. In Aliasing in Object-Oriented Programming. Types, Analysis and Verification, pages 196–232. 2013. doi: 10.1007/978-3-642-36946-9_8. URL http://dx.doi.org/10.1007/978-3-642-36946-9_8.

Bjarne Steensgaard. Points-to analysis in almost linear time. pages 32–41, 1996.

Frank Tip. A survey of program slicing techniques. J. Prog. Lang., 3(3), 1995.

Antti Valmari. Stubborn sets for reduced state space generation. In Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings], pages 491–515, 1989. doi: 10.1007/3-540-53863-1_36. URL http://dx.doi.org/10.1007/3-540-53863-1_36.

Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. A concurrent portfolio approach to SMT solving. In Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings, pages 715–720, 2009. doi: 10.1007/978-3-642-02658-4_60. URL http://dx.doi.org/10.1007/978-3-642-02658-4_60.

B. Woolf. Null object. In Pattern languages of program design 3, pages 5–18. Addison-Wesley Longman Publishing Co., Inc., 1997.

Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. (JAIR), 32:565–606, 2008. doi: 10.1613/jair.2490. URL http://dx.doi.org/10.1613/jair.2490.

Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in Android applications. 2015.

Xin Zhang, Mayur Naik, and Hongseok Yang. Finding optimum abstractions in parametric dataflow analysis. In PLDI, pages 365–376, 2013. doi: 10.1145/2462156.2462185.

Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in datalog. In PLDI, page 27, 2014. doi: 10.1145/2594291.2594327.

Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. SmartDroid: an automatic system for revealing UI-based trigger conditions in Android applications. In SPSM@CCS, pages 93–104, 2012.

Zhi-Hua Zhou. Ensemble Methods: Foundations and Algorithms. Chapman & Hall/CRC, 1st edition, 2012. ISBN 1439830037, 9781439830031.

Appendix A

Soundness proof for jumping analysis

In this section, we prove the soundness of our framework for jumping analyses (Theorem 5.4.1). The central challenge of the proof lies in proving the soundness of the A-Jump rule given a sound relevance relation (e.g., one satisfying Condition 5.3.1). For reference, the concrete syntax of the language we use to formalize jumping analysis is given in Figure 5.1 and explained in Section 5.1. The concrete semantics of this language are given in Figure 5.2 and explained in Section 5.1.1. The abstract semantics of jumping analysis are given in Figure 5.3 and explained in Section 5.2.2.

A.1 Assumptions

We begin by stating a few assumed conditions that the proof relies on. In addition, we restate the relevance soundness condition from the body of this paper with one additional requirement: $t_{\mathsf{init}} \in T_{\text{rel}}$. This requirement is not restrictive, since it is trivial to adapt any relevance relation to satisfy this requirement. We introduce this requirement as a technical device because it maintains invariant that the relevance relation always returns at least one transition to jump to, making the proof much cleaner.

.

Condition A.1.1 (Relevance soundness)

If $\langle R, \ell_{\text{post}} \rangle \rightsquigarrow T_{\text{rel}}$, $\langle \sigma, \ell_{\text{pre}} \rangle \xrightarrow[T]{}^{*} \langle \sigma', \ell_{\text{post}} \rangle$, $t_{\text{irrel}} : \ \ell_1 \dashv\!\! [c] \!\!\rightarrow \ell_2 \in P - T_{\text{rel}}$, and $\vdash \{ R_{\text{pre}} \}\ c\ \{ R \}$, then $t_{\mathsf{init}} \in T_{\text{rel}}$ and either

(a) $R_{\text{pre}} \models R$, or (b) $\exists\ T_1, T_2$ s.t. $T = T_1 \mathbin{\hat{}} T_2$, $t_{\text{irrel}} \notin T_2$ and $T_{\text{rel}} \cap T_2 \neq \emptyset$.

Condition A.1.2 (Soundness of command evaluation)

If $\langle \sigma_{\text{pre}}, c\rangle \Downarrow \sigma_{\text{post}}$ and $\vdash \{R_{\text{pre}}\}\ c\ \{R_{\text{post}}\}$ such that $\sigma_{\text{post}} \in \gamma(R_{\text{post}})$, then $\sigma_{\text{pre}} \in \gamma(R_{\text{pre}})$.

These conditions are assumptions about the inputs to the framework; that is, we require a sound abstract semantics for commands and a sound relevance relation.

Condition A.1.3 (Sanity of analysis for commands)

For all R, c, $\vdash \{R_{\text{pre}}\}\ c\ \{R\}$ for some R_{pre}.

This condition says that if we have a command and an abstract state in hand, we can always run our analysis for commands and get a pre-state.

Condition A.1.4 (Label pairs uniquely define a transition)

Each pair of labels is involved in at most one transition; that is, (a) If $t_0 = \ell \dashv\![c_0]\!\mapsto \ell' \in P$ and $t_1 = \ell \dashv\![c_1]\!\mapsto \ell' \in P$, then $c_0 = c_1$ and $t_0 = t_1$. Furthermore, (b) If $\langle \sigma, \ell\rangle \underset{t}{\rightarrow} \langle \sigma', \ell'\rangle$ and $t = \ell_0 \dashv\![c]\!\mapsto \ell_1$, then $\ell = \ell_0$ and $\ell' = \ell_1$.

Condition A.1.5 (Program non-empty)

$P \neq \emptyset$.

These are simply a well-formedness condition that we impose on the input programs P that we wish to analyze. The program non-empty restriction simply makes the proof cleaner without restricting the set of input programs in a meaningful way; it is trivial to statically analyze empty programs. We require each pair of labels to define at most one transition so there can be no ambiguity regarding which transition the small-step evaluation relation visited given a pair of labels.

Condition A.1.6 (Form of initial transition)

$t_{\mathsf{init}} \in T = \ell_{\mathsf{dummy}} \dashv\![\texttt{skip}]\!\mapsto \ell_{\mathsf{entry}}$.

Condition A.1.7 (Initial transition is a no-op)

If $\langle \sigma, \ell\rangle \underset{t_{\mathsf{init}}}{\rightarrow} \langle \sigma', \ell'\rangle$, then $\sigma = \sigma'$.

Condition A.1.8 (Initial transition is always visited)

If $\langle \sigma_{\mathsf{dummy}}, \ell_{\mathsf{dummy}} \rangle \xrightarrow[T]{}^* \langle \sigma', \ell' \rangle$, and $T \neq \emptyset$, then $t_{\mathsf{init}} \in T$.

Condition A.1.9 (Execution from initial label starts from initial state)

If $\langle \sigma, \ell_{\mathsf{dummy}} \rangle \xrightarrow[T]{}^* \langle \sigma', \ell' \rangle$, then $\sigma = \sigma_{\mathsf{dummy}}$.

Condition A.1.10 (Initial state is unconstrained)

If $\sigma_{\mathsf{dummy}} \in \gamma(R)$, then $R = \top$.

These five conditions are assumptions about the special initial transition t_{init} and the special initial state σ_{dummy}. The proof is much cleaner if we can rely on the fact that a relevant transition always exists, and adding our own special no-op transition t_{init} accomplishes this goal without restricting the class of input programs we can consider. We assume that the initial transition goes from the initial "dummy" label ℓ_{dummy} to the entry label of the program ℓ_{entry} and that executing the transition does not change the concrete state. We also assume that any program execution starting from the special dummy label ℓ_{dummy} starts in the special state σ_{dummy}. Furthermore, we assume that every non-empty program execution starting from the dummy label ℓ_{dummy} visits the initial transition t_{init}.

Finally, we assume that the only abstract state that includes σ_{dummy} is $\top$; that is, we cannot assume that the initial concrete state has any particular structure. This is important for allowing us to show that it is sound to refute queries that cannot be produced by any execution of the program P. Otherwise, we would be allowed assume that the constraints encoded in the query held in the initial state, and thus would be unable to refute anything.

A.2 Key Lemmas

Lemma A.2.1 (Soundness of transition evaluation) If $\langle \sigma_{\text{pre}}, \ell_{\text{pre}} \rangle \xrightarrow[t]{} \langle \sigma_{\text{post}}, \ell_{\text{post}} \rangle$ and $I \vdash t$ such that $\sigma_{\text{post}} \in \gamma(I(\ell_{\text{post}}))$, then $\sigma_{\text{pre}} \in \gamma(I(\ell_{\text{pre}}))$.

Proof A.2.1 By induction on the derivation of $\langle \sigma_{\text{pre}}, \ell_{\text{pre}} \rangle \xrightarrow[t]{} \langle \sigma_{\text{post}}, \ell_{\text{post}} \rangle$ and leveraging Condition A.1.2.

This lemma lifts the soundness of command evaluation to soundness for the single-step transition relation, which is required for the proof since the premises of C-STEP, A-STEP, and A-JUMP involve this relation.

Lemma A.2.2 (Relation of single-step evaluation and command evaluation) If $\langle \sigma, \ell \rangle \xrightarrow[t]{} \langle \sigma', \ell' \rangle$ and $t\colon \ell \dashv\!c\!\mapsto \ell'$, then $\langle \sigma, c \rangle \Downarrow \sigma'$.

Proof A.2.2 By induction on the derivation of $\langle \sigma_{\text{pre}}, \ell_{\text{pre}} \rangle \xrightarrow[t]{} \langle \sigma_{\text{post}}, \ell_{\text{post}} \rangle$.

This lemma clarifies the relationship between the single-step transition relation and the concrete semantics for commands, which is useful because the relevance soundness condition gives a guarantee about the execution of commands, but the abstract semantics reason primarily about transitions.

Lemma A.2.3 (Unrolling concrete executions w.r.t a transition) If $\langle \sigma_{\text{pre}}, \ell_{\text{pre}} \rangle \xrightarrow[T]{}^{*} \langle \sigma_{\text{post}}, \ell_{\text{post}} \rangle$, and $t \in T$, then $\langle \sigma_{\text{pre}}, \ell_{\text{pre}} \rangle \xrightarrow[T_{\text{pre}}]{}^{*} \langle \sigma, \ell \rangle$, $\langle \sigma, \ell \rangle \xrightarrow[t]{} \langle \sigma', \ell' \rangle$, and $\langle \sigma', \ell' \rangle \xrightarrow[T_{\text{post}}]{}^{*} \langle \sigma_{\text{post}}, \ell_{\text{post}} \rangle$ such that $T = T_{\text{pre}} \mathbin{\hat{}} \{t\} \mathbin{\hat{}} T_{\text{post}}$.

Proof A.2.3 By induction on the derivation of $\langle \sigma_{\text{pre}}, \ell_{\text{pre}} \rangle \xrightarrow[T]{}^{*} \langle \sigma_{\text{post}}, \ell_{\text{post}} \rangle$.

This lemma allows us to decompose a multi-step concrete derivation that we know visits a certain transition t into a multi-step derivation preceding the visit of t, the single-step visit of t, and a multi-step derivation succeeding the visit of t. This lemma is needed to allow us to transform a concrete derivation into one that matches an abstract derivation given that we know they both visit the same transition.

The next lemma is **the** key lemma needed to make the proof go through.

Lemma A.2.4 (Irrelevant transitions preserve concretization property) If $\langle \sigma_{\text{pre}}, \ell_{\text{pre}} \rangle \xrightarrow[T]{}^{*} \langle \sigma_{\text{post}}, \ell_{\text{post}} \rangle$ such that $\sigma_{\text{post}} \in \gamma(R_{\text{post}})$, and for all $t\colon\ \ell_1 \dashv\!c\!\mapsto \ell_2 \in T$, $\vdash \{R'\}\ c\ \{R_{\text{post}}\}$ and $R' \models R_{\text{post}}$, then $\sigma_{\text{pre}} \in \gamma(R_{\text{post}})$.

Proof A.2.4 By induction on the structure of $\langle \sigma_{\text{pre}}, \ell_{\text{pre}} \rangle \xrightarrow[T]{}^{*} \langle \sigma_{\text{post}}, \ell_{\text{post}} \rangle$.

Case C-STOP: Then $\sigma_{\text{pre}} = \sigma_{\text{post}}$ and since $\sigma_{\text{post}} \in \gamma(R_{\text{post}})$, clearly $\sigma_{\text{pre}} \in \gamma(R_{\text{post}})$.

Case C-STEP: Then

$$\frac{\mathscr{C}_1 :: \langle \sigma_{\text{pre}}, \ell_{\text{pre}} \rangle \xrightarrow[T_{\text{pre}}]{}^{*} \langle \sigma', \ell' \rangle \quad \mathscr{C}_2 :: \langle \sigma', \ell' \rangle \xrightarrow[t]{} \langle \sigma_{\text{post}}, \ell_{\text{post}} \rangle}{\langle \sigma_{\text{pre}}, \ell_{\text{pre}} \rangle \xrightarrow[T_{\text{pre}} \,\hat{}\, \{t\}]{}^{*} \langle \sigma_{\text{post}}, \ell_{\text{post}} \rangle}$$

Since $T = T_{\text{pre}} \,\hat{}\, \{\, t \,\}$, clearly $T_{\text{pre}} \subseteq T$. By the assumptions of the theorem that for all $t\colon\ \ell_1 \dashv\!\![c]\!\!\mapsto \ell_2 \in T$, $\vdash \{R'\}\ c\ \{R_{\text{post}}\}$ and $R' \models R_{\text{post}}$ and the fact that $T_{\text{pre}} \subseteq T$, we know that these assumptions hold for T_{pre} also: ($\mathscr{F}_1$::) $t\colon\ \ell_1 \dashv\!\![c]\!\!\mapsto \ell_2 \in T_{\text{pre}}$, $\vdash \{R'\}\ c\ \{R_{\text{post}}\}$ and $R' \models R_{\text{post}}$.

We can apply Lemma A.2.2 to $t\colon\ \ell' \dashv\!\![c]\!\!\mapsto \ell_{\text{post}}$ and $\mathscr{C}_2$ to obtain ($\mathscr{F}_2$::) $\langle \sigma', c \rangle \Downarrow \sigma_{\text{post}}$. We can then apply Condition A.1.3 to c and R_{post} to obtain ($\mathscr{F}_3$::) $\vdash \{R'\}\ c\ \{R_{\text{post}}\}$ for some R'.

We can then apply Condition A.1.2 to ($\mathscr{F}_2$::) $\langle \sigma', c \rangle \Downarrow \sigma_{\text{post}}$, ($\mathscr{F}_3$::) $\vdash \{R'\}\ c\ \{R_{\text{post}}\}$, and theorem assumption $\sigma_{\text{post}} \in \gamma(R_{\text{post}})$ to obtain $\sigma' \in \gamma(R')$. Because $\sigma' \in \gamma(R')$ and $R' \models R_{\text{post}}$, we know ($\mathscr{F}_4$::) $\sigma' \in \gamma(R_{\text{post}})$.

Finally, we can apply the IH to $\mathscr{C}_1$, ($\mathscr{F}_4$::) $\sigma' \in \gamma(R_{\text{post}})$, and ($\mathscr{F}_1$::) $t\colon\ \ell_1 \dashv\!\![c]\!\!\mapsto \ell_2 \in T_{\text{pre}}$, $\vdash \{R'\}\ c\ \{R_{\text{post}}\}$ and $R' \models R_{\text{post}}$ to obtain $\sigma_{\text{pre}} \in \gamma(R_{\text{post}})$, as required.

A.3 Soundness proof

Theorem A.3.1 (Soundness of jumping analysis)

If $\mathscr{T}_1$:: $\langle \sigma_{\mathsf{dummy}}, \ell_{\mathsf{dummy}} \rangle \xrightarrow[T]{}^{*} \langle \sigma_{\text{post}}, \ell_{\text{post}} \rangle$ and $\mathscr{T}_2$:: $I \vdash \ell_{\text{post}}$ such that $\sigma_{\text{post}} \in \gamma(I(\ell_{\text{post}}))$, then $\sigma_{\mathsf{dummy}} \in \gamma(I(\ell_{\mathsf{dummy}}))$.

Proof A.3.1 By induction on the derivation of $\langle \sigma_{\mathsf{dummy}}, \ell_{\mathsf{dummy}} \rangle \xrightarrow[T]{}^{*} \langle \sigma_{\text{post}}, \ell_{\text{post}} \rangle$.

Case (1) C-STOP:

Then $\sigma_{\mathsf{dummy}} = \sigma_{\text{post}}$ and $\ell_{\mathsf{dummy}} = \ell_{\text{post}}$. Since $\sigma_{\text{post}} \in \gamma(I(\ell_{\text{post}}))$ is an assumption of the theorem, the desired result $\sigma_{\mathsf{dummy}} \in \gamma(I(\ell_{\mathsf{dummy}}))$ is immediate.

Case (2) C-STEP:

$$\frac{\mathscr{C}_1 :: \langle \sigma_{\mathsf{dummy}}, \ell_{\mathsf{dummy}} \rangle \xrightarrow[T_{\mathsf{entry}}]{}{}^{*} \langle \sigma', \ell' \rangle \qquad \mathscr{C}_2 :: \langle \sigma', \ell' \rangle \xrightarrow[t_{\text{post}}]{} \langle \sigma_{\text{post}}, \ell_{\text{post}} \rangle}{\langle \sigma_{\mathsf{dummy}}, \ell_{\mathsf{dummy}} \rangle \xrightarrow[T]{}{}^{*} \langle \sigma_{\text{post}}, \ell_{\text{post}} \rangle}$$

The only relevant abstract rule is A-JUMP.

Case A-JUMP: The last abstract rule applied was:

A-JUMP

$$\frac{\begin{array}{c}\mathscr{A}_1 :: I(\ell_{\text{post}}) \models R \quad \mathscr{A}_2 :: \langle R, \ell_{\text{post}} \rangle \rightsquigarrow T_{\text{rel}} \\ \mathscr{A}_3 :: I \vdash t \text{ for all } t : \ell_i \dashv\![c_{ij}]\!\mapsto \ell_j \in T_{\text{rel}} \qquad \mathscr{A}_4 :: R \models I(\ell_j) \text{ for all } \ell_j \quad \mathscr{A}_5 :: I \vdash \ell_i \text{ for all } \ell_i\end{array}}{I \vdash \ell_{\text{post}}}$$

We proceed on cases by: (A) $T_{\mathsf{entry}} = \emptyset$ and (B) $T_{\mathsf{entry}} \neq \emptyset$.

Case (2-A) $T_{\mathsf{entry}} = \emptyset$:

Then $T = \emptyset \,\hat{}\, \{t_{\text{post}}\}$, so $T = \{\, t_{\text{post}} \,\}$ and plainly $T \neq \emptyset$. By applying Condition A.1.8 to theorem assumption $\mathscr{T}_2$ and the derived fact $T \neq \emptyset$, we obtain $t_{\mathsf{init}} \in T$. Since T is a singleton set containing only t_{post}, it follows from $t_{\mathsf{init}} \in T$ that $t_{\text{post}} = t_{\mathsf{init}}$.

By applying Condition A.1.7 to this fact and $\mathscr{C}_2$, we can conclude that $\sigma' = \sigma'_{\text{post}}$. Since $\sigma' = \sigma'_{\text{post}}$ and the assumption of the theorem that $\sigma_{\text{post}} \in \gamma(I(\ell_{\text{post}}))$, we can conclude that $\sigma' \in \gamma(I(\ell_{\text{post}}))$. By Condition A.1.6, $t_{\mathsf{init}} = \ell_{\mathsf{dummy}} \dashv\![\texttt{skip}]\!\mapsto \ell_{\mathsf{entry}}$. Applying Condition A.1.4(b) to $\mathscr{C}_2$ and the fact that $t_{\text{post}} = t_{\mathsf{init}}$ $= \ell_{\mathsf{dummy}} \dashv\![\texttt{skip}]\!\mapsto \ell_{\mathsf{entry}}$, we can reason that $\ell' = \ell_{\mathsf{dummy}}$. Finally, we can apply Condition A.1.9 to $\mathscr{C}_2$ and $\ell' = \ell_{\mathsf{dummy}}$ to obtain $\sigma' = \sigma_{\mathsf{dummy}}$. Since we have derived fact $\sigma' \in \gamma(I(\ell_{\text{post}}))$, $\sigma' = \sigma_{\mathsf{dummy}}$, and $\ell_{\text{post}} =$ ℓ_{dummy}, we can conclude $\sigma_{\mathsf{dummy}} \in \gamma(I(\ell_{\mathsf{dummy}}))$, as required.

Case (2-B) $T_{\mathsf{entry}} \neq \emptyset$:

Plainly, either t_{post} is a relevant transition or it is not. Thus, there are two sub-cases to consider here: (I) t_{post} $\in T_{\text{rel}}$, and (II) $t_{\text{post}} \notin T_{\text{rel}}$.

Case (2-B-I) $t_{\text{post}} \in T_{\text{rel}}$:

By $\mathscr{A}_3$, we know $I \vdash t$ for all $t\colon\ \ell_i \dashv\!\![c_{ij}]\!\!\mapsto \ell_j \in T_{\text{rel}}$. Combining this fact with the assumption of this case that $t_{\text{post}} \in T_{\text{rel}}$, we can choose $t = t_{\text{post}}$ to obtain $I \vdash t_{\text{post}}$. We can then apply Lemma A.2.1 to concrete sub-derivation $\mathscr{C}_2$, derived fact $I \vdash t_{\text{post}}$, and theorem assumption $(\mathscr{T}_3)$ $\sigma_{\text{post}} \in \gamma(I(\ell_{\text{post}}))$ to obtain $\ell' \in \gamma(I(\ell'))$.

We can apply Condition A.1.4 to $\mathscr{C}_2$ and t_{post} to write t_{post} as $t_{\text{post}}\colon\ \ell' \dashv\!\![c_{\text{post}}]\!\!\mapsto \ell_{\text{post}}$. Since $t_{\text{post}} \in T_{\text{rel}}$ by the assumption of this case and we know $(\mathscr{A}_5)$ $I \vdash \ell_i$ for all ℓ_i for all $t\colon\ \ell_i \dashv\!\![c_{ij}]\!\!\mapsto \ell_j \in T_{\text{rel}}$, we can choose $\ell_i = \ell'$ to obtain $I \vdash \ell'$. We can then apply the IH to concrete sub-derivation $\mathscr{C}_1$, derived fact $I \vdash \ell'$, and derived fact $I(\ell') \in \gamma(I(\ell'))$ to obtain $\sigma_{\mathsf{dummy}} \in \gamma(I(\ell_{\mathsf{dummy}}))$, as required.

Case (2-B-II) $t_{\text{post}} \notin T_{\text{rel}}$:

We know the transitions T traversed by a concrete execution are a subset of the program transitions P (i.e., $T \subseteq P$), so we can use this fact and the assumption of this case that $t_{\text{post}} \notin T_{\text{rel}}$ to reason that (*) $t_{\text{post}} \in P$ - T_{rel}.

We can apply Condition A.1.1 to the T-JUMP assumption $\langle R, \ell_{\text{post}} \rangle \rightsquigarrow T_{\text{rel}}$ and Condition A.1.3 to obtain the facts $(\mathscr{F}_1)$ $t_{\mathsf{init}} \in T_{\text{rel}}$, for all $t\colon\ \ell_1 \dashv\!\![c]\!\!\mapsto \ell_2 \in P$ - T_{rel}, $\vdash \{R'\}\ c\ \{R\}$ (using Condition A.1.3 once more)), and either (a) $\exists\, T_1, T_2$ s.t. $T = T_1\ \hat{}\ T_2$, $t_{\text{post}} \notin T_2$ and $T_{\text{rel}} \cap T_2 \neq \emptyset$, or (b) $R' \models R$. We proceed by cases depending on whether the control-feasibility condition (a) holds or the data-relevance condition (b) holds.

Case (2-B-II-a) $\exists\, T_1, T_2$ s.t. $T = T_1\ \hat{}\ T_2$, $t_{\text{post}} \notin T_2$, and $T_{\text{rel}} \cap T_2 \neq \emptyset$:

Because $T = T_{\mathsf{entry}}\ \hat{}\ \{t_{\text{post}}\}$, the only way we can write $T = T_1\ \hat{}\ T_2$ such that $t \notin T_2$ is to choose $T_1 = T$ and $T_2 = \emptyset$. However, this leads us to a contradiction because an assumption of this case is that $T_{\text{rel}} \cap T_2 \neq \emptyset$, but we require $T_2 = \emptyset$ and clearly $T_{\text{rel}} \cap \emptyset = \emptyset$.

Case (2-B-II-b) $\forall\, t\colon\ \ell_1 \dashv\!\![c]\!\!\mapsto \ell_2 \in P$ - T_{rel}, $\vdash \{R'\}\ c\ \{R\}$, and $R' \models R$:

We can apply Condition A.1.8 to concrete sub-derivation $\mathscr{C}_1$:: the assumptions of the current case (2-B) $T_{\mathsf{entry}} \neq \emptyset$ to obtain $t_{\mathsf{init}} \in T_{\mathsf{entry}}$. Since we also know that $(\mathscr{F}_1)$ $t_{\mathsf{init}} \in T_{\text{rel}}$, note that $T_{\mathsf{entry}} \cap T_{\text{rel}} \neq \emptyset$. Now, let us select a transition $t_{\text{rel}} \in T_{\textsf{entry}} \cap T_{\text{rel}}$ that is the **last** relevant transition visited in the concrete trace $T_{\textsf{entry}}$. Such a transition is guaranteed to exist because we have just shown that $T_{\textsf{entry}} \cap T_{\text{rel}} \neq \emptyset$. More formally, let $T = t_0, \ldots, t_n$. We choose t_{rel} to be a transition t_i such that $t_i \in T$, $t_i \in T_{\text{rel}}$, and for all $t_j \in T$ where $j > i$, $t_j \notin T_{\text{rel}}$. In prose, t_{rel} is the **last** transition visited by the concrete execution that is in also in the set of relevant transitions T_{rel}; all transitions in T occurring after t_{rel} are not in T_{rel}.

Next, we can apply Lemma A.2.3 to concrete sub-derivation $\mathscr{C}_1$ and the fact that $t_{\text{rel}} \in T_{\textsf{entry}}$ to obtain
$(\mathscr{F}_2)$ $\langle \sigma_{\textsf{dummy}}, \ell_{\textsf{dummy}} \rangle \xrightarrow[T_{\text{pre}}]{}^{*} \langle \sigma_{\text{pre}}, \ell_{\text{pre}} \rangle$,
$(\mathscr{F}_3)$ $\langle \sigma_{\text{pre}}, \ell_{\text{pre}} \rangle \xrightarrow[t_{\text{rel}}]{} \langle \sigma'_{\text{pre}}, \ell'_{\text{pre}} \rangle$, and
$(\mathscr{F}_4)$ $\langle \sigma'_{\text{pre}}, \ell'_{\text{pre}} \rangle \xrightarrow[T_{\text{irrel}}]{}^{*} \langle \sigma', \ell' \rangle$.
By our method of selecting t_{rel}, we know $(\mathscr{F}_5)$ for all $t_{\text{irrel}} \in T_{\text{irrel}}$, $t \notin T_{\text{rel}}$.

We can apply Condition A.1.4 to $\mathscr{C}_2$ and t_{post} to write t_{post} as $t_{\text{post}}\colon\ \ell' \dashv\!\![c_{\text{post}}]\!\!\mapsto \ell_{\text{post}}$. We can apply Lemma A.2.2 to $\mathscr{C}_2$:: and $t_{\text{post}}\colon\ \ell' \dashv\!\![c_{\text{post}}]\!\!\mapsto \ell_{\text{post}}$ to obtain $(\mathscr{F}_5)$ $\langle \sigma', c_{\text{post}} \rangle \Downarrow \sigma_{\text{post}}$.

Next, we can apply the assumption of case (2-B-II-b) $\forall\, t\colon\ \ell_1 \dashv\!\![c]\!\!\mapsto \ell_2 \in P$ - T_{rel}, $\vdash \{R'\}\ c\ \{R\}$ to fact (*) $t_{\text{post}} \in P$ - T_{rel} (choosing $t = t_{\text{post}}$, $c = c_{\text{post}}$, and $R = I(\ell_{\text{post}})$) to determine that $(\mathscr{F}_6)$ $R' \models I(\ell_{\text{post}})$ (note that this also yields $\vdash \{R'\}\ c\ \{I(\ell_{\text{post}})\}$ as a consequence of choosing $R = I(\ell_{\text{post}})$). We can then apply Condition A.1.2 to $(\mathscr{F}_5)$ $\langle \sigma', c_{\text{post}} \rangle \Downarrow \sigma_{\text{post}}$, $\vdash \{R'\}\ c\ \{I(\ell_{\text{post}})\}$, and theorem assumption $(\mathscr{T}_3)$ $\sigma_{\text{post}} \in \gamma(I(\ell_{\text{post}}))$ to obtain $\sigma' \in \gamma(R')$. Since $(\mathscr{F}_6)$ $R' \models I(\ell_{\text{post}})$, by definition of $\models$ we know $\gamma(R') \subseteq \gamma(I(\ell_{\text{post}}))$. Since $\gamma(R') \subseteq \gamma(I(\ell_{\text{post}}))$ and $\sigma' \in \gamma(R')$, clearly $\sigma' \in \gamma(I(\ell_{\text{post}}))$.

We can apply Lemma A.2.4 to $(\mathscr{F}_4)$ $\langle \sigma'_{\text{pre}}, \ell'_{\text{pre}} \rangle \xrightarrow[T_{\text{irrel}}]{}^{*} \langle \sigma', \ell' \rangle$, $\sigma' \in \gamma(I(\ell_{\text{post}}))$, and the previously derived facts that for all $t_{\text{irrel}} \in T_{\text{irrel}}$, $t \notin T_{\text{rel}}$, $t_{\text{irrel}}\colon\ \ell_1 \dashv\!\![c_{\text{irrel}}]\!\!\mapsto \ell_2 \in T_{\text{irrel}}$, and $\vdash \{R'\}\ c\ \{I(\ell_{\text{post}})\}$, and using the assumption of this case $R' \models R$ to derive $R' \models I(\ell_{\text{post}})$ (choosing $R = I(\ell_{\text{post}})$), we obtain the fact that $(\mathscr{F}_7)$ $\sigma'_{\text{pre}} \in \gamma(I(\ell_{\text{post}}))$.

By A-JUMP premise $(\mathscr{A}_3)$ $I \vdash t$ for all $t\colon\ \ell_i \dashv\!\![c_{ij}]\!\!\mapsto \ell_j \in T_{\text{rel}}$. Since $t_{\text{rel}}\colon\ \ell_{\text{pre}} \dashv\!\![c]\!\!\mapsto \ell'_{\text{pre}} \in T_{\text{rel}}$, and by A-JUMP premise $\mathscr{A}_4$ we know $R \models I(\ell_j)$ for all ℓ_j where $t\colon\ \ell_i \dashv\!\![c_{ij}]\!\!\mapsto \ell_j \in T_{\text{rel}}$, we can select $\ell_j = \ell'_{\text{pre}}$ to obtain $R \models I(\ell'_{\text{pre}})$. By $\mathscr{A}_1$ we have $I(\ell_{\text{post}}) \models R$, which means $I(\ell_{\text{post}}) \models R \models I(\ell'_{\text{pre}})$. This implies $\gamma(I(\ell_{\text{post}})) \subseteq \gamma(R) \subseteq \gamma(I(\ell'_{\text{pre}}))$ by definition of $\models$. Because $(\mathscr{F}_7)$ $\sigma'_{\text{pre}} \in \gamma(I(\ell_{\text{post}}))$, clearly $\sigma'_{\text{pre}} \in \gamma(I(\ell'_{\text{pre}}))$.

We can now apply the transition soundness lemma (Lemma A.2.1) to $(\mathscr{F}_3)$ $\langle \sigma_{\text{pre}}, \ell_{\text{pre}} \rangle \xrightarrow[t_{\text{rel}}]{} \langle \sigma'_{\text{pre}}, \ell'_{\text{pre}} \rangle$, A-JUMP premise ($\mathscr{A}_3$) $I \vdash t$ for all $t\colon \ell_i \dashv\!\![c_{ij}]\!\!\mapsto \ell_j \in T_{\text{rel}}$ where $t\colon \ell_i \dashv\!\![c_{ij}]\!\!\mapsto \ell_j \in T_{\text{rel}}$ (choosing $t = t_{\text{rel}}$, since $t_{\text{rel}} \in T_{\text{rel}}$), and derived fact $\sigma'_{\text{pre}} \in \gamma(I(\ell'_{\text{pre}}))$ to obtain $\sigma_{\text{pre}} \in \gamma(I(\ell_{\text{pre}}))$.

Finally, we can apply the IH to ($\mathscr{F}_2$) $\langle \sigma_{\mathsf{dummy}}, \ell_{\mathsf{dummy}} \rangle \xrightarrow[T_{\text{pre}}]{}^{*} \langle \sigma_{\text{pre}}, \ell_{\text{pre}} \rangle$, A-JUMP premise $\mathscr{A}_5$:: $I \vdash \ell_i$ for all ℓ_i where $t\colon \ell_i \dashv\!\![c_{ij}]\!\!\mapsto \ell_j \in T_{\text{rel}}$ (choosing $t = t_{\text{rel}}\colon\ \ell_{\text{pre}} \dashv\!\![c]\!\!\mapsto \ell'_{\text{pre}}$ and consequently $\ell_i = \ell_{\text{pre}}$), and derived fact $\sigma_{\text{pre}} \in \gamma(I(\ell_{\text{pre}}))$ to obtain $\sigma_{\mathsf{dummy}} \in \gamma(I(\ell_{\mathsf{dummy}}))$, as required.