Type-Intertwined Separation Logic

by

Devin Coughlin

B.S., Stanford University, 2003

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

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

This thesis entitled:
Type-Intertwined Separation Logic
written by Devin Coughlin
has been approved for the Department of Computer Science

Prof. Bor-Yuh Evan Chang

Dr. Amer Diwan

Prof. Ranjit Jhala

Prof. Sriram Sankaranarayanan

Prof. Jeremy G. Siek

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.

Coughlin, Devin (Ph.D., Computer Science)

Type-Intertwined Separation Logic

Thesis directed by Prof. Bor-Yuh Evan Chang

Abstract Static program analysis can improve programmer productivity and software reliability by definitively ruling out entire classes of programmer mistakes. For mainstream imperative languages such as C, C++, and Java, static analysis about the heap—memory that is dynamically allocated at run time—is particularly challenging because heap memory acts as global, mutable state.

This dissertation describes how to soundly combine two static analyses that each take vastly different approaches to reasoning about the heap: type systems and separation logic. Traditional type systems take an alias-agnostic, global view of the heap that affords both fast verification and light-weight annotation of invariants holding over the entire program. Separation logic, in contrast, provides an alias-aware, local view of the heap in which invariants can vary at each program point.

In this work, I show how type systems and separation logic can be safely and efficiently combined. The result is **type-intertwined separation logic**, an analysis that applies traditional type-based reasoning to some regions of the program and separation logic to others—converting between analysis representations at region boundaries—and summarizes some portions of the heap with coarse type invariants and others with precise separation logic invariants.

The key challenge that this dissertation addresses is the communication and preservation of heap invariants between analyses. I tackle this challenge with two core contributions. The first is **type-consistent summarization and materialization**, which enables type-intertwined separation logic to both leverage and selectively violate the global type invariant. This mechanism allows the analysis to efficiently and precisely verify invariants that hold almost everywhere. Second, I describe **gated separating conjunction**, a non-commutative strengthening of standard separating conjunction that expresses local “dis-pointing” relationships between sub-heaps. Gated separation enables local heap reasoning by permitting the separation logic to frame out portions of memory and prevent the type system from interfering with its contents—an operation that would be unsound in type-intertwined analysis with only standard separating conjunction. With these two contributions, type-intertwined separation logic combines the benefits of both type-like global reasoning and separation-logic-style local reasoning in a single analysis.

Dedication

To my parents.

Acknowledgements

I have had the privilege of working with two excellent advisors during my graduate career. My first advisor, Amer Diwan, taught me what it meant to be part of the research community. I still remember the shiver that went down my spine when he said to me “your field” and meant static analysis. Later, when Amer left for Google, Bor-Yuh Evan Chang took a big chance on an unproven student and pushed me to think in ways I didn’t realize possible. Evan: it has been a true joy to watch the Programming Languages and Verification group at CU Boulder grow and thrive under your leadership.

As a junior graduate student, I had the extreme good fortune to have as mentors two senior students, Christoph Reichenbach and Todd Mytkowicz. To this day, I aspire to be as gracious, kind, and receptive to newcomers in the field as they were with me. I would like to thank my other collaborators, including Yi-Fan Tsai, Jeremy Siek, and Xavier Rival as well as the junior students I have had the pleasure of working with: Alex Beale, Ross Holland, and Shawn Meier. I have relied greatly on my fellow students and labmates for emotional support: Arlen Cox, Rhonda Hoenigman, Jonathan Turner, Aleksandar Chakarov, Alexandra Gendreau, and Amin Ben Sassi—our lunches, hikes, and late–night Game of Thrones-watching got me through the rough times.

Of course, I could not have done this without the constant support of my parents, Sharon Devine and Curtiss Coughlin. Mom, Dad—thank you for putting up with me for all these years.

Contents

Chapter

1 Introduction 1

1.1 Types and Separation Logic: Disparate Approaches to Taming the Heap 2

1.1.1 Types are Flow-Insensitive, Alias-Agnostic Heap Invariants 3

1.1.2 Separation Logic Allows Alias-Aware, Local Heap Reasoning 5

1.2 Thesis Statement 9

1.3 Dissertation Outline 10

2 Motivation: Verifying Invariants that Hold Almost Everywhere 12

2.1 Background: Reflective Method Call 12

2.2 Specifying Required Relationships with Dependent Refinement Types 14

2.3 Problem: Imperative Updates Violate Relationships 15

2.4 Almost-Everywhere Hypothesis 17

2.5 Inspiration for the Almost-Everywhere Hypothesis 18

3 Overview: Intertwining Type Checking and Separation Logic 22

3.1 Flow-Insensitive Dependent Types for Reflection Safety in Objective-C 23

3.2 Leveraging Type Invariants During Symbolic Analysis 29

3.3 Local Heap Reasoning with Gated Separation 35

4 Leveraging Almost-Everywhere Heap Invariants 43

4.1 A Language with Mutable Objects and Reflection 44

4.1.1 Syntax 44

4.1.2 Concrete State 46

4.1.3 Concrete Semantics 47

4.2 A Flow-Insensitive Dependent Refinement Type System 49

4.2.1 Refinement Types For Reflection 49

4.2.2 Concretization of Types 49

4.2.3 Subtyping with Refinements 53

4.2.4 Identifier Substitution 55

4.2.5 Expression Typing 56

4.3 Symbolic Analysis 60

4.3.1 Symbolic State 60

4.3.2 Concretization of Symbolic State 62

4.3.3 Symbolic Execution 67

4.4 Handoff and Invariant Conversion 70

4.4.1 Handoff Between Analyses 71

4.4.2 From Type Environments to Symbolic States and Back Again. 72

4.4.3 Materialization from Type-Consistent Heaps 76

4.4.4 Summarizing Symbolic Objects Back Into Types. 80

4.5 Soundness of Intertwined Analysis 82

4.6 Case Study: Checking Reflective Call Safety 83

4.6.1 Prototype Tool 83

4.6.2 Benchmarks 88

4.6.3 Prototype Performance 90

4.6.4 Alarms 93

4.7 Related Work 96

5 Type-Intertwined Framing with Gated Separation 99

5.1 Gated Separation 101

5.1.1 Memory Formulas and Concretization 101

5.1.2 Axioms of Gated Separation 102

5.2 Gated Separation in a Traditional Separation Logic 107

5.2.1 Syntax 108

5.2.2 Purely Symbolic Static Semantics 109

5.2.3 Language Requirements for Gated Framing. 114

5.3 Type-Intertwined Separation Logic 116

5.3.1 Type Checking the Command Language 117

5.3.2 Type-Intertwined State 119

5.3.3 Static Semantics of Type-Intertwined Separation Logic 122

5.4 Related Work 126

6 Measuring Enforcement Windows with Symbolic Trace Interpretation 127

6.1 Introduction 127

6.2 Overview and Metrics 132

6.2.1 Preliminaries: Trace Instructions 132

6.2.2 Measuring: Where, What, and How 135

6.3 Measurement Framework 143

6.3.1 Symbolic Trace Interpretation 143

6.3.2 Implementation: Dynamic Symbolic Heap 146

6.3.3 Sufficiency for Static Analysis Design 148

6.4 Measurements 150

6.4.1 Case Study: Bugs and Program Evolution 151

6.4.2 Distribution of Enforcement Distances 157

6.4.3 Threats to Validity 159

6.5 Related Work 159

6.6 Summary of Symbolic Trace Interpretation 160

7 Conclusions and Future Work 162

Bibliography 164

Appendix

A Additional Proofs for FISSILE Type Analysis 173

A.1 Lemmas Concerning Abstract State 173

A.2 Lemmas Concerning Concrete Execution 174

A.3 Details of the Proof of Intertwined Soundness FISSILE Type Analysis 176

Tables

Table

4.1 A suite of reflection benchmarks in Objective-C. 89

4.2 Annotation Burden. 90

4.3 Precision, Premises, and Performance. 92

4.4 Characterization of false alarms under FISSILE type analysis. 95

6.1 Framework Sufficiency for Analysis Design. 149

6.2 Distances get shorter after bug fixes. 152

Figures

Figure

1.1 Traditional type invariants make weak guarantees about the entire heap. 4

1.2 Traditional types are alias agnostic and cannot distinguish between Case 1, Case 2, and Case 3. In contrast, separation logic makes strong dis-aliasing guarantees and thus can determine that Case 1 is the only one that applies. 6

1.3 Separation logic makes strong dis-aliasing guarantees, allowing it to analyze the footprint of an operation in isolation, without consideration of the frame. 8

2.1 Reflection can be used for decoupled callbacks. 13

2.2 Imperative updates may temporarily violate required relationships. 15

2.3 Heap mutation requires reasoning about a large amount of context. 19

2.4 Reason for hope. 21

3.1 Type-intertwined analysis. 22

3.2 Verifying reflective call safety requires knowing responds-to relationships between objects and selectors. 24

3.3 The heap is split into immediately type-inconsistent and almost type-consistent regions. 30

3.4 Example: Verifying an almost-everywhere invariant. 32

3.5 Framing out before switching to type checking. 35

3.6 The traditional frame rule with $*$ is unsound in type-intertwined separation logic. . . 38

3.7 Gated separation prevents the foregate from pointing into the aftgate. 40

3.8 The frame rule with gated separation (⧏) is sound in type-intertwined separation logic. 41

4.1 A core imperative programming language with objects and reflective method call. . . 45

4.2 Concrete state. 46

4.3 Concrete execution 48

4.4 Subtyping of refinement types used for verifying reflective call safety. 53

4.5 Subenvironments under substitution. 56

4.6 Typing of expressions with refinement relationships between storage locations. 57

4.7 The symbolic analysis state splits type environments into types lifted to values and the locations where values are stored. 60

4.8 Non-type-intertwined rules for symbolic analysis. 68

4.9 Formal version of `CustomImage` callback example. 70

4.10 Analysis handoff via environment typeification and symbolization. 71

4.11 Symbolization and typeification of the stack. 73

4.12 Materialization and summarization. 77

4.13 Missile enables programmers to specify almost-everywhere invariants in C and Objective-C with dependent refinement type annotations (shaded region). 84

4.14 Missile integrates with the `Xcode` development environment to visualize mixed flow-insensitive and path-sensitive alarms. 85

4.15 Missile's retry heuristics cannot show that this example is safe. 88

5.1 Gated separating conjunction (⧏) is a non-commutative strengthening of separating conjunction that additionally constrains the range of one sub-heap to be disjoint from the domain of the other. Shaded regions indicate differences with standard separating conjunction ($\ast$). 100

5.2 Axiom Schemata of Gated Separation 103

5.3 A core imperative command language with globals and a mutable heap. 108

5.4 The key challenge is ensuring writes do not violate gated separation. 110

5.5 Concrete Semantics. 115

5.6 Types for the command language. 117

5.7 Type checking rules for the command language. 118

5.8 Type-intertwined symbolic state. 119

5.9 Type-Intertwined Separation Logic. 123

6.1 A potential validation scope. 128

6.2 Inlining depth as a metric for complexity. 130

6.3 Concrete and symbolic trace interpretation of a short example. The left-hand–side of the figure is explained in Section 6.2.1, the right in Section 6.2.2. 134

6.4 Distance metrics capturing combinations of control versus data reasoning and static versus dynamic reasoning. 137

6.5 Call tree for inlining depth. 138

6.6 Symbolic trace interpretation for inlining depth. 144

6.7 Studying reported buggy dereference sites and the code evolution using enforcement distances. 153

6.8 Misuse of the `java.io.File` API. 154

6.9 Program evolution violates programmer expectation about a tree invariant. 154

6.10 Distribution of dereference site measurements for DaCapo. Data enforcement distances are overwhelming short. Control distances can get very long for heap locations, suggesting non-operational heap reasoning (e.g., by encapsulation or invariants). . . 156

6.11 Trading off data distance to reduce control distance. 158

Chapter 1

Introduction

Programmers make mistakes. In aggregate, such mistakes—bugs—are expensive, costing the United States tens of billions of dollars per year [78]. Even individually they can be catastrophic. For example, simple programmer mistakes that permit an invalid write past the end of a memory buffer can allow remote code execution in C—a serious security vulnerability. Such buffer overruns have led to the Code Red [26], Slammer [77], and Conficker worms [87], each of which cost billions of dollars [108]. Even reads past the end of a memory buffer can have terrible consequences: the Heartbleed [2] vulnerability in OpenSSL allowed remote clients to read portions of a server's memory, exposing users' plain-text passwords to the entire world.

Static program analysis can help eliminate these bugs. Static analysis techniques can definitively rule out entire classes of programmer mistakes by constructing an approximation of the run-time behavior of a program—without actually running the program . If this approximation includes all possible behavior of the program (i.e., is a sound overapproximation) and yet does not include the bug in question then the program cannot manifest the bug. Such overapproximate techniques serve as a complement to testing (and other underapproximate approaches) and have the potential to enhance software security, improve reliability, and reduce development cost.

Unfortunately, with overapproximation comes imprecision: if the approximation of a program's behavior is so coarse that it includes buggy behavior—even when the program itself has no bugs—then the static analysis will incorrectly report a bug. Such false alarms inhibit adoption of static analysis because they force the programmer to either ignore the results of the analysis (defeating the purpose of the analysis in the first place) or contort their programming style to work around imprecision in the analysis.

Reasoning about the heap—memory that is allocated dynamically, at run time, and accessed indirectly—is particularly important for static analysis of mainstream programming languages (such as C, C++, and Java), which make heavy use of imperative heap updates. In these languages, the heap acts as global, mutable state, so the key challenge for analysis is determining the effect of writes to heap cells on prior reasoning about the contents of the heap. This challenge manifests in two ways. First, the potential for pointer aliasing—where two pointers point to the same heap cell—requires an analysis to consider the case where mutation through one pointer may change storage accessed through another. Second, the global nature of the heap means that any piece of code can potentially change the contents of any heap cell—and so a sound analysis must account for the potential of any called function to change the heap. For both manifestations, the crucial analysis requirement is to tame an "action at a distance" through the heap.

1.1 Types and Separation Logic: Disparate Approaches to Taming the Heap

The goal of this dissertation is to combine two analyses that take vastly different approaches to taming "action at a distance": type systems and separation-logic-based static analysis. Type systems combat action at a distance by acknowledging that the heap is global—that any line of code could potentially modify any portion of the heap—and enforcing relatively weak invariants everywhere. If these invariants (i.e., types) are not strong enough to prove the property of interest (typically freedom from a class of untrapped run-time errors) then the type system emits an alarm. In contrast, separation-logic–based static analysis (henceforth "separation logic") assumes that the heap is local—that it is possible to restrict each line of code to a fixed portion of the heap (its **footprint**)—and ensures that strong invariants about other portions of the heap cannot be violated by the line in question. If this footprint cannot be inferred or is not sufficient, then separation logic emits an alarm. In the rest of this section, I provide a brief overview of the two very different approaches that type systems and separation logic take to taming the heap.

1.1.1 Types are Flow-Insensitive, Alias-Agnostic Heap Invariants

Static type systems [20] ensure that programs are free from a class of relatively shallow but extremely important safety bugs—in the words of Robin Milner, they guarantee that "well-typed programs do not go wrong" [74]. Static typing occupies an apparent sweet spot between developer burden and safety guarantees: type systems are the only form of modular specification and verification to be widely adopted by mainstream programmers. Types form the first line of defense against programmer errors—not only in conventional languages (like C, C++, and Java) but also in emerging settings (such as GPU programming [59]) and even for traditionally dynamically-typed languages (such as JavaScript). While type systems are particularly well-suited to ruling out untrapped run-time errors—especially in modular code designed for reuse, such as frameworks and libraries—type-like reasoning has been applied to a wide variety of other problems, including pointer analysis [97], and region-based memory memory management [102].

Traditional type checking can be viewed as a form of static analysis in which the type system enforces a single type invariant at all program points. That is, the type system can assume that the global type invariant holds before each statement and correspondingly must guarantee that it holds after. Although imprecise by static analysis standards, this style of reasoning is powerful because it allows effective reasoning about global data: at any point, the type system can assume the global type invariant holds, without regard to context. In particular, the type system does not have to consider control flow to determine whether a given statement is safe. For this reason, type systems are often called **flow-insensitive**. It is this flow-insensitivity that affords type systems their typically-low annotation burden : modular annotations need not specify the effect of a function because all functions must maintain the global invariant. Such invariants also allow relatively simple, unification-based inference of types [74].

The inherent imprecision of a single, global invariant has motivated the development of flow-sensitive type systems [3, 41, 45, 91, 98, 105]. These type systems drop global constraints on mutable storage locations and instead focus on tracking facts on values as they flow through the program.

```
class Circle {        class Point {
  Point center;         int xPos;
  int radius;           int yPos;
};                    };
```

(a) Global type definition. The global type invariant holds at all program points and summarizes all instances of a type indistinguishably.

```
1  void moveRight(Circle c) {
2    c.center.xPos += 5;
3  }
```

(b) Types assume and guarantee the global type invariant.

Figure 1.1: Traditional type invariants make weak guarantees about the entire heap.

Such systems typically have more in common with static analysis techniques—such as the need for specified or inferred loop invariants and function summaries—than with their flow-insensitive cousins. In this document, I use the term “type system” to mean a traditional, flow-insensitive type system. Another response to imprecision in type systems has been the development of mixed static-dynamic typing—including gradual typing [96] and hybrid type checking [42]—in which obligations that cannot be discharged statically are guaranteed by additional checking at run time. Here, again, I use “type system” to mean a solely static type system.

Type environments constrain the heap. A type environment maps local variables to the types of those local variables. While this mapping explicitly describes only locals, it implicitly constrains the entire heap reachable from those variables. I illustrate this implicit constraint in Figure 1.1, which defines two classes: `Circle` and `Point`, in a Java-like language (Figure 1.1a). Consider the type environment in the `moveRight` method in Figure 1.1b: it guarantees that the local variable (parameter) `c` contains a reference to (i.e., the address of) a `Circle`. Now, the type of `Circle` says that every instance must have a `center` field that contains a reference to an instance of `Point` and the type of `Point` requires that its `xPos` field contain an integer. That is, the type of `c` does not constrain just its local storage, but also the **entire heap** that is reachable from `c` by dereferencing pointers stored in fields. Assuming no null pointers (let us say, as is common in mainstream languages, that null pointer dereferences are not type errors), this constraint is strong enough to show that the field update at line 2 is free of run-time type errors. The type invariant guarantees that fields in the path `c.center.xPos` exist and that the heap cell accessed by that path contains an integer before the update—and so it will contain one after the update, as well. The type environment is sufficient to rule out run-time errors—but is it still a very weak heap constraint.

Types are coarse summaries. From a static analysis perspective, a type is an abstract summary of a potentially unbounded number of storage locations that coarsely summarizes all instances of a type indistinguishably. This imprecision is extremely important for light-weight, modular annotation—in contrast with other flow-insensitive reasoning about memory (i.e., pointer analysis [6, 37, 97], alias analysis [107])—because annotated abstract locations are type names, which are already familiar to programmers.

Unfortunately, the coarseness of traditional types makes them agnostic to aliasing: they are not expressive enough to describe aliasing and dis-aliasing relationships. I show an example of a program that types are not expressive enough to verify in Figure 1.2a. Here the programmer uses the `assert` at line 7 as a request to statically prove that the value stored in the `c1.center.xPos` remains unchanged across the call to `moveRight()`. The key to proving this assertion is showing that that `c1.center.xPos` and `c2.center.xPos` are two different cells on the heap—that is that `c1` and `c2` are dis-aliased, as are the fields `c1.center` and `c2.center`. Type-based reasoning alone is not strong enough to prove this assertion: it can determine that `c1.center` and `c2.center` both contain `Points`—but not whether these points are the different or the same. With type-based reasoning alone, there are three possible cases. I illustrate these cases in Figures 1.2b–1.2d. The actual case is Figure 1.2b: `c1` and `c2` each point to distinct objects, as do their `center` fields. But type-based reasoning cannot rule out two other cases: (1) where `c1` and `c2` each point to distinct `Circles`, but their `center` fields point to the same `Point` (Figure 1.2c); and (2) where `c1` and `c2` point to the same circle (Figure 1.2d). In these last two cases, the assertion may not hold—so a sound type-based analysis must emit a false alarm.

1.1.2 Separation Logic Allows Alias-Aware, Local Heap Reasoning

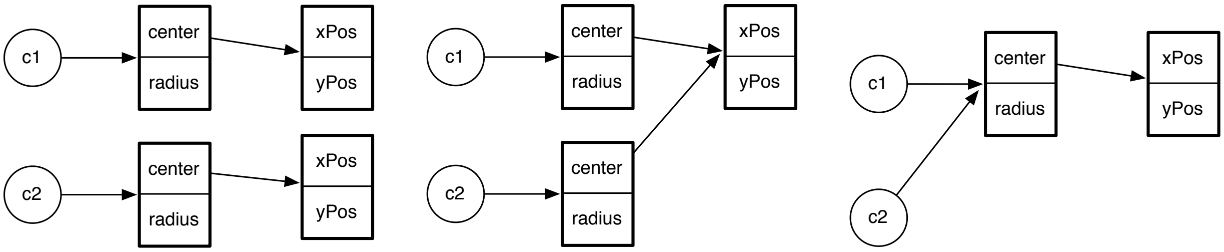
In contrast with type systems—which traditionally provide flow-insensitive, alias-agnostic guarantees about the global reachable heap—separation logic [88] performs flow-sensitive, alias-aware

```
Circle c1 = new Circle();
Circle c2 = new Circle();
c1.center = new Point();
c2.center = new Point();

int saved = c1.center.xPos;
moveRight(c2);

assert(c1.center.xPos == saved);
```

(a) The assertion cannot be proven with type-based reasoning alone.



(b) Case 1

(c) Case 2

(d) Case 3

Figure 1.2: Traditional types are alias agnostic and cannot distinguish between Case 1, Case 2, and Case 3. In contrast, separation logic makes strong dis-aliasing guarantees and thus can determine that Case 1 is the only one that applies.

reasoning about local portions of the heap. Historically, separation logic arose as a substructural extension of Hoare's program logic [58], designed to treat memory as a resource and thus allow elegant Hoare-style reasoning about allocation and mutation in the heap. Separation logic has been applied to reasoning about data structures (shape analysis [9, 22]) and concurrency [80].

At the core of separation logic is the assumption that heap invariants can be separated into disjoint regions with the strong guarantee that the heap cells represented in one region are not represented in another. Safely splitting the heap in this way requires potentially costly reasoning about allocation and dis-aliasing (to determine which addresses and thus storage locations are distinct) but the payoff is significant: it enables two key analysis capabilities: (1) precise reasoning about heap mutation with strong updates and (2) local reasoning about data via framing.

Strong updates. Precise reasoning about heap mutation is a challenge, even for flow-sensitive analyses, because any of loss of precision in determining the identity of the cell being mutated results in a corresponding imprecision the abstract value stored at that cell. For complicated heap graphs, such imprecision can quickly cascade, limiting the utility of analysis . Separation logic sidesteps this concern by tracking allocation precisely enough to guarantee that each abstract cell corresponds to exactly one concrete cell, allowing the effect of abstract assignment transfer functions to mirror the destructive nature of their concrete counterparts. With this precision, separation logic can distinguish between the three aliases cases presented in Figure 1.2 and determine that Case 1 (Figure 1.2b) is the result of executing the code in Figure 1.1—a level of precision not possible with type-based reasoning.

Framing. Perhaps the most critical capability of separation logic is the ability to split the heap into two disjoint regions: (1) the **footprint**, which the operation may read and write from, and (2) the **frame**, which the operation is not permitted to touch.

The frame rule is an inference rule of separation logic that allows the logic (and thus an analysis built upon the logic) to choose a potential footprint for an operation—or even for a called function—and analyze the operation or function over the footprint alone, dropping the frame from consideration completely. Then the post-heap derived from considering the effects of the operation

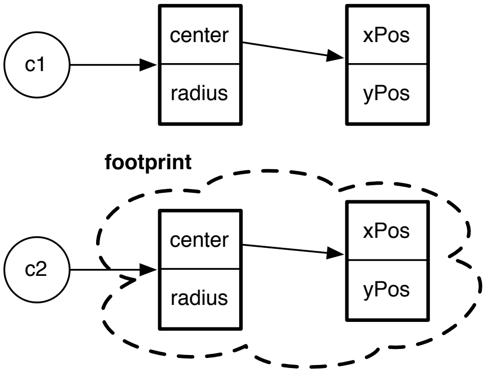


Figure 1.3: Separation logic makes strong dis-aliasing guarantees, allowing it to analyze the footprint of an operation in isolation, without consideration of the frame.

on the footprint can be trivially combined with the unchanged frame and analysis can continue. This combination is sound because separation logic, crucially, ensures that the operation cannot possibly change any heap cells in the frame. That is, with separation logic the analysis can identify a local portion of the heap, shear it off, and analyze part of the program **as if the local portion were the entire heap**.

This style of analysis can help to tame the heap, making it possible to reason about non-interference of the kind required to verify the example (previously described) in Figure 1.2. As described above, the key to verifying that example is reasoning about `c1.center.xPos` and `c2.center.xPos` as distinct heap cells. As I showed in Figure 1.2, traditional type systems are not precise enough to represent this analysis fact. With separation logic, this is possible.

Figure 1.3 shows an illustration of a view that separation logic could take of the heap when trying to determine the effect of the call to `moveRight()` at line 6 in Figure 1.2a. In contrast with a type-based view of the heap, separation logic can reason precisely enough about allocation (that `new` creates fresh cells) and assignment (the analysis can perform strong updates to heap cells) to determine that (1) locals `c1` and `c2` contain the addresses of distinct `Circle`s and (2) that the addresses of the `Point`s stored in the `center` fields of `c1` and `c2` are themselves distinct.

With this precise reasoning about aliasing, an analysis based on separation logic can separate the heap into two definitely distinct regions: that which `moveRight()` touches (the footprint) and that which it does not (the frame). In Figure 1.3, the footprint (surrounded in dotted lines) is the object pointed-to by `c2` and the object pointed to by `c2.center`; the remaining heap cells are in the frame. Because separation logic guarantees that a portion of code cannot access memory outside of its designated footprint, the analysis can be sure that `moveRight()` does not change the circle pointed-to by `c1`—and thus that the assertion at line 7 will definitely pass.

1.2 Thesis Statement

As we have seen, type systems and separation logic take wildly different approaches to taming the imperative heap—each with its benefits and drawbacks. Types tame the heap by enforcing an invariant over the entire reachable heap that is the same at all program points. This invariant must be weak enough that an analysis can ensure it is maintained without strong updates. In contrast, separation logic enforces more precise invariants over local portions of the heap, framing out the rest. This framing requires strong dis-aliasing guarantees—guarantees which can also be used to perform strong updates.

The thesis of this dissertation is that these two styles of reasoning are compatible:

Types systems and separation logic can be safely and efficiently combined in a manner that preserves the benefits of global reasoning for type systems and local reasoning for separation logic.

That is, I will describe an analysis that (1) applies traditional type-based reasoning to some portions of the program and separation logic to others and (2) summarizes some portions of the heap with type invariants and others with separation logic invariants. By "safely" I mean that the analysis is sound; by "efficiently" I mean that the analysis can be applied to large, real-world programs and run fast enough to be incorporated into integrated development environments. By the "benefits of global reasoning" I mean that the analysis can rely upon a flow-insensitive type invariant in type-checked portions of the program; while by "benefits of local reasoning" I mean that the portions checked in separation logic can employ a frame rule.

The philosophy behind this approach is to **keep the type system as simple as possible**. The goal is to employ what is essentially an "off-the-shelf" type system design and thus retain the well-established benefits of type-based reasoning: fast checking times and ease of annotation. Rather then complicate the type system to be flow-sensitive, or to reason directly about aliasing, I propose to confine the inherent complexity of precise reasoning about heap mutation to the separation logic portion of the analysis.

The end result is an analysis in which each component analysis is true to itself: types act like types and separation logic acts like separation logic. In contrast with abstraction refinement schemes [7, 55]—which employ a single analysis that allows varying levels of abstraction—this approach mixes [64] two distinct analyses, each with their own transfer functions. The key challenge that this dissertation addresses is the communication and preservation of heap invariants between the type system and the separation logic.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows. Chapter 2 motivates the combination of types and separation logic with an example problem—verifying imperative updates of dependent types specifying relationships—that benefits from both type-based and separation-logic-style reasoning. I also present evidence for what I call the "Almost-Everywhere" hypothesis. This hypothesis is a conjecture about how programmers enforce important safety properties—if it holds for a given invariant of interest, then type-intertwined separation logic can be successful at verifying it. In Chapter 3, I give a high-level overview of how these two styles of static reasoning can be safely combined into type-intertwined separation logic. Chapter 4 formally describes almost-everywhere heap invariants and demonstrates that type invariants can be soundly and efficiently communicated between a type analysis and separation logic, on demand. In Chapter 5, I describe how this approach can admit a type-intertwined frame rule by extending separation logic with a strengthened form of spatial conjunction that we call "gated separation." In Chapter 6, I present the inspiration for this work: a dynamic analysis to determine whether a static analysis is insufficient. Finally, I present conclusions and suggest future work in Chapter 7.

Chapter 2

Motivation: Verifying Invariants that Hold Almost Everywhere

Modular verification of just about any interesting property of programs requires the specification and inference of invariants. One particularly rich mechanism for specifying such invariants is dependent refinement types [106], which have been applied extensively to, for example, checking array bounds [27, 90, 91, 105]. These types are compelling because they permit the specification of relationships in a type system framework that naturally admits modular checking. For example, a modular refinement type system can relate an array with an in-bounds index or a memory location with the lock that serializes access to it.

A less well-studied problem that also falls into a refinement type framework is modularly verifying the safety of reflective method call in dynamic languages.

2.1 Background: Reflective Method Call

Reflective method call is a language feature that enables programmers to invoke a method via a run-time string value called a **selector** rather than specify the name of the method directly in the source code. This language feature is relied upon heavily in dynamic languages, such as Ruby and Python, as a means to decouple client and framework code—but is also commonly used in more static languages, like Java, C#, and Objective-C. Yet while they are powerful and convenient, reflective method calls introduce a new kind of run-time error: the method named by the selector may not exist. In contrast with traditional static analysis for reflection—which has focused on the problem of determining the exact targets of reflective method calls [14, 16, 23, 47, 72, 103] to aid in whole-program analysis—we are concerned with **reflection safety**: ensuring that the target exists.

Our key observation about modularly verifying reflective call safety is that the essential property to capture and track through the program is the relationship between a **responder** (the object on which the method is invoked) and a valid selector. In particular, the verifier does not need to determine the actual string value as long as it can ensure that the "responds-to" relationship between the object and the selector holds at the reflective call site. This observation is crucial because the point where this relationship invariant is established and where the selector is known (usually in client code) is likely far removed from the point where the reflective call is performed and at which the invariant is relied upon (usually in framework code).

```
1  class Callback
2    var sel: String = ...
3    var obj: Object ⦇ ↾respondsTo sel()→void ⦈  = ...
4
5    def doCallback()
6       performSelector(self.obj, self.sel)
```

Figure 2.1: Reflection can be used for decoupled callbacks.

Figure 2.1 gives an example of how programmers use reflection to decouple components (ignore the shaded portion for now—this is a dependent type annotation, which we discuss below). Here the programmer has created a `Callback` class with two fields: `sel`, which stores a string containing the name of a method (that is, a selector) and `obj`, which stores the object (the responder) on which the selector stored in `sel` will be called. The call to `performSelector()` on line 6 performs a reflective dispatch: it invokes the method indicated by the selector on the object. The key thing to note about this idiom is that the `Callback` object is completely oblivious to both the type of the responder (it can be any object) and the contents of the selector string—and so it is possible that the required relationship between the responder and the selector (that the value stored in `obj` responds to the value stored in `sel`) may not hold and thus that the call will fail.

2.2 Specifying Required Relationships with Dependent Refinement Types

The required relationship between `obj` and `sel` in `Callback` can be guaranteed with a dependent refinement type.

Dependent Refinement Types. **Refinement types** $\{ v : B \mid R(v) \}$ consist of two components: the base type B, which comes from the underlying type system, and the (optional) refinement formula R, which add restrictions on the value v beyond those imposed by the base type. For example, the refinement type $\{ v : \mathrm{Int} \mid v \geq 0 \}$ expresses not only that the value must be an Int (a traditional type constraint) but also that it must be greater than or equal to zero (a richer constraint than expressible in traditional type systems). Refinement types are particularly useful in two cases. First, when the refinements are restricted to a language in which validity of formulas involving implication is easily checkable (e.g., the refinements are in a theory supported by an SMT solver, such as in [11, 90]), then typing and subtyping checks can be discharged by off-the-shelf solvers. Second, when the built-in (base) type system is too weak to reason about the property of interest (such as proving memory safety in C [27, 91] or when reasoning about dynamic languages [24, 25, 43]), then refinement types offer a mechanism for specifying and reasoning about type-like properties above and beyond the base type system. As a notational convenience, we write refinements without the bound variable and assume the bound variable is used as the first argument of all atomic relations. For example, we write $\mathrm{Int} \upharpoonright \geq 0$ instead of $\{ v : \mathrm{Int} \mid v \geq 0 \}$.

With **dependent** refinement types, the refinement formula can refer to program expressions—and in particular, to local variables and fields. So, for example, the dependent type $\mathrm{Int} \upharpoonright \geq y$ constrains its inhabitants to be greater than or equal to the value stored in local variable `y`. These references allow dependent types to express required relationships. For example, ascribing local `x` to have type $\mathrm{Int} \upharpoonright \geq y$ expresses the requirement that `x` $\geq$ `y`.

Refinement Types for Reflection. For verifying safety of reflective method calls, the crucial required relationship is the responds-to relationship between responder and selector. The shaded portion of line 3 in Figure 2.2 shows an example type annotation expressing the required relationship between fields `obj` and `sel`. This annotation says that object `obj` should have a method with the name of the string value stored in field `sel` and that the method should have the type signature ()→`void` (that is, it should take no parameters and return `void`). If the type system can ensure that this relationship holds, then the reflective call at line 6 is guaranteed to succeed.

Condit et al.'s Deputy [27] dependent type system demonstrated how such relationships can be checked in a **flow-insensitive** manner to ensure memory safety in C. (There, the relationship of interest is between an array and an in-bounds index.) Deputy checks relationships by applying Hoare's backwards-assignment rule for weakest preconditions to a flow-insensitive type environment, ensuring that the required relationship holds both before and after every assignment—if not, it will generate an alarm. In essence, Deputy makes the following implicit hypothesis about relationships between storage locations:

Hypothesis (Deputy). ***All*** *relationships hold* ***all*** *of the time.*

That is, Deputy assumes programmers establish important relationships between storage locations atomically and never break them.

2.3 Problem: Imperative Updates Violate Relationships

Unfortunately, the Deputy hypothesis is overly optimistic. It ignores the common case where a programmer updates both ends of a relationship (the referring location and the referred-to location) in two separate steps. Suppose the developer adds an `update()` method (Figure 2.2) to the `Callback` class introduced in Figure 2.1.

```
7 def update(val o: Object ⦇⌈respondsTo s()→void⦈, val s: String)
8   this.sel = s ✘
9   this.obj = o ✔
```

Figure 2.2: Imperative updates may temporarily violate required relationships.

This method takes two parameters as input: an object `o` and a selector string `s`. Further, let us suppose that the developer has specified a refinement type (line 7) that guarantees that `o` responds to `s`. Then a Deputy-style, flow-insensitive approach will generate an alarm at line 8 (marked by an ✘) because after executing the first assignment the `this.obj` field does not respond to the updated `this.sel`. In other words, the assignment violates the flow-insensitive invariant that `this.obj` must **always** respond to `this.sel`. Note that changing the order of the assignments does not help here; if the programmer updates the `obj` field first then the new `obj` will not respond to the old `sel`.

The flow-insensitive alarm at line 8 is a false alarm. Even though the programmer has violated the required relationship, she restores it at line 9 (✔) and does not rely on it holding for the time when the relationship is violated (i.e., there is no reflective call). Deputy's flow-insensitive approach cannot tolerate such temporary violations. If the first step breaks the relationship, a flow-insensitive analysis will always report an error regardless of whether a later step re-establishes the relationship. While this particular violation could be avoided in languages that have parallel updates, this language feature is not typically supported in mainstream imperative languages.

Type systems can check required relationships in languages without parallel updates with flow-sensitive reasoning. Flow-sensitive type systems [3, 41, 45, 91, 98, 105] calculate per-program point type invariants—in essence, they allow the type of a storage location to change at each program point. This style of reasoning is much more precise than flow-insensitive reasoning but is also made more expensive by the need to reason about the effect of mutation on storage locations. Rondon et al's Low-Level Liquid Types [89, 91] infers flow-sensitive dependent types to ensure memory safety. Here, the required relationship is that a pointer is in-bounds for a buffer—and the flow-sensitive reasoning ensures the relationship holds at each dereference. In essence, Low-Level Liquid Types makes the following implicit hypothesis about relationships between storage locations:

Hypothesis (Low-Level Liquid Types). *A relationship will hold only at the point it is relied upon; it will vary freely at all other program points.*

That is, the fully flow-sensitive reasoning in Low-Level Liquid Types assumes programmers only establish important relationships right before they are used and that these relationships will not hold (and thus will require precise reasoning) everywhere else. This hypothesis is very pessimistic—it assumes the worst-scenario about potential relationships and therefore reasons precisely even when such precision may not be needed.

2.4 Almost-Everywhere Hypothesis

The underlying premise of this work is that there is a class of relationships for which Deputy's flow-insensitive approach is **almost** right. This premise rests on two hypotheses about how programmers break and maintain important relationships between storage locations.

Hypothesis 1. ***All*** *relationships hold* ***most*** *of the time.*

That is, we hypothesize that the periods of execution during which programmers break relationships are relatively short. We believe that while programmers can reason flow-sensitively (i.e. by simulation) about local relationships and flow-insensitively (i.e. at the type-invariant level) about global relationships, they cannot reason both globally and flow-sensitively—the cognitive load of reasoning flow-sensitively about the entire program is simply too high. We therefore expect developers to re-establish broken relationships quickly to convince themselves the program is safe.

Hypothesis 2. ***Most*** *relationships hold* ***all*** *of the time.*

That is, we hypothesize that developers do not violate a large number of relationships simultaneously. We believe that even when reasoning locally programmers are incapable of keeping track of a large number of broken relationships at the same time—and so will keep such violations to a minimum.

We call relationships for which the above two hypotheses hold "almost-everywhere relationships' and claim that an analysis can check them quickly and precisely enough to be practical. We will describe an analysis that leverages these hypotheses to check relationship invariants in Chapter 3—but first, we describe our reasoning for proposing these hypotheses.

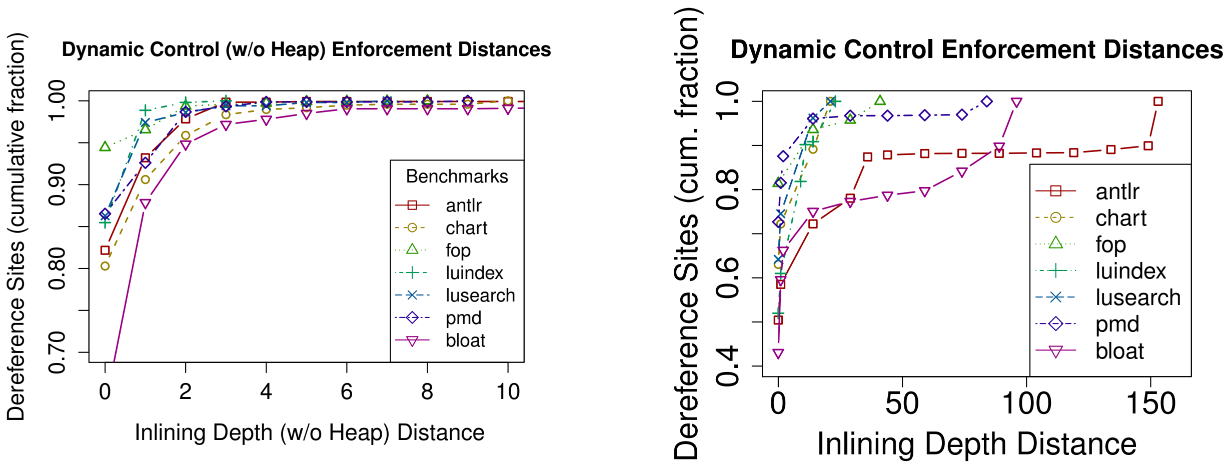
2.5 Inspiration for the Almost-Everywhere Hypothesis

The inspiration for the almost-everywhere hypothesis—and for type-intertwined separation logic—came from a study [29] that we performed to explore the limits of static reasoning about null pointer dereferences in Java programs. We describe the measurement apparatus for these experiments fully in Chapter 6—here we focus on two key results from the study, which was designed to explore the limits of purely operational static reasoning.

Operational vs. Invariant-based Reasoning By operational reasoning, we mean reasoning by simulating code to determine the exact effect of a sequence of instructions. For a computer, this style of reasoning is not particularly difficult to perform—for example, by a precise, separation-logic-based symbolic execution—but it is expensive in terms of computational resources and, of course, might not even terminate. For a programmer (i.e., a human being) this style of reasoning is even more heroic: she must careful simulate each line of code in her head, one-by-one—expensive in terms of cognitive load. In both cases, the total length of operational reasoning is necessarily limited: computers by memory and computational time; and programmers by human cognition.

These limits can be overcome by invariant-based reasoning. Rather than precisely simulate large chunks of code, invariant-based reasoning assumes an **invariant**—an imprecise abstraction of the program's state or operation—for a particular context. From the point of view of static analysis, an invariant is a fact to be verified or discovered about the program. To a programmer, invariants are constraints (sometimes unstated, sometimes even only faintly conceived) that the programmer should conform to so that the program behaves correctly. By assuming and guaranteeing appropriate invariants, both automated analyses and human programmers can limit the scope of operational reasoning and yet still reason about the entire behavior of the program. Just as a child can stick her nose an inch away from her coloring book and scribble with reckless abandon—knowing that if she stays between the lines then the picture will emerge when she pulls her head back—so can an analysis or programmer conform to a strong-enough invariant to ensure correct behavior of the program as a whole.

Experiment: The Absurdity of Purely Operational Heap Reasoning. As described in Chapter 1, the key challenge for static heap reasoning is the need for an analysis to definitively rule out "action at a distance", in which one part of the program invalidates a heap invariant relied upon by another. The key inspiration for type-intertwined separation logic was an experiment to measure how much purely operational reasoning would be necessary to rule out such action at a distance when verifying null dereference safety, a simple but important safety property in Java programs. We collected dynamic traces of programs in the DaCapo [13] benchmark suite and interpreted these traces with a framework that captures various measures of how much context would be required for operational reasoning to verify these programs. I provide the details of this analysis in Chapter 6—here I focus on the results for two key measures of context.



(a) Excluding sites requiring heap reasoning

(b) Including sites requiring heap reasoning

Figure 2.3: Heap mutation requires reasoning about a large amount of context.

Figure 2.3 shows two graphs, each displaying the cumulative fraction of dereference sites where purely operational reasoning would require at least k levels of inlining (i.e., a k-callstrings [95] level of context sensitivity) for a perfectly precise interprocedural analysis to prove safe (see Section 6.3.1 for a detailed description of how these measurements were calculated from dynamic traces). Because we measured k with a dynamic analysis, the observed value is a lower-bound for the actual number of inlinings—that is, k is necessary but perhaps not sufficient. Figure 2.3a shows the cumulative distribution for dereference sites that do not require heap reasoning, while Figure 2.3b gives the same curve but for all sites—including those which require reasoning about the heap. There are two salient features of this figure. The first feature is that the amount of context required for operational reasoning about values that flow through the heap is orders of magnitude greater than that for non-heap-related reasoning. Consider the bloat benchmark. Figure 2.3a shows that for dereference sites that do not involve values that flow through the heap, an analysis could reason about 95% of observed sites with only 3 levels of inlining. In contrast, when sites involving the heap are included, reasoning about 95% of sites would require almost 100 levels of inlining context. This results show that mutation through the heap renders operational reasoning about even simple safety properties much more challenging. The second key feature is that the amount of context required for heap-based reasoning is too large to be plausible for a typical programmer to keep in her head. For human beings, purely operational reasoning on the scale required to show that dereferences succeed would be absurd.

And yet, the incidence of bugs in computer programs is very low. Estimates of the rate of bugs differ depending on programming language, project, and developer sophistication—but fall within the range of 0.1–30 bugs per thousand lines of new code [10, 76, 84] for mainstream languages like C, C++, and Java. That is, at least 97% of newly written lines of code are contain no bugs! It seems that humans are effective enough at reasoning about the behavior of their programs to prevent most bugs. Because purely operational human reasoning would be absurd, we expect that programmers are employing mostly invariant-based reasoning with only short stretches of operational reasoning mixed in. This conjecture is the basis for our Hypothesis 1: that all relationship invariants hold most of the time—and suggests that a static analysis that mimics this mixture of reasoning could be effective.

A second experiment offered inspiration for Hypothesis 2. Figure 2.4 shows the distribution of sizes of required reasoning scopes for null dereference safety along a data rather than a control dimension. Here, the data dimension is flow count—the number of distinct fields through which a value flows from the point at which it can be operationally shown to be be non-null (i.e., an allocation or a null-comparison) to the dereference site. This distribution shows that for the vast

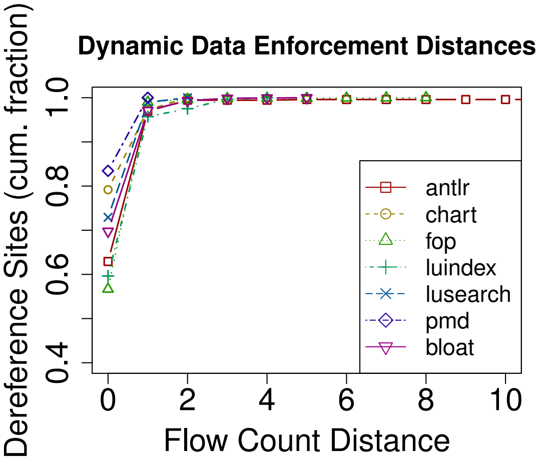


Figure 2.4: Reason for hope.

majority of sites (more than 97%), the observed lower bound for precise heap reasoning is two storage cells or fewer. If this lower bind is tight, then for a given site an analysis need reason precisely about only 2 or fewer storage cells on the heap. This small number of cells inspired Hypothesis 2: that developers do not violate a large number of relationships simultaneously.

Chapter 3

Overview: Intertwining Type Checking and Separation Logic

In this chapter, I provide a high-level overview of type-intertwined separation logic, a static analysis technique designed around the almost-everywhere hypothesis described in Chapter 2. Type-intertwined separation logic combines type checking and separation logic, applying a flow-insensitive type analysis to some parts of the program and a path-sensitive, separation-logic-based symbolic analysis to others.

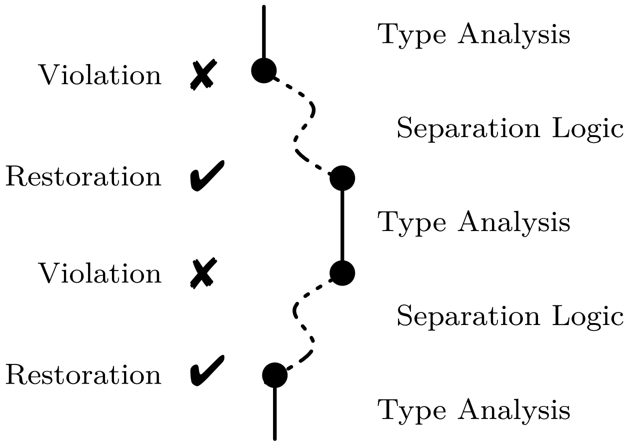


Figure 3.1: Type-intertwined analysis.

The strategy of when to switch between the two forms of analysis can be either user-specified or heuristic, guided by the violation and restoration of global type invariants. I illustrate this switching pictorially in Figure 3.1. Recall the `update()` method from the `Callback` class in Figure 2.2—there the programmer updated two related storage locations, temporarily violating and then restoring the relationship that would be required everywhere by a Deputy-style global type invariant. Our strategy for checking such almost-everywhere invariants applies the flow-insensitive type analysis to the regions where the type invariant holds and switches to the symbolic analysis when the programmer violates the invariant—that is, when there is a flow-insensitive type error. The symbolic analysis continues until the programmer restores the invariant, at which point the type analysis resumes. Sometimes the symbolic analysis may benefit from access to precise invariants (such as aliasing relationships) involving code before the point of the type error—I discuss our heuristics for when to switch between analyses in detail in Section 4.6.1.

In this chapter, I provide an overview of three core contributions for intertwining type systems and separation logic. The first contribution is a type analysis, a novel **dependent type system** for **reflective method safety**, which I present in Section 3.1. The second is **type-consistent materialization and summarization** (Section 3.2), a mechanism that allows a type-intertwined symbolic analysis to both **leverage** and **selectively violate** a global type invariant. Finally, in Section 3.3 I describe how to enrich separation logic with **gated separation**—a non-commutative strengthening of separating conjunction that enables local heap reasoning in type-intertwined separation logic.

Throughout this chapter, I present an example that illustrates the main challenges in permitting temporary violations of type consistency with respect to heap-allocated objects. This example is drawn from verifying reflective call safety in real-world Objective-C code, which requires such temporary violations to be able to use simple, global type invariants.

3.1 Flow-Insensitive Dependent Types for Reflection Safety in Objective-C

In this section, I present an example of how programmers use reflective method call to avoid boilerplate code and to decouple components in Objective-C, a language that makes pervasive use of reflection. I then provide an overview of a flow-insensitive dependent type system for verifying reflection safety in Objective-C that is **almost** precise enough check this example. I describe this type system formally in Section 4.2.

```
@interface Button
- (void)drawState:(String * ⦇↾in {‘Up’, ‘Down’}⦈ )state {
  String *m = ...
  CustomImage *image = ...
  m = ["draw" append:state];
  [image setDelegate:self selector:m];
  [image draw];
}
- (void)drawUp { ... }
- (void)drawDown { ... }
@end
@interface CustomImage {
  Object * ⦇↾respondsTo sel()→void⦈  obj;
  String *sel;
}
- (void)setDelegate:(Object * ⦇↾respondsTo s()→void⦈ )o
           selector:(String *)s {
  self->obj = o;
  self->sel = s;
}
- (void)draw {
  [self->obj performSelector:self->sel];
}
@end
```

Figure 3.2: Verifying reflective call safety requires knowing responds-to relationships between objects and selectors.

A real-world example. Objective-C, like C++, is an object-oriented layer on top of C that adds classes and methods. We will describe its syntax as needed. Figure 3.2 shows an example, adapted from the ShortcutRecorder[1] library, of typical reflection use in Objective-C. Ignore the annotations in double parentheses ⦇·⦈ for now—these denote our additions to the language of types. The `Button` class (lines 1–11) contains a `drawState:` method (lines 2–8) that draws the button as either up or down, according to whether the caller passes the string `"Up"` or `"Down"` as the `state` argument. A class is defined within `@interface...@end` blocks; an instance method definition begins with `-`. Methods are defined and called using an infix notation inspired by Smalltalk. For example, the code at line 6 calls the `setDelegate:selector:` method on the `image` object with `self` as the first argument and `m` as the second. This call is analogous to `image->setDelegateSelector(self,m)` in C++.

Now, a `Button` object draws itself by using the `CustomImage` to call either `drawUp` or `drawDown`. The `CustomImage` sets up a drawing context and **reflectively** calls the passed-in selector on the passed-in delegate at line 22—the delegate and selector pair form, in essence, a callback, similar to the `Callback` example discussed in Figure 2.2. This syntax `[`o `performSelector:`s`]` for reflective call in Objective-C is analogous to o`.send(`s`)` in Ruby, `getattr(`o`,`s`)()` in Python, and o`[`s`]()` in JavaScript. In this case, the delegate is set to the `Button` object itself, and the selector is constructed by appending the passed-in `state` string to the string constant `"draw"` (lines 5–6). Constructing the selector dynamically reduces boilerplate by avoiding, for example, a series of `if` statements inspecting the `state` variable. Using reflection for callbacks also improves decoupling—`CustomImage` is agnostic to the identity of the delegate. This delegate idiom is one common way responder-selector pairs arise in Objective-C and other dynamic languages.

The use of reflection in this example comes at a cost: while the Objective-C type system statically checks that directly called methods exist, it cannot do so for reflective calls—these are only checked at run time. In this work, we present an analysis that enables modular static checking of reflective call safety while still maintaining the benefits of reduced boilerplate. To prove that the

[1] https://github.com/shortcutrecorder/shortcutrecorder

program is reflection-safe, we use refinement types [42, 46, 90] to ensure that the responder does, in fact, respond to the selector.

To see how these "responds-to" relationships arise, consider the reflective call at line 22. It will throw a run-time error if the receiver does not have a method with the name specified in the argument—conversely, to be safe, it is sufficient that `self->obj` responds to `self->sel`. There is an unexpressed invariant requiring that for every instance of `CustomImage`, the object stored in the `obj` field must respond to the selector stored in the `sel` field. We capture this invariant by applying the respondsTo selector()→`void` refinement to the `obj` field at line 13. This refinement expresses the desired **relationship** between the `obj` field and the `sel` field. The method signature ()→`void` states that the `sel` field holds the name of a method that takes zero parameters with return type `void`. This relationship expresses an intuitive invariant that, unfortunately, does not quite hold everywhere. Still, our analysis is capable of using this **almost**-everywhere invariant to check that the required relationships hold when needed.

Working backward, we see that the `setDelegate:selector:` method updates the `obj` and `sel` fields with the values passed as parameters—this demonstrates the need, in a modular analysis, for respondsTo refinements to apply to parameters as well as fields. We annotate parameter `o` to require that it responds to `s`. In order for this relationship to hold on the parameters, any time the method is called, the first argument must respond to the second. Thus at the call to `setDelegate:selector:` at line 6, the analysis must ensure that `self` responds to `m`. In the caller (i.e., the client of `CustomImage`), we know a precise type, `Button`, for the first argument (while from the callee's point of view it is merely `Object`). This means we know that, from the caller's point of view, the delegate will respond to the selector if the selector is a method on `Button`—so if we limit the values `m` can take on to either `"drawUp"` or `"drawDown"` the respondsTo refinement in the callee will be satisfied. We write in for a refinement that limits strings to one of a set of string constants (i.e., a union of singletons). For simplicity in presentation, this is our only string refinement, although more complex string reasoning is possible.

Subtyping with refinement types. Our approach relies on a subtyping judgment $\Gamma \vdash$

$T_1 <: T_2$ in the dependent-refinement type system that is a static over-approximation of semantic inclusion (i.e., under a type environment Γ, the concretization of type T_1 is contained in the concretization of T_2). As an example, we consider informally subtyping with the respondsTo and the in refinements for reflective call safety. For in refinements, this is straightforward: an in-refined string is a subtype of another string if the possible constant values permitted by the first are a subset of those required by the second. The situation for subtyping the respondsTo refinement is complicated by the fact that a relationship refinement can refer to the **contents** of related storage locations. Consider the `Button *` type in the `drawState:` method. The type environment, Γ, limits the local variable `m` to hold either `"drawUp"` or `"drawDown"`. Because (1) `Button *` is a subtype of `Object *` in the base Objective-C type system and (2) `Button` has both a `drawUp` and a `drawDown` method, it is the case that `Button *` is a subtype of $\texttt{Object *} \restriction \mathsf{respondsTo}\,\mathsf{m}$. This relationship is specific to the environment. If, for example, $\Gamma(\mathsf{m})$ instead had refinement $\mathsf{in}\,\{\text{'Fred'}\}$ the above subtyping relationship would not hold. Note that for presentation we have elided the method type on the respondsTo here; we do so whenever it is not relevant to the discussion.

Type checking field assignments. We check field assignments flow-insensitively with a weakest-preconditions–based approach similar to the Deputy type system [27] (although extended to handle subtyping). Here, we focus on why flow-insensitive typing raises alarms for the field assignments `self->obj = o` at line 18 and `self->sel = s` at line 19 (see Section 4.2.5 for more details on how checking proceeds).

To check the first assignment, we first augment the type environment with fresh locals representing the fields of the assigned-to object and then check the assignment as if it were a local update. Conceptually, we **temporarily** bring field storage locations into scope and give them local names. Let this augmented type environment be:

$$\begin{aligned}\Gamma_{\mathrm{a}} \quad = \quad & \Gamma[\mathsf{o} : \texttt{Object *} \restriction \mathsf{respondsTo}\,\mathsf{s}] \\ & [\mathsf{s} : \texttt{String *} \\ & [\mathsf{obj} : \texttt{Object *} \restriction \mathsf{respondsTo}\,\mathsf{sel}] \\ & [\mathsf{sel} : \texttt{String *}\end{aligned}$$

for some Γ and where we explicitly show the two respondsTo refinements (for presentation, we use the field names `obj` and `sel` as the fresh locals). Checking the assignment to `obj` requires the traditional subtyping check $\Gamma_{\mathrm{a}} \vdash \Gamma_{\mathrm{a}}(\mathsf{o}) <: \Gamma_{\mathrm{a}}(\mathsf{obj})$—that is, that:

$$\Gamma_{\mathrm{a}} \vdash \texttt{Object *} \restriction \mathsf{respondsTo}\,\mathsf{s} <: \texttt{Object *} \restriction \mathsf{respondsTo}\,\mathsf{sel}$$

This subtyping constraint—which does not hold—expresses the requirement that the relationship between obj and sel should be preserved across the assignment.

While this subtyping check is what would be prescribed in a standard, non-dependent type system, we want a similar check to be required (and thus also cause a flow-insensitive type error) when checking the next line (line 19) where `sel` is updated. This update mutates a storage location that is referred-to in a dependent refinement but does not itself have a dependent type. Our weakest-preconditions–based approach uses environment subtyping under substitution to ensure that the referencing type is not invalidated. Because the type environment Γ_{a} is a flow-insensitive invariant, we require it to hold after the assignment. Now, treating type environments as a formula, the weakest precondition of Γ_{a} with respect to the assignment `self->sel = s` is $\Gamma_{\mathrm{a}}[\mathsf{sel} \mapsto \mathsf{s}]$—that is, if Γ_{a} substituting s for sel holds before the assignment then Γ_{a} will definitely hold after (this is the backwards Hoare rule for assignment applied to a type environment). But, because of flow-insensitivity, again, we can assume that Γ_{a} holds before the assignment. So, conceptually, in order to show that the assignment is safe, it suffices to show that every state that satisfies Γ_{a} also satisfies $\Gamma_{\mathrm{a}}[\mathsf{sel} \mapsto \mathsf{s}]$. Treating type environments as logical formulas, this is essentially showing that $\Gamma_{\mathrm{a}} \Rightarrow \Gamma_{\mathrm{a}}[\mathsf{sel} \mapsto \mathsf{s}]$. From a types perspective, we can satisfy this implication with subtyping by showing that:

$$\Gamma_{\mathrm{a}}(x[\mathsf{sel} \mapsto \mathsf{s}]) <: T[\mathsf{sel} \mapsto \mathsf{s}] \text{ for all } x : T \text{ in } \Gamma_{\mathrm{a}}$$

Here we write $e[x \mapsto y]$ and $T[x \mapsto y]$ for substituting x with y in expression e and type T, respectively. When x is obj and T is $\texttt{Object *} \restriction \mathsf{respondsTo}\,\mathsf{sel}$ we find a requirement similar to that for `self->obj = o`, but with the subtyping relation reversed:

$$\Gamma_{\mathrm{a}} \vdash \texttt{Object *} \restriction \mathsf{respondsTo}\,\mathsf{sel} <: \texttt{Object *} \restriction \mathsf{respondsTo}\,\mathsf{s}$$

The augmented type environment Γ_{a} does not guarantee this requirement, either—so the type system would generate an alarm at line 19, as well.

The key thing to note about these two assignments is that although each is unsafe in isolation, considered in combination, they are safe. The first assignment breaks the invariant that `obj` should respond to `selector`, but the second restores it. These temporary violations cannot be tolerated by a flow-insensitive type analysis because, in imperative languages, flow-insensitive types on storage locations really perform two duties. First, they express facts about values: any value read from a variable with type $\mathsf{respondsTo}\ \mathsf{m}$ can be assumed to respond to the value stored in location m. But second, they express constraints on mutable storage: for the fact to universally hold, the type system must disallow any write to that variable of a value that does not respond to m. These constraints are fine for standard types but are problematic for relationships that are established or updated in multiple steps.

3.2 Leveraging Type Invariants During Symbolic Analysis

As we argued in Section 2.3, moving to a flow-sensitive treatment of typing is **too pessimistic**. Our work is motivated by the observation that although programmers do sometimes violate refinement relationships, most of the time these relationships hold—they are **almost** flow-insensitive (Hypothesis 1). And further, even when some relationship is violated, most other relationships are not (Hypothesis 2). We say such a heap is **almost type-consistent**. To set up this notion, we first make explicit a standard notion of type-consistency.

Definition 1 (Type-Consistency)**.** *A storage location is **type-consistent** if the values stored in it and all locations in its reachable heap conform to the requirements imposed by their flow-insensitive refinement type annotations.*

Thus, a storage location is **type-inconsistent** if either (1) the value stored in it **immediately** without pointer dereferences violates a type constraint; or (2) there is a type-inconsistent location **transitively** in its reachable heap. We distinguish these two cases of **immediately type-inconsistent** versus only **transitively type-inconsistent**.

In this work we rely on two premises about how programmers violate refinement relationships over storage locations on the heap—these are the two almost-everywhere hypotheses from Chapter 2 instantiated for type consistency:

Premise 1 **All** of the heap is type-consistent **most** of the time.

Premise 2 **Most** of the heap is not immediately type-inconsistent **all** of the time. In other words, only a few locations are responsible for breaking the global type invariant at any time.

Following **Premise 1**, we apply type analysis when the heap is type-consistent and switch to symbolic analysis when the type invariant is violated (this is the approach illustrated in Figure 3.1) Under this premise, these periods of violation are bounded in execution—and short enough that the path explosion from precise symbolic analysis is manageable.

Premise 2 is at the core of our approach to soundly handling temporary type violations on heap locations. The key idea is a view of the heap as being made up of two separate regions: (a) a small number of individual locations that are allowed to be immediately type-inconsistent and (b) an **almost type-consistent** region consisting of (fully) type-consistent or only transitively type-inconsistent locations, which we illustrate pictorially in Figure 3.3.

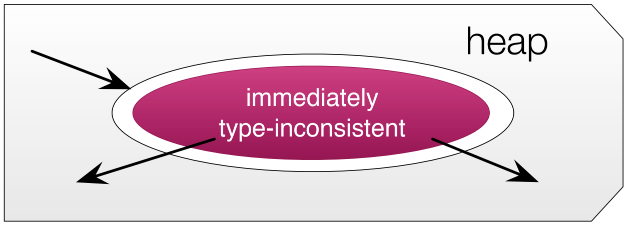


Figure 3.3: The heap is split into immediately type-inconsistent and almost type-consistent regions.

Here, the dark node represents one location that is immediately type-inconsistent, while the light area around it is not immediately type-inconsistent. Note that there may be pointers (shown as arrows) from the light region to the immediately type-inconsistent region. We call the light region the "almost type-consistent heap" because the objects in it are only transitively type-inconsistent.

In the analysis, the locations in the almost type-consistent heap are summarized and represented by an atomic assertion ok, while the possibly immediately type-inconsistent locations are materialized and explicitly given in a separation logic symbolic memory (described below). As we will see, the key analysis operations involve moving objects between these two regions.

Handoff Example. To more concretely illustrate our approach, we now walk though how the analysis intertwines flow-insensitive and path-sensitive reasoning as well as how it reasons about the heap. Figure 3.4 describes the verification of the `setDelegate:selector:` method from Figure 3.2. The boxed regions indicate analysis invariants at each program point. We also provide graphical representations of these invariants (we describe these below).

The type analysis will detect that the assignment at line 18 violates the flow-insensitive invariant, which is described by a type environment Γ mapping local variables to their required types. It will then back up to a program point where the **global type invariant holds** (marked by ①) and switch to the symbolic analysis (corresponding to the handoff by ✘ in Figure 3.1). At this point, the analysis symbolizes the type environment (②), splitting it into a symbolic state. For this example, we provide representations of symbolic state in both formula and graphical form.

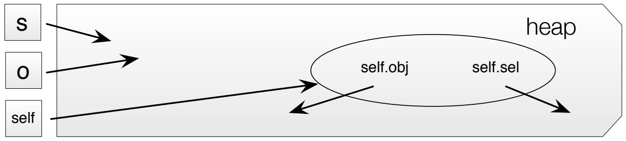
Symbolic static in formula form. As a formula, the symbolic state $\widetilde{\mathrm{E}} \mid\mid \widetilde{\mathrm{H}} \mid\mid \widetilde{\Gamma}$ consists of three components: $\widetilde{\mathrm{E}}$, an environment mapping local variables to the symbolic values stored in them; $\widetilde{\mathrm{H}}$, a separation-logic-based representation of the heap; and $\widetilde{\Gamma}$, a value typing mapping symbolic values to refinement types lifted to symbolic values. As we will see, this value typing describes facts known about symbolic values—expressing, for example, that $\widetilde{\mathrm{s}}$ is a string or that $\widetilde{\mathrm{o}}$ responds to $\widetilde{\mathrm{s}}$. As a notational convenience, we name symbolic values by the storage location they initially came from. For example, the initially symbolized symbolic environment $\widetilde{\mathrm{E}}$ says that the symbolic value $\widetilde{\mathrm{o}}$ is stored in the local variable o at the time of the split. Symbolic separation-logic heaps $\widetilde{\mathrm{H}}$ can be the empty heap emp, a single materialized object (e.g., $\widetilde{\mathrm{self}} \mapsto \{\mathsf{obj} \mapsto \widetilde{\mathsf{obj}} * \mathsf{sel} \mapsto \widetilde{\mathrm{sel}}\}$), the separating conjunction of two symbolic sub-heaps $\widetilde{\mathrm{H}}_1 * \widetilde{\mathrm{H}}_2$, or the (non-standard) formula literal ok, whose concretization includes all concrete sub-heaps whose fields are not **immediately inconsistent** with their declared field types.

① Global type invariant holds:

$$\boxed{\Gamma = \mathsf{self} : \mathsf{CustomImage}{*}, \mathsf{o} : \mathsf{Object}{*} \restriction \mathsf{r2}\,\mathsf{s}, \mathsf{s} : \mathsf{String}}$$

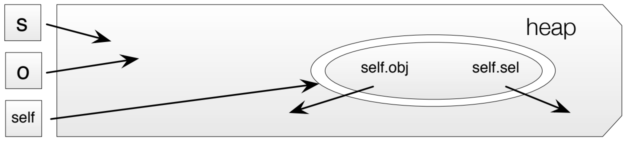
② Symbolize type environment:

$$\begin{aligned}
&\widetilde{E} = \mathsf{self} \mapsto \widetilde{\mathrm{self}} \circledast \mathsf{o} \mapsto \widetilde{\mathrm{o}} \circledast \mathsf{s} \mapsto \widetilde{\mathrm{s}} \mid\mid \\
&\widetilde{H} = \mathsf{ok} \mid\mid \\
&\widetilde{\Gamma}_{②} = \widetilde{\mathrm{self}} : \mathsf{CustomImage}{*}, \widetilde{\mathrm{o}} : \mathsf{Object}{*} \restriction \mathsf{r2}\,\widetilde{\mathrm{s}}, \widetilde{\mathrm{s}} : \mathsf{String}
\end{aligned}$$



③ Type-consistent materialization:

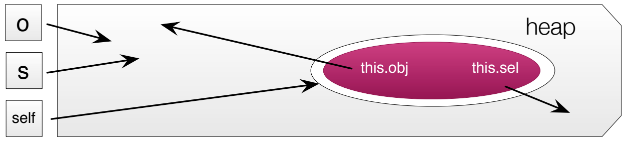
$$\begin{aligned}
&\widetilde{H} = \mathsf{ok} \circledast \widetilde{\mathbf{self}} \mapsto \{\mathbf{obj} \mapsto \widetilde{\mathbf{obj}} \circledast \mathbf{sel} \mapsto \widetilde{\mathbf{sel}}\} \mid\mid \\
&\widetilde{\Gamma}_{③} = \widetilde{\mathrm{self}} : \mathsf{CustomImage}{*}, \widetilde{\mathrm{o}} : \mathsf{Object}{*} \restriction \mathsf{r2}\,\widetilde{\mathrm{s}}, \widetilde{\mathrm{s}} : \mathsf{String}, \widetilde{\mathrm{obj}} : \mathsf{Object}{*} \restriction \mathsf{r2}\,\widetilde{\mathrm{sel}}, \widetilde{\mathrm{sel}} : \mathsf{String}
\end{aligned}$$



```
self->obj = o;
```

④ Strong update for obj:

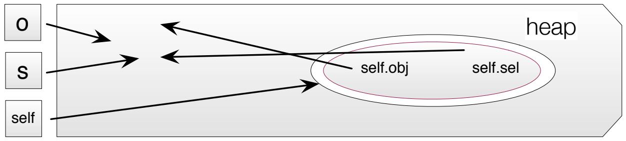
$$\boxed{\widetilde{H} = \mathsf{ok} \circledast \widetilde{\mathrm{self}} \mapsto \{\mathsf{obj} \mapsto \widetilde{\mathbf{o}} \circledast \mathsf{sel} \mapsto \widetilde{\mathrm{sel}}\} \mid\mid \widetilde{\Gamma}_{③}}$$



```
self->sel = s;
```

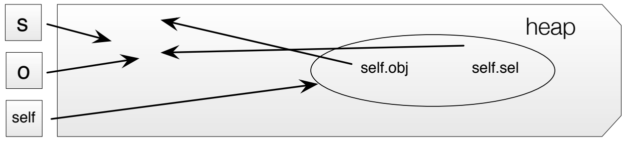
⑤ Strong update for sel:

$$\boxed{\widetilde{H} = \mathsf{ok} \circledast \widetilde{\mathrm{self}} \mapsto \{\mathsf{obj} \mapsto \widetilde{\mathrm{o}} \circledast \mathsf{sel} \mapsto \widetilde{\mathbf{s}}\} \mid\mid \widetilde{\Gamma}_{③}}$$



⑥ Type-consistent summarization:

$$\boxed{\widetilde{H} = \mathsf{ok} \mid\mid \widetilde{\Gamma}_{③}}$$



⑦ Global type invariant restored:

$$\boxed{\Gamma = \mathsf{self} : \mathsf{CustomImage}{*}, \mathsf{o} : \mathsf{Object}{*} \restriction \mathsf{r2}\,\mathsf{s}, \mathsf{s} : \mathsf{String}}$$

Figure 3.4: Example: Verifying an almost-everywhere invariant.

Symbolic state in graphical form. We also provide graphical representations of these symbolic state invariants. The square boxes on the left side of these figures represent the stack: local variables s, o, and self. The right region represents the heap. Black arrows are pointers—so, for example, the invariant at point ② says that the self local variable contains a pointer to an object on the heap with fields obj and self. Graphically, we make a distinction between portions of the heap that are summarized in ok and those that are explicitly materialized: we represent summarized storage as contiguous with the heap (the oval at point ②, for example) and materialized storage as cut out (the inset oval at ③, for example). Magenta cells (in ④, for example) represent storage that is immediately type-inconsistent, while grey cells represent storage that is at most transitively type-inconsistent.

Symbolization. Symbolization (②) enables the symbolic analysis to reason about the contents of memory locations (i.e., $\widetilde{\mathrm{E}}$ and $\widetilde{\mathrm{H}}$) separately from facts ($\widetilde{\Gamma}$) known about values—and, crucially, allows the values stored in these locations to be inconsistent with the invariants required by their declared types. The key soundness criteria for a newly split symbolic state is that (1) it must have locals storing values that have the same types as specified in the type environment and (2) it must make no assumptions about aliasing. We describe symbolization formally in Section 4.4.2.

Immediately after the symbolization, the constraints in the global type environment still hold, so all fields on the heap must be type-consistent. To capture this condition, the symbolic analysis initially assumes that the entire symbolic heap is represented by ok. In this example, the symbolic analysis can initially assume that the type environment holds and so the value stored in parameter o must respond to the value stored in parameter s. We indicate this in the value typing by giving $\widetilde{\mathrm{o}}$ the symbolic type $\mathsf{Object}{*} \upharpoonright \mathsf{r2}\,\widetilde{\mathrm{s}}$. (For brevity, we represent the $\mathsf{respondsTo}$ relation as $\mathsf{r2}$ and omit the symbolic environment, which does not change in this example). Note that the refinement for $\widetilde{\mathrm{o}}$ is dependent on a symbolic value $\widetilde{\mathrm{s}}$ and not a local variable.

Materialization. Before the analysis can reason about the field updates, it must perform **type-consistent materialization** (③) from ok. This materialization makes the storage for the object pointed to by $\widetilde{\mathrm{self}}$ explicit on the symbolic heap, which now says that $\widetilde{\mathrm{self}}.\mathsf{obj}$ contains the symbolic value $\widetilde{\mathrm{obj}}$ and $\widetilde{\mathrm{self}}.\mathsf{sel}$ contains the symbolic value $\widetilde{\mathrm{sel}}$. Because this storage comes from ok, the analysis can assume the invariant required by the dependent refinement types on the fields of $\widetilde{\mathrm{self}}$ (a CustomImage object) hold. To reflect this, the analysis adds value typings (symbolic facts) to $\widetilde{\Gamma}$ for the fresh symbolic values stored in those fields: $\widetilde{\mathrm{obj}}$ is guaranteed to respond to $\widetilde{\mathrm{sel}}$ with the value typing $\mathsf{Object}{*} \restriction \mathsf{r2}\ \widetilde{\mathrm{sel}}$ and $\widetilde{\mathrm{sel}}$ is guaranteed to be a string. In essence, this process pulls information on demand from from the type analysis into the symbolic analysis. We formalize this materialization in Section 4.4.3. The value typing $\widetilde{\Gamma}$ does not change from this point on, so we do not repeat its contents in our boxed invariants. With explicit storage materialized in a separation logic representation of the heap, the symbolic analysis can now perform **strong updates** as it interprets the field writes to obj and sel.

Summarization. After the first write (④), obj may not respond to sel, and so the object may be immediately type-inconsistent (magenta, in the graphical representation)—but after the second write (⑤), the object has returned to a not immediately type-inconsistent state (because the pure facts say that $\widetilde{\mathrm{o}}$ responds to $\widetilde{\mathrm{s}}$). The analysis can now perform **type-consistent summarization** (⑥) and safely summarize the storage for $\widetilde{\mathrm{self}}$ back into ok. Further, now that the symbolic heap consists solely of ok and the locals are consistent with their declared types, it must be the case that the **global invariant has been restored**. That is, if no part of the heap is immediately type-inconsistent, then all parts of the heap must be type-consistent. At this point (⑦), the analysis switches back to flow-insensitively checking the rest of the program.

With this approach, the analysis must reason with expensive symbolic analysis only when a global invariant is violated. Further, even during symbolic analysis, the analysis must reason explicitly only about those objects for which an invariant is violated—all others can be summarized in ok. Given this mechanism, we can easily allow for more than one materialization at a time as long as we disjunctively account for possible aliasing with already materialized locations—Premise 2 suggests that this explosion is also manageable.

3.3 Local Heap Reasoning with Gated Separation

As we saw in Section 3.2, the symbolic analysis can switch back to type checking when the entire heap has been summarized into ok. This approach negotiates the inherent mismatch between the global, alias-agnostic reasoning in type systems on the one hand and the local, alias-aware reasoning on the other by granting the type analysis ownership of the **entire heap**. That is, when checking a type block, the approach outlined in Section 3.2 initially assumes that the entire heap is solely constrained by the type environment. Correspondingly, when switching from separation logic to types, it must guarantee that the entire heap can be fully described by traditional types.

Problem: All-or-nothing handoff is insufficient. Unfortunately, this "all-or-nothing" approach is not always sufficient—some programs require a more flexible approach that permits controlled, cross-type-analysis preservation of separation logic invariants.

```
    ⟦t
1     - (CustomImage *)createWithDelegate:(Object * ⦇↾respondsTo s()→void⦈ )o
2                                 selector:(String *)s
        ⟦s
3         CustomImage *image = [CustomImage alloc];
4         image->obj = o;
          ⟦t
5           [self someMethod];
          ⟧t
6         image->sel = s;
        ⟧s
7       return image;
      }
    ⟧t
```

Figure 3.5: Framing out before switching to type checking.

I give an example of such a program in Figure 3.5. The -createWithDelegate:selector: method creates a new CustomImage object, initializing its obj and sel fields with the passed-in parameters (ignore the ⟦·⟧ annotations for the moment—I will dicuss them in a moment). The programmer allocates a new CustomImage object at line 3, updates its obj field at line 4, calls another method at line 5 and then updates the sel field (line 6) and returns the initialized image. Let us assume that the modular analysis does not have access to the source code of -someMethod but that it does have a type signature (that the method takes no parameter and has no return value). The type-intertwined approach described in Section 3.2 can handle calls to such methods by fully switching to type checking to check the call. In the example above, I describe an intertwined analysis strategy with nested $\llbracket \cdot \rrbracket$ annotations: a $\llbracket_{\mathrm{s}} \cdot \rrbracket_{\mathrm{s}}$ block indicates that a region of code should be checked with separation logic, while $\llbracket_{\mathrm{t}} \cdot \rrbracket_{\mathrm{t}}$ indicates a type-checked block. With this strategy, the analysis will start with type checking and then switch to separation logic to reason about the allocation and assignment. It will then switch to **nested** type checking for the method call, returning to separation logic afterward. Then, after the image has been fully initialized it will switch back to type checking before the return.

Unfortunately, this strategy is out of reach for the analysis as described in Section 3.2. The key complication in this example is that the programmer calls the method when only one of the two related fields in `CustomImage` have been initialized—that is, at line 5 the **global type invariant does not hold**. The conditions for handoff described in Section 3.2—that the heap consists entirely of `ok`—do not apply because the immediately type-inconsistent storage for the newly allocated image cannot be summarized into `ok`. Futher, even if that analysis could switch to types at line 5, it would lose the separation-logic-level must-alias guarantee that the value passed in the parameter `o` is the same as the value stored in the field `obj`, which is crucial to showing that returned `CustomImage` is not immediately-type-inconsistent at line 7.

Need: Framing across intertwined type blocks. What is needed, when switching from separation logic to type analysis, is the ability to "frame out" the portion of the heap for the newly allocated `CustomImage` and thus prevent a nested type analysis block from accessing the framed-out memory. A sound framing out would rule out two unsound behaviors. First, it would prevent the type analysis from relying upon a type invariant (in this case, the responds-to relationship between `obj` and `sel`) when it it does not really hold; and second, it would ensure that the type-checked code does not invalidate a separation logic heap invariant behind the back of the symbolic analysis (in this case, that `obj` and `o` contain the same value).

Unfortunately, **the traditional frame rule is unsound for type-intertwined analysis**.

This rule, which we described informally in Section 1.1.2, allows separation logic to shear off a disjoint portion of the heap (the frame) and analyze a command with respect to only the portion of the heap that it accesses (the footprint). In separation logic alone, this rule is sound because the analysis can ensure that the command does not access memory that is disjoint from the footprint (that is, is joined via a separating conjunction $*$ with the footprint.)

Types, however, do not respect the traditional frame rule. This is because type-checking allows access to the entire reachable heap. Consider a faulty version of the `-setDelegate:selector:` method from Figure 3.2 that has been modified to call the `-draw` method while invariant is violated:

```
- (void)setDelegate:(Object * ⦇⌈respondsTo s()→void⦈ )o
          selector:(String *)s {
  self->obj = o;
  [self draw];
  self->sel = s;
}
- (void)draw {
  [self->obj performSelector:self->sel];
}
```

Here, the call to the `-draw` method may cause a reflection safety error because at the time the method is called, `self->obj` may not respond to `self->sel`. Adding the traditional frame rule to our analysis could, unsoundly, cause this erroneous version to type check. I illustrate this unsoundness in Figure 3.6.

This unsound verification proceeds similarly to sound one presented in Figure 3.4 (although I omit the graphical representation of invariants). As before, the global type invariant holds upon entry to the method (①). The analysis will similarly symbolize a new symbolic state that splits the type invariant Γ into a symbolic state and materialize storage for $\widetilde{\mathrm{self}}$ to be able to perform a strong update for the assignment to `self->obj`. The analysis state after this assignment (at ②) leaves the storage for $\widetilde{\mathrm{self}}$ in a type-inconsistent state because $\widetilde{\mathrm{o}}$ does not respond to $\widetilde{\mathrm{sel}}$.

The traditional frame rule would allow the analysis to unsoundly frame out this inconsistent storage, leaving the symbolic heap (shown shaded at ③) to consist of solely the ok token. This is a signal to the analysis that the entire heap is type consistent (clearly, it is not) and so the analysis determines that the global type invariant is restored (④) and switches back to type checking without

① **Global type invariant holds:**

$$\boxed{\Gamma = \mathsf{self} : \mathsf{CustomImage}{*}, \mathsf{o} : \mathsf{Object}{*} \restriction \mathsf{r2}\,\mathsf{s}, \mathsf{s} : \mathsf{String}}$$

$\vdots$

```
self->obj = o;
```

② **After symbolization, materialization, strong update:**

$$\boxed{\begin{array}{l} \widetilde{\mathrm{E}} = \mathsf{self} \mapsto \widetilde{\mathrm{self}} \ast \mathsf{o} \mapsto \widetilde{\mathrm{o}} \ast \mathsf{s} \mapsto \widetilde{\mathrm{s}} \;|| \\ \widetilde{\mathrm{H}} = \mathsf{ok} \ast \widetilde{\mathrm{self}} \mapsto \{\mathsf{obj} \mapsto \widetilde{\mathbf{o}} \ast \mathsf{sel} \mapsto \widetilde{\mathrm{sel}}\} \;|| \\ \widetilde{\Gamma} = \widetilde{\mathrm{self}} : \mathsf{CustomImage}{*}, \widetilde{\mathrm{o}} : \mathsf{Object}{*} \restriction \mathsf{r2}\,\widetilde{\mathrm{s}}, \widetilde{\mathrm{s}} : \mathsf{String}, \widetilde{\mathrm{obj}} : \mathsf{Object}{*} \restriction \mathsf{r2}\,\widetilde{\mathrm{sel}}, \widetilde{\mathrm{sel}} : \mathsf{String} \end{array}}$$

③ **Frame out storage for $\widetilde{\mathrm{self}}$:**

$$\boxed{\begin{array}{l} \widetilde{\mathrm{E}} = \mathsf{self} \mapsto \widetilde{\mathrm{self}} \ast \mathsf{o} \mapsto \widetilde{\mathrm{o}} \ast \mathsf{s} \mapsto \widetilde{\mathrm{s}} \;|| \\ \widetilde{\mathrm{H}} = \mathsf{ok} \;\; || \\ \widetilde{\Gamma} = \widetilde{\mathrm{self}} : \mathsf{CustomImage}{*}, \widetilde{\mathrm{o}} : \mathsf{Object}{*} \restriction \mathsf{r2}\,\widetilde{\mathrm{s}}, \widetilde{\mathrm{s}} : \mathsf{String}, \widetilde{\mathrm{obj}} : \mathsf{Object}{*} \restriction \mathsf{r2}\,\widetilde{\mathrm{sel}}, \widetilde{\mathrm{sel}} : \mathsf{String} \end{array}}$$

④ **Global type invariant unsoundly appears restored**:

$$\boxed{\Gamma = \mathsf{self} : \mathsf{CustomImage}{*}, \mathsf{o} : \mathsf{Object}{*} \restriction \mathsf{r2}\,\mathsf{s}, \mathsf{s} : \mathsf{String}}$$

```
[self draw];
```

$\vdots$

```
self->sel = s;
```

Figure 3.6: The traditional frame rule with $\ast$ is unsound in type-intertwined separation logic.

an alarm—an unsound false negative.

The crux of the unsoundness is a fundamental mismatch between how separation logic invariants and type environments constrain the heap. As I described in Section 1.1, separation logic memory formulas divide the heap into disjoint regions and make the strong guarantee that a command will not access memory outside of its allowed region. A key thing to note is that separation logic formulas allow pointers outside of its region—it just prevents those pointers from being dereferenced (by generating an alarm). In contrast, a type environment must allow access to the entire heap reachable from its roots—it cannot shear off a disjoint portion of the heap if that portion is reachable from some root. Because the storage for $\widetilde{\mathrm{self}}$ is reachable from the self root, it cannot be safely framed from out of view of any typechecked code that has access to the root. Ultimately, a stronger guarantee than the disjointness provided by traditional separating conjunction is needed to prevent type-intertwined interference.

Contribution: Gated Separation. We enrich separation logic with **gated separating conjunction**, a non-commutative strengthening of separating conjunction that prevents type-intertwined interference. Like traditional separating conjunction, gated separating conjunction constrains two sub-heaps to be disjoint—but it additionally constrains the **range** of one sub-heap (what we call the **foregate**) to be disjoint with the domain of the other (the **aftgate**). This strengthening ensures that the foregate does not directly point into the aftgate but freely allows pointers from aftgate to the foregate. We write the gated separation of a symbolic foregate heap from a symbolic aftgate heap as $\widetilde{\mathrm{H}}_{\mathrm{fore}} \triangleleft\!\!\triangleleft \widetilde{\mathrm{H}}_{\mathrm{aft}}$. The direction of the triangle is mean to convey the direction in which pointers are allowed: from aftgate to forgate.

As we will see, this disjoint "dis-pointing" relationship is strong enough to enable sound framing across a type-intertwined block via the application of the **type-intertwined frame rule**. Consider again the `-createWithDelegate:selector:` method from Figure 3.5, in which the programmer partially initializes a `CustomImage` object and calls a method, requiring a switch to type checking. In Figure 3.7, I show a graphical representation of the state of memory immediately before the method call at line 5. There are four local variables: parameters `s` and `o`, the `self` variable (containing the current receiver), and `image`. Both `s` and `o` contain a pointer to unknown locations on the heap, and `image` contains a pointer to the newly allocated `CustomImage` object. At this point, the image's `obj` field has been initialized to contain the same value as `o`—but image's `sel` points to some unknown location on the heap. Because `obj` may not respond to `sel`, the object is immediately type-inconsistent—so we mark it magenta.

The memory locations inside the dashed area have the key property that while they point into the rest of the heap, there are no pointers from the rest of the heap into the dashed area. In other words, the dashed area is gate-separated from both the rest of the heap and the storage for `self`. With this guarantee, there is no sequence of dereferences starting from `self` and ending in the inconsistent memory. For this reason, the immediately type-inconsistent storage for the image can be safely framed out. Gated separating conjunction ensures that there is no way that a command given access to the rest of the heap (the foregate) can access the aftgate (the dashed area)—even if

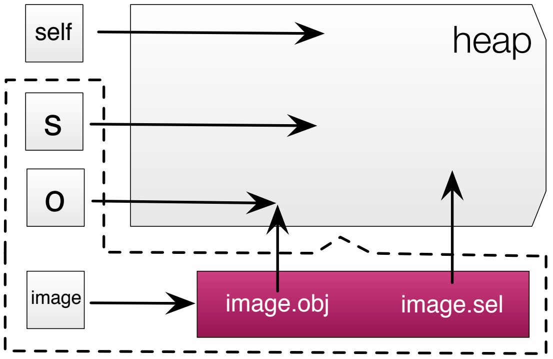


Figure 3.7: Gated separation prevents the foregate from pointing into the aftgate.

analysis of the command (or some subcommand) switches to type checking.

In Figure 3.8, I present an example verification that shows how gated separation allows an analysis to switch to typechecking even when memory in the aftgate is immediately type inconsistent. This example verifies the `-createWithDelegate:selector:` method from Figure 3.5. Verification starts by symbolizing a symbolic state (①) from the type environement. As discussed previously, a newly symbolized state has a fully type-consistent heap—so the heap consists solely of the `ok` token. Allocation creates fresh storage—a new address—so it is guaranteed that any existing pointers on the heap do not point to it. For this reason, at point ② we mark the newly allocated `CustomImage` object as gate separated (shown shaded) from the rest of the heap. We add symbolic values ($\widetilde{\mathrm{obj}}$ and $\widetilde{\mathrm{sel}}$) for the values in the fields of this storage but do not assume any facts about them. After the first assignment (③), the storage for the image object is immediately type-inconsistent; this state corresponds to the graphical representation in Figure 3.7.

Because the storage for the object is gate-separated from the rest of the heap, the analysis can soundly frame it out, yielding the invariant at point ④ (shown indented). Note that it also removes the stack roots `o`, `s`, and `image`. With this storage removed, the heap now consists solely of `ok`—and so the analysis determines that it is safe to switch to type checking with a new type environment (point ⑤) containing only the `self` root. The type checker treats the type environment as a flow-insensitive invariant and so guarantees that the invariant also holds at point ⑥ after checking the call to `-someMethod`. The analysis can then symbolize this type invariant, yielding the

$$\boxed{\Gamma = \mathsf{self} : \mathsf{SomeObject}{*}, \mathsf{o} : \mathsf{Object}{*} \restriction \mathsf{r2}\,\mathsf{s}, \mathsf{s} : \mathsf{String}}$$

① **After symbolization:**

$$\boxed{\begin{array}{l} \widetilde{\mathrm{E}} = \mathsf{self} \mapsto \widetilde{\mathrm{self}} \ast \mathsf{o} \mapsto \widetilde{\mathrm{o}} \ast \mathsf{s} \mapsto \widetilde{\mathrm{s}} \mid\mid \\ \widetilde{\mathrm{H}} = \mathsf{ok} \mid\mid \\ \widetilde{\Gamma} = \widetilde{\mathrm{self}} : \mathsf{SomeObject}{*}, \widetilde{\mathrm{o}} : \mathsf{Object}{*} \restriction \mathsf{r2}\,\widetilde{\mathrm{s}}, \widetilde{\mathrm{s}} : \mathsf{String} \end{array}}$$

```
CustomImage *image = [CustomImage alloc];
```

② **Freshly allocated memory is gate separated:**

$$\boxed{\begin{array}{l} \widetilde{\mathrm{E}} = \mathsf{self} \mapsto \widetilde{\mathrm{self}} \ast \mathsf{o} \mapsto \widetilde{\mathrm{o}} \ast \mathsf{s} \mapsto \widetilde{\mathrm{s}} \ast \mathsf{image} \mapsto \widetilde{\mathrm{image}} \mid\mid \\ \widetilde{\mathrm{H}} = \mathsf{ok} \lessdot \widetilde{\mathrm{image}} \mapsto \{\mathsf{obj} \mapsto \widetilde{\mathsf{obj}} \ast \mathsf{sel} \mapsto \widetilde{\mathrm{sel}}\} \mid\mid \\ \widetilde{\Gamma} = \widetilde{\mathrm{self}} : \mathsf{SomeObject}{*}, \widetilde{\mathrm{o}} : \mathsf{Object}{*} \restriction \mathsf{r2}\,\widetilde{\mathrm{s}}, \widetilde{\mathrm{s}} : \mathsf{String}, \widetilde{\mathrm{image}} : \mathsf{CustomImage}{*} \end{array}}$$

```
image->obj = o;
```

③ **After strong update:**

$$\boxed{\begin{array}{l} \widetilde{\mathrm{E}} = \mathsf{self} \mapsto \widetilde{\mathrm{self}} \ast \mathsf{o} \mapsto \widetilde{\mathrm{o}} \ast \mathsf{s} \mapsto \widetilde{\mathrm{s}} \ast \mathsf{image} \mapsto \widetilde{\mathrm{image}} \mid\mid \\ \widetilde{\mathrm{H}} = \mathsf{ok} \lessdot \widetilde{\mathrm{image}} \mapsto \{\mathsf{obj} \mapsto \boldsymbol{\widetilde{\mathrm{o}}} \ast \mathsf{sel} \mapsto \widetilde{\mathrm{sel}}\} \mid\mid \\ \widetilde{\Gamma} = \widetilde{\mathrm{self}} : \mathsf{SomeObject}{*}, \widetilde{\mathrm{o}} : \mathsf{Object}{*} \restriction \mathsf{r2}\,\widetilde{\mathrm{s}}, \widetilde{\mathrm{s}} : \mathsf{String}, \widetilde{\mathrm{image}} : \mathsf{CustomImage}{*} \end{array}}$$

④ **Frame out storage for** $\widetilde{\mathrm{image}}$**:**

$$\boxed{\begin{array}{l} \widetilde{\mathrm{E}} = \mathsf{self} \mapsto \widetilde{\mathrm{self}} \mid\mid \\ \widetilde{\mathrm{H} =} \mathsf{ok} \mid\mid \\ \widetilde{\Gamma} = \widetilde{\mathrm{self}} : \mathsf{SomeObject}{*} \end{array}}$$

⑤ **Global type invariant for** self **root holds**:

$$\boxed{\Gamma = \mathsf{self} : \mathsf{SomeObject}{*}}$$

```
[self someMethod];
```

⑥ **Global type invariant for** self **root still holds**:

$$\boxed{\Gamma = \mathsf{self} : \mathsf{SomeObject}{*}}$$

⑦ **Symbolize type environment for** self **root:**

$$\boxed{\begin{array}{l} \widetilde{\mathrm{E}} = \mathsf{self} \mapsto \widetilde{\mathrm{self}}' \mid\mid \\ \widetilde{\mathrm{H} =} \mathsf{ok} \mid\mid \\ \widetilde{\Gamma} = \widetilde{\mathrm{self}}' : \mathsf{SomeObject}{*} \end{array}}$$

⑧ **Reincorporate frame:**

$$\boxed{\begin{array}{l} \widetilde{\mathrm{E}} = \mathsf{self} \mapsto \widetilde{\mathrm{self}}' \ast \mathsf{o} \mapsto \widetilde{\mathrm{o}} \ast \mathsf{s} \mapsto \widetilde{\mathrm{s}} \ast \mathsf{image} \mapsto \widetilde{\mathrm{image}} \mid\mid \\ \widetilde{\mathrm{H}} = \mathsf{ok} \lessdot \widetilde{\mathrm{image}} \mapsto \{\mathsf{obj} \mapsto \widetilde{\mathrm{o}} \ast \mathsf{sel} \mapsto \widetilde{\mathrm{sel}}\} \mid\mid \\ \widetilde{\Gamma} = \widetilde{\mathrm{self}}' : \mathsf{SomeObject}{*}, \widetilde{\mathrm{o}} : \mathsf{Object}{*} \restriction \mathsf{r2}\,\widetilde{\mathrm{s}}, \widetilde{\mathrm{s}} : \mathsf{String}, \widetilde{\mathrm{image}} : \mathsf{CustomImage}{*} \end{array}}$$

```
image->sel = s;
  ⋮
```

Figure 3.8: The frame rule with gated separation ($\lessdot$) is sound in type-intertwined separation logic.

symbolic state at point ⑦. The symbolic environment $\widetilde{\mathrm{E}}$ in this state contains only the single self binding. Note that the symbolized value stored in that local variable is $\widetilde{\mathrm{self}}'$ and not $\widetilde{\mathrm{self}}$. This is because, in Objective-C, self acts like a true local variable—unlike this in Java, it can be written to. Thus, like for any other local variable, the analysis cannot assume that it contains the same value before and after handoff to type checking. After symbolization, the analysis can now safely reincorporate the framed-out heap and its roots back into the symbolic memory (point ⑧). Gated separation ensures that this operation is sound: the type checked portion of the program could not have interfered with its contents.

I provide a formal characterization of gated separation in Chapter 5, including a discussion of its concretization and axiom schemata (Section 5.1); a description of the challenge of establishing and maintaining gated separating conjunction in a program logic (Section 5.2); and a derivation of the type-intertwined frame rule (Section 5.3).

Chapter 4

Leveraging Almost-Everywhere Heap Invariants

As I described in Chapter 3, a successful strategy for tolerating temporary violations of almost-everywhere invariants is to intertwine a flow-insensitive type analysis with a path-sensitive symbolic analysis, applying the type analysis when the invariant holds and switching to the symbolic analysis when it does not. The key to our approach—and the most important contribution described in this dissertation—is a mechanism in the symbolic analysis to materialize memory from and then summarize back into the almost type-consistent heap. This mechanism enables the symbolic analysis to leverage and selectively violate the global type invariant over heap locations.

In this chapter, I describe and characterize Fissile Type Analysis, a type-intertwined analysis that uses this mechanism to check dependent refinement types in languages with mutable heaps. In Section 4.1, I describe a core expression language with objects and reflective method call. I present a novel, flow-insensitive type system for checking safety of reflective method calls in Section 4.2. Unfortunately, as I showed in Section 2.3, flow-insensitive typing alone is often too imprecise to check relationships between storage locations that are updated separately. Fissile Type Analysis tolerates such temporary type violations by switching to a symbolic analysis (Section 4.3) that runs until the invariant is restored. Crucially, the symbolic analysis can materialize and summarize from the almost type-consistent heap, a process I describe in Section 4.4. I provide a proof of soundness for Fissile Type Analysis in Section 4.5 and evaluate the effectiveness of the analysis in a case study verifying reflection safety in large, real-world Objective-C applications (Section 4.6). Portions of this chapter appeared in my POPL 2014 [28] paper “FISSILE Type Analysis: Modular Checking of Almost Everywhere Invariants”, which was joint work with Bor-Yuh Evan Chang.

4.1 A Language with Mutable Objects and Reflection

In this section, I briefly describe the syntax, concrete state, and semantics of a core expression language with three features of interest: objects, reflective method call, and mutable fields. The first two features provide a property to verify: the safety of reflective method calls; the third exposes the difficulty of reasoning about a mutable heap.

4.1.1 Syntax

We describe Fissile type analysis over a core imperative programming language of expressions e with objects and reflective method call. We give the syntax and types for this language in Figure 4.1. For presentation purposes, we have only three types of values: unit, strings, and objects. We assume disjoint syntactic classes of identifiers for program variables x, y, z, field names f, method names m, and parameter names p, as well as a distinguished identifier ‘self’ that stands for the receiver object in methods. Program expressions include literals for unit $\langle\rangle$, strings c, objects $\{\overline{\mathrm{var}\ f : T \;=\; e}, \overline{\mathrm{def}\ m(\overline{p : T_p}) : B_{\mathrm{ret}} \;=\; e}\}$. The ‘var’ declarations specify mutable fields f of types T, and the ‘def’ declarations describe methods m with parameters p of types T_p and with return type B_{ret}. The return type is a base type, which does not itself have refinements (but could have refinements on its fields in the case of an object base type). An overline stands for a sequence of items. Objects are heap allocated. Local variable binding ‘let $x : T = e_1$ in e_2’ binds a local variable x of type T initialized to the value of e_1 whose scope is e_2. We include one string operation for illustration: string append $x_1 \mathbin{@} x_2$. Then, we have reads of locals x and fields $x.f$, writes to fields $x.f := y$, basic control structures for sequencing $e_1; e_2$ and branching $e_1 \mathbin{[\!]} e_2$.

For presentation, we use non-deterministic branching, as the guard condition of an ‘if-then-else’ expression has no effect on flow-insensitive type checking. (This condition can be reflected in a symbolic analysis by strengthening the symbolic state with the guard condition, as is standard.) Finally, we have two method call forms: one for direct calls $z.m(\overline{x})$ and one for reflective calls

identifiers			
x, y, z, p, self			local variables
f			field names
m			method names
expressions			
e	$::=$	$\langle\rangle \mid c$	unit, string literals
	$\mid$	$\{\overline{\text{var } f : T_f \ = \ x},$ $\overline{\text{def } m(\overline{p : T_p}) : B_{\text{ret}} \ = \ e}\}$	object literals
	$\mid$	$\text{let } x \,:\, T = e_1 \text{ in } e_2$	local allocation
	$\mid$	$x_1 \mathbin{@} x_2$	string append
	$\mid$	x	local read
	$\mid$	$x.f$	field read
	$\mid$	$x.f := y$	field write
	$\mid$	$e_1 \ ; e_2$	sequencing
	$\mid$	$e_1 \mathbin{[]} e_2$	branching
	$\mid$	$z.m(\overline{x})$	direct method call
	$\mid$	$z.[y](\overline{x})$	reflective method call
types			
T^ι	$::=$	$B \ \upharpoonright \ R^\iota_1, \ldots, R^\iota_n$	refinement types
base types			
B	$::=$	$\text{Unit} \mid \text{Str}$	unit type, string type
	$\mid$	$\{\overline{\text{var } f : T_f}, \ \overline{\text{def } m(\overline{p : T_p}) \to B_{\text{ret}}}\}$	object types
refinements			
R^{I}	$::=$	$\mathsf{in}\ \{c_1, \ldots, c_n\}$	constant string value
	$\mid$	$\mathsf{respondsTo}\ \iota(\overline{p : T_p}) \to B_{\text{ret}}$	object responds-to

Figure 4.1: A core imperative programming language with objects and reflective method call.

$$\begin{array}{rlll}
\text{concrete environments} & E & ::= & \cdot \mid E[x : v] \\
\text{concrete heaps} & H & ::= & \cdot \mid H[a : \langle o, B \rangle] \\
\text{concrete objects} & \mathrm{o} & ::= & \cdot \mid o[f : v] \mid o[m : e] \\
\text{values} & \mathrm{v} & ::= & a \mid s \mid \langle\rangle \\
\text{global addresses} & \mathrm{a} & &
\end{array}$$

Figure 4.2: Concrete state.

$z.[y](\overline{x})$. A call allocates an activation record for the receiver object z and parameters $\overline{x}$; it then dispatches with the direct name m or the reflective selector y. Types T are a base type B for either unit Unit, strings Str, or objects $\{\overline{\text{var } f : T_f},\ \overline{\text{def } m(\overline{p : T_p}) \rightarrow B_{\text{ret}}}\}$ with a set of refinements R, which are interpreted conjunctively.

We present our approach as a framework parameterized by the language of dependent refinements R needed to specify the invariants of interest. Because these refinements refer to storage locations, they should be parametric with respect to the syntactic class of identifiers I. We decorate with a superscript R^{L}, R^{F}, or R^{S} when we want to emphasize or make clear over which syntactic class of identifiers the refinement ranges: locals x, fields f, or symbolic values $\widetilde{v}$ (see Section 4.3), respectively. We write ι to indicate an identifier when the class of identifier is irrelevent because of parametricity. Because types include refinements, types are parametrized as well—written T^{L}, T^{F}, or T^{S}—as are type environments Γ^{L}, Γ^{F}, Γ^{S}. Unadorned types T and environments Γ are implicitly parameterized over locals.

4.1.2 Concrete State

We describe the concrete state for this core language in Figure 4.2. The concrete state consists of a pair (E, H) of a local variable environment E and a heap H. Concrete environments are maps from local variable identifiers x to values v. Heaps are maps from concrete addresses a to a pair $\langle o, B \rangle$ of an object o and object base type B—in essence, objects are tagged with their concrete type. This type tag has no significance for concrete execution; we refer to it in concretization of type environments (Section 4.2.2) in order to describe type consistency. Objects o are maps from field identifiers f to field values v and also from method names m to method bodies, which are syntactic expressions e. Values v are either object addresses a, strings s, or the unit value $\langle\rangle$.

4.1.3 Concrete Semantics

We provide a big-step operational semantics specifying our concrete execution in Figure 4.3. A judgment of the form $E \vdash [H]\, e\, [r]$ means that in a concrete local environment E and starting with a heap H, executing expression e results in r. Where r can be either a pair $H' \downarrow v$ of a heap H' and value v, or an explicit error, err.

We indicate map lookup with parentheses, so, e.g., $E(x)$ in the E-Var rule indicates the result of looking up the value in E that is bound to the identifier x. We indicate map update with square brackets, so e.g., $o' = o[f : v]$ in the rule E-Write-Field indicates that o' is the result of updating object o with a binding mapping field f to value v.

Many of the rules are standard; we now the describe the non-standard rules. The E-Object-Literal describes execution of object literal expressions. It creates a new object o mapping (1) field names to the result of looking up the initializing variable names in the environment and (2) method names to the method bodies specified in literal. The rule constructs the type of the object, B, and adds a mapping from a fresh address a to the object/tag pair $\langle o, B\rangle$ to the heap. The result of executing the rule is the pair of the new heap H' and the address a. The string append rule (E-Str-Append) looks up the values stored in the two variable names and appends them.

The rule for direct method call, E-Call-Direct, looks up the address stored in the receiver local variable z and then looks up the object o stored at that address and then looks up the expression e (method body) with name m on that object. It then creates a new local variable environment E' mapping the formal parameters p to the actuals. Here, for convenience, we assume that all methods draw from the same sequence of parameter names, so, e.g., the first parameter name is always p_1, the second (if it exists) is always p_2, etc. E' also includes an entry mapping self to the value of the receiver. It then executes the method body e in the E'. The result of executing a direct method call is the resultant heap, H' and value v.

Reflective method call (E-Call-Refl) is the same as E-Call-Direct except that the name of

$$\boxed{E \vdash [H]\, e\, [r]}$$

E-Unit

$$\frac{}{E \vdash [H]\, \langle\rangle\, [H \downarrow \langle\rangle]}$$

E-Str-Literal

$$\frac{}{E \vdash [H]\, c\, [H \downarrow c]}$$

E-Object-Literal

$$\frac{\begin{array}{c} o = \overline{[f : E(x_f)]}\ \overline{[m : e]} \\ B = \{\overline{\text{var } f : T_f,}\ \ \overline{\text{def } m(\overline{p : T_p}) \rightarrow B_{\text{ret}}}\} \qquad a \notin \text{dom}(H) \qquad H' = H[a : \langle o, B\rangle] \end{array}}{E \vdash [H]\, \{\overline{\text{var } f : T_{\text{f}} \ = \ x_f,}\ \ \overline{\text{def } m(\overline{p : T_p}) : B_{\text{ret}} \ = \ e}\}\, [H' \downarrow a]}$$

E-Str-Append

$$\frac{}{E \vdash [H]\, x_1 \mathbin{@} x_2\, [H'' \downarrow E(x_1)E(x_2)]}$$

E-Var

$$\frac{}{E \vdash [H]\, x\, [H \downarrow E(x)]}$$

E-Let

$$\frac{E \vdash [H]\, e_1\, [H' \downarrow v_1] \qquad E[x : v_1] \vdash [H']\, e_2\, [r]}{E \vdash [H]\, \text{let } x : T = e_1 \text{ in } e_2\, [r]}$$

E-Read-Field

$$\frac{\langle o, B\rangle = H(E(x))}{E \vdash [H]\, x.f\, [H \downarrow o(f)]}$$

E-Write-Field

$$\frac{a = E(x) \qquad v = E(y) \qquad \langle o, B\rangle = H(a) \qquad o' = o[f : v] \qquad H' = H[a : \langle o', B\rangle]}{E \vdash [H]\, x.f = y\, [H' \downarrow v]}$$

E-Seq

$$\frac{E \vdash [H]\, e_1\, [H' \downarrow v_1] \qquad E \vdash [H']\, e_2\, [r]}{E \vdash [H]\, e_1 \ ; \ e_2\, [r]}$$

E-Branch-Left

$$\frac{E \vdash [H]\, e_1\, [r]}{E \vdash [H]\, e_1 \mathbin{[]} e_2\, [r]}$$

E-Branch-Right

$$\frac{E \vdash [H]\, e_2\, [r]}{E \vdash [H]\, e_1 \mathbin{[]} e_2\, [r]}$$

E-Call-Direct

$$\frac{\langle o, B\rangle = H(E(z)) \qquad e = o(m) \qquad E' = \overline{p : E(x)}, \text{self} : E(z) \qquad E' \vdash [H]\, e\, [r]}{E \vdash [H]\, z.m(\overline{x})\, [r]}$$

E-Call-Refl

$$\frac{m = E(y) \qquad \langle o, B\rangle = H(E(z)) \qquad e = o(m) \qquad E' = \overline{p : E(x)}, \text{self} : E(z) \qquad E' \vdash [H]\, e\, [r]}{E \vdash [H]\, z.[y](\overline{x})\, [r]}$$

Selected Error Cases

E-Call-Refl-Lookup-Err

$$\frac{m = E(y) \qquad \langle o, B\rangle = H(E(z)) \qquad m \notin \text{dom}(o)}{E \vdash [H]\, z.[y](\overline{x})\, [\mathsf{err}]}$$

E-Write-Field-Deref-Err

$$\frac{v = E(x) \qquad v \notin \text{dom}(H)}{E \vdash [H]\, x.f = y\, [\mathsf{err}]}$$

E-Seq-Err-1

$$\frac{E \vdash [H]\, e_1\, [\mathsf{err}]}{E \vdash [H]\, e_1 \ ; \ e_2\, [\mathsf{err}]}$$

Figure 4.3: Concrete execution

the method is looked up in environment rather than specified directly in the expression.

In addition to the normal rules shown in Figure 4.3, we also have error rules that produce err when none of these rules apply. So, for example, we have the E-Call-Refl-Lookup-Err rule for reflective call. This rule generates an error when the receiver of a reflective call does not have a method with the name specified in the selector.

4.2 A Flow-Insensitive Dependent Refinement Type System

Errors in reflective method calls (caused by, for example, the E-Call-Refl-Lookup-Err rule above) can be statically—if imprecisely—ruled out with a flow-insensitive dependent refinement system, as we described informally in Section 3.1. This section formalizes our flow-insensitive type analysis for checking dependent refinements, including rules for subtyping (Section 4.2.3) and for typing expressions (Section 4.2.5).

4.2.1 Refinement Types For Reflection

To verify reflective call safety, we instantiate Fissile Type Analysis with refinements specific to reflection safety. As we have seen, the key property is the '$\mathsf{respondsTo}\ \iota(\overline{p : T_p}) \rightarrow B_{\mathrm{ret}}$' refinement that says an object must respond to the value named by ι with the given method type. As a dependent type invariant on storage locations, the refinement constrains **both** the storage location on which the refinement is applied and the storage location named by ι. That is, the former must a hold a responder for the selector in the latter. We also need some refinements on string values, such as the union of singletons: '$\mathsf{in}\ \{c_1, \ldots, c_n\}$' which says the value is one of the following (string) literals $c_1, \ldots, c_n$.

4.2.2 Concretization of Types

The concretization of a type typically gives a set of values that are in the type. In our case, however, types also imply constraints on ancillary structures. For example, the concretization of an object type constrains the heap so that the fields of that object are appropriate. Similarly, the concretization of a local type constrains the local variable environment, and the concretization of a dependent field type constrains the parent object of the field.

Type Environments. The concretization of a type environment is a set of pairs of concrete environments and concrete heaps for which 1) the values stored in local variables conform to the requirements of the local types in the type environment and 2) the objects stored at all addresses in the heap conform to the requirements of the type for that address.

$$\gamma : \mathsf{TypeEnvironments} \rightarrow \mathcal{P}(\mathsf{Environments} \times \mathsf{Heaps})$$

$$\gamma(\Gamma) \triangleq \left\{ (E, H) \;\middle|\; \begin{array}{l} \text{for all } x : T^{\mathrm{L}} \text{ in } \Gamma \text{ exists a } v \text{ where} \\ \qquad (E, H, v) \in \gamma(T^{\mathrm{L}}) \text{ and } E(x) = v \\[1ex] \text{and for all } a : \langle o, B \rangle \text{ in } H \\ \qquad (H, a) \in \gamma(B) \end{array} \right\}$$

Local Types. Local types are adorned with local refinements that may imply a relationship between a value of that type and the local variables, so the concretization of a local type puts restrictions on concrete environments in addition to requiring that the value be a member of the base type. If the base type is an object type, the value is an address which must point to an object on the heap, so the concretization constrains potential heaps as well.

$$\gamma : \mathsf{Types}^{\mathrm{L}} \rightarrow \mathcal{P}(\mathsf{Environments} \times \mathsf{Heaps} \times \mathsf{Values})$$

$$\gamma(B \upharpoonright R^{\mathrm{L}}_1, \cdots, R^{\mathrm{L}}_n) \triangleq \left\{ (E, H, v) \;\middle|\; \begin{array}{l} (H, v) \in \gamma(B) \text{ and} \\[1ex] \text{for all } i \text{ where } 1 \leq i \leq n \\ \qquad (E, H, v) \in \gamma(R^{\mathrm{L}}_i) \end{array} \right\}$$

Local Refinements Local refinements constrain values (as well as the concrete environment and heap) independently of base types. The concretization of a local 'in' refinement is straightforward: it restricts the values to be one of the explicitly allowed string constants but imposes no constraints on the environment or heap. The concretization of a local '$\mathsf{respondsTo}$' refinement requires the values to be addresses with the additional constraint that the object on the heap at that address has a method with the name of the string value stored in the local variable referred to in the refinement. It further requires that the method is in the concretization of the method type signature specified by the refinements for a receiver of type B.

$\gamma : \mathsf{Refinements}^{\mathrm{L}} \rightarrow \mathcal{P}(\mathsf{Environments} \times \mathsf{Heaps} \times \mathsf{Values})$

$$\gamma(\mathsf{in}\ (c_1, \ldots, c_n)) \triangleq \left\{(E, H, s) \;\middle|\; s \in \{c_1, \ldots, c_n\} \;\right\}$$

$$\gamma(\mathsf{respondsTo}\ x(\overline{p : T_p}) \rightarrow B_{\mathrm{ret}}) \triangleq \left\{ (E, H, a) \;\middle|\; \begin{array}{l} \text{exists string } m \text{, object } o \\ \text{and base type } B \text{ where} \\ m = E(x) \\ H(a) = \langle o, B \rangle \text{ and} \\ o(m) \in \gamma(B, (\overline{p : T_p}) \rightarrow B_{\mathrm{ret}}) \end{array} \right\}$$

Base Types. Base types have no refinements (although if a base type is an object type its **fields** may have field refinements). The concretization of a base type is a set of heap-value pairs where the heap may be constrained for object types. The concretizations of Str and Unit is straightforward. For object types, the value must be an address that points to an object that has the required fields and methods. The presence of the required fields is enforced by forcing the object to be in the concretization of the field environment (the type environment mapping field names to field types). The concretization also requires the object to have methods that match their declared type signatures for a receiver of the type B.

$\gamma : \mathsf{BaseTypes} \rightarrow \mathcal{P}(\mathsf{Heaps} \times \mathsf{Values})$

$$\gamma(\mathrm{Str}) \triangleq \left\{(H, v) \;\middle|\; v \text{ is a string } \right\}$$

$$\gamma(\mathrm{Unit}) \triangleq \left\{(H, v) \;\middle|\; v \text{ is } \langle\rangle \; \right\}$$

The concretization of an object type $B = \{\overline{\text{var } f : T_f},\ \overline{\text{def } m(\overline{p : T_p}) \rightarrow B_{\mathrm{ret}}}\}$ is:

$$\gamma(B) \triangleq \left\{ (H, a) \;\middle|\; \begin{array}{l} \text{exists } o \text{ where} \\ \quad H(a) = \langle o, B \rangle \text{ and} \\ \quad o(m) \in \gamma(B, (\overline{p : T_p}) \rightarrow B_{\mathrm{ret}}) \text{ for all methods } m \text{ and} \\ \quad (H, o) \in \gamma(\overline{f : T_f}). \end{array} \right\}$$

As we will see, the concretization of method signatures does not concretize its arguments, so the definition is well-founded.

Field Types. Field types, like local types, consist of a base type B and a sequence of refinements R_i^{F}. However, for field types the refinements refer to **field locations** on the parent object rather than local variable locations—that is, field types can restrict the values stored in other fields of their containing objects. The concretization of a field type reflects this: it limits the possible objects o to those permitted by the field refinements.

$$\gamma : \mathsf{Types}^{\mathrm{F}} \rightarrow \mathcal{P}(\mathsf{Heaps} \times \mathsf{Objects} \times \mathsf{Values})$$

$$\gamma(B \restriction R_1^{\mathrm{F}}, \cdots, R_n^{\mathrm{F}}) \triangleq \left\{ (H, o, v) \;\middle|\; \begin{array}{c} (H, v) \in \gamma(B) \text{ and} \\ \text{for all } i \text{ where } 1 \leq i \leq n \\ (H, o, v) \in \gamma(R_i^{\mathrm{F}}) \end{array} \right\}$$

Field Refinements. Field refinements are analogous to local refinements except that they impose requirements on the object containing the field rather than the local variables. The concretization of an 'in' refinement is essentially as in the local case. The concretization of '$\mathsf{respondsTo}\ f(\overline{p : T_p}) \rightarrow B_{\mathrm{ret}}$' relates **two** objects: o the object on which the concrete value is stored as a field and o' the object whose address is the concrete value. We require o' to have a method with the name of whatever string is stored in $o(f)$ and force that method to be in the concretization of the method type signature specified in the refinement.

$$\gamma : \mathsf{Refinements}^{\mathrm{F}} \rightarrow \mathcal{P}(\mathsf{Heaps} \times \mathsf{Objects} \times \mathsf{Values})$$

$$\gamma(\mathsf{in}\ (c_1, \ldots, c_n)) \triangleq \left\{ (H, o, s) \;\middle|\; s \in \{c_1, \ldots, c_n\} \right\}$$

$$\gamma(\mathsf{respondsTo}\ f(\overline{p : T_p}) \rightarrow B_{\mathrm{ret}}) \triangleq \left\{ (H, o, a) \;\middle|\; \begin{array}{l} \text{exists string } m, \text{ object } o' \\ \text{and base type } B \text{ where} \\ m = o(f) \\ H(a) = \langle o', B \rangle \text{ and} \\ o'(m) \in \gamma(B, (\overline{p : T_p}) \rightarrow B_{\mathrm{ret}}) \end{array} \right\}$$

Field Type Environments. Field type environments Γ^{F} are analogous to local type environments, except that they constrain the values that can be stored into the fields of an object rather than local variables.

$$\gamma : \mathsf{TypeEnvironments}^{\mathrm{F}} \rightarrow \mathcal{P}(\mathsf{Heaps} \times \mathsf{Objects})$$

$$\gamma(\Gamma^{\mathrm{F}}) \triangleq \left\{ (H, o) \;\middle|\; \begin{array}{l} \text{for all } f : T^{\mathrm{F}} \text{ in } \Gamma^{\mathrm{F}} \\ \quad (H, o, v) \in \gamma(T^{\mathrm{F}}) \text{ and } o(f) = v \end{array} \right\}$$

$$\boxed{\Gamma^{\mathrm{I}} \vdash T_1^{\mathrm{I}} <: T_2^{\mathrm{I}}}$$

$$\textsc{Sub-Component}\quad \frac{\forall_{i\in 1\ldots m}[\exists_{S\subseteq\{R_1^{\mathrm{I}},\ldots,R_n^{\mathrm{I}}\}}\ \Gamma^{\mathrm{I}} \vdash B \upharpoonright S <: B \upharpoonright Q_i]}{\Gamma^{\mathrm{I}} \vdash B \upharpoonright R_1^{\mathrm{I}},\ldots,R_n^{\mathrm{I}} <: B \upharpoonright Q_1^{\mathrm{I}},\ldots,Q_m^{\mathrm{I}}}$$

$$\textsc{Sub-Weaken}\quad \frac{\{Q_1^{\mathrm{I}},\ldots,Q_m^{\mathrm{I}}\} \subseteq \{R_1^{\mathrm{I}},\ldots,R_n^{\mathrm{I}}\}}{\Gamma^{\mathrm{I}} \vdash B \upharpoonright R_1^{\mathrm{I}},\ldots,R_n^{\mathrm{I}} <: B \upharpoonright Q_1^{\mathrm{I}},\ldots,Q_m^{\mathrm{I}}}$$

$$\textsc{Sub-Str-In-Refl}\quad \frac{\{c_1,\ldots,c_m\} \subseteq \{k_1,\ldots,k_n\}}{\Gamma^{\mathrm{I}} \vdash \mathrm{Str} \upharpoonright \mathsf{in}\,\{c_1,\ldots,c_m\} <: \mathrm{Str} \upharpoonright \mathsf{in}\,\{k_1,\ldots,k_n\}}$$

$$\textsc{Sub-Obj-RespondsTo-Refl}\quad \frac{\Gamma^{\mathrm{I}} \vdash \Gamma^{\mathrm{I}}(x) <: \mathrm{Str} \upharpoonright \mathsf{in}\,\{c_1,\ldots,c_n\} \qquad \forall_{i\in 1..n}\ B \text{ has a method named } c_i \text{ with signature } (\overline{p:T_p}){\to}B_{\mathrm{ret}}}{\Gamma^{\mathrm{I}} \vdash B \upharpoonright \cdots <: B \upharpoonright \mathsf{respondsTo}\ \iota(\overline{p:T_p}){\to}B_{\mathrm{ret}}}$$

$$\textsc{Sub-Refl}\quad \frac{}{\Gamma^{\mathrm{I}} \vdash T^{\mathrm{I}} <: T^{\mathrm{I}}} \qquad\qquad \textsc{Sub-Trans}\quad \frac{\Gamma^{\mathrm{I}} \vdash T_1^{\mathrm{I}} <: T_2^{\mathrm{I}} \qquad \Gamma^{\mathrm{I}} \vdash T_2^{\mathrm{I}} <: T_3^{\mathrm{I}}}{\Gamma^{\mathrm{I}} \vdash T_1^{\mathrm{I}} <: T_3^{\mathrm{I}}}$$

Figure 4.4: Subtyping of refinement types used for verifying reflective call safety.

Method Type Signatures. A method type signature $(\overline{p:T_p}){\to}B_{\mathrm{ret}}$ consists of a list of pairs of parameter names p with types T_p (i.e., a local type environment) and a return type $B_{\mathrm{ret}} \upharpoonright$. The concretization of a method signature in combination with a base type B yields the set of method bodies (expressions) that type check in a type environment corresponding to the parameters of the signature extended with a binding for self of base type B lifted to a local type with no refinements.

$$\gamma : \mathsf{BaseTypes} \times \mathsf{TypeEnvironments}^{\mathrm{L}} \times \mathsf{Types}^{\mathrm{L}} \to \mathcal{P}(\mathsf{Expressions})$$

$$\gamma(B, (\overline{p:T_p}){\to}B_{\mathrm{ret}}) \triangleq \left\{ e \;\middle|\; \overline{p:T_p}, \mathrm{self} : B \upharpoonright\ \vdash e : B_{\mathrm{ret}} \upharpoonright \right\}$$

4.2.3 Subtyping with Refinements

In Figure 4.4, we give a simple syntactic subtyping for the refinements for our example client: verifying reflective call safety. As discussed in Section 4.2.1, the language of refinements and the subtyping procedure are simply parameters to the framework. What is required is these components are parametrized, in turn, by a class I of identifiers ι and that the subtyping procedure is sound with respect to concrete inclusion. We emphasize the first point in the above by showing judgment forms indexed by the class of identifiers I.

Subtyping of dependent types is always with respect to an environment—the rule to introduce a refinement may refer to the environment (and thus the types of other variables) in order to establish a relationship with that variable. The Sub-Component rule permits component-wise refinement subtyping. It says that a dependent type $B_1 \upharpoonright R_1^{\mathrm{I}}, \ldots, R_n^{\mathrm{I}}$ is a subtype of another dependent type $B_2 \upharpoonright Q_1^{\mathrm{I}}, \ldots, Q_m^{\mathrm{I}}$ if the base types match and for each refinement Q_i^{I} on the super type, there is some set of refinements S from the subtype that are sufficient to derive it. Rule Sub-Weaken says that a type can always be weakened by dropping refinements. Sub-Str-In-Refl says that an in refinement constraining a value to be one in a set of string constants can be weakened by adding additional constants to the set. The Sub-Obj-RespondsTo-Refl introduces the respondsTo refinement by combining information from base object types and the environment to introduce respondsTo. This rule says that for any location ι that is one of a set of selector strings $c_1, \ldots, c_n$, then any object of base type B with methods of the appropriate signature for all $c_1, \ldots, c_n$ responds to the method named by ι with that signature. Finally, Sub-Refl and Sub-Trans say that subtyping is reflexive and transitive. For our purposes, it does not matter how subtyping is checked as long as it is a sound approximation of semantic subtyping. We could, for example, use an SMT solver as in Liquid Types [90] and also replace the type rules for string operations with an off-the-shelf string solver to make the abstract semantics more precise.

Soundness of Subtyping Here we show that subtyping is a sound over-approximation of inclusion under the appropriate concretization. Note the two different forms: although we can represent the subtyping judgment for difference classes of types with a single, parametrized system of inference rules, the **meaning** of the instantiated systems are different for locals and fields.

Lemma 1 (Soundness of Subtyping):

(1) If $\Gamma^{\mathrm{L}} \vdash T_1^{\mathrm{L}} <: T_2^{\mathrm{L}}$ *and* $(E, H, v) \in \gamma(T_1^{\mathrm{L}})$ *where* $(E, H) \in \gamma(\Gamma^{\mathrm{L}})$ *then* $(E, H, v) \in \gamma(T_2^{\mathrm{L}})$*; and*

(2) If $\Gamma^{\mathrm{F}} \vdash T_1^{\mathrm{F}} <: T_2^{\mathrm{F}}$ *and* $(H, o, v) \in \gamma(T_1^{\mathrm{F}})$ *where* $(H, o) \in \gamma(\Gamma^{\mathrm{F}})$ *then* $(H, o, v) \in \gamma(T_2^{\mathrm{F}})$*,*

Proof. In each case, by induction on the derivation of the subtyping relation. □

.

We will need a similar result for the subtyping in the symbolic analysis. (See Lemma 2 in Section 4.3.2 for details.)

4.2.4 Identifier Substitution

Throughout this work we treat various maps as substitutions between classes of identifiers. Here, we formally describe the effect of these substitutions on dependent refinements. We have three classes of identifiers: local variable names (denoted with x, y, and z), field names (f, g) and symbolic variables ($\widetilde{x}$, $\widetilde{y}$, $\widetilde{a}$). (For a description of this latter class, see Section 4.3.1.) As before, we use I to indicate a definition that is parametric in the identifier class and ι to indicate an indicate an indentifier in that parametric class.

We denote maps from I to I to with θ. We denote maps from either locals or fields to symbolic values with $\tilde{\theta}$ and rely on context to disambiguate. Abusing notation, we denote maps from symbolic values to either locals or fields (again, disambiguated by context) as $\tilde{\theta}^{-1}$. Note that when we use a substitution from symbolic values to identifiers, we always require the substitution to be 1-1 and thus invertible.

We define identifier substitution under a map by:

$$B \restriction R_1^{\mathrm{I}}, \cdots, R_n^{\mathrm{I}}\ [\theta] \triangleq B \restriction R_1^{\mathrm{I}}\ [\theta], \cdots, R_n^{\mathrm{I}}\ [\theta]$$

where

$$\mathsf{in}\ (c_1, \ldots, c_n)\ [\theta] \triangleq \mathsf{in}\ (c_1, \ldots, c_n)$$

and

$$\mathsf{respondsTo}\ \iota(\overline{p : T_p}) \rightarrow B_{\mathrm{ret}}\ [\theta] \triangleq \mathsf{respondsTo}\ \theta(\iota)(\overline{p : T_p}) \rightarrow B_{\mathrm{ret}}$$

We write $\Gamma_1^{\mathrm{I}} <:_\theta \Gamma_2^{\mathrm{I}}$ (Figure 4.5) to mean that a type environment Γ_1^{I} is a subenvironment of an environment Γ_2^{I} under the identifier substitution θ. We define this formally in Sub-Env. Here $T^{\mathrm{I}}[\theta]$ means the type T^{I} but with all variable references in $T^{\mathrm{I}}[\theta]$'s refinements replaced with their substitute in θ. We use this style of substitution for both Hoare-style weakest precondition

$$\boxed{\Gamma^{\mathrm{I}}_1 <:_\theta \Gamma^{\mathrm{I}}_2}$$

$$\textsc{Sub-Env}\quad \frac{\Gamma^{\mathrm{I}}_1 \vdash \Gamma^{\mathrm{I}}_1(\theta(\iota)) <: T^{\mathrm{I}}[\theta] \qquad \text{for all } \iota : T^{\mathrm{I}} \in \Gamma^{\mathrm{I}}_2}{\Gamma <:_\theta \Gamma'}$$

Figure 4.5: Subenvironments under substitution.

calculations (as described in Section 3.1 and Section 4.2.5) and for conversion of invariants between type and symbolic domains (Section 4.4).

4.2.5 Expression Typing

Figure 4.6 defines a fairly standard flow-insensitive typing of expressions. A judgment of the form $\Gamma \vdash e : T$ says that an expression e has type T in a typing environment Γ, where Γ is a finite map from variable identifiers to types, which we view as the types assigned to program variables (i.e., $\Gamma^{\mathrm{L}} \vdash e : T^{\mathrm{L}}$ for emphasis). The standard typing judgment form demonstrates that Γ is a flow-insensitive invariant.

Figure 4.6a gives the standard typing rules for control structures, locals, and subsumption. The usual T-Seq and T-Branch rules show that the type system is completely flow-insensitive. The T-Read-Local rule is also standard, as is the rule for introducing new let-bound immutable local variables. For simplicity, we introduce subtyping only in let-bindings and assume our language is an intermediate form that has made explicit where to apply subsumption. From a soundness point of view, there is no difficulty with using the standard subsumption rule instead.

In Figure 4.6b, we give rules specific for the ‘in’ refinement used for reflection safety. These rules simply capture an abstract semantics of the string operations for the ‘in’ refinement. Rule T-String-Lit-Refl says that a string literal c is the singleton c, and rule T-Append-String-Refl says if we know that x_1 and x_2 both correspond to a set of possible string constants, then the operation yields the append of all pairs. We elide the analogous rule for string append where there are no refinements, as we assume that is part of the underlying base type system.

$$\boxed{\Gamma \vdash e : T}$$

$$\textsc{T-Seq}\quad \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 \,;\, e_2 : T_2} \qquad \textsc{T-Branch}\quad \frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 \mathbin{[]} e_2 : T} \qquad \textsc{T-Read-Local}\quad \frac{}{\Gamma, x : T \vdash x : T}$$

$$\textsc{T-Let}\quad \frac{\begin{array}{c}\Gamma \vdash e_1 : T_1 \\ \Gamma \vdash T_1 <: T_1' \qquad \Gamma, x : T_1' \vdash e_2 : T_2 \qquad x \notin \text{dom}(\Gamma) \qquad x \notin \text{fv}(T_1') \qquad x \notin \text{fv}(T_2) \qquad x \notin \text{fv}(\Gamma)\end{array}}{\Gamma \vdash \text{let } x : T_1' = e_1 \text{ in } e_2 : T_2}$$

(a) Standard typing for control structures and locals.

$$\textsc{T-String-Lit-Refl}\quad \frac{}{\Gamma \vdash c : \text{Str} \upharpoonleft \mathsf{in}\ \{c\}} \qquad \textsc{T-Append-String-Refl}\quad \frac{\Gamma(x_1) = \text{Str} \upharpoonleft \mathsf{in}\ \{c_1 \ldots, c_n\} \qquad \Gamma(x_2) = \text{Str} \upharpoonleft \mathsf{in}\ \{k_1 \ldots, k_m\}}{\Gamma \vdash x_1 \mathbin{@} x_2 : \text{Str} \upharpoonleft \mathsf{in}\ \{c_1 k_1, \ldots, c_n k_1, \ldots, c_n k_m\}}$$

(b) Refinement typing specific to reflection safety.

$$\textsc{T-Write-Field}\quad \frac{T = \Gamma(x) \qquad \Gamma, \text{fieldtypes}(T) <:_{[f:y]} \text{fieldtypes}(T)}{\Gamma \vdash x.f := y : \text{Unit}} \qquad \textsc{T-Read-Field}\quad \frac{\Gamma(x) = T_\text{x}}{\Gamma \vdash x.f : T_\text{x}.f\upharpoonleft}$$

$$\textsc{T-Method-Call}\quad \frac{\Gamma(z) = \{\cdots, \text{def } m(\overline{p : T_p}) \to B_\text{ret}, \cdots\} \qquad \Gamma <:_{[\overline{p:x}, \text{self}:z]} \overline{p : T_p}, \text{self} : \Gamma(z)}{\Gamma \vdash z.m(\overline{x}) : B_\text{ret} \upharpoonleft}$$

$$\textsc{T-Object-Literal}\quad \frac{\begin{array}{c}\Gamma <:_{[\overline{f:x_f}]} \overline{f : T_f} \\ \overline{p : T_p}, \text{self} : B \upharpoonleft \vdash e : B_\text{ret} \upharpoonleft \qquad \text{for all methods} \qquad B = \{\overline{\text{var } f : T_f},\ \overline{\text{def } m(\overline{p : T_p}) \to B_\text{ret}}\}\end{array}}{\Gamma \vdash \{\overline{\text{var } f : T_\text{f} = x_f},\ \overline{\text{def } m(\overline{p : T_p}) : B_\text{ret} = e}\} : B \upharpoonleft}$$

(c) Non-standard rules for flow-insensitive dependent types.

$$\textsc{T-Reflective-Method-Call}\quad \frac{\Gamma(z) = \{\cdots\} \upharpoonleft \mathsf{respondsTo}\ y(\overline{p : T_p}) \to B_\text{ret} \qquad \Gamma <:_{[\overline{p:x}, \text{self}:z]} \overline{p : T_p}, \text{self} : \Gamma(z)}{\Gamma \vdash z.[y](\overline{x}) : B_\text{ret} \upharpoonleft}$$

(d) Reflective method calls.

Figure 4.6: Typing of expressions with refinement relationships between storage locations.

Dependent refinements complicate flow-insensitive checking of some language features. We describe checking these features in Figure 4.6c in a way similar to Deputy [27], in which the effect of an assignment is essentially interpreted using Hoare's backward rule for assignment—although we have added subtyping. The T-Write-Field rule checks the refinement relationship preservation condition for a write to a field. In particular, we need to verify that the write to field f of x does not invalidate the relationships of the object fields stored in x. To do so, we rearrange the pre-state by conceptually bringing the fields of x into scope—"locally-naming heap locations." This rearrangement yields the check

$$\Gamma, \mathrm{fieldtypes}(T) \;<:_{[f:y]}\; \mathrm{fieldtypes}(T)$$

where fieldtypes(T) yields an environment consisting of the field declarations of an object type T:

$$\mathrm{fieldtypes}(\{\mathrm{var}\ f : T_{\mathrm{f}}, \ldots\} \;\upharpoonright\; \ldots) \stackrel{\mathrm{def}}{=} f : T_{\mathrm{f}}, \ldots \ \text{for each field}\ f\ .$$

We walked through an example of applying this rule in Section 3.1.

A field read given by T-Read-Field is relatively straightforward. We essentially yield the type of the f field of T_{x}, which we write as $T_{\mathrm{x}}.f$. However, we drop any relationship refinements with other fields, as they would be ill-formed in Γ. We write this dropping of relationship refinements as $T{\upharpoonright}$. Direct method call with the T-Method-Call rule is quite similar to write, we check that the pre-state over the actual arguments $\overline{x}$ conforms to the types of the formal parameters $\overline{p}$. The return type of a method must be a base type, and we show that it has no refinements when used as a type. The T-Object-Literal describes type checking object literals and thus modular analysis of methods. Object literals (or anonymous objects) consist of two components: a set of field declarations and a set of method declarations. A field declaration var $f : T_f \;=\; x_f$ declares a field f of type T_f and sets its initial value to that stored in local variable x_f. Our treatment of field initializers is analogous to T-Write-Field except that all initializers are considered simultaneously. Essentially, we promote the fields in the object to local variables and check that the values of the initializers in the local environment meet the requirements of the refinements on the field storage locations. A method declaration def $m(\overline{p : T_p}) : B_{\text{ret}} \;=\; e$ specifies a method called m which takes a sequence of parameters $\overline{p}$ with types $\overline{T_p}$ and has return type B_{ret} and body e. Our checking of method bodies is standard: we check the method body in a type environment with the parameter types and a special variable self representing the receiver.

The T-Reflective-Method-Call rule (Figure 4.6d) demonstrates checking of reflection safety. We require that the responder object z has a refinement guaranteeing that it responds to the selector y with method type signature $(\overline{p : T_p}) \to B_{\text{ret}}$. The arguments to the call are checked against the specified types of the parameters via $\Gamma \;<:_{[\overline{p{:}x},\text{self}:z]}\; \overline{p : T_p}, \text{self} : \Gamma(z)$. We write $\Gamma \;<:_{\theta}\; \Gamma'$ as the lifting of subtyping to type environments under a substitution θ from variables on the right to variables on the left. The type of the call is then the return base type of the method (as expected) without any refinements $B_{\text{ret}} \restriction$.

Another Instantiation: Typing for array refinements. Fissile Type Analysis is applicable to more than just reflection safety, because it is parametrized by the language of refinements and a decision procedure for over-approximating semantic inclusion. To provide context for our approach, we sketch another instantiation that checks array-bounds safety—a property considered by many prior works—with a few refinements and rules. Suppose we augment our programming language to include array allocation and array access and add two refinements: (1) $\mathsf{hasLength}$, which indicates that an array has the specified (non-zero) length and (2) $\mathsf{indexedBy}$, which indicates that the specified index is a valid index for the array—that is, that the index is in bounds. When analyzing an array access $e[x]$, we check that x is a valid index into the array e by requiring that e's type has the refinement $\mathsf{indexedBy}\ x$. We could introduce this refinement via subtyping with the following rule, which says that x is a valid index for an array of length y if the environment Γ restricts x and y such that $x \geq 0$ and $x < y$:

$$\frac{[\![\Gamma]\!] \vdash_{\text{SMT}} x \geq 0 \wedge x < y}{\Gamma \vdash B \restriction \mathsf{hasLength}\ y \;<:\; B \restriction \mathsf{indexedBy}\ x}$$

We could verify that this condition holds by encoding the environment into a linear arithmetic formula (written $[\![\Gamma]\!]$) and checking entailment with an SMT solver (written as the judgment

$$\begin{array}{lcll}
\widetilde{\Sigma} & ::= & (\widetilde{\Gamma}, \widetilde{H}) & \text{symbolic states} \\
\widetilde{E} & ::= & \cdot \mid \widetilde{E}[x : \widetilde{v}] & \text{symbolic environments} \\
\widetilde{H} & ::= & \mathsf{emp} \mid \widetilde{a} : \tilde{o} \mid \widetilde{H}_1 * \widetilde{H}_2 \mid \mathsf{ok} & \text{symbolic heaps} \\
\widetilde{\Gamma} & ::= & \cdot \mid \widetilde{\Gamma}[\widetilde{v} : T^{\mathrm{S}}] & \text{symbolic facts} \\
\widetilde{P} & ::= & \widetilde{\Sigma} \downarrow \widetilde{v} \mid \widetilde{P}_1 \vee \widetilde{P}_2 \mid \mathsf{false} & \text{symbolic paths} \\
\tilde{o} & ::= & \mathsf{emp} \mid f : \widetilde{v} \mid \tilde{o}_1 * \tilde{o}_2 & \text{symbolic objects} \\
\multicolumn{3}{l}{\widetilde{v}, \widetilde{a}, \widetilde{x}, \widetilde{y}, \widetilde{z}} & \text{symbolic values}
\end{array}$$

Figure 4.7: The symbolic analysis state splits type environments into types lifted to values and the locations where values are stored.

$\phi_1 \vdash_{\mathrm{SMT}} \phi_2$ for formulas ϕ_1 and ϕ_2). Here we map meta-variables x and y in the typing judgment to logical variables of the same name in the SMT entailment checking judgment. In general, it should be possible to take any symbolic relational domain (e.g., [33, 34, 63, 75]) and lift it to a refinement, as long as there both a means for statically overapproximating inclusion and a mechanism to establish the relationship (either by inspecting the type environment flow-insensitively or as the result of a sound symbolic analysis, such as abstract interpretation [31]).

4.3 Symbolic Analysis

The type analysis described in the previous section is efficient but coarse. It is flow-insensitive—constraining all storage locations to be a fixed type and the heap to be always in a consistent state. When these constraints hold, we get a simple and fast analysis. When they are temporarily violated, our overall analysis can switch (as described in Chapter 3) to a path-sensitive symbolic analysis. We now describe this symbolic analysis, including its abstract state (Section 4.3.1), concretization (Section 4.3.2), and static semantics (Section 4.3.3). This section discusses the symbolic analysis in isolation; we describe handoff between types and the symbolic analysis, as well as leveraging the almost type-consistent heap during symbolic analysis, in Section 4.4.

4.3.1 Symbolic State

In the symbolic analysis, we split (hence "Fissile" Type Analysis) a type environment Γ into a symbolic environment $\widetilde{E}$ and a symbolic state $\widetilde{\Sigma}$ (Figure 4.7). A symbolic environment $\widetilde{E}$

provides variable context: it maps variables to symbolic values $\widetilde{v}$ that represent their values. A symbolic state $\widetilde{\Sigma}$ consists of two components: a symbolic fact context $\widetilde{\Gamma}$, mapping symbolic values to the facts (symbolic types) known about them and a symbolic heap $\widetilde{H}$. A symbolic heap $\widetilde{H}$ contains a partially-materialized sub-heap that maps addresses ($\widetilde{a}$) to symbolic objects ($\tilde{o}$), which are themselves maps from field names (f) to symbolic values. We write symbolic objects and heaps using the separating conjunction $*$ notation borrowed from separation logic [88] to state that we refer to disjoint storage locations. (As we showed in Section 3.3, type-intertwined symbolic analysis cannot soundly support framing with $*$, a key benefit of separating conjunction in traditional separation logic. We describe gated separating conjunction, a strengthening of $*$ that does allow type-intertwined framing, in Chapter 5.)

Symbolic values $\widetilde{v}$ correspond to existential, logic variables. For clarity, we often use $\widetilde{a}$ to express a symbolic value that is an address and similarly use $\widetilde{x}, \widetilde{y}, \widetilde{z}$ for values stored in the corresponding program variables x, y, z. Relationship refinements in $\widetilde{\Gamma}$ are expressed in terms of types lifted to symbolic values (T^{S})—that is, the refinements state relationship facts between values and not storage locations (like the refinements in Γ for typing). Our overall analysis state is a symbolic path set $\widetilde{P}$, which is a disjunctive set of singleton paths $\widetilde{\Sigma} \downarrow \widetilde{x}$. A singleton path is a pair of a symbolic state and a symbolic value corresponding to the return state and value, respectively.

The symbolic heap $\widetilde{H}$ enables treating heap locations much like stack locations, capturing relationships in the symbolic context $\widetilde{\Gamma}$, though certainly more care is required with the heap due to aliasing. A symbolic heap $\widetilde{H}$ can be empty emp, a single materialized object $\widetilde{a} : \tilde{o}$ with base address $\widetilde{a}$ and fields given by $\tilde{o}$, or a separating conjunction of sub-heaps $\widetilde{H}_1 * \widetilde{H}_2$. Lastly and most importantly, a sub-heap can be ok, which represents an arbitrary but **almost type-consistent heap**. This formula essentially grants permission to materialize from the almost type-consistent heap and, as discussed in Section 3.2, is the key mechanism for soundly transitioning between the type and symbolic analyses.

4.3.2 Concretization of Symbolic State

In this section we formally describe the meaning of symbolic state in terms of a concretization function to sets of concrete states (recall we described the concrete state in Section 4.1.2). Throughout this section, as in the concretization in the types domain (Section 4.2.2) we overload γ to mean concretization of various abstract components. In some cases, however (e.g., for base types B), the same abstract component will have a **different** concretization in the types domain than in the symbolic domain. In these cases we write $\widetilde{\gamma}$ to mean concretization in the symbolic domain.

Symbolic Paths. The concretization of a single symbolic path $(\widetilde{\Gamma}, \widetilde{H}) \downarrow \widetilde{v}$ given a symbolic environment yields a set of triples $(E,\, H,\, v)$ of a concrete environment E, a concrete heap H, and a concrete value v. In order for this triple to be in the concretization of the symbolic path, there must exist a valuation V mapping symbolic values to concrete values (essentially giving meaning to the concrete values) and a splitting of the heap into two components: the type-consistent heap H^{ok} and the materialized heap H^{mat} (here we write $H_1 * H_2$ to mean the disjoint combination of two concrete heaps). The concretization of a symbolic path forces (1) the valuation and the concrete environment to agree with the concretization of the symbolic environment $\widetilde{E}$; (2) the valuation and the two heaps to agree with both the concretization of the symbolic heap and the concretization of the symbolic fact map; and (3) the valuation of the symbolic result of the path must be the concrete value in the triple. In many ways, the valuation in the concretization of a symbolic domain is similar to the environment in the concretization of type environments from the type domain, since the concretization of the symbolic facts T^{S} in $\widetilde{\Gamma}$ constrains the valuation similarly to how the concretization of types T constrains an environment E in the types domain.

The concretization of a conjunction of symbolic paths $\widetilde{P}_1 \vee \widetilde{P}_2$ yields the union of the concretizations of the constituent subpaths. Note that this means that each subpath gets its own valuation. The concretization of false—a path that is not reachable—is empty.

$$\gamma : (\mathsf{SymEnv} \times \mathsf{SymPath}) \to \mathcal{P}(\mathsf{Envs} \times \mathsf{Heaps} \times \mathsf{Values})$$

$$\gamma(\widetilde{E}, (\widetilde{\Gamma}, \widetilde{H}) \downarrow \widetilde{v}) \triangleq \left\{ (E, H, v) \;\middle|\; \begin{array}{l} \text{Exists valuation } V : \mathsf{SymVal} \rightarrow \mathsf{Val} \text{ and} \\ \text{heaps } H^{\text{ok}},\ H^{\text{mat}} \text{ where} \\ \quad (V, E) \in \gamma(\widetilde{E}) \text{ and} \\ \quad (V, H^{\text{ok}}, H^{\text{mat}}) \in \gamma(\widetilde{H}) \cap \gamma(\widetilde{\Gamma}) \text{ and} \\ \quad H = H^{\text{ok}} \ast H^{\text{mat}} \text{ and} \\ \quad V(\widetilde{v}) = v \end{array} \right\}$$

$$\gamma(\widetilde{E}, \widetilde{P}_1 \vee \widetilde{P}_2) \triangleq \gamma(\widetilde{E}, \widetilde{P}_1) \cup \gamma(\widetilde{E}, \widetilde{P}_2)$$

$$\gamma(\widetilde{E}, \mathsf{false}) \triangleq \{\}$$

Symbolic States. A symbolic state consists of the state elements of a symbolic path but does not have a return value, so the concretization of a symbolic state $\widetilde{\Sigma}$ with respect to a symbolic environment $\widetilde{E}$ is a set of pairs (E, H) of concrete environments and heaps. The concretization of a disjunction of symbolic states (produced, for example, as the result of materialization) consists of the union of concretizations of the constituent states.

$\gamma : (\mathsf{SymEnv} \times \mathsf{SymState}) \rightarrow \mathcal{P}(\mathsf{Env} \times \mathsf{Heap})$

$$\gamma(\widetilde{E}, \widetilde{\Sigma}) \triangleq \left\{ (E, H) \;\middle|\; \text{Exists } \widetilde{x} \text{ where } (E, H, v) \in \gamma(\widetilde{\Sigma} \downarrow \widetilde{x}) \right\}$$

$$\gamma(\widetilde{E}, \widetilde{\Sigma}_1 \vee \widetilde{\Sigma}_2) \triangleq \gamma(\widetilde{E}, \widetilde{\Sigma}_1) \cup \gamma(\widetilde{E}, \widetilde{\Sigma}_2)$$

Symbolic Environment. The concretization of a symbolic environment yields a valuation and a concrete environment where the values stored in the concrete environment must agree with the valuation of the symbolic values stored in the symbolic environment. $\gamma : \mathsf{SymEnv} \rightarrow \mathcal{P}(\mathsf{Valuation} \times \mathsf{Env})$

$$\gamma(\widetilde{E}) \triangleq \left\{ (V, E) \;\middle|\; V(\widetilde{x}) = E(x) \text{ for all } x : \widetilde{x} \text{ in } \widetilde{E} \right\}$$

Symbolic Heaps. The concretization of a symbolic heap gives structure to the splitting of a heap into H^{ok} and H^{mat} and constraints the valuation of the symbolic values (both addresses and field values) specified on the symbolic heap to match the corresponding concrete values stored in the concrete heaps. The concretization of emp forces both H^{ok} and H^{mat} to be the empty concrete heap $(\cdot)$ and leaves the valuation unspecified. The concretization of ok puts constraints on neither H^{ok} nor the valuation, but forces H^{mat} to be empty. The concretization of an explicitly materialized symbolic heap mapping pair forces H^{ok} to be empty and H^{mat} to consist solely of a single-object concrete heap mapping the valuation of the symbolic address to a concrete object in the concretization of the symbolic object. It puts no constraints on the tag of the object (although this will be constrained by the symbolic concretization of base types, as we will see). Finally, the concretization of two symbolic sub-heaps consists of the component-wise concrete separating conjunction of the type-consistent and materialized components of concretization of the sub-heaps.

$\gamma : \mathsf{SymHeap} \to \mathcal{P}(\mathsf{Valuation} \times \mathsf{Heap} \times \mathsf{Heap})$

$$\gamma(\mathsf{emp}) \triangleq \left\{(V, H^{\mathrm{ok}}, H^{\mathrm{mat}}) \;\middle|\; H^{\mathrm{ok}} = \cdot \text{ and } H^{\mathrm{mat}} = \cdot \;\right\}$$

$$\gamma(\mathsf{ok}) \triangleq \left\{(V, H^{\mathrm{ok}}, H^{\mathrm{mat}}) \;\middle|\; H^{\mathrm{mat}} = \cdot \;\right\}$$

$$\gamma(\widetilde{a} : \widetilde{o}) \triangleq \left\{ (V, H^{\mathrm{ok}}, H^{\mathrm{mat}}) \;\middle|\; \begin{array}{l} H^{\mathrm{ok}} = \cdot \text{ and exists } a,\, o,\, B, \text{ where} \\ \quad H^{\mathrm{mat}} = a : \langle o, B\rangle \text{ and } V(\widetilde{a}) = a \text{ and} \\ \quad (V, o) \in \gamma(\widetilde{o}) \end{array} \right\}$$

$$\gamma(\widetilde{H}_1 \ast \widetilde{H}_2) \triangleq \left\{ (V, H^{\mathrm{ok}}, H^{\mathrm{mat}}) \;\middle|\; \begin{array}{l} (V, H_1^{\mathrm{ok}}, H_1^{\mathrm{mat}}) \in \gamma(\widetilde{H}_1) \text{ and} \\ (V, H_2^{\mathrm{ok}}, H_2^{\mathrm{mat}}) \in \gamma(\widetilde{H}_2) \text{ and} \\ H^{\mathrm{ok}} = H_1^{\mathrm{ok}} \ast H_2^{\mathrm{ok}} \text{ and} \\ H^{\mathrm{mat}} = H_1^{\mathrm{mat}} \ast H_2^{\mathrm{mat}} \end{array} \right\}$$

Symbolic Objects. The concretization of a symbolic object is a set of pairs of a valuation and a concrete object where the values stored in the fields of the concrete object are the same as valuation of the symbolic values associated with the fields in the symbolic object.

$\gamma : \mathsf{SymObject} \to \mathcal{P}(\mathsf{Valuation} \times \mathsf{Object})$

$$\gamma(\widetilde{o}) \triangleq \left\{(V, o) \;\middle|\; o(f) = V(\widetilde{x}) \text{ for each } f : \widetilde{x} \text{ in } \widetilde{o} \;\right\}$$

Symbolic Fact Map. The concretization of a symbolic fact map in the symbolic domain is similar to the concretization of a type environment in the type domain, except that here there are two heaps (H^{ok} and H^{mat}) and here the symbolic facts (symbolic types) constrain the valuation rather than an environment. Further, unlike the concretization of a type environment, which forces the entire heap to be type-consistent, the concretization of a symbolic fact map **only forces those addresses in H^{ok} to be immediately type consistent.** (We will explore this in more detail when discussing the symbolic concretization $\widetilde{\gamma}$ of base types.)

$$\gamma : \mathsf{SymFactMap} \to \mathcal{P}(\mathsf{Valuation} \times \mathsf{Heap} \times \mathsf{Heap})$$

$$\gamma(\widetilde{\Gamma}) \triangleq \left\{ (V, H^{\text{ok}}, H^{\text{mat}}) \;\middle|\; \begin{array}{l} \text{for each } \widetilde{x} : T^{\text{S}} \text{ in } \widetilde{\Gamma} \\ \quad (V, H^{\text{ok}}, H^{\text{mat}}, V(\widetilde{x})) \in \gamma(T^{\text{S}}) \\ \\ \text{and for all } a : \langle o, B\rangle \text{ in } H^{\text{ok}} \\ \quad\quad (H^{\text{ok}}, H^{\text{mat}}, a) \in \widetilde{\gamma}(B) \end{array} \right\}$$

Symbolic Facts. The concretization of a symbolic fact (symbolic type) is similar to that of the concretization of a regular type in the type domain, except, again, that it constrains two heaps and constrains a valuation rather than a concrete environment. Note that the concretization of the base type uses the symbolic concretization ($\widetilde{\gamma}$) of base types rather than the type concretization (γ) of base types. We use two separate symbols here because the standard overloading of γ would be ambiguous.

$$\gamma : \mathsf{Types}^{\text{S}} \to \mathcal{P}(\mathsf{Valuation} \times \mathsf{Heap} \times \mathsf{Heap})$$

$$\gamma(B \upharpoonright R^{\text{S}}_1, \cdots, R^{\text{S}}_n) \triangleq \left\{ (V, H^{\text{ok}}, H^{\text{mat}}, v) \;\middle|\; \begin{array}{l} (H^{\text{ok}}, H^{\text{mat}}, v) \in \widetilde{\gamma}(B) \text{ and} \\ \text{for all } i \text{ where } 1 \leq i \leq n \\ \quad (V, H^{\text{ok}} \ast H^{\text{mat}}, v) \in \gamma(R^{\text{S}}_i) \end{array} \right\}$$

Symbolic Refinements. The concretization of a symbolic refinement is similar to that of local refinements except that it constrains the valuation rather then a concrete environment. Note that unlike many symbolic concretizations, this concretization constrains only a single heap. This is because symbolic refinements represent relationships between **values** and thus cannot be violated.

$$\gamma : \mathsf{Refinements}^{\text{S}} \to \mathcal{P}(\mathsf{Valuation} \times \mathsf{Heap} \times \mathsf{Value})$$

$$\gamma(\mathsf{in}\ (c_1, \ldots, c_n)) \triangleq \{(V, H, s) \mid s \in \{c_1, \ldots, c_n\} \;\}$$

$$\gamma(\mathsf{respondsTo}\ \widetilde{x}(\overline{p : T_p}) \to B_{\text{ret}}) \triangleq \left\{ (V, H, a) \;\middle|\; \begin{array}{l} \text{exists string } m \text{, object } o \\ \text{and base type } B \text{ where} \\ m = V(\widetilde{x}) \\ H(a) = \langle o, B\rangle \text{ and} \\ o(m) \in \gamma(B, (\overline{p : T_p}) \to B_{\text{ret}}) \end{array} \right\}$$

Symbolic Concretization of Base Types. The symbolic concretization of base types (again, note the use of $\widetilde{\gamma}$ rather than γ) is structurally similar to that of the type domain concretization of base types, but the differences here are crucial. First, as is typical for the symbolic domain, there are two heaps, H^{ok} and H^{mat}. Second, for object types, note that whether the fields are constrained by the field environment for the type depends on whether the address of the object is in H^{ok} or not. If the address is in H^{ok}, then the field environment is constrained (with the **symbolic** concretization $\widetilde{\gamma}$). But if the address is **not** in H^{ok} then the fields are asserted to exist (and have some value stored in them), but the object is not required to be immediately type-consistent. In essence, the symbolic concretization traverses all reachable addresses and only forces those that are in H^{ok} to immediately conform to the requirements of the base type's type environment.

$\widetilde{\gamma} : \mathsf{BaseTypes} \to \mathcal{P}(\mathsf{Heap} \times \mathsf{Heap} \times \mathsf{Value})$

$$\widetilde{\gamma}(\text{Str}) \triangleq \left\{(H^{\text{ok}}, H^{\text{mat}}, v) \;\middle|\; v \text{ is a string } \right\}$$
$$\widetilde{\gamma}(\text{Unit}) \triangleq \left\{(H^{\text{ok}}, H^{\text{mat}}, v) \;\middle|\; v \text{ is a } \langle\rangle \right\}$$

The symbolic concretization of an object type $B = \{\overline{\text{var } f : T_f}, \;\; \overline{\text{def } m(\overline{p : T_p}) \to B_{\text{ret}}}\}$ is:

$$\widetilde{\gamma}(B) \triangleq \left\{ (H^{\text{ok}}, H^{\text{mat}}, a) \;\middle|\; \begin{array}{l} \text{exists } o \text{ where} \\ \quad H^{\text{ok}} * H^{\text{mat}}(a) = \langle o, B\rangle \text{ and} \\ \quad f \in \text{dom}(o) \text{ for all fields } f \\ \quad o(m) \in \gamma(B, (\overline{p : T_p}) \to B_{\text{ret}}) \text{ for all methods } m \text{ and} \\ \quad \text{if } a \in \text{dom}(H^{\text{ok}}) \text{ then} \\ \quad\quad (H^{\text{ok}}, H^{\text{mat}}, o) \in \widetilde{\gamma}(\overline{f : T_f}) \end{array} \right\}$$

As we will see (Lemma 3 in Section 4.4.2), a key property of the symbolic concretization of base types is that when H^{mat} is empty—that is, when the entire heap is almost type-consistent—then the meaning of a base type in the symbolic domain is that same as the meaning of the type in the types domain.

Symbolic Concretization of Field Types. The symbolic concretization of a field type is similar that for the type domain concretization of field types, but again there are two heaps and the base type is forced to be in the symbolic concretization of base types $\widetilde{\gamma}$, rather than γ.

$\widetilde{\gamma} : \mathsf{Types}^{\text{F}} \to \mathcal{P}(\mathsf{Heap} \times \mathsf{Heap} \times \mathsf{Object} \times \mathsf{Value})$

$$\widetilde{\gamma}(B \upharpoonright R^{\mathrm{F}}_1, \cdots, R^{\mathrm{F}}_n) \triangleq \left\{ (H^{\mathrm{ok}}, H^{\mathrm{mat}}, o, v) \;\middle|\; \begin{array}{l} (H^{\mathrm{ok}}, H^{\mathrm{mat}}, v) \in \widetilde{\gamma}(B) \text{ and} \\ \text{for all } i \text{ where } 1 \leq i \leq n \\ \qquad (H^{\mathrm{ok}} * H^{\mathrm{mat}}, o, v) \in \gamma(R^{\mathrm{F}}_n) \end{array} \right\}$$

Symbolic Concretization of Field Type Environments. Again, the symbolic concretization of a field type environment is similar to that in the type domain, except for the splitting of the heap and the recursive call to $\widetilde{\gamma}$ rather than γ.

$$\widetilde{\gamma} : \mathsf{Environments}^{\mathrm{F}} \to \mathcal{P}(\mathsf{Heap} \times \mathsf{Heap} \times \mathsf{Object})$$

$$\widetilde{\gamma}(\Gamma^{\mathrm{F}}) \triangleq \left\{ (H^{\mathrm{ok}}, H^{\mathrm{mat}}, o) \;\middle|\; \begin{array}{l} \text{for all } f : T^{\mathrm{F}} \text{ in } \Gamma^{\mathrm{F}} \text{ exists } v \text{ where} \\ \quad (H^{\mathrm{ok}}, H^{\mathrm{mat}}, o, v) \in \widetilde{\gamma}(T^{\mathrm{F}}) \text{ and } o(f) = v \end{array} \right\}$$

Soundness of Subtyping in the Symbolic Domain With concretization of types in the symbolic domain defined, we can now show that subtyping in this domain is also a sound over-approximation of including under concretization, similar to Lemma 1.

Lemma 2 (Soundness of Subtyping in the Symbolic Domain):

(1) If $\Gamma^{\mathrm{S}} \vdash T^{\mathrm{S}}_1 <: T^{\mathrm{S}}_2$ *and* $(V, H^{\mathrm{ok}}, H^{\mathrm{mat}}, v) \in \gamma(T^{\mathrm{S}}_1)$ *where* $(V, H^{\mathrm{ok}}, H^{\mathrm{mat}}) \in \gamma(\Gamma^{\mathrm{S}})$ *then* $(V, H^{\mathrm{ok}}, H^{\mathrm{mat}}, v) \in \gamma(T^{\mathrm{S}}_2)$.

(2) If $\Gamma^{\mathrm{F}} \vdash T^{\mathrm{F}}_1 <: T^{\mathrm{F}}_2$ *and* $(H^{\mathrm{ok}}, H^{\mathrm{mat}}, v) \in \widetilde{\gamma}(T^{\mathrm{F}}_1)$ *where* $(H^{\mathrm{ok}}, H^{\mathrm{mat}}) \in \widetilde{\gamma}(\Gamma^{\mathrm{F}})$ *then* $(H^{\mathrm{ok}}, H^{\mathrm{mat}}, v) \in \widetilde{\gamma}(T^{\mathrm{F}}_2)$.

Proof. By induction on the derivation of the subtyping relation. □

4.3.3 Symbolic Execution

Here, we formalize a symbolic execution [66], which is a disjunctive, path-sensitive analysis. Thus, our overall analysis state $\widetilde{P}$ is a disjunction of states. A single path $\widetilde{\Sigma} \downarrow \widetilde{x}$ is a pair of a symbolic environment and a symbolic value corresponding to the state on return and the return value of the path, respectively. Recall that we give a full description of our symbolic analysis state in Figure 4.7 in Section 4.3.

$$\boxed{\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e\, \{\widetilde{P}\}}$$

$$\textsc{Sym-Branch}\quad \frac{\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_1\, \{\widetilde{P}_1\} \qquad \widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_2\, \{\widetilde{P}_2\}}{\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_1 \talloblong e_2\, \{\widetilde{P}_1 \vee \widetilde{P}_2\}}$$

$$\textsc{Sym-Read-Local}\quad \frac{}{\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, x\, \{\widetilde{\Sigma} \downarrow \widetilde{E}(x)\}}$$

$$\textsc{Sym-Let}\quad \frac{\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_1\, \{\bigvee_i (\widetilde{\Sigma}_i \downarrow \widetilde{y}_i)\} \qquad \widetilde{x} \notin \widetilde{E} \qquad \widetilde{E}[x : \widetilde{y}_i] \vdash \{\widetilde{\Sigma}_i\}\, e_2\, \{\widetilde{P}_i\} \quad \text{for all } i}{\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, \text{let } x : T_1 = e_1 \text{ in } e_2\, \{\bigvee_i \widetilde{P}_i\}}$$

$$\textsc{Sym-Read-Field}\quad \frac{}{\widetilde{E} \vdash \{\widetilde{\Gamma}, \widetilde{H}\}\, x.f\, \{\widetilde{\Gamma}, \widetilde{H} \downarrow \widetilde{H}(\widetilde{E}(x))(f)\}}$$

$$\textsc{Sym-Write-Field}\quad \frac{}{\widetilde{E} \vdash \{\widetilde{\Gamma}, \widetilde{H}\}\, x.f := y\, \{\widetilde{\Gamma}, \widetilde{H}\big[\widetilde{E}(x) : \big(\widetilde{H}(\widetilde{E}(x))[f : \widetilde{E}(y)]\big)\big] \downarrow \widetilde{E}(y)\}}$$

$$\textsc{Sym-Seq}\quad \frac{\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_1\, \{\widetilde{P}'\} \qquad \widetilde{E} \vdash \{\widetilde{P}'\}\, e_2\, \{\widetilde{P}''\}}{\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_1 \,;\, e_2\, \{\widetilde{P}''\}}$$

$$\boxed{\widetilde{E} \vdash \{\widetilde{P}\}\, e\, \{\widetilde{P}'\}}$$

$$\textsc{Sym-Cases}\quad \frac{\widetilde{E} \vdash \{\widetilde{\Sigma}_i\}\, e\, \{\widetilde{P}_i\} \qquad \text{for all } i}{\widetilde{E} \vdash \{\bigvee_i \widetilde{\Sigma}_i \downarrow \widetilde{x}_i\}\, e\, \{\bigvee_i \widetilde{P}_i\}}$$

Figure 4.8: Non-type-intertwined rules for symbolic analysis.

In Figure 4.8, we describe a forward symbolic execution. Here we discuss only the "purely" symbolic rules—we describe rules for handoff with types and materialization from ok in Section 4.4. The judgment $\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e\, \{\widetilde{P}\}$ says that in the context of a given symbolic local variable environment $\widetilde{E}$ and with a symbolic state $\widetilde{\Sigma}$ on input, expression e symbolically evaluates to a disjunction of states $\widetilde{P}$ on output. So, for example, the Sym-Branch rule for non-deterministic branches says that if the left-hand–side e_1 of a branch in state $\widetilde{\Sigma}$ with local variable environment $\widetilde{E}$ evaluates to a disjunction of state-value pairs $\widetilde{P}_1$ and the right-hand–side e_2 evaluates to a disjunction $\widetilde{P}_2$ then symbolically executing the branch in $\widetilde{\Sigma}$ with $\widetilde{E}$ results in the disjunction of both of those disjunctions. That is, symbolically executing a branch evaluates each side of the branch with no loss of precision at the cost of a doubling of possible states (i.e., a symbolic join).

The Sym-Read-Local rule describes symbolic execution of reading a local variable x. This evaluation returns a single path with the state unchanged. The resultant value is the result of looking up the local address for x in the local variable environment $\widetilde{E}$. The Sym-Let rule describes symbolic evaluation of let expressions. This rule evaluates the initialization expression e_1 to a disjunction of paths, binds the resultant symbolic value to the (fresh) local variable in the environment $\widetilde{E}$ and executes the body e_2 in the resultant state. Note that in symbolic execution (in contrast with type checking), the declared type of the newly bound variable does not constrain the value stored into it. The Sym-Read-Field rule describes symbolic execution of reading a field f of a base address in x. The base object is required to be materialized, so the analysis just needs to look up the value in the materialized heap. The Sym-Write-Field rule describes symbolic execution of writing the value of a local variable y to a field f of a base address x. It requires that the object at the base address $\widetilde{E}(x)$ already be materialized and updates the appropriate field in the symbolic heap $\widetilde{H}$. This is a strong update—there is no need to lose precision on a heap write.

The Sym-Cases rule derives judgments of a slightly different form than the previous rules: it takes disjunctions of paths rather than single symbolic states, splits them into those single symbolic states, executes the expression in each of those, and then disjunctively combines each of the results. This form is the top-level judgment for symbolic execution. For example, the Sym-Seq rule relies on this judgment form to describe symbolically executing one expression after another.

Unlike traditional symbolic analysis, our type-intertwined approach can soundly ensure termination by falling back to type checking (as we will describe in Section 4.4). In practice, this allows the analysis to switch types at the end of loop bodies to cut back edges and cut recursion with method summaries.

4.4 Handoff and Invariant Conversion

We now walk through a modified version of the example from Section 3.2 to describe the key type-intertwined components of our analysis, including handoff between the type analysis and symbolic execution (and vice versa) and materialization/summarization from the almost type-consistent heap.

```
1  {
2    var obj: {} ↾ respondsTo sel = ...,
3    var sel: Str = ...,
4    def update(o: {} ↾ respondsTo s, s: Str): Unit =
5      self.obj := o;
6      self.sel := s
7  }
```

Figure 4.9: Formal version of `CustomImage` callback example.

Figure 4.9 shows a formal version of the `CustomImage` callback example. Here the `update` method updates the `obj` and `sel` fields in sequence. Recall that the first assignment breaks the type invariant and the second assignment restores it. We illustrate the core operations behind type-intertwined analysis by walking through this example. When checking this method, the type analysis will produce a flow-insensitive type error for the assignment at line 5 and so will switch (handoff) to symbolic execution. As we saw in the overview example in Figure 3.4, the analysis will (1) "symbolize" a suitable symbolic analysis state from the type environment (Section 4.4.2), (2) materialize storage from the almost type-consistent heap (Section 4.4.3), (4) symbolically execute the two field writes, (5) summarize storage for the fields back into the almost type-consistent heap

$$\boxed{\Gamma \vdash e : T}$$

T-Symbolic-Handoff

$$\frac{\widetilde{E} \vdash \{\widetilde{\Gamma}, \mathsf{ok}\}\, e\, \{\bigvee_i (\widetilde{\Gamma}_i, \mathsf{ok}) \downarrow \widetilde{x}_i\} \qquad \begin{array}{c} \Gamma \xrightarrow{\mathsf{symbolize}} \widetilde{\Gamma}, \widetilde{E} \\ \widetilde{\Gamma}_i, \widetilde{E} \xrightarrow{\mathsf{typeify}} \Gamma \end{array} \qquad \widetilde{\Gamma}_i \vdash \widetilde{\Gamma}_i(\widetilde{x}_i) <: T[\widetilde{E}] \qquad \text{for all } i}{\Gamma \vdash e : T}$$

$$\boxed{\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e\, \{\widetilde{P}\}}$$

Sym-Type-Handoff

$$\frac{\widetilde{\Gamma}, \widetilde{E} \xrightarrow{\mathsf{typeify}} \Gamma \qquad \Gamma \vdash e : T \qquad \Gamma \xrightarrow{\mathsf{symbolize}} \widetilde{\Gamma}', \widetilde{E} \qquad \widetilde{z} \notin \mathrm{dom}(\widetilde{\Gamma}')}{\widetilde{E} \vdash \{\widetilde{\Gamma}, \mathsf{ok}\}\, e\, \{\widetilde{\Gamma}'[\widetilde{z} : T[\widetilde{E}]], \mathsf{ok} \downarrow \widetilde{z}\}}$$

Figure 4.10: Analysis handoff via environment typeification and symbolization.

(Section 4.4.4), and (5) attempt to "typeify" the resultant symbolic analysis state back to the original type environment (Section 4.4.2). In the rest of this section I describe these operations in detail and demonstrate their soundness.

4.4.1 Handoff Between Analyses

In Figure 4.10, we describe the handoff process that determines when and how to switch from type checking to symbolic execution and from symbolic The T-Sym-Handoff rule formalizes handoff between type checking and symbolic analysis. Roughly speaking, this rule says the type checker can switch to the symbolic analysis to check an expression e in a type environment Γ by creating ("symbolizing") a symbolic state representing Γ, symbolically executing e in that state. and then ensuring that the resultant symbolic states conform to ("typeify to") the global type invariant. (We will describe symbolization and typeification judgments in detail later (Section 4.4.2)—here we discuss the handoff rules themselves.)

More precisely, the T-Sym-Handoff rule says that the type analysis can check an expression e in a type environment Γ if it can split the type environment (via symbolization) into a pair of a symbolic fact map $\widetilde{\Gamma}$ and symbolic environment $\widetilde{E}$. It can then symbolically execute the expression in that environment with a symbolic heap originally consisting solely of ok. (The symbolic execution can assume the entire heap is initially not immediately type-inconsistent because it just switched from type checking.) After symbolic execution, the resultant symbolic environments and fact maps must be consistent with ("typeify to") the original Γ, and the symbolic facts about the resulting symbolic values must be consistent with the inferred type T of the expression. Here $T[\tilde{\theta}]$ converts the standard type T to a symbolic type (fact) T^{S} with a substitution $\tilde{\theta}$ that replaces all variable references in T's refinements with symbolic values, as described in Section 4.2.4. We lift the subtyping judgment $\cdot \vdash \cdot <: \cdot$ to symbolic types in the expected way—this judgment over-approximates concrete inclusion in the symbolic domain (Lemma 2).

The key aspect of this handoff is that although the symbolic execution is free to violate any of the flow-insensitive constraints imposed by Γ, it must restore them to return to type checking. Both the initially symbolized heap **and** the finally typeified heap must consist solely of ok—the symbolic analysis can safely assume the entire heap is consistent on entry; but it must guarantee the consistency is restored on exit.

We describe the analogous handoff from symbolic execution to type analysis in the Sym-Type-Handoff rule. This rule says that the symbolic execution can switch to type checking for an expression e if the heap consists solely of ok and it can typeify the symbolic environment $\widetilde{E}$ and fact map $\widetilde{\Gamma}$ to a type environment Γ. Then, if the expression is well-typed in Γ and has type T then symbolic execution can continue in a symbolic state such that type environment symbolizes to a new fact map $\widetilde{\Gamma}'$ extended with a fact binding for the result of the expression. Again, the symbolic heap must be fully type-consistent (consist solely of ok) both before and after handoff. (We describe how this requirement can be soundly relaxed in Chapter 5.)

4.4.2 From Type Environments to Symbolic States and Back Again.

Symbolization splits a type environment Γ (which expresses type constraints on local variables) into a symbolic fact map $\widetilde{\Gamma}$ (expressing facts about symbolic values) and a symbolic local variable state $\widetilde{E}$ (expressing where those values are stored). For example, consider the type environment

$$\boxed{\begin{array}{l}\widetilde{\Gamma}, \widetilde{E} \xrightarrow{\mathsf{typeify}} \Gamma \\ \Gamma \xrightarrow{\mathsf{symbolize}} \widetilde{\Gamma}, \widetilde{E}\end{array}}$$

$$\textsc{C-Stack-Symbolize}\quad \frac{\widetilde{E} \text{ is 1-1} \qquad \Gamma <:_{\widetilde{E}^{-1}} \widetilde{\Gamma}}{\Gamma \xrightarrow{\mathsf{symbolize}} \widetilde{\Gamma}, \widetilde{E}} \qquad\qquad \textsc{C-Stack-Typeify}\quad \frac{\widetilde{\Gamma} <:_{\widetilde{E}} \Gamma}{\widetilde{\Gamma}, \widetilde{E} \xrightarrow{\mathsf{typeify}} \Gamma}$$

$$\boxed{\Gamma <:_{\tilde{\theta}^{-1}} \widetilde{\Gamma} \qquad \widetilde{\Gamma} <:_{\tilde{\theta}} \Gamma}$$

$$\textsc{Sub-Types-Facts}\quad \frac{\Gamma \vdash \Gamma(\tilde{\theta}^{-1}(\tilde{x})) <: T^{\mathrm{S}}[\tilde{\theta}^{-1}] \quad \text{for all } \tilde{x} : T^{\mathrm{S}} \in \widetilde{\Gamma}}{\Gamma <:_{\tilde{\theta}^{-1}} \widetilde{\Gamma}} \qquad \textsc{Sub-Facts-Types}\quad \frac{\widetilde{\Gamma} \vdash \widetilde{\Gamma}(\tilde{\theta}(x)) <: T[\tilde{\theta}] \quad \text{for all } x : T \in \Gamma}{\widetilde{\Gamma} <:_{\tilde{\theta}} \Gamma}$$

Figure 4.11: Symbolization and typeification of the stack.

above at line 5 in Figure 4.9, immediately before the first write:

$$\Gamma = [\mathsf{o} : \{\} \restriction \mathsf{respondsTo}\ \mathsf{s}][\mathsf{s} : \mathsf{Str}][\mathsf{self} : \mathrm{T}_{\mathrm{Image}}]$$

where $\mathrm{T}_{\mathrm{Image}} = \{\text{var } \mathsf{obj} : \{\} \restriction \mathsf{respondsTo}\ \mathsf{sel}, \text{var } \mathsf{sel} : \mathsf{Str}\}$. We can symbolize this environment to create a symbolic environment where $\widetilde{E} = [\mathsf{o} : \tilde{\mathsf{o}}][\mathsf{s} : \tilde{\mathsf{s}}][\mathsf{self} : \widetilde{\mathsf{self}}]$. Here we have created fresh symbolic names to represent the values stored on the stack: $\tilde{\mathsf{o}}$ is the name of the value stored in local o, $\tilde{\mathsf{s}}$ in local s, etc. These symbolic values represent concrete values from a type environment in which the storage location refinement relationships hold, so we can safely assume that **values** initially stored in those locations have the equivalent relationships, expressed as lifted types:

$$\widetilde{\Gamma} = [\tilde{\mathsf{o}} : \{\} \restriction \mathsf{respondsTo}\ \tilde{\mathsf{s}}][\tilde{\mathsf{s}} : \mathsf{Str}][\widetilde{\mathsf{self}} : \mathrm{T}_{\mathrm{Image}}]$$

Note that the refinement on $\tilde{\mathsf{o}}$ refers to symbolic value $\tilde{\mathsf{s}}$ and not storage location s, but that the refinements on the types of the **fields** of the base type of $\widetilde{\mathsf{self}}$'s fact $\mathrm{T}_{\mathrm{Image}}$ still refer to (field) storage locations. These field refinements on the base object type are, in essence, a "promise" that if any explicit storage for those fields is later materialized, it must be consistent with $\mathrm{T}_{\mathrm{Image}}$ when summarized back into ok.

We formalize type environment symbolization in rule C-Stack-Symbolize (Figure 4.11), which captures the requirement that the symbolized state must over-approximate the original type environment. We note that $\widetilde{E}^{-1}$ forms a substitution map from symbolic values to local variable names and require that the symbolized fact map $\widetilde{\Gamma}$ under that substitution be an over-approximate environment of the original type environment Γ (rule Sub-Types-Facts). In essence, any assumptions that the symbolic analysis initially makes about the symbolic facts must also hold in original type environment. That $\widetilde{E}$ is one-to-one ensures that the inverse exists but more importantly encodes the requirement that the newly symbolized environment makes no assumptions about aliasing between values stored on the stack in local variables. Note that when symbolizing a local variable with type $B \restriction R^{\mathrm{L}}$ in a type environment, we do not lift the base type B to the symbolic domain nor do we create storage for any of B's fields. That is, refinements on the **fields** of an object base type remain refinements over fields, expressing both facts about the field contents **and** constraints on those storage locations. This interpretation is what permits materialized, immediately type-inconsistent objects to point back into ok (i.e., the almost type-consistent region). As we detail in Section 4.4.3, with this interpretation our analysis **materializes** storage for objects from ok **on demand**, which is not only more efficient but is required in the presence of recursion.

Soundness of Handoff An important concern for our intertwined approach is whether information is transferred soundly between the type analysis and the symbolic analysis (i.e., we have a sound reduced product [32]). In particular, symbolization and materialization (Section 4.4.3) "pull" information from the type invariant on demand during symbolic execution and then permit temporary violations of the global heap invariant in some locations. We take an abstract interpretation-based approach [31] to soundness, which is critical for expressing almost type-consistent heaps and connecting the soundness of type checking with the soundness of symbolic analysis. In this section, we connect, via concretization, the different meanings of object types and their associated reachable heaps in the two analyses.

At handoff, the analysis requires that the explicitly materialized heap be empty. The following lemma states that under those conditions (i.e., when the entire heap is not immediately type-inconsistent), the meaning of base types in the symbolic domain ($\widetilde{\gamma}$) is the same as the meaning of base types in the type (γ) domain:

Lemma 3 (Equivalence of Typed and Symbolic Base Types):

$$\gamma(B) = \left\{(H, v) \;\middle|\; (H, \cdot, v) \in \widetilde{\gamma}(B) \right\}$$

Proof. By induction on the structure of B. □

This property is crucial for reasoning about the relationship between types and the symbolic domain.

While standard subtyping relates the concretizations of types within the same domain, subtyping under substitution relates concretizations in different domains. Here the soundness property essentially states that the syntactic transformation between types over identifiers of different classes over-approximates the analogous semantic transformation between concretizations.

With these lemmas for type substitution, we can now prove the soundness of subtyping under substitution. Here the soundness property essentially states that the syntactic transformation between types over identifiers of different classes over-approximates the analogous semantic transformation between concretizations.

Lemma 4 (Soundness of Subtyping Under Substitution):

(1) If $\Gamma^{\mathrm{L}}_1 <:_{\theta} \Gamma^{\mathrm{L}}_2$ *then*

$$\gamma(\Gamma^{\mathrm{L}}_1) \subseteq \left\{(E, H) \mid (E \circ \theta, H) \in \gamma(\Gamma^{\mathrm{L}}_2) \right\}$$

(2) If $\Gamma^{\mathrm{L}} <:_{\tilde{\theta}^{-1}} \Gamma^{\mathrm{S}}$ *then*

$$\gamma(\Gamma^{\mathrm{L}}) \subseteq \left\{(E, H) \mid (E \circ \tilde{\theta}^{-1}, H, \cdot) \in \gamma(\Gamma^{\mathrm{S}}) \right\}$$

(3) If $\Gamma^{\mathrm{S}} <:_{\tilde{\theta}} \Gamma^{\mathrm{L}}$ *then*

$$\left\{(V, H^{\mathrm{ok}}, H^{\mathrm{mat}}) \in \gamma(\Gamma^{\mathrm{S}}) \mid H^{\mathrm{mat}} = \cdot \right\} \subseteq \left\{(V, H^{\mathrm{ok}}, H^{\mathrm{mat}}) \mid (V \circ \tilde{\theta}, H^{\mathrm{ok}}) \in \gamma(\Gamma^{\mathrm{L}}) \right\}$$

(4) If $\Gamma^{\mathrm{S}} <:_{\tilde{\theta}} \Gamma^{\mathrm{F}}$ *then*

$$\gamma(\Gamma^{\mathrm{S}}) \subseteq \left\{(V, H^{\mathrm{ok}}, H^{\mathrm{mat}}) \mid (H^{\mathrm{ok}}, H^{\mathrm{mat}}, V \circ \tilde{\theta}) \in \widetilde{\gamma}(\Gamma^{\mathrm{F}}) \right\}$$

(5) If $\Gamma^{\mathrm{F}} <:_{\tilde{\theta}^{-1}} \Gamma^{\mathrm{S}}$ *then*

$$\widetilde{\gamma}(\Gamma^{\mathrm{F}}) \subseteq \left\{(H^{\mathrm{ok}}, H^{\mathrm{mat}}, o) \mid (o \circ \tilde{\theta}^{-1}, H^{\mathrm{ok}}, H^{\mathrm{mat}}) \in \gamma(\Gamma^{\mathrm{S}}) \right\}$$

Proof. Straightforward application of a technical lemma (Lemma 11) giving semantic meaning to transformations between the concretizations, the soundness of subtyping (Lemma 1), the equivalence of type domain base types and symbolic domain base types when the materialized heap is empty (Lemma 3), and a technical lemma on weakening of environments (Lemma 12). □

We rely on on this over-approximation under transformation to prove soundness of handoff (Lemma 5, below) as well as ensure the safety of method calls and field initialization (Theorem 1), and to prove the soundness materialization (Lemma 7) and summarization (Lemma 9).

In the case of handoff, the T-Sym-Handoff and Sym-Type-Handoff rules (Figure 4.10) describe the conditions under which the type analysis can switch to the symbolic analysis and vice versa in terms of typeification and symbolization. The following lemma shows that these judgments, combined with the constraint that the symbolic heap consist of only ok at the time of handoff, is sufficient to guarantee that no potential concrete states are dropped when switching between the two analyses.

Lemma 5 (Soundness of Handoff):

(1) If $\Gamma^{\mathrm{L}} \; \overrightarrow{\mathsf{symbolize}} \; \widetilde{\Gamma}, \widetilde{E}$ *then* $\gamma(\Gamma^{\mathrm{L}}) \subseteq \gamma(\widetilde{E}, (\widetilde{\Gamma}, \mathsf{ok}))$ *and*

(2) If $\widetilde{\Gamma}, \widetilde{E} \; \overrightarrow{\mathsf{typeify}} \; \Gamma^{\mathrm{L}}$ *then* $\gamma(\widetilde{E}, (\widetilde{\Gamma}, \mathsf{ok})) \subseteq \gamma(\Gamma^{\mathrm{L}})$

Proof. By application of Lemma 4 and the definitions of concretization of type environments and symbolic states. □

We rely on this lemma in our proof of soundness of Fissile type analysis (see, e.g., the *T-Sym-Handoff* sub-case of the **E-Branch-Left** case in Theorem 1).

While symbolization and typeification of **states** govern the handoff between type analysis and symbolic analysis, symbolization and typeification of **objects** constrain the materialization and summarization of storage during symbolic analysis—which we now discuss.

4.4.3 Materialization from Type-Consistent Heaps

Returning to the callback example, recall that the analysis has symbolized a state corresponding to the type environment immediately before line 5. A symbolic heap $\widetilde{H}$ consists of two separate regions: (1) the materialized heap, a precise region with explicit storage that supports strong updates and allows field values to differ from their declared types (i.e., permits immediate type-inconsistency) and (2) the ok, a summarized region in which all locations are either type-consistent or only transitively type-inconsistent. In a newly symbolized analysis state, $\widetilde{H}$ consists of solely ok. Before the field write at line 5 can proceed, the analysis must first materialize storage for the $\mathrm{T}_{\mathrm{Image}}$ object pointed to by $\widetilde{\mathsf{self}}$ to get:

$$\widetilde{H} = \mathsf{ok} * \widetilde{\mathsf{self}} \mapsto \{\mathsf{obj} : \widetilde{\mathsf{obj}}, \mathsf{sel} : \widetilde{\mathsf{sel}}\}$$

and a new fact map $\widetilde{\Gamma}$ that contains the additional facts: $\widetilde{\mathsf{obj}} : \{\} \upharpoonright \mathsf{respondsTo} \quad \widetilde{\mathsf{sel}}$ and $\widetilde{\mathsf{sel}} : \mathsf{Str}$.

$$\boxed{\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e\, \{\widetilde{P}\}}$$

$$\textsc{Sym-Materialize}\quad \frac{\widetilde{\Sigma}\ \overrightarrow{\mathsf{materialize}}\ \bigvee_i \widetilde{\Sigma}'_i \qquad \widetilde{E} \vdash \{\widetilde{\Sigma}'_i\}\, e\, \{\widetilde{P}_i\} \qquad \text{for all } i}{\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e\, \{\bigvee_i \widetilde{P}_i\}} \qquad \textsc{Sym-Summarize}\quad \frac{\widetilde{\Sigma}\ \overrightarrow{\mathsf{summarize}}\ \widetilde{\Sigma}' \qquad \widetilde{E} \vdash \{\widetilde{\Sigma}'\}\, e\, \{\widetilde{P}\}}{\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e\, \{\widetilde{P}\}}$$

$$\boxed{\widetilde{\Sigma}\ \overrightarrow{\mathsf{materialize}}\ \bigvee_i \widetilde{\Sigma}'_i \qquad \widetilde{\Sigma}\ \overrightarrow{\mathsf{summarize}}\ \widetilde{\Sigma}'}$$

$$\textsc{M-Materialize}\quad \frac{\widetilde{\Sigma} = \widetilde{\Gamma}, \widetilde{H} \qquad \mathsf{ok} \in \widetilde{H} \qquad \widetilde{a} \notin \mathrm{dom}(\widetilde{H}) \qquad \widetilde{\Gamma}(\widetilde{a}) = B\ \upharpoonright\ \cdots \qquad B\ \overrightarrow{\mathsf{symbolize}}\ \widetilde{\Gamma}^{\mathrm{fields}}, \tilde{o} \qquad \widetilde{\Gamma}' = \widetilde{\Gamma}, \widetilde{\Gamma}^{\mathrm{fields}}}{\widetilde{\Sigma}\ \overrightarrow{\mathsf{materialize}}\ \left((\widetilde{\Gamma}', (\widetilde{H} * \widetilde{a} : \tilde{o})) \vee \bigvee_{\widetilde{y} \in \mathrm{mayalias}_{\widetilde{\Sigma}}(\widetilde{a})} \widetilde{\Sigma}|_{\widetilde{a} = \widetilde{y}} \right)}$$

$$\textsc{M-Summarize}\quad \frac{\mathsf{ok} \in \widetilde{H} \qquad \widetilde{\Gamma}(\widetilde{a}) = B\ \upharpoonright\ \cdots \qquad \widetilde{\Gamma}, \tilde{o}\ \overrightarrow{\mathsf{typeify}}\ B}{\widetilde{\Gamma}, (\widetilde{H} * \widetilde{a} : \tilde{o})\ \overrightarrow{\mathsf{summarize}}\ \widetilde{\Gamma}, \widetilde{H}}$$

$$\boxed{\begin{array}{l} \widetilde{\Gamma}, \tilde{o}\ \overrightarrow{\mathsf{typeify}}\ B \\ B\ \overrightarrow{\mathsf{symbolize}}\ \widetilde{\Gamma}, \tilde{o} \end{array}}$$

$$\textsc{C-Object-Typeify}\quad \frac{\widetilde{\Gamma}\ <:_{\tilde{o}}\ \mathrm{fieldtypes}(B)}{\widetilde{\Gamma}, \tilde{o}\ \overrightarrow{\mathsf{typeify}}\ B} \qquad \textsc{C-Object-Symbolize}\quad \frac{\Gamma = \mathrm{fieldtypes(B)} \qquad \tilde{o} \text{ is 1-1} \qquad \Gamma\ <:_{\tilde{o}^{-1}}\ \widetilde{\Gamma}}{B\ \overrightarrow{\mathsf{symbolize}}\ \widetilde{\Gamma}, \tilde{o}}$$

Figure 4.12: Materialization and summarization.

We formalize materialization and summarization in Figure 4.12, which extends our symbolic execution with these operations. The Sym-Materialize rule says that our symbolic executor is allowed to non-deterministically materialize any symbolic address from ok into the materialized heap. This materialization may require considering multiple possible aliasing relationships with the already materialized addresses, so it produces a disjunction of possible materialized states all of which must be symbolically executed. In practice, we limit materialization to the base object of field reads and writes. Rule Sym-Summarize covers the opposite operation: the symbolic executor can non-deterministically summarize a materialized object back into ok. We describe the details of materialization here and summarization in Section 4.4.4.

The C-Object-Symbolize and M-Materialize rules describe type-consistent materialization of object storage. Creating symbolic storage for an object type is very similar to symbolizing a type environment. As rule C-Object-Symbolize defines, the analysis can symbolize a type B to a symbolic object $\tilde{o}$ (mapping field names to fresh symbolic values) and a fact map $\widetilde{\Gamma}$ (facts about those values) if the assumed facts about the values are no stronger than those guaranteed by the object's field types. Once the analysis has symbolized an object, it adds the new object storage to the explicit heap and facts about the fresh symbolic values to the fact map in rule M-Materialize.

For the symbolic analysis to perform strong updates, it must maintain the key invariant that any two objects' storage locations in the explicit heap are definitely separate. When materializing an arbitrary object, the evaluator must consider whether any of the already materialized objects aliases with the newly materialized object and case split on these possibilities. The split is required because any two distinct symbolic values may in fact represent the same concrete value. In M-Materialize, for any input state $\widetilde{\Sigma}$ in which the heap contains ok, the symbolic analysis is free to materialize the object stored at a symbolic address $\widetilde{a}$ from the type-consistent heap. For the case where the materialized symbolic address does not alias any address already on the explicit heap, we symbolize a new symbolic object $\tilde{o}$ with fresh symbolic values as described above from the base type of $\widetilde{a}$. In the case where the address may alias some address $\widetilde{y}$ on the materialized heap, we must assert that $\widetilde{a} = \widetilde{y}$. We write $\widetilde{\Sigma}|_{\widetilde{a}=\widetilde{y}}$ for any sound constraining of $\widetilde{\Sigma}$ with the equality (in practice, we implement it by substituting one name for the other and applying a meet $\sqcap$ in the symbolic facts $\widetilde{\Gamma}$). We also leave unspecified the $\mathrm{mayalias}_{\widetilde{\Sigma}}(\widetilde{a})$ that should soundly yield the set of addresses that may-alias $\widetilde{a}$; our implementation uses a type-based check to rule out simple non-aliases.

This rule is quite general. It permits an arbitrary number of locations to be immediately type-inconsistent without any constraints on connectivity, ownership, or non-aliasing. To simplify the formalization of type-consistent materialization, we restrict relations expressed by refinements in the heap to be among fields within objects. Relations between fields are captured because all of fields of the object are symbolized at the same time (see C-Object-Symbolize). Supporting cross-object relations would merely require materializing multiple objects while disjunctively considering all possible aliasing relationships and then symbolizing their fields simultaneously within each configuration. It would also be possible to just materialize the fields corresponding to the specific relationships that we wish to violate by using a field-split model [69, 81] of objects.

Soundness of Materialization Before we can characterize the effect of materialization on symbolic states, we must describe the effect of materialization on the symbolic concretization of base types.

Lemma 6 (Type-Consistent Materialization for Types): *If* $(H^{\mathrm{ok}} * a : \langle o_a, B_a\rangle, H^{\mathrm{mat}}, v) \in \widetilde{\gamma}(B)$ *then* $(H^{\mathrm{ok}}, H^{\mathrm{mat}} * a : \langle o_a, B_a\rangle, v) \in \widetilde{\gamma}(B)$*.*

Proof. By induction on the structure of B. □

This lemma says that if a given value v is in the symbolic concretization of a base type B and that concretization has an object of type B_a stored in the type-consistent heap at address a, then moving the storage for a from the type-consistent heap to the materialized heap will not cause the type of v to change from the perspective of the symbolic analysis.

We will use this lemma to prove soundness of materialization, but first we must discuss the soundness criteria for two operations relied upon in the M-Materialize rule: may-alias analysis and constraint of a symbolic state for equality of symbolic values.

Condition 1 (Soundness of mayalias)**.**

For all $(V, H^{\mathrm{ok}}, H^{\mathrm{mat}}) \in \gamma(\widetilde{H})$:

if $y \in \mathrm{dom}(\widetilde{H})$ and $V(\widetilde{y}) = V(\widetilde{a})$ then $\widetilde{y} \in \mathrm{mayalias}_{\widetilde{\Gamma},\widetilde{H}}(\widetilde{a})$

Condition 1 says that the may-alias analysis employed during materialization must be sound—that is, if the concretization of a state $\widetilde{\Gamma}, \widetilde{H}$ yields a valuation V in which the values associated with $\widetilde{y}$

and $\widetilde{a}$ are the same, then the alias analysis must report this possibility.

Condition 2 (Soundness of State Equality Constraint)**.**

$$\left\{ (V, H^{\text{ok}}, H^{\text{mat}}) \in \gamma(\widetilde{\Sigma}) \;\middle|\; V(\widetilde{x}) = V(\widetilde{y}) \right\} \subseteq \gamma(\widetilde{\Sigma}|_{\widetilde{x}=\widetilde{y}}).$$

Condition 2 says that constraint of the symbolic state such that two symbolic values are equal must be sound—that is, the constraint must not erroneously throw out a state in which the concrete values represented by the symbolic values are equal.

With these conditions in place, we can now prove the soundness of materialization from the type consistent heap:

Lemma 7 (Soundness of Type-Consistent Materialization):

If $\mathsf{ok} \in \widetilde{H}$ *and* $\widetilde{a} \notin \mathrm{dom}(\widetilde{H})$ *and* $\widetilde{\Gamma}(\widetilde{a}) = B \restriction \cdots$ *and* $B \xrightarrow{\mathsf{symbolize}} \widetilde{\Gamma}^{\text{fields}}, \tilde{o}$ *and* $\widetilde{\Gamma}' = \widetilde{\Gamma}, \widetilde{\Gamma}^{\text{fields}}$ *then for all* $\widetilde{E}$ *where* $\mathrm{rng}(\widetilde{E}) \subseteq \mathrm{dom}(\widetilde{\Gamma})$*:*

$$\gamma(\widetilde{E}, (\widetilde{\Gamma}, \widetilde{H})) \subseteq \gamma\left(\widetilde{E}, (\widetilde{\Gamma}', (\widetilde{H} * \widetilde{a} : \tilde{o})) \vee \bigvee_{\widetilde{y} \in \mathrm{mayalias}_{\widetilde{\Sigma}}(\widetilde{a})} \widetilde{\Sigma}|_{\widetilde{a}=\widetilde{y}}\right).$$

Proof. By Lemma 6, Lemma 12, Lemma 4, and Conditions 1 and 2. □

This lemma proves the soundness of the M-Materialize rule; it says that if the required preconditions hold, the operation of materializing storage from ok will not erroneously throw out potential concrete states. We use this lemma in the overall proof of soundness of Fissile type analysis (see the *Sym-Materialize* sub-case of the **E-Branch-Left** case in Theorem 1).

4.4.4 Summarizing Symbolic Objects Back Into Types.

Continuing our running example, with the symbolization and materialization complete, the analysis now executes the field writes at lines 5 and 6. At this point, the symbolic heap at line 6 is:

$$\widetilde{H} = \mathsf{ok} * \widetilde{\mathsf{self}} \mapsto \{\mathsf{obj} : \widetilde{\mathsf{o}}, \mathsf{sel} : \widetilde{\mathsf{s}}\}.$$

That is, the fields now contain the values passed in as parameters. But recall that $\widetilde{\Gamma}(\widetilde{\mathsf{o}}) = \{\} \restriction \mathsf{respondsTo}\ \widetilde{\mathsf{s}}$ and $\widetilde{\Gamma}(\widetilde{\mathsf{self}}) = \mathrm{T}_{\mathrm{Image}}$. In this state, the value stored in field obj again responds to the value stored in field sel—the flow-insensitive type invariant ($\mathrm{T}_{\mathrm{Image}}$) promised by $\widetilde{\Gamma}(\widetilde{\mathsf{self}})$ again holds—and thus the object can be safely summarized back into the ok.

We describe this process in rule M-Summarize (Figure 4.12), which says that a symbolic address $\widetilde{a}$ pointing to a materialized object $\tilde{o}$ can be summarized (i.e., removed from the explicit heap) if the object is consistent with (i.e., can be "typeified" to) the base type required of the address in the fact map. Typeifying a symbolic object $\tilde{o}$ to an object type B (rule C-Object-Typeify) is analogous to symbolization except that it goes in the other direction. We require that the symbolic fact map be over-approximated by the field types of B, nicely converting it to the symbolic domain using $\tilde{o}$ as the substitution. Note that $\tilde{o}$ does not need to be one-to-one; the observation that this constraint is irrelevant for typeification captures that types are agnostic to aliasing.

Once all materialized objects have been summarized (and thus $\widetilde{H} = \mathsf{ok}$), the checker can end the handoff to symbolic analysis and resume type checking (back to rule T-Sym-Handoff in Figure 4.10) as long as the symbolic locals are consistent with the original Γ for all symbolic paths (rule C-Stack-Typeify in Figure 4.11) and the returned symbolic values have facts consistent with the return type of the expression.

Soundness of Summarization As we did for materialization, for soundness of summarization we will first characterize the effect of summarization on the concretization of a base type:

Lemma 8 (Soundness of Type-Consistent Summarization for Types): *If* $(H^{\mathrm{ok}}, H^{\mathrm{mat}} * a : \langle o_a, B_a\rangle, v) \in \widetilde{\gamma}(B)$ *and* $V(\widetilde{a}) = a$ *and* $\widetilde{\Gamma}(\widetilde{a}) = B_a \upharpoonright \cdots$ *and* $(V, H^{\mathrm{ok}}, H^{\mathrm{mat}} * a : \langle o_a, B_a\rangle) \in \widetilde{\gamma}(\widetilde{\Gamma})$ *and* $(V, o_a) \in \gamma(\tilde{o})$ *and* $\widetilde{\Gamma}, \tilde{o}\ \overrightarrow{\mathsf{typeify}}\ B_a$ *then* $(H^{\mathrm{ok}} * a : \langle o_a, B_a\rangle, H^{\mathrm{mat}}, v) \in \widetilde{\gamma}(B)$.

Proof. By induction on the structure of B and with Lemma 4. □

This lemma says that if a given value v is in the symbolic concretization of a base type B and that concretization has an object o_a of type B_a stored in the materialized heap at address a and the preconditions for summarization hold (i.e., that the object is not immediately type-inconsistent), then moving moving the storage for a from the materialized heap to the type-consistent heap will not cause the type of v to change. With this helper lemma in place, we can then prove the soundness of summarization for states:

Lemma 9 (Soundness of Type-Consistent Summarization for States): *If* $\widetilde{\Gamma}, \tilde{o} \ \overrightarrow{\mathsf{typeify}} \ B$ *and* $\mathsf{ok} \in \widetilde{H}$ *and* $\widetilde{\Gamma}(\widetilde{a}) = B \upharpoonright \cdots$ *then for all* $\widetilde{E}$*:*

$$\gamma(\widetilde{E}, (\widetilde{\Gamma}, \widetilde{H} * \widetilde{a} : \tilde{o})) \subseteq \gamma(\widetilde{E}, (\widetilde{\Gamma}, \widetilde{H})) \ .$$

Proof. By Lemma 8 and the definition of the concretization of symbolic states. □

This lemma proves the soundness of the M-Summarize rule. It says that if the symbolic execution determines that a materialized symbolic object is not immediately type inconsistent, then the symbolic storage for that object can be summarized back into ok without erroneously dropping potential concrete states. We rely on this lemma in the *Sym-Summarize* sub-case of the **E-Branch-Left** case in Theorem 1.

4.5 Soundness of Intertwined Analysis

We can now present our main theorem: the proof of soundness of combined type analysis and symbolic execution with almost type-consistent heaps. The theorem states soundness of all three forms of static judgments: type checking, state-to-path symbolic execution, and path-to-path symbolic execution. Note that we prove that the result of execution is a heap-value pair; that is, it is not an error state err.

Theorem 1 (Soundness of Fissile Type Analysis)**.**
If $E \vdash [H]\, e\, [r]$ *then*

(1) If $\Gamma^{\mathrm{L}} \vdash e : T^{\mathrm{L}}$ *and* $(E, H) \in \gamma(\Gamma^{\mathrm{L}})$ *then* $r = H' \downarrow v'$ *where* $(E, H') \in \gamma(\Gamma^{\mathrm{L}})$ *and* $(E, H', v') \in \gamma(T^{\mathrm{L}})$*; and*

(2) If $\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e\, \{\widetilde{P}\}$ *and* $(E, H) \in \gamma(\widetilde{E}, \widetilde{\Sigma})$ *then* $r = H' \downarrow v'$ *where* $(E, H', v') \in \gamma(\widetilde{E}, \widetilde{P})$*; and*

(3) If $\widetilde{E} \vdash \{\widetilde{P}\}\, e\, \{\widetilde{P}'\}$ *and* $(E, H, v) \in \gamma(\widetilde{E}, \widetilde{P})$ *then* $r = H' \downarrow v'$ *where* $(E, H', v') \in \gamma(\widetilde{E}, \widetilde{P}')$*.*

Proof. By induction on the derivation of $E \vdash [H]\, e\, [r]$. □

I provide the interesting cases, as well additional required technical lemmas in Chapter A. The key feature of the proof is a nested simultaneous induction on each of the static judgment forms to account for the non-syntax-directed nature of the T-Sym-Handoff, Sym-Types-Handoff, Sym-Materialize, and Sym-Handoff rules.

4.6 Case Study: Checking Reflective Call Safety

We implemented our Fissile type analysis approach to checking almost flow-insensitive invariants in a prototype method reflection safety checker for Objective-C. We evaluate our prototype, a plugin to the clang static analyzer, by investigating the following questions: What is the increased type annotation cost for checking reflection safety? How much does the mixed Fissile approach improve precision? Do our premises about how programmers violate relationships hold in practice? Is our intertwined “almost type” analysis as fast as we hope? We also discuss a bug found by our tool—surprising in a mature application. The bug fix that we proposed was accepted by the application developer. Finally, we show that our tool is capable of alerting inexperienced Objective-C developers to a common source of beginner bugs, typos in selector names, by applying it to snippets culled from postings by confused developers on online forums.

4.6.1 Prototype Tool

We have implemented our approach in a tool, Missile, that verifies dependent refinement types in C and Objective-C. This tool supports a rich specification language that enables programmers to extend their existing type declarations with dependent refinements expressing almost-everywhere invariants. Missile is implemented as a plugin for the `clang` [1] static analyzer and integrates with the `Xcode` [61] development environment to provide feedback to the programmer, including graphical abstract counterexamples when reporting alarms.

Specification Language. Missile enables developers to specify almost-everywhere invariants on storage locations alongside the storage declaration. I show such a specification in the shaded region of Figure 4.13. Here, the programmer has annotated the `obj` field of the Objective-C

Callback class with the refinement {respondsTo sel} (specifying that the object stored in obj should have a method with the name stored in sel). The __attribute__ annotation mechanism ensures that source code extended with MISSILE specifications remains compatible with other compilers and thus will not interfere with the developer's existing build system.

```
@interface Callback {
  Object *obj __attribute__((missile("{respondsTo sel}"))) ;
  SEL sel;
}
@end
```

Figure 4.13: MISSILE enables programmers to specify almost-everywhere invariants in C and Objective-C with dependent refinement type annotations (shaded region).

MISSILE supports refinements specifying a variety of relationship requirements. For example, the refinement

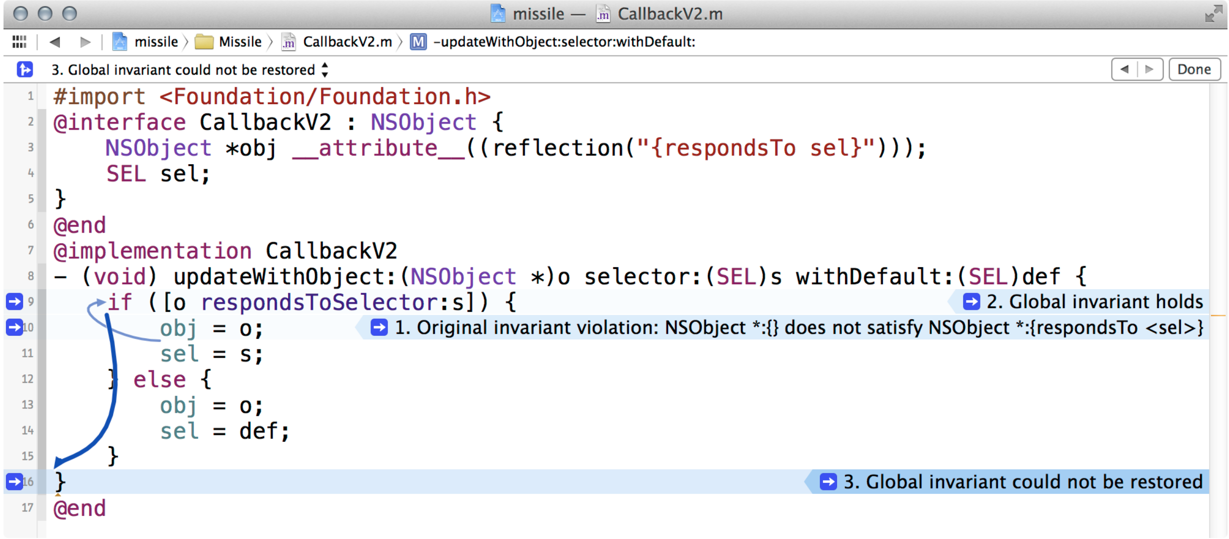
{valueIn ('didClick', 'didDoubleClick')}

refines string types to ensure that the value is one of those specified; this refinement is useful for verifying reflective method call. The formula refinement embeds quantifier-free linear arithmetic into types, so the annotation

{formula (and (gte _v_ 0) (lt _v_ len))}

could express the specification that a value in this type (represented by the bound variable _v_) should be an in-bounds index for an array of length len. Because MISSILE builds on the parametric Fissile Type Analysis framework, it can be easily extended to support any refinement with a procedure for statically over-approximating semantic inclusion (e.g., subtyping, implication with an SMT solver, string solvers, etc.).

In addition to field refinements—useful for expressing data structure invariants—MISSILE also supports refinements among locals as well as refinements relating parameters, the method receiver, and the return value. The tool also provides a traditional assert mechanism to specify invariants at individual program points.



3. Global invariant could not be restored

```
#import <Foundation/Foundation.h>
@interface CallbackV2 : NSObject {
    NSObject *obj __attribute__((reflection("{respondsTo sel}")));
    SEL sel;
}
@end
@implementation CallbackV2
- (void) updateWithObject:(NSObject *)o selector:(SEL)s withDefault:(SEL)def {
    if ([o respondsToSelector:s]) {
        obj = o;
        sel = s;
    } else {
        obj = o;
        sel = def;
    }
}
@end
```

2. Global invariant holds

1. Original invariant violation: NSObject *:{} does not satisfy NSObject *:{respondsTo <sel>}

3. Global invariant could not be restored

Figure 4.14: MISSILE integrates with the Xcode development environment to visualize mixed flow-insensitive and path-sensitive alarms.

Visual Abstract Counterexamples The intertwined approach MISSILE takes to verification poses a challenge when presenting alarms to the user: how should the tool explain the combination of global flow-insensitive and local path-sensitive reasoning that lead to the alarm? MISSILE integrates with the XCODE development environment to visually display these abstract counterexamples. Figure 4.14 provides an example of this visualization, which shows the explanation for an alarm in a modified version of the method verified in Section 3.2.

In this version, the programmer has added an additional `def` parameter, providing a default selector to be stored if `o` does not respond to `s`. Unfortunately, the new version has a bug: if `o` does not respond to `def` then the `Callback` object may be left in an inconsistent state if the `else` branch executes. MISSILE visually displays this reasoning with a sequence of arrows rendered above the path in question. The arrows in this figure are all displayed by the tool in the development environment—we have not added them after the fact. The tool shows the flow-insensitive invariant violation (the small arrow marked 1) and indicates the location where the symbolic analysis took over and assumed the global invariant held (marked here with 2). We use the thin arrows to indicate the path taken by the analysis on which it could not guarantee invariant restoration. The programmer can quiet this alarm by supplying a parameter annotation (checked at method call sites) on `o`, `__attribute__((reflection("{respondsTo def}")))` specifying that it must respond to `def`.

Handoff Heuristics. When Missile detects a flow-insensitive type error, it can switch to path-sensitive reasoning to determine that the type error is a false alarm as described in the handoff rules from Section 4.4. These rules characterize when it is sound to switch between analysis styles—but not when it is effective to do so. Here, we describe the key heuristics that Missile employs to switch between type checking and symbolic execution in practice.

Missile's heuristics are driven by the lexical structure of the source code location at which the type error occurred. Upon a type error, the tool first starts symbolically executing immediately prior to the statement containing the type error. It will symbolically execute until it passes the location of the type error and either (1) it can safely return to type checking or (2) it attempts to retry symbolic execution of a larger chunk of code in the hope that more context will allow it to determine that it is safe to to return to type checking. The tool can safely return to type checking when one of two criteria are met:

- **The global type invariant is restored on all paths.** In this case, the symbolic execution has demonstrated that the type error is a false alarm and so the analysis switches back to type checking at the point of restoration. In an effort to limit the path explosion during symbolic execution, Missile returns back to type checking optimistically, as soon as it is sound to do so.

- **All symbolic paths to the type error are found to be unreachable.** Again, in this case the type error is a false alarm and so the the analysis returns back to type checking.

In some cases, the symbolic execution cannot safely return to type checking, but it also cannot continue symbolic execution:

- **Unhandled language constructs**. Missile's symbolic executor is limited in the language constructs that it can handle. For example, it does not attempt to unroll while loops nor does it symbolically execute switch statements. Further, because it does not have access to the whole program, it cannot symbolically execute dynamically-dispatched Objective-C method calls without a summary of the method.

- **Summary preconditions not met.** Even if a method is annotated with a summary, the symbolic execution will not be able to apply the summary if it cannot show that the summary preconditions hold.

- **End of lexical scope.** An implementation limitation of Missile's symbolic executor is that it operates over lexical portions of the program's abstract syntax tree (this makes it easier to switch back and forth with the type system). If the symbolic executor reaches the end of the entire lexical syntax over which it has access (for example, the body of a while loop, or the else branch of an if statement) and the type invariant is not restored then it will report that it could not determine whether the type error was a false alarm—even if symbolically executing out of the scope could show the type invariant was restored.

In these cases, the tool attempts to back up and retry symbolic execution of a larger chunk of code.

Missile heuristically retries symbolic execution with increasingly larger portions of the abstract syntax tree containing the original type error. It starts by symbolically executing from the initial type error up to (potentially) the end of the brace statement containing the type error. If the symbolic execution cannot show that the type invariant is restored before the end of this statement, it retries from the beginning of the brace statement. This extra scope may allow the symbolic execution to establish additional aliasing relationships to show the type invariant is restored. If this does not succeed, it retries from beginning of any containing if statement. This retry allows the symbolic execution to take advantage of the if statement's guard condition—as it does, for example, to determine that `o` responds to `s` in Figure 4.14. If this extra scope is not enough, it continues with increasingly larger containing brace and if statements, until it tries symbolically executing the entire method body. If symbolically executing the entire method does not succeed, it reports the original flow-insensitive type error to the user.

```
1 - (void) performCallback {
2     [this callSomeMethod];
3
4     Obj *o = self->obj;
5     SEL s = self->sel;
6     [o performSelector:s];
7 }
```

Figure 4.15: Missile's retry heuristics cannot show that this example is safe.

While these retries can mitigate some of the shortcomings of the lexically-based symbolic executor, the approach has its limitations. Suppose the `Callback` class has a `performCallback` method as shown in Figure 4.15. The reflective call at line 6 is safe. But the heuristics above will result in a false alarm. After the initial type error, the analysis will attempt execute the statement at line 6. This symbolic execution will fail because, starting from line 6 it cannot show that the preconditions for `performSelector:` (that `o` always responds `s`) hold. The retry heuristics will then retry symbolic execution from the beginning of the method—but the call at line 2 will force the symbolic execution to return to type checking. The analysis will then continue in type checking mode until it reaches line 6, at which point it will hit the type error again and give up. In this case, a successful approach would be to retry symbolic execution starting from line 4—but our heuristics are not intelligent enough to attempt to do this. In general, our heuristic approach relies on being able to successfully symbolic execute from the beginning of a scope to the triggering type error—if this is not possible, Missile will not be able to show it is a false alarm.

4.6.2 Benchmarks

We evaluate our approach to checking reflection safety on a suite of real-world benchmarks in Objective-C that we collected from the open-source community. This suite consists of 6 libraries and 3 large applications. Table 4.1 provides the size of these benchmarks. The "Lines of Code" count includes project headers but excludes comments and whitespace. The "Methods" column indicates the total number of methods, and the "Reflective Call Sites" column gives the number of calls to

Benchmark	Lines of Code	Reflective Call Sites	Methods
OAuth	1248	7	92
SCRecorder	2716	12	200
ZipKit	3301	28	165
Sparkle	5290	40	320
ASIHTTP	13565	68	707
OmniFrwks	160769	192	7611
Vienna	37348	186	2261
Skim	60211	207	3010
Adium	176632	587	8723
Combined	461080	1327	23089

Table 4.1: A suite of reflection benchmarks in Objective-C.

system library methods that perform reflection, either directly or as part of some other operation.

I first describe our library benchmarks. OAuth performs OAuth Consumer authentication and uses reflective calls to inform clients when the authentication succeeds or fails. Sparkle is a very widely-used automatic update library that uses reflection to communicate across threads, to avoid boilerplate, and to interact with the client application. ZipKit is a library that reads and writes compressed archives—it also uses reflection to communicate across threads. SCRecorder is a library that developers embed to allow their users to record custom keyboard shortcuts; it is the source of our motivating example from Chapter 2. ASIHTTP is a library that performs web services calls; it uses reflection to interact with the client application and also to communicate between threads. Finally, the OmniFrwks are a very large collection of base libraries providing common functionality to OmniGroup applications—including the widely used OmniGraffle application. Our three application benchmarks are Vienna, an RSS newsreader; Skim, a PDF reader; and Adium, an instant message chat client. The OmniFrwks are noteworthy because they are very large and have been in continuous development since 1997. The fact that our tool can run on them provides evidence for the kind of real-world Objective-C that we can handle—a codebase that would be challenging for a purely symbolic analysis.

Benchmark	Total Annotations / Per Reflective Site	Symbolic Annotations / Per Reflective Site
OAuth	5 / 0.71	0 / 0.00
SCRecorder	9 / 0.75	4 / 0.33
ZipKit	0 / 0.00	0 / 0.00
Sparkle	0 / 0.00	0 / 0.00
ASIHTTP	2 / 0.03	2 / 0.03
OmniFrwks	49 / 0.26	2 / 0.01
Vienna	24 / 0.13	4 / 0.02
Skim	7 / 0.03	0 / 0.00
Adium	40 / 0.07	0 / 0.00
Combined	136 / 0.10	12 / 0.01

Table 4.2: Annotation Burden.

4.6.3 Prototype Performance

In this section, we describe our prototype's performance on a benchmark suite in terms of (1) the developer cost to add annotations for modular reflection checking; (2) the improvement in precision over Deputy-style flow-insensitive checking; (3) whether our key conjectured premises about almost-everywhere invariants hold; and (4) the cost in running time of the analysis.

The developer cost to add modular reflection checking. To measure the developer cost of adding modular reflection checking, we seeded potential type errors by first annotating the reflection requirements on 76 system library functions (i.e., with respondsTo refinements). These are requirements imposed by the system API enriched to check for method reflection errors.

Then, we added annotations to quiet as many alarms as possible. We characterize these annotations in Table 4.2. The 'Total Annotations" column lists the total number of annotations and the average number of annotations required per reflective callsite, while the "Symbolic Annotations" column lists gives the number of symbolic summaries required, in total and per reflective call site. These summaries expose the storage for getters and setters without exposing their implementation with abstract predicates [82]. The annotation column values give an indication of how much work it would be for developers to modularly check their use of reflection. We do not include annotations on the system library in these numbers. All annotations are **checked**—they emit a static type error if their requirements are not met.

We observe that our benchmarks fall into three categories, depending on how they use reflection. **Clients** of reflective APIs, such as Sparkle and ZipKit, have a very low (essentially zero) annotation burden. In contrast, benchmarks that **expose** reflective interfaces, such as SCRecorder and OAuth have a higher annotation burden. This is perhaps not surprising, since annotations are the mechanism through which interfaces expose requirements to clients. In the middle are those that use reflection in both ways: parts of OmniFrwks do expose a reflective API, but they also use internal reflection quite significantly. Our application benchmarks also fall in this category: they are structured into modular application frameworks and a core application client.

Over our entire benchmark suite we find that we need 0.10 annotations and 0.01 symbolic summaries per reflective callsite (row “Combined,” columns “Total Annotations” and “Symbolic Annotations”). In other words, on average, the programmer should expect to write one annotation for every 10 uses of reflection and a single symbolic heap effect summary for every 100 uses of reflection. Importantly, note that almost all of our annotations are extremely lightweight refinement annotations, like respondsTo—only 0.05% of methods required a symbolic summary. Even there, the summaries were very simple because they were on leaf methods, such as setters. This overall low annotation burden highlights a key benefit of our optimistic mostly flow-insensitive approach: whenever the reflection relationship is preserved flow-insensitively, no method summary is needed. Contrast this to a modular flow-sensitive approach where a summary is needed on all methods to describe their potential effects on reflection-related fields.

Improved precision. We verified reflection safety on our benchmarks using two configurations: a completely flow-insensitive analysis (no switching) and our mixed Fissile approach. Table 4.3 compares the number of static type errors reported by each. “Check Sites” gives the number of program sites where some annotation was checked; “FI Type Errors” indicates the number of check sites where a flow-insensitive type analysis produces a type error; “Fissile Type Errors” indicates the number of check sites where we emit a static type error and the corresponding percent reduction from the flow-insensitive approach. Fissile type analysis sometimes significantly reduces

Benchmark	Check	FI	Fissile	Successful	Max.	Analysis Time
	Sites	Type	Type Errors	Symbolic	Mats.	(Rate)
		Errors	(% Reduced)	Sections		
OAuth	11	7	2 (- 71%)	7	1	0.24s (5.3 kloc/s)
SCRecorder	15	2	0 (-100%)	2	2	0.28s (10.8 kloc/s)
ZipKit	28	0	0 (–)	0	0	0.10s (33.0 kloc/s)
Sparkle	40	4	1 (- 75%)	3	1	0.67s (7.9 kloc/s)
ASIHTTP	68	50	10 (- 80%)	59	2	0.50s (27.2 kloc/s)
OmniFrwks	259	82	74 (- 10%)	9	1	4.25s (37.8 kloc/s)
Vienna	207	59	38 (- 36%)	28	2	2.79s (13.4 kloc/s)
Skim	212	43	43 (- 0%)	0	0	2.49s (24.1 kloc/s)
Adium	648	87	70 (- 20%)	17	1	8.79s (20.1 kloc/s)
Combined	1488	334	238 (- 29%)	125	2	20.09s (23.0 kloc/s)

Table 4.3: Precision, Premises, and Performance.

the number of static type alarms (e.g., ASIHTTP)—and by 29% in our combined benchmark suite.

The number of Fissile static type alarms ranges from 0 (for SCRecorder and ZipKit) to 74 (for OmniFrwks, our most challenging benchmark). Pessimistically viewing our tool as a post-development analysis, we manually triaged all the reported static type errors to determine if they could manifest at run-time as true bugs (see discussion on bugs below) or otherwise are false alarms due to static over-approximation. The single biggest source of false alarms were reflection calls on objects pulled from collection classes. Retrofitting Objective-C's underlying type system for parametric polymorphism (such has been done for Java with generics) would directly improve precision for this case. At the same time, as discussed below, the efficiency of Fissile makes it feasible to instead consider it as a development-time type checker where a small number of code rewritings or cast insertions are not unreasonable (especially if most casts would go away altogether with generic types).

Premises. We designed Fissile type analysis around two core premises (Section 3.2): (1) that most of the program can be checked flow-insensitively and (2) that even when a flow-insensitive relationship between heap storage locations is violated, most other relationships on the heap remain intact. Table 4.3 shows the result of our investigation into these premises on our benchmark suite. "Successful Symbolic Sections" gives the number of times our analysis successfully switched from type checking to symbolic execution and back again, while “Max. Mats.” gives the maximum number of materialized objects ever present in the explicit heap (this includes unsuccessful symbolic sections). These results indicate that the number of times the analysis switches to symbolic execution and back is quite low, even for large programs—Premise 1 appears to hold empirically. The maximum number of simultaneous materializations is also low—Premise 2 holds as well empirically. Note that we need more than the single materialization that would be possible with a non-disjunctive flow-sensitive analysis.

Modular reflection checking at interactive speeds. Our two core premises hold, enabling FISSILE type analysis to soundly verify “almost everywhere” invariants quickly. The “Analysis Time” column in Table 4.3 indicates the speed of our analysis on each benchmark, in both absolute terms and in lines of code per second. Analysis times range from less than a second for our smaller (around 1,000 lines of code) benchmarks to around 9 seconds (for our largest, about 180,000 lines of code). These results include only the time to run our analysis: they do not include parsing or clang’s base type checker. Our goal with these measurements is to determine the additional compile-time penalty a developer would incur when adding our analysis to her existing work-flow. Expressed as a rate (thousands of lines of code per second), our analysis ranges from about 5 kloc/s to around 38 kloc/s, with a weighted average of 23.0 kloc/s. In general, the larger benchmarks show a faster rate because they amortize the high cost of checking system headers (which are typically more than 100 kloc) over larger compilation units. The “Combined” row treats all of the benchmarks together as a combined workload. Experiments were performed on a 4-core 2.6 GHz Intel Core i7 laptop with 16GB of RAM running OS X 10.8.2. We used clang 3.2 (trunk 165236) compiled in “Release+Asserts” mode to perform the analysis and xcodebuild 4.6/4H127 to drive the build.

4.6.4 Alarms

Finding bugs. When running our tool on the Vienna benchmark, we found a real reflection bug in a mature application:

```
NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
[nc addObserver:self selector:"autoCollapseFolder"
          name:"MA_Notify_AutoCollapseFolder" object:nil];
```

Here an object registers interest in being notified whenever any code in the project auto-collapses a folder. This notification takes the form of a reflective callback: the autoCollapseFolder method of self will be called. Unfortunately, self has no such method. Our analysis detects this error and issues an alarm. We reported the bug to the developers; they acknowledged it as a bug and fixed it[1] .

Our tool was also useful in finding bugs in beginner Objective-C code. We used it to statically detect run-time errors in 12 code snippets culled from mailing lists and discussion forums. These novice reflective errors fell into three different categories: (1) typos in selector names, (2) intending to reflectively call a method with a selector stored in a variable but instead passing in a constant selector with the **name** of the variable, and (3) passing the wrong responder into a reflective call, typically a field of self instead of self itself (even experts are susceptible this last kind of bug). These results show that our tool can statically detect a common class of novice errors; they provide evidence in favor of including reflective call checking with Fissile type analysis in the compiler.

False Alarms. The false alarm rate of our analysis ranges from 0% (for SCRecorder and ZipKit) to as high as 30% (for OmniFrwks), shown in Table 4.3. We characterize the major sources of these false alarms in Table 4.4. The two largest sources of false alarms are the tool's handling of library collections classes (the "Collections" column) and the inability to express dependent type constraints that cross class boundaries ("Cross-Kind"). We will discuss these two sources in more detail below. The other specific sources of false alarms are language constructs, such as switch statements, that our prototype tool cannot handle ("Unhandled Construct") and a missing annotation on a reflective library class. The "Uncategorized" column gives the number of alarms that we did not specifically categorize—typically because doing so would have required determining all the targets of dynamically dispatched methods. It is possible that many of these uncategorized alarms could be eliminated by extending the tool to either (1) make a whole-program assumption or (2) support much richer method summaries and summary verification. Both of these extensions

[1] https://github.com/ViennaRSS/vienna-rss/pull/85

Benchmark	Collections	Cross-Kind	Unhandled Construct	Missing Annotation	Uncategorized
OAuth	2 (100%)	-	-	-	-
Sparkle	-	-	-	-	1 (100%)
ASIHTTP	1 (10%)	-	-	-	9 (90%)
OmniFrwks	52 (70%)	4 (5%)	2 (3%)	1 (1%)	15 (20%)
Vienna	8 (21%)	4 (12%)	-	0	26 (76%)
Skim	23 (53%)	14 (33%)	2 (5%)	-	4 (9%)
Adium	41 (59%)	-	-	-	29 (41%)
Combined	127 (53%)	22 (9%)	4 (2%)	1 (0%)	84 (35%)

Table 4.4: Characterization of false alarms under Fissile type analysis.

are possible—but not without losing the key benefits of our approach: modular analysis and easy-to-write annotations.

Collections-Related False Alarms. The single biggest source of false alarms (53% on our combined workload) are reflective calls over collections classes, in which collections reflectively call a passed-in selector over all of the objects they contain:

```
[array makeObjectsPerformSelector:selector]
```

In this case, ideally, we would allow the user to refine the types of objects stored in the collection to restrict them so that they respond to `selector`. Unfortunately, our implementation strategy of building on Objective-C's base type system is a hindrance in this case because Objective-C does not support parametric polymorphism or generics. In order to take this approach we would first have to implement generics for Objective-C—a large undertaking. We believe this would be a relatively straight-forward (although labor intensive) process and would remove most of the collections-related false alarms. (Clearly, we would have to implement it to be sure.)

Cross-Kind–Related Alarms. Another major source of false alarms (9%) comes from a limitation of our approach to keeping types as simple as possible: we only permit checking of relationships between storage locations "of the same kind": that is, parameters refinements can only refer to other parameters, locals refinements to other locals, and field refinements to other fields of the **same** object. This means that we cannot express that, e.g., `o->field` responds to `sel` in the following example drawn from ASIHTTP:

```
- (void)performSelector:(String *)sel onTarget:(Foo *)o {
        [o->field performSelector:sel];
}
```

We similarly do not allow refinement relationships involving local variables whose address is taken (as these are, effectively, on the the global heap). We believe that investigating these "cross-kind" relationships—perhaps with some kind of ownership types—could be a fruitful area of future research.

4.7 Related Work

Refinement Types. Dependent refinement types [46, 106] enable programmers to restrict types based on the value of program expressions and thus rule out certain classes of run-time errors, such as out-of-bounds array accesses. Extending dependent types to imperative languages [21, 99, 105] has generally led to flow-sensitive type systems because mutation may change the value of a variable referred to in a type. The high burden that flow-sensitive type annotations impose on the programmer motivates sophisticated inference schemes [90], of which CSolve [91] is perhaps the closest work to ours. In contrast to CSolve, which performs flow-sensitive checking of inferred flow-sensitive types with at most one materialization, we use path-sensitive checking of flow-insensitive annotations [27] and support arbitrary materialization with a disjunctive symbolic analysis, as opposed to proving non-aliasing for one materialization (e.g., [3, 5, 41]). DJS [24] checks dependent refinements in JavaScript, including the safety of dynamic field accesses—a problem similar to reflective method call safety—but supports only single materialization and employs a flow-sensitive heap.

Symbolic Execution. Symbolic execution [66] is a precise path-by-path program exploration technique that is primarily used in the context of bug finding. Because of today's fast SMT solving technology, there has been a recent explosion in techniques (e.g., [17, 48, 50, 93]) that have significantly improved the effectiveness of symbolic execution. The SMPP approach [54] leverages SMT technology combined with abstract interpretation on path programs to lift a symbolic-execution–based technique to exhaustive verification. This technique can be seen as applying a fixed one level of analysis switching between a top-level symbolic executor and an abstract interpreter for loops. In contrast, our technique alternates between symbolic execution and type analysis on demand (starting with a top-level type analysis). Our approach of switching between type checking and symbolic execution is similar to the MIX system [64] for simple types. A significant difference is that our approach enables the symbolic executor to leverage the heap-consistency invariant enforced by the type analysis through a type-consistent materialization operation, which is critical for our rich refinement relationship invariants, whereas the symbolic and type analyses in MIX interact minimally with respect to the heap.

Object Invariants. The notion of temporary violations of an invariant is also reminiscent of the large body of work on object invariants (see [39] for an overview). We remark on two perspective differences that make FISSILE complementary to this work. First, the points where the invariant is assumed and where they may be violated is not based on the program structure (e.g., inside a method or not) but instead is based on the analysis being applied (i.e., type or symbolic). Second, the symbolic analysis takes a more global view of the heap and decides specifically which objects may violate the global type invariant. Issues like reentrancy and multi-object invariants are not as salient in FISSILE, but are possible at the cost of separate symbolic summaries or more expensive, disjunctive analysis in certain complex situations.

Separation Logic. On materialized heap locations, our symbolic analysis works over separation logic [9, 88] formulas. We define an on-demand materialization [92] that is universal in separation-logic–based analyzers [8, 22, 38, 52, 73]. However, our materialization operator pulls out heap cells that are summarized and validated **independently** using a refinement type analysis. Bi-abductive shape analyses [19, 51] are modular analyses that try to infer a symbolic summary for each method. Our analysis is modular using a fast, flow-insensitive type analysis with few uses of symbolic summaries. Bi-abduction and our technique could complement each other nicely in that (1) we do not require symbolic summaries on all methods—only those that violate type consistency across method boundaries—and (2) bi-abduction could be applied to generate candidate symbolic summaries.

Reflection. Most prior work on reflection analysis has focused on whole-program **resolution**: determining, at a reflective site, what method is called (either statically [16, 23, 103] or dynamically [14, 47]). We address the problem of modular static checking of reflective call safety: ensuring that the receiver responds to the selector, in languages with imperative update. Politz et al. [86] describe a type system that modularly checks reflection safety by combining occurrence typing [101] with first-class member names specified by string patterns. In contrast, we treat the "responds-to" **relationship** as first-class (i.e., we permit the user to specify it with a dependent refinement), allowing us to (1) check relationships between mutable fields and (2) express that an object responds to two completely unknown (i.e. potentially identical) selectors. Livshits et al. [72] **assume** reflection safety and leverage this assumption to improve precision of callgraph construction.

Chapter 5

Type-Intertwined Framing with Gated Separation

In the previous chapter, I presented a type-intertwined symbolic analysis that could both leverage and selectively violate a global type invariant by materializing and summarizing from what we called the almost type-consistent heap. Even though this symbolic analysis reasoned about the heap with separation-logic-style heap invariants, it could not soundly apply a key benefit of separation logic: local reasoning about the heap via the frame rule. Unfortunately—as I described in Section 3.3—the traditional frame rule is not sound for type-intertwined analyses.

In this chapter, I describe an extension of separation logic with a new form of spatial conjunction—**gated separating conjunction**—that allows sound type-intertwined framing. In Section 5.1, I formally characterize gated separating conjunction by defining its concretization and providing key axiom schemata, with a particular focus on how it interacts with traditional separating conjunction. In Section 5.2, I show how gated separation can be incorporated into a standard separation-logic-based program logic (i.e., one without type-intertwining)—including how gated separation interacts with heap allocation and mutation—and demonstrate that it obeys its own, gate-separated version of the frame rule. Finally, in Section 5.3, I present a fully type-intertwined separation logic, including a version of the gated frame rule that is sound for almost type-consistent heaps.

$$M ::= \mathsf{emp} \mid \widehat{a}_1 \mapsto \widehat{a}_2 \mid \mathsf{true} \mid M_1 \ast M_2 \mid \boxed{M_1 \mathbin{\triangleleft\!\!\ast} M_2} \qquad \widehat{a} \in \mathsf{SymbolicAddrs}$$

(a) Memory formulas

$$\begin{array}{rl} \text{stores} & \sigma \in \mathsf{Stores} = \mathsf{Addrs} \rightharpoonup_{\text{fin}} \mathsf{Addrs} \\ \text{valuations} & V \in \mathsf{Valuations} = \mathsf{SymbolicAddrs} \to \mathsf{Addrs} \end{array}$$

(b) Concrete memory

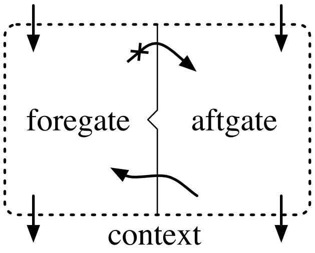
$$\boxed{\sigma \vDash_V M}$$

$$[\,] \vDash_V \mathsf{emp} \qquad [V(\widehat{a}_1) \mapsto V(\widehat{a}_2)] \vDash_V \widehat{a}_1 \mapsto \widehat{a}_2 \qquad \sigma \vDash_V \mathsf{true}$$

$$\begin{array}{rl} \sigma \vDash_V M_1 \ast M_2 & \text{iff } \sigma = \sigma_1 \cup \sigma_2 \text{ for some } \sigma_1, \sigma_2 \\ & \quad \text{where } \sigma_1 \vDash_V M_1 \text{ and } \sigma_2 \vDash_V M_2 \\ & \quad \text{and } \mathrm{dom}(\sigma_1) \cap \mathrm{dom}(\sigma_2) = \emptyset \end{array}$$

$$\begin{array}{rl} \sigma \vDash_V M_1 \mathbin{\triangleleft\!\!\ast} M_2 & \text{iff } \sigma = \sigma_1 \cup \sigma_2 \text{ for some } \sigma_1, \sigma_2 \\ & \quad \text{where } \sigma_1 \vDash_V M_1 \text{ and } \sigma_2 \vDash_V M_2 \\ & \quad \text{and } \mathrm{dom}(\sigma_1) \cap \mathrm{dom}(\sigma_2) = \emptyset \\ & \quad \boxed{\text{and } \mathrm{rng}(\sigma_1) \cap \mathrm{dom}(\sigma_2) = \emptyset} \end{array}$$

(c) Memory concretization (via model relation)



(d) Graphical representation of gated separation

Figure 5.1: Gated separating conjunction ($\mathbin{\triangleleft\!\!\ast}$) is a non-commutative strengthening of separating conjunction that additionally constrains the range of one sub-heap to be disjoint from the domain of the other. Shaded regions indicate differences with standard separating conjunction ($\ast$).

5.1 Gated Separation

Gated separating conjunction is a non-commutative strengthening of traditional separating conjunction that separates a heap into two disjoint sub-heaps: a **foregate** and an **aftgate**. The additional, crucial constraint is that that the foregate sub-heap must not directly point into the aftgate sub-heap—although the aftgate may point back into the foregate. As we described in Section 3.3, this stronger separation between foregate and aftgate enables a type-intertwined frame rule to grant the type analysis access to the foregate portion of the heap and guarantee that the aftgate remains untouched. In this section, we focus on gated separation itself—we will formally describe the details of the type-intertwined frame rule in Section 5.3.3. Here, we define the syntax and concretization of gated separating conjunction (Section 5.1.1) for a simpler formula language. We describe key axioms for this operator, showing both its similarities and differences with standard separating conjunction, as well as how the two conjunctive operators interact, in Section 5.1.2.

5.1.1 Memory Formulas and Concretization

Figure 5.1 describes the syntax and concretization of gated separation for a simple separation logic in which addresses point only to single values. We employ this simple model of memory for explanatory purposes—as we will show in Section 5.1.2, gated separation is also applicable to more complex memory representations. Figure 5.1a gives the syntax of memory formulas: a formula can be empty emp; a single heap cell $\widehat{a}_1 \mapsto \widehat{a}_2$ with (symbolic) address $\widehat{a}_1$ storing address $\widehat{a}_2$; an arbitrary heap true (which we include as the simplest memory formula that is not precise; or a separating conjunction of sub-heaps $M_1 * M_2$. Lastly and most importantly, it can be a gated separating conjunction $M_1 \lessdot\!\!\!\lhd M_2$ (shown shaded, for emphasis), which separates a foregate M_1 from an aftgate M_2. We use the symbol $\lessdot\!\!\!\lhd$ to connote that the aftgate sub-heap can directly point into the foregate but not the other way around.

We define concretization for gated separation (Figures 5.1b and 5.1c) in terms of a model relation. For our explanatory model of memory, a concrete store σ is a finite map from addresses to addresses (which in this model are the only form of values) and a valuation (or interpretation) V maps symbolic addresses to concrete addresses. Here a relation of the form $\sigma \vDash_V M$ says that a store σ is a model for the formula M under valuation V—so the empty formula emp is modeled by the empty store; a singleton formula $\widehat{a}_1 \mapsto \widehat{a}_2$ is modeled by a store with exactly one cell, mapping the valuation of $\widehat{a}_1$ to the valuation of $\widehat{a}_2$; and true is modeled by any store. Our concretization of separating conjunction is entirely standard: a store σ is a model for $M_1 * M_2$ iff σ is the union of two stores σ_1 and σ_2 that are models of M_1 and M_2 respectively and that have disjoint domains—that is, the addresses of the stores in σ_1 and σ_2 are distinct.

The concretization of gated separating conjunction ($\mathrel{\triangleleft\!\!\!\ast}$) is a non-commutative strengthening of that for normal separating conjunction: in addition to the usual disjoint domain restriction on the store, we require that the **range** of the left sub-heap (the foregate) be disjoint from the **domain** of the right sub-heap (the aftgate). In other words, the foregate must not **directly** point into the aftgate (but pointers in the other direction are allowed). This "gate" between the foregate sub-heap and aftgate sub-heap enables the type-intertwined frame rule—it protects the aftgate sub-heap from type-intertwined interference (Section 5.3.3). Also crucially for local reasoning, the gated separation constraint is not too strong: it does not restrict the foregate from reaching the aftgate via a **third** disjoint sub-heap (which we sometimes call the **context**). We graphically illustrate the gated separation constraint in Figure 5.1d. There, the normal arrows show allowed pointers, while the crossed-out arrow indicates the dis-pointing relationship between foregate and aftgate.

5.1.2 Axioms of Gated Separation

Gated separating conjunction is similar in many ways to standard separating conjunction, but it also differs in key respects. We give the key axiom schemata characterizing gated separating conjunction—and in particular describing how it interacts with normal separating conjunction—in Figure 5.2. Here we write $M_1 \Rightarrow M_2$ to mean that for all stores σ and valuations V, $\sigma \vDash_V M_1$ implies $\sigma \vDash_V M_2$.

Gated separating conjunction shares many properties with standard separating conjunction.

(1) Neutral and Absorbing Elements

$$\mathsf{emp} \mathbin{\lhd\!\!\!\ast} M \Leftrightarrow M \qquad M \mathbin{\lhd\!\!\!\ast} \mathsf{emp} \Leftrightarrow M$$

$$\mathsf{true} \mathbin{\lhd\!\!\!\ast} M \Rightarrow \mathsf{true} \qquad M \mathbin{\lhd\!\!\!\ast} \mathsf{true} \Rightarrow \mathsf{true}$$

(2) Associativity

$$(M_1 \mathbin{\lhd\!\!\!\ast} M_2) \mathbin{\lhd\!\!\!\ast} M_3 \Leftrightarrow M_1 \mathbin{\lhd\!\!\!\ast} (M_2 \mathbin{\lhd\!\!\!\ast} M_3)$$

(3) $\lhd\!\!\!\ast$ Weakening

$$M_1 \mathbin{\lhd\!\!\!\ast} M_2 \Rightarrow M_1 \ast M_2$$

(4) Foregate Shrinking

$$(M_1 \ast M_2) \mathbin{\lhd\!\!\!\ast} M_3 \Rightarrow M_1 \ast (M_2 \mathbin{\lhd\!\!\!\ast} M_3)$$

(5) Aftgate Shrinking

$$M_1 \mathbin{\lhd\!\!\!\ast} (M_2 \ast M_3) \Rightarrow (M_1 \mathbin{\lhd\!\!\!\ast} M_2) \ast M_3$$

(6) Gate Partitioning

$$(M_1 \ast M_2) \mathbin{\lhd\!\!\!\ast} (M_3 \ast M_4) \Rightarrow (M_1 \mathbin{\lhd\!\!\!\ast} M_3) \ast (M_2 \mathbin{\lhd\!\!\!\ast} M_4)$$

(7) $\ast$ Strengthening

$$K[M_1 \ast M_2] \Rightarrow K[M_1 \mathbin{\lhd\!\!\!\ast} M_2] \qquad \text{if } \overline{\text{outptrs}}(M_1) \sqsubseteq \underline{\text{domaddrs}}(K)$$

(8) Aftgate Strengthening

$$M \mathbin{\lhd\!\!\!\ast} (K[M_1 \ast M_2]) \Rightarrow M \mathbin{\lhd\!\!\!\ast} (K[M_1 \mathbin{\lhd\!\!\!\ast} M_2]) \quad \text{if } \overline{\text{outptrs}}(M_1) \sqsubseteq \underline{\text{addrs}}(M)$$

$$\begin{array}{lcll} K & ::= & \bullet \mid K \olessthan M \mid M \olessthan K & \text{memory contexts} \\ \olessthan & ::= & \ast \mid \lhd\!\!\!\ast & \text{separating conjunctions} \end{array}$$

$$\begin{array}{rr} \overline{\text{outptrs}} : \mathsf{Formulas} \to \mathcal{P}_{\text{fin}}(\mathsf{SymbolicAddrs})^{\top} & \overline{\text{outptrs}}(\widehat{a}_1 \mapsto \widehat{a}_2) \triangleq \{\widehat{a}_2\} \\ \underline{\text{domaddrs}} : \mathsf{Formulas} \to \mathcal{P}_{\text{fin}}(\mathsf{SymbolicAddrs})^{\top} & \underline{\text{domaddrs}}(\widehat{a}_1 \mapsto \widehat{a}_2) \triangleq \{\widehat{a}_1\} \\ \underline{\text{addrs}} : \mathsf{Formulas} \to \mathcal{P}_{\text{fin}}(\mathsf{SymbolicAddrs})^{\top} & \underline{\text{addrs}}(\widehat{a}_1 \mapsto \widehat{a}_2) \triangleq \{\widehat{a}_1, \widehat{a}_2\} \end{array}$$

Figure 5.2: Axiom Schemata of Gated Separation

Like $*$, the gated separation operator $\mathbin{\triangleleft\!\!\ast}$ has emp and true as neutral and absorbing elements, respectively (Schema 1). Also, like separating conjunction, the gated version is associative (Schema 2). Gated separation is similarly monotone with respect to implication. That is, it supports the inference rule:

$$\frac{M_1 \Rightarrow M_1' \qquad M_2 \Rightarrow M_2'}{M_1 \mathbin{\triangleleft\!\!\ast} M_2 \Rightarrow M_1' \mathbin{\triangleleft\!\!\ast} M_2'}$$

Unlike normal separating conjunction, however, the restriction gated separating conjunction imposes on the left sub-heap differs from that imposed on the right—gated separation is not commutative:

$$M_1 \mathbin{\triangleleft\!\!\ast} M_2 \not\Rightarrow M_2 \mathbin{\triangleleft\!\!\ast} M_1$$

Extending separation logic with gated separating conjunction yields an ordered logic, in which the exchange rule (for gate-separated conjuncts) is inadmissible. Note that rearranging $*$-separated conjuncts **within** a given branch of a gated separating conjunction is allowed, by the monotone property of $\mathbin{\triangleleft\!\!\ast}$ and the commutativity of $*$.

Weakening Gated Separation. Gated separating conjunction can always be weakened to normal standard conjunction (Schema 3). This property is evident from the definition given in Figure 5.1c: the concretization of gated separating conjunction is identical to that for standard separating conjunction, except that it adds the additional range restriction on the context. When gated separating conjunction interacts with traditional separating conjunction, we can also always **selectively** loosen the gated separation constraint. That is, it is safe push in $\mathbin{\triangleleft\!\!\ast}$ over $*$ to shrink the foregate (Schema 4), shrink the aftgate (Axiom 5), or partition the gate (Schema 6) to create two separate foregates and aftgates (the analogous pushing-out is not always safe).

Strengthening Standard Separation. Ordinary separating conjunction can sometimes be strengthened to gated separating conjunction—but doing so requires additional auxiliary information. Following the concretization, we can strengthen $M_1 * M_2$ to $M_1 \mathbin{\triangleleft\!\!\ast} M_2$ if we can prove that the range in any concretization of M_1 is disjoint from the domain of any concretization of M_2. Let us define an **out-pointer** of a memory M as an address in the range of (a concretization of)

M that is not also in the domain of (that concretization of) M. Then to strengthen $M_1 * M_2$ to $M_1 \mathbin{\triangleleft\!\!\!*} M_2$, it is sufficient to show that any out-pointer in M_1 **is not** in the domain of M_2 because the domains of M_1 and M_2 are known to be disjoint by $*$. But from an abstraction perspective, reasoning directly about address disequality is expensive and negates a key benefit of separation logic.

Instead, Schema 7 shows an indirect way of deriving this condition. Conceptually, we can strengthen $*$ with $\mathbin{\triangleleft\!\!\!*}$ if we can show that every out-pointer of the potential foregate **must point somewhere other than** into the aftgate. We write $\overline{\mathrm{outptrs}}(M)$ for an over-approximation of the out-pointers of M in terms of symbolic addresses and $\underline{\mathrm{domaddrs}}(M)$ for an under-approximation of the domain addresses of M. For the moment, let us ignore over- and under-approximations, we will explain the need for approximation in more detail a bit later. We define K to be a memory context (i.e., a memory formula with a hole), write $K[M]$ for the syntactic plugging of M for the hole $\bullet$ in K, and lift $\underline{\mathrm{domaddrs}}(K) \triangleq \underline{\mathrm{domaddrs}}(K[\mathsf{emp}])$. So the side-condition of Schema 7 says that we can strengthen $*$ to $\mathbin{\triangleleft\!\!\!*}$ if we can show that the out-pointers of the potential foregate M_1 are addresses of cells in the surrounding context K (and thus not addresses of cells in the potential aftgate M_2).

Schema 8 shows the analogous way of deriving the gated condition to strengthen a $*$ to a $\mathbin{\triangleleft\!\!\!*}$ by leveraging the stronger constraint of a gated separation in the surrounding context: if an over-approximation of the out-pointers of M_1 is fully contained in an under-approximation of the addresses in either the domain or range of the outer foregate M (written $\underline{\mathrm{addrs}}(M)$), then M_1 can be made into the foregate for M_2. This axiom is sound because any concrete address in either the domain or the range of the outer foregate M is guaranteed not to be in the domain of M_2 (which is part of the outer aftgate).

As we will see in Section 5.2, it is often useful to allocate fresh storage in the aft of a top-level gate (this is safe because fresh memory, by definition, cannot be pointed-to by any other memory), initialize that storage, and then "evict" that storage into the foregate. Because the evicted storage may point to un-evicted locations still in the aftgate, this eviction is not safe in general. In terms of formulas, eviction is the replacing of $M \mathbin{\triangleleft\!\!\!*} (M_1 * M_2)$ with $(M * M_1) \mathbin{\triangleleft\!\!\!*} M_2$, which is safe when $\overline{\text{outptrs}}(M_1) \sqsubseteq \underline{\text{addrs}}(M)$. Note that eviction is derivable from Aftgate Strengthening (8), Associativity (2), and $\lessdot$ Weakening (3).

Soundness. Unfortunately, we have to consider over- and under-approximations in the above because of the presence of summaries in formulas. As a simple example of a summary, the formula true represents any heap, so neither the outside symbolic addresses that the formula points to nor the symbolic addresses of its domain can be syntactically determined. In this case, the worst-case–scenario must be considered. That is, in Schemas 7 and 8, we should over-approximate the set of out-pointers of the potential foregate M_1 and under-approximate the set of addresses in the surrounding context (respectively, K and M).

These address approximation functions return either a finite set of symbolic variables or $\top$, written $\mathcal{P}_{\text{fin}}(\mathsf{SymbolicAddrs})^{\top}$. We can view this set $\mathcal{P}_{\text{fin}}(\mathsf{SymbolicAddrs})^{\top}$ as a join semi-lattice with the usual lattice operations $\sqcup$ and $\sqsubseteq$ defined as union and inclusion for finite subsets lifted with $\top$. We give meaning to the address approximation functions in the following soundness criteria:

Condition 3 (Soundness of Address Approximation Functions)**.** The address approximation functions $\overline{\text{outptrs}}$, $\underline{\text{addrs}}$, and $\underline{\text{domaddrs}}$ are **sound** if for all memory formulas M we have $\sigma \vDash_V M$ implies:

(1) $\text{rng}(\sigma) \setminus \text{dom}(\sigma) \subseteq [\![\overline{\text{outptrs}}(M)]\!]^V$; and

(2) $[\![\underline{\text{addrs}}(M)]\!]^V \subseteq \text{dom}(\sigma) \cup \text{rng}(\sigma)$; and

(3) $[\![\underline{\text{domaddrs}}(M)]\!]^V \subseteq \text{dom}(\sigma)$

where $[\![\cdot]\!]^V : \mathcal{P}_{\text{fin}}(\mathsf{SymbolicAddrs})^{\top} \to \mathcal{P}(\mathsf{SymbolicAddrs})$ is a concretization function for approximations of addresses defined as follows:

$$\begin{array}{rcl} [\![\top]\!]^V & \triangleq & \mathsf{SymbolicAddrs} \\ [\![\{\widehat{a}_1, \ldots, \widehat{a}_n\}]\!]^V & \triangleq & \{V(\widehat{a}_1), \ldots, V(\widehat{a}_n)\} \end{array}$$

Theorem 2 (The Axioms of Gated Separation are Sound)**.** *If $\overline{outptrs}$, $\underline{domaddrs}$, and $\underline{addrs}$ meet Condition 3, then Axioms 1-7 of gated separation are sound.*

Proof. Straight-forward application of definition of concretization. By cases on K for axioms 7 and 8. □

Logic Extensions. The form of these axioms are general enough that they are useful both for more complicated concrete models of memory and for more expressive formulas abstracting memory. We show in Section 5.3 how to extend them to operate over memory formulas that summarize portions of the heap with types.

We note that our axioms do not require precise reasoning about the domain or range addresses of a summary formula, which would be in conflict with the premise of summarization. Instead, for the strengthening axioms (Schemas 7 and 8), we use approximations that are not difficult to define for many commonly-used summary forms, in combination with what can derived from separation and gated separation, respectively. These address approximation functions must simply satisfy the soundness conditions (Condition 3). In the case of inductive summaries widely used in separation-logic–based inductive shape analysis [9, 22], these address approximation functions can be defined to be precise. For example, for the standard inductive predicate defining a singly-linked list starting at $\widehat{a}$—$\mathsf{list}(\widehat{a})$, we can define $\overline{\text{outptrs}}(\mathsf{list}(\widehat{a})) \triangleq \emptyset$, as there is no out-pointer from the summary. Similarly, for the standard singly-linked list segment from $\widehat{a}_1$ to $\widehat{a}_2$—$\mathsf{ls}(\widehat{a}_1, \widehat{a}_2)$—we can define precisely $\overline{\text{outptrs}}(\mathsf{ls}(\widehat{a}_1, \widehat{a}_2)) \triangleq \{\widehat{a}_2\}$, as the only out-pointer of the list segment is the endpoint $\widehat{a}_2$.

5.2 Gated Separation in a Traditional Separation Logic

As we saw in the previous section, gated separation is a strengthening of separating conjunction that can be used to show a local "dis-pointing" relationship between a foregate an aftgate. Ultimately, we will show (Section 5.3.3) that the stronger constraint of gated separation allows us to recover framing with type-intertwined analysis. In this section, we demonstrate the challenges of maintaining gated separation in the context of a traditional (i.e., not type-intertwined) separation-logic-based program logic that has been extended with gated separating conjunction. As we will see, the main

identifiers

x, y, z			global variables

commands

c	$::=$	$x = \text{alloc } y$	heap allocation
	$\mid$	$x = \langle\rangle$	write unit to global
	$\mid$	$x = *y$	heap read
	$\mid$	$*x = y$	heap write
	$\mid$	$c_1 \; ; \; c_2$	sequencing

Figure 5.3: A core imperative command language with globals and a mutable heap.

challenge is positively ensuring that heap mutation does not violate a gated separation constraint. In particular, if the memory cell being updated is in any foregate, then we must make sure that the value being written is not an address of a cell in the corresponding aftgates. At the same time, gated separation enables a stronger form of local reasoning: it affords its own gated version of the frame rule, which we discuss in Section 5.2.2.

5.2.1 Syntax

In Figure 5.3, I describe a core command language with a C-like mutable heap. For simplicity of presentation, the only values are the unit value $\langle\rangle$ and pointers. We use this simpler language, rather the reflection language from Chapter 4, because the core problem—reasoning about type consistency—requires neither object types nor dependent types.

The language has global variables (x, y, z) only—there are no local variables. A command c can be '$x = \text{alloc } y$', which allocates a fresh cell on the heap, initializes it with the contents of y and stores its address in x. The '$x = \langle\rangle$' command writes the unit value to the global variable x, while '$x = *y$' performs a heap read: it dereferences the address contained in y and stores the resultant value into the global variable x. The '$*x = y$' writes to the heap: it updates the cell pointed-to by the address in x to contain the value in y. The only control operator is sequencing '$c_1 \; ; \; c_2$', which performs c_1 followed by c_2. We assume additional control-flow statements can be defined in terms of these atomic commands in the standard way.

5.2.2 Purely Symbolic Static Semantics

We describe a static semantics for this core language in Figure 5.4. We write $\vdash \{M\}\, c\, \{M'\}$ for the judgment form defining a standard partial-correctness Hoare triple. This judgement states "If a concrete store σ satisfies M and an execution of command c from σ terminates, then the resulting store satisfies M'." Here the key challenge is ensuring that heap writes of address do not cause the written-to cell to point into a region that gated separation constrains it to not point into. Even writes to global variables can invalidate gated separation because in our model of memory global variables are cells on the heap (that is, global variable names act as addresses).

Heap Writes. A write $*x = y$ involves explicitly accessing three cells: reading the cell for y to get the value to write (the value cell $y \mapsto \widehat{\mathbf{v}}$), reading the cell for x to find the cell to write to (the pointer-read cell $x \mapsto \widehat{a}$), and writing to the cell pointed to by x (the write cell $\widehat{a} \mapsto -$); for clarity, we highlight the value being written in bold $\widehat{\mathbf{v}}$—although not italic, this symbol is still a meta-variable. With gated separation, there is an implicit constraint on a conceptual fourth cell if $\widehat{\mathbf{v}}$ is an address: the cell whose address is the value being written (the constrained cell $\widehat{\mathbf{v}} \mapsto -$). In particular, the constraint is that it cannot be in an aftgate behind the write cell. But proving directly that the constrained cell is not any aftgate behind the write cell is at best a very expensive global case analysis, negating a key benefit of separation logic. Instead, we follow the perspective of the strengthening axioms (Schemas 7 and 8) from Section 5.1.2 in proving positively that the constrained cell is in some other sub-heap.

The S-Heap-Write-FoundCellElsewhere-Com heap write rule is similar to Schema 7. In order to write the value $\widehat{\mathbf{v}}$, the rule requires that cells for the two global variables are on the heap (i.e., $x \mapsto \widehat{a}, y \mapsto \widehat{\mathbf{v}} \in M$). Then, we can perform the write if we can decompose M to find two sub-heaps M_1 and $K_2[\widehat{a} \mapsto -]$ where (1) the cell pointed-to by the value being written (the potentially constrained cell) is definitely in M_1 (via checking $\widehat{\mathbf{v}} \in \underline{\text{domaddrs}}(M_1)$) and (2) constrained cell is either in a foregate $\lessdot$ of or disjoint $*$ from the write cell ($\widehat{a} \mapsto -$). This is sound because it ensures that there is no way the constrained cell can be in a foregate with respect to the write cell. Note

Heap writes $\boxed{\vdash \{M\}\, c\, \{M\}}$

S-Heap-Write-FoundCellElsewhere-Com

$$\frac{M = K_1[M_1 \olessthan K_2[\widehat{a} \mapsto -]] \qquad y \mapsto \widehat{\mathbf{v}} \in M \qquad x \mapsto \widehat{a} \in M \qquad \widehat{\mathbf{v}} \in \underline{\mathrm{domaddrs}}(M_1)}{\vdash \{M\}\ {*x} = y\, \{K_1[M_1 \olessthan K_2[\widehat{a} \mapsto \widehat{\mathbf{v}}]]\}}$$

S-Heap-Write-FoundValueInForegate-Com

$$\frac{M = K_1[M_1 \mathrel{\triangleleft\!\!\!\ast} K_2[\widehat{a} \mapsto -]] \qquad y \mapsto \widehat{\mathbf{v}} \in M \qquad x \mapsto \widehat{a} \in M \qquad \widehat{\mathbf{v}} \in \underline{\mathrm{addrs}}(M_1)}{\vdash \{M\}\ {*x} = y\, \{K_1[M_1 \mathrel{\triangleleft\!\!\!\ast} K_2[\widehat{a} \mapsto \widehat{\mathbf{v}}]]\}}$$

S-Heap-Write-InSame

$$\frac{M = K[\widehat{a} \mapsto - \mathbin{\boldsymbol{*}} y \mapsto \widehat{\mathbf{v}}] \qquad x \mapsto \widehat{a} \in M}{\vdash \{M\}\ {*x} = y\, \{K[\widehat{a} \mapsto \widehat{\mathbf{v}} \mathbin{\boldsymbol{*}} y \mapsto \widehat{\mathbf{v}}]\}}$$

Heap reads

S-Heap-Read-FoundCellElsewhere-Com

$$\frac{M = K_1[M_1 \olessthan K_2[x \mapsto -]] \qquad y \mapsto \widehat{a} \in M \qquad \widehat{a} \mapsto \widehat{\mathbf{v}} \in M \qquad \widehat{\mathbf{v}} \in \underline{\mathrm{domaddrs}}(M_1)}{\vdash \{M\}\, x = {*y}\, \{K_1[M_1 \olessthan K_2[x \mapsto \widehat{\mathbf{v}}]]\}}$$

S-Heap-Read-FoundValueInForegate-Com

$$\frac{M = K_1[M_1 \mathrel{\triangleleft\!\!\!\ast} K_2[x \mapsto -]] \qquad y \mapsto \widehat{a} \in M \qquad \widehat{a} \mapsto \widehat{\mathbf{v}} \in M \qquad \widehat{\mathbf{v}} \in \underline{\mathrm{addrs}}(M_1)}{\vdash \{M\}\, x = {*y}\, \{K_1[M_1 \mathrel{\triangleleft\!\!\!\ast} K_2[x \mapsto \widehat{\mathbf{v}}]]\}}$$

S-Heap-Read-InSame

$$\frac{M = K[x \mapsto - \mathbin{\boldsymbol{*}} \widehat{a} \mapsto \widehat{\mathbf{v}}] \qquad y \mapsto \widehat{a} \in M}{\vdash \{M\}\, x = {*y}\, \{K[x \mapsto \widehat{v} \mathbin{\boldsymbol{*}} \widehat{a} \mapsto \widehat{\mathbf{v}}]\}}$$

Allocation

S-Alloc-Com

$$\frac{M = K[x \mapsto -] \qquad y \mapsto \widehat{v} \in M \qquad \widehat{a} \notin M}{\vdash \{M\}\, x = \mathrm{alloc}\ y\, \{K[\mathsf{emp}] \mathrel{\triangleleft\!\!\!\ast} \widehat{a} \mapsto \widehat{v} \mathrel{\triangleleft\!\!\!\ast} x \mapsto \widehat{a}\}}$$

Miscellaneous rules

S-WriteUnit-Com

$$\frac{M = K[x \mapsto -] \qquad \widehat{v} \notin M}{\vdash \{M\}\, x = \langle\rangle\, \{K[x \mapsto \widehat{v}]\}}$$

S-Seq-Com

$$\frac{\vdash \{M\}\, c_1\, \{M'\} \qquad \vdash \{M'\}\, c_2\, \{M''\}}{\vdash \{M\}\, c_1\ ;\ c_2\, \{M''\}}$$

Gated framing

S-GatedFrame-Com

$$\frac{\vdash \{M_{\text{fore}}\}\, c\, \{M'_{\text{fore}}\}}{\vdash \{K[M_{\text{fore}} \mathrel{\triangleleft\!\!\!\ast} M_{\text{aft}}]\}\, c\, \{K[M'_{\text{fore}} \mathrel{\triangleleft\!\!\!\ast} M_{\text{aft}}]\}}$$

Figure 5.4: The key challenge is ensuring writes do not violate gated separation.

that it is unsound to permit the write in the remaining case where the constrained cell is in a foregate with respect to the write cell (for then, the post-heap would look like $K_1[K_2[x \mapsto \widehat{\mathbf{v}}] \lessdot M_1]$ and thus would explicitly violate gated separation). This rule can be read as global rule where M encompasses the entire analysis state, though it also does not preclude framing out some context (with only a loss of precision). At the same, it is unsatisfactory from a local reasoning perspective because it would never apply if M contained only the three access cells. In essence, like Schema 7, this rule never directly leverages gated separation.

The S-Heap-Write-FoundValueInForegate-Com heap write rule does leverage gated separation, in a manner analogous to Schema 8. This rule requires a similar decomposition to that in S-Heap-Write-FoundCellElsewhere-Com. The key difference is that if M_1 is gate separated from the write cell we can safely write the value if it is **any** address in M_1. Writing such a value is safe because if the value is in the range of M_1 then there is no way the constrained cell can be in K_2—otherwise, the gated separation constraint on the pre-state would not hold. If the value is in the domain of M_1 then the same argument as S-Heap-Write-FoundCellElsewhere-Com holds. Similarly to that rule, the constraint in the other direction (where the value-to-write $\widehat{\mathbf{v}}$ appears in the aftgate with respect to the write cell) is not sound.

Finally, the S-Heap-Write-InSame rule says that a heap write is safe if the read-cell (cell from which the value will be read) and the write cell (to the cell to which the value will be written) are "adjacent" within the same "gate region". Here the fact that the read-cell and write-cell are adjacent means that it is allowable to perform the write regardless of the gating constraints in the context K—otherwise, the gated constraint would be violated in the prestate.

Heap Reads. The rules for heap reads $x = *y$ are very similar to those for heap writes—this reflects the fact that in this core language, global variables act like addresses on the heap and thus we can establish gate-separation-style dis-pointing relationships between global variable cells and other portions of the heap. The S-Heap-Read-FoundCellElsewhere-Com rule is very similar to S-Heap-Write-FoundCellElsewhere-Com except that for the read case, we must (1) have global y pointing to an address that can be dereferenced and (2) ensure that updating the global variable cell for x with the read value $\widehat{\mathbf{v}}$ does not violate a gated constraint. Each of the other two rules for heap reads, S-Heap-Read-FoundValueInForegate-Com and S-Heap-Read-InSame take the same for form as their heap write counterparts, except that the write-cells are now global variables and the read-cells are addresses on the heap.

Allocation. Even though an allocation 'x = alloc y' writes to a global variable, it does not require explicit reasoning to avoid violating gated separation because newly allocated cells are always fresh. The S-Alloc-Com rule describes the effect of an allocation. This rule requires that the heap can be broken up into a context K and a global variable cell for x where, anywhere on the heap, the global variable y contains value $\widehat{v}$. Under these conditions, allocation creates a new cell $\widehat{a} \mapsto \widehat{v}$ with fresh address $\widehat{a}$ that is gate-separated from the rest of heap. Because this address is fresh, there is no way that the rest of the heap in K can violate a gated separation constraint by pointing into $\widehat{a} \mapsto \widehat{v}$ (K cannot point into $x \mapsto \widehat{a}$ because local variable addresses cannot be stored on the heap). There is also no way that a pointer ($\widehat{v}$) out of $\widehat{a} \mapsto \widehat{v} \lessdot\!\!\lessdot x \mapsto \widehat{a}$ can violate any existing gated separation constraint inside K because the former is conjoined with K at the top level. The fact that freshly allocated allocated storage cannot be pointed into by any pre-existing memory is a useful invariant, as we discussed in Section 3.3.

Miscellaneous Rules. Like allocation, the '$x = \langle\rangle$' command does not require reasoning about gated separation. As the S-WriteUnit-Com shows, we can assign the unit value to a global variable without regard to any other cell. In this case, we (somewhat imprecisely) treat the unit value a fresh symbolic variable. Sequencing (S-Seq-Com) is entirely standard.

Gated Framing. For standard separating conjunction, the frame rule enables local reasoning about the heap by allowing an analysis or program logic to "frame out" a portion memory that is irrelevant to the command of interest. The analysis can then analyze the command on the command's footprint and be guaranteed that (1) the framed-out portion of memory could not have been accessed by the command and (2) that the portion of the heap represented by the post-state footprint is disjoint from the frame.

For modular reasoning with gated separation, it would be similarly advantageous to frame out the aftgate, analyze a command with respect to the foregate alone, and be guaranteed that (1) command has not altered the contents of the aftgate and (2) that the new foregate is prevented from directly pointing into the aftgate. Ultimately, the goal is to support a global type summary in the foregate such that for a memory gate-split into $M_{\text{fore}} \mathrel{\triangleleft\!\!\mid} M_{\text{aft}}$ we know that the global summary cannot "reach" cells in M_{aft} through cells in M_{fore}. Then, we can analyze a command with respect to the foregate footprint M_{fore}—even permitting arbitrary materialization and reduction from the global summary in M_{fore}—with type-intertwined framing. We discuss gated separating with global summaries in Section 5.3.3—here we show that gated separation supports its own version of the frame rule in a standard separation logic.

The S-GatedFrame-Com rule in Figure 5.4 describes the frame rule for gated separation in separation logic. This rule requires that the analysis can decompose the heap into three components: a context K, a foregate M_{fore}, and an aftgate M_{aft}. Then, the analysis can determine the result of executing a command c in M_{fore} alone and be guaranteed that it can replace M_{fore} with M'_{fore} in the pre-state to obtain a valid post-state over the entire heap—and in particular, that the doing so will not violate a gated separation constraint. This rule shows that, like standard separating conjunction, gated separating conjunction admits local heap reasoning.

Example 1 (Applying the Gated Frame Rule)**.** Consider the read command `x` $=$ $*$`x` (like following a link of a linked list). As above, we highlight the value being written in **bold**. In the pre-state, we have that the memory from variable `y` cannot be reached via the memory from variable `x`: for example, the gated separation $\mathrel{\triangleleft\!\!\mid}$ implies that $\widehat{a}_2 \neq \widehat{a}_3$ (in addition to $\widehat{a}_1 \neq \widehat{a}_3$ from ordinary separation). We frame out the memory from variable `y` (shown shaded).

$$\dfrac{\dfrac{}{\vdash\ \{\mathtt{x} \mapsto \widehat{a}_1 \ast \widehat{a}_1 \mapsto \widehat{\mathbf{a}}_{\mathbf{2}}\}\ \mathtt{x} = {*}\mathtt{x}\ \{\mathtt{x} \mapsto \widehat{\mathbf{a}}_{\mathbf{2}} \ast \widehat{a}_1 \mapsto \widehat{\mathbf{a}}_{\mathbf{2}}\}}}{\vdash\ \{(\mathtt{x} \mapsto \widehat{a}_1 \ast \widehat{a}_1 \mapsto \widehat{a}_2) \mathrel{\triangleleft\!\!\mid} (\mathtt{y} \mapsto \widehat{a}_3 \ast \widehat{a}_3 \mapsto \widehat{a}_4)\}\ \mathtt{x} = {*}\mathtt{x}\ \{(\mathtt{x} \mapsto \widehat{\mathbf{a}}_{\mathbf{2}} \ast \widehat{a}_1 \mapsto \widehat{a}_2) \mathrel{\triangleleft\!\!\mid} (\mathtt{y} \mapsto \widehat{a}_3 \ast \widehat{a}_3 \mapsto \widehat{a}_4)\}}$$

By the gating constraint in the pre-state, no value obtained by dereferencing in the memory from variable `x` can ever alias $\widehat{a}_3$, for instance, and thus the gating constraint holds in the post-state.

Note that if we are not concerned about type-intertwined framing (Section 5.3), we can recover the standard frame rule by replacing $\lhd\!\!\!\!\ast$ with $*$ in the above.

5.2.3 Language Requirements for Gated Framing.

Like with the frame rule for ordinary separation, gated framing cannot be applied to arbitrary commands. A command that havocs the entire heap, for example, cannot be supported with either the ordinary frame rule or the gated frame rule. For the ordinary frame rule, the command must not dereference the framed out portion of the heap. In other words, it must be able to "do the same thing" when evaluated on the footprint alone as when evaluated on the footprint and the frame. For the gated frame rule, the additional constraint is that the command must not be able to fabricate pointers "out of nothing" that happen to be addresses in the framed-out aftgate. In essence, the gate-frameable condition is that the command must be "oblivious" to the existence of the framed out portion of the heap, which is entirely reasonable in the context of well-behaved languages, such as Java and JavaScript (either statically or dynamically typed). For languages where fabrication of pointers is possible, such as C, one would have to ensure these properties for a given command statically as part of the analysis.

Concrete Semantics. I give the semantics for the core command language from Section 5.2.1 as a big-step semantics in Figure 5.5 and show (Lemma 10) that the language meets the requirements for gated framing. For simplicity in presentation, we consider global program variables to also be addresses (i.e., $\mathsf{Vars} \subseteq \mathsf{Addrs}$), that is, program variables in memory formulas conceptually stand for their respective allocation addresses. Here, a judgment of the form $\vdash \langle\sigma\rangle\, c\, \langle\sigma'\rangle$ says that in pre-state σ command c big-step evaluates to post-state r where r is either a concrete post-state σ' or an error state err.

These rules are quite standard. The notation $\sigma[a : v]$ indicates the result of updating a store σ with a binding from address a to value v. The C-WriteUnit-Com rule, for example, says that the effect of command writing the unit value $\langle\rangle$ to a global variable x should be to update the binding for x in the prestate store with the unit value (recall that we treat global variables as a special class of addresses on the heap). The C-Alloc-Com rule describes allocation: it says that an allocation 'x

$\boxed{\vdash \langle\sigma\rangle\, c\, \langle r\rangle}$

C-WriteUnit-Com
$$\frac{x \in \mathrm{dom}(\sigma) \qquad \sigma' = \sigma[x : \langle\rangle]}{\vdash \langle\sigma\rangle\, x = \langle\rangle\, \langle\sigma'\rangle}$$

C-Alloc-Com
$$\frac{x \in \mathrm{dom}(\sigma) \qquad a \notin \sigma \qquad \sigma' = \sigma[a : \sigma(y)][x : a]}{\vdash \langle\sigma\rangle\, x = \mathrm{alloc}\ y\, \langle\sigma'\rangle}$$

C-Heap-Read-Com
$$\frac{x \in \mathrm{dom}(\sigma) \qquad \sigma' = \sigma[x : \sigma(\sigma(y))]}{\vdash \langle\sigma\rangle\, x = *y\, \langle\sigma'\rangle}$$

C-Heap-Write-Com
$$\frac{\sigma' = \sigma[\sigma(x) : \sigma(y))]}{\vdash \langle\sigma\rangle\, *x = y\, \langle\sigma'\rangle}$$

C-Seq-Com
$$\frac{\vdash \langle\sigma\rangle\, c_1\, \langle\sigma'\rangle \qquad \vdash \langle\sigma'\rangle\, c_2\, \langle\sigma''\rangle}{\vdash \langle\sigma\rangle\, c_1\ ;\ c_2\, \langle\sigma''\rangle}$$

Error Cases

C-WriteUnit-Err-Com
$$\frac{x \notin \mathrm{dom}(\sigma)}{\vdash \langle\sigma\rangle\, x = \langle\rangle\, \langle\mathsf{err}\rangle}$$

C-Alloc-Err-Com
$$\frac{x \notin \mathrm{dom}(\sigma)\ \text{or}\ y \notin \mathrm{dom}(\sigma)}{\vdash \langle\sigma\rangle\, x = \mathrm{alloc}\ y\, \langle\mathsf{err}\rangle}$$

C-Heap-Read-Err-Com
$$\frac{x \notin \mathrm{dom}(\sigma)\ \text{or}\ y \notin \mathrm{dom}(\sigma)\ \text{or}\ \sigma(y) \notin \mathrm{dom}(\sigma)}{\vdash \langle\sigma\rangle\, x = *y\, \langle\mathsf{err}\rangle}$$

C-Heap-Write-Err-Com
$$\frac{x \notin \mathrm{dom}(\sigma)\ \text{or}\ \sigma(x) \notin \mathrm{dom}(\sigma)\ \text{or}\ y \notin \mathrm{dom}(\sigma)}{\vdash \langle\sigma\rangle\, *x = y\, \langle\mathsf{err}\rangle}$$

C-Seq-Err1-Com
$$\frac{\vdash \langle\sigma\rangle\, c_1\, \langle\mathsf{err}\rangle}{\vdash \langle\sigma\rangle\, c_1\ ;\ c_2\, \langle\mathsf{err}\rangle}$$

C-Seq-Err2-Com
$$\frac{\vdash \langle\sigma\rangle\, c_1\, \langle\sigma'\rangle \qquad \vdash \langle\sigma'\rangle\, c_2\, \langle\mathsf{err}\rangle}{\vdash \langle\sigma\rangle\, c_1\ ;\ c_2\, \langle\mathsf{err}\rangle}$$

Figure 5.5: Concrete Semantics.

$=$ alloc y‘ should add a fresh address a to the store, binding it to whatever value is contained in global variable y, and update x's binding to contain the new address. We describe heap reads ‘$x =$ *y’ in rule C-Heap-Read-Com. This rule looks up the address stored in global variable y and then writes the value stored at that address into global variable x. Heap writes ‘*$x = y$’ analogously look up the value stored in global y and store it in the the cell for the address stored in global x. Finally, C-Seq-Com shows the standard sequence rule. We also—as is standard for a big-step semantics—have explicit error cases. These cases cover ways in which a non-error rule could fail to apply. The C-Alloc-Err-Com rule, for example, applies both when x is not in the domain of the store σ and also when $\sigma(y)$ does not exist. The rest of the error rules cover the analogous error versions of the success cases.

The following lemma shows that the concrete semantics given in Figure 5.5 are gate-frameable:

Lemma 10 (Gate-Frameable Concrete Semantics.): *If* $\mathcal{C} :: \vdash \langle \sigma_{\text{fore}} \lhd \sigma_{\text{aft}} \rangle \, c \, \langle r \rangle$ *then* $\vdash \langle \sigma_{\text{fore}} \rangle \, c \, \langle r_{\text{fore}} \rangle$ *and*

(1) *If* $r \neq \textsf{err}$ *then either*

  (a) $r_{\text{fore}} = \textsf{err}$ *or*

  (b) $r_{\text{fore}} = \sigma'_{\text{fore}}$ *where* $r = \sigma'_{\text{fore}} \lhd \sigma_{\text{aft}}$*;*

and

(2) *if* $r = \textsf{err}$ *then* $r_{\text{fore}} = \textsf{err}$*.*

Proof. By induction on the derivation of $\mathcal{C}$. □

Here we overload $\lhd$ so that $\sigma_1 \lhd \sigma_2$ means that the domains of σ_1 and σ_2 are disjoint, as are the range of σ_1 and the domain of σ_2. This lemma is crucial to showing that the gated frame rule (Figure 5.4) is sound.

5.3 Type-Intertwined Separation Logic

As illustrated in Section 3.3, framing with standard separation is unsound with materialization from a global summary. In our case, we wish to materialize from a global type invariant but also apply framing for modular reasoning during symbolic analysis. In this section, we show how gated separation enables a combination of global materialization and framing by, in essence, constraining the "globalness" of the global summary. The result is a truly type-intertwined separation logic, including both a global summary of an almost type-consistent heap and local type-intertwined reasoning via a frame rule for gated separating conjunction. We describe the global type invariant—a constraint on reachability from a set of program variable roots—and type checking rules in Section 5.3.1. In Section 5.3.2, we describe the type-intertwined symbolic state and show the relationship between the global almost type-consistent summary and the types of values. Finally, in Section 5.3.3, we provide a static semantics for type-intertwined separation logic and show how its treatment of allocation and gated framing differ from the rules for traditional separation logic extended with gated separation.

types

T	$::=$	unit $\mid$ ref T	types
Γ	$::=$	$\cdot \mid \Gamma[x : T]$	type environments

Figure 5.6: Types for the command language.

5.3.1 Type Checking the Command Language

We designed the command language from Section 5.2 to be typed with a very simple type system.

Types. We describe the types for this simple system in Figure 5.6. A type T can be either the unit type 'unit' (whose single value is $\langle\rangle$) or a type 'ref T_1', which is the type of a mutable pointer to a value of type T_1. These types are simple but still rich enough to express an inductively defined constraint on the reachable heap. As is standard, a type environment Γ is a finite mapping from global variable identifiers x to types T.

The concretization of a type yields a set of pairs of a constrained store and value (in a similar fashion to the concretization of base types in the reflection language described in Section 4.2.2).

$\gamma : \mathsf{Types} \to \mathcal{P}(\mathsf{Stores} \times \mathsf{Values})$

$$\gamma(\text{unit}) \triangleq \left\{(\sigma, v) \;\middle|\; v \text{ is } \langle\rangle \;\right\}$$

$$\gamma(\text{ref } T) \triangleq \left\{(\sigma, a) \;\middle|\; (\sigma, \sigma(a)) \in \gamma(T) \;\right\}$$

The concretization of a type environment constrains yields a set of stores where each binding the type environment constrains the heap and the of value stored in the global variable (address) with the binding name.

$\gamma : \mathsf{TypeEnvironments} \to \mathcal{P}(\mathsf{Stores})$

$$\gamma(\Gamma) \triangleq \left\{\sigma \;\middle|\; \begin{array}{l} (\sigma, \sigma(x)) \in \gamma(T) \text{ for all } x : T \text{ in } \Gamma\text{; and} \\ \text{rng}(\sigma) \cap \mathsf{Vars} = \emptyset. \end{array} \right\}$$

$$\textsc{T-Write-Unit-Com}\ \frac{\Gamma(x) = \mathrm{unit}}{\Gamma \vdash x = \langle\rangle} \qquad \textsc{T-Alloc-Com}\ \frac{\Gamma(x) = \mathrm{ref}\ T \qquad \Gamma(y) = T}{\Gamma \vdash x = \mathrm{alloc}\ y}$$

$$\textsc{T-Heap-Read-Com}\ \frac{\Gamma(x) = T \qquad \Gamma(y) = \mathrm{ref}\ T}{\Gamma \vdash x = *y} \qquad \textsc{T-Heap-Write-Com}\ \frac{\Gamma(x) = \mathrm{ref}\ T \qquad \Gamma(y) = T}{\Gamma \vdash *x = y} \qquad \textsc{T-Seq-Com}\ \frac{\Gamma \vdash c_1 \qquad \Gamma \vdash c_2}{\Gamma \vdash c_1\ ;\ c_2}$$

Figure 5.7: Type checking rules for the command language.

We also enforce the constraint that even though global variables act as addresses, the addresses themselves can never be stored on the heap. This is a reasonable constraint: the command language does not have an addressof (&) operator.

Type checking rules. We give rules for typechecking the command language in Figure 5.7. These rules are quite standard. The key property of this type system is that it flow-insensitively constrains the reachable heap (and only the reachable heap) from a given typed storage location (either a global variable or a dynamically allocated cell on the heap) to consistent with the location's declared type.

The T-Write-Unit-Com rule says that the unit literal $\langle\rangle$ can be written to a global variable x if x has type 'unit' in the type environment Γ. Note that even though global variable locations are mutable and concretize to heap cells, we do not give them a ref type—such types are reserved for addresses of dynamically allocated memory. We type allocation with the T-Alloc-Com rule, which says that the address of a newly allocated cell that has been initialized with a value of type T can be stored in a global variable of type 'ref T'. The T-Heap-Read-Com and T-Heap-Write-Com describe typing of heap reads and writes, respectively. The value read from a dynamically allocated address of type 'ref T' can be safely placed in a global variable of type T, while the a value read from a global variable of type T can be written to an address of type 'ref T'. Finally, the T-Seq-Com shows that type environments are flow-insensitive: two sequenced sub-commands must be well-typed with respect the same type environment Γ.

symbolic state

M	$::=$	$\cdots$	gated memories
	$\mid$	ok	global type invariant summary
$\widehat{\Gamma}$	$::=$	$\cdot \mid \widehat{\Gamma}[\widehat{v} : T]$	value typing
Σ	$::=$	$M \text{ ¦ } \widehat{\Gamma}$	state
Π	$::=$	$\Sigma \mid \Pi \vee \Pi \mid \mathsf{false}$	symbolic paths

Figure 5.8: Type-intertwined symbolic state.

5.3.2 Type-Intertwined State

Type-intertwined separation logic requires a richer abstract state (presented in Figure 5.8) than that for the traditional separation logic (Section 5.2). For type-intertwined separation, we extend memory formulas M with the ok atomic assertion, which represents a portion of the heap that is not immediately type-inconsistent: that is, the values contained in storage locations summarized in ok are at worst transitively type-inconsistent (Section 3.2) with their location's declared types.

In contrast with the standard separation logic presented in Section 5.2—where the abstract state consisted solely of a symbolic memory M—in type-intertwined separation logic a symbolic state Σ consists of a pair $M \text{ ¦ } \widehat{\Gamma}$ of a memory M and a value typing $\widehat{\Gamma}$. As we saw in the reflection symbolic analysis presented in Section 4.3.1, a value typing maps symbolic values $\widehat{v}$ to the **promised** type T of a value—the type that the value will have when its entire reachable heap is type-consistent. Unlike the reflection analysis, here the promised types do not have refinements so they do not express relationships with other symbolic values. We also allow disjunctive symbolic paths Π, similar to the reflection analysis. Here, however, these disjunctions result not from branching in the command language (there is none) but—as we will see—arise from the potential for multiple aliasing relationships when materializing from ok.

Concretization. Like concretization in the reflection analysis—and unlike concretization for traditional separation logic—a concretization in type-intertwined separation logic yields a valuation V mapping symbolic value to concrete values and **two** disjoint stores, one for the almost type-consistent ok portion of the heap and one for the explicitly materialized portion.

$\gamma : \mathsf{SymbolicMemory} \to \mathcal{P}(\mathsf{Valuation} \times \mathsf{Store} \times \mathsf{Store})$

$$\gamma(\mathsf{emp}) \triangleq \left\{ (V, \sigma^{\mathrm{ok}}, \sigma^{\mathrm{mat}}) \;\middle|\; \sigma^{\mathrm{ok}} = \cdot \text{ and } \sigma^{\mathrm{mat}} = \cdot \;\right\}$$

$$\gamma(\mathsf{ok}) \triangleq \left\{ (V, \sigma^{\mathrm{ok}}, \sigma^{\mathrm{mat}}) \;\middle|\; \sigma^{\mathrm{mat}} = \cdot \;\right\}$$

$$\gamma(\widehat{a} \mapsto \widehat{v}) \triangleq \left\{ (V, \sigma^{\mathrm{ok}}, \sigma^{\mathrm{mat}}) \;\middle|\; \sigma^{\mathrm{ok}} = \cdot \text{ and } \sigma^{\mathrm{mat}} = [V(\widehat{a}) \mapsto V(\widehat{v})] \;\right\}$$

$$\gamma(M_1 \ast M_2) \triangleq \left\{ (V, \sigma_1^{\mathrm{ok}} \ast \sigma_2^{\mathrm{ok}}, \sigma_1^{\mathrm{mat}} \ast \sigma_2^{\mathrm{mat}}) \;\middle|\; \begin{array}{l} (V, \sigma_1^{\mathrm{ok}}, \sigma_1^{\mathrm{mat}}) \in \gamma(M_1) \text{ and} \\ (V, \sigma_2^{\mathrm{ok}}, \sigma_2^{\mathrm{mat}}) \in \gamma(M_2) \text{ and} \\ \text{exists } \sigma \text{ where } \sigma = \sigma_1^{\mathrm{ok}} \ast \sigma_2^{\mathrm{ok}} \ast \sigma_1^{\mathrm{mat}} \ast \sigma_2^{\mathrm{mat}} \end{array} \right\}$$

$$\gamma(M_1 \mathbin{\triangleleft\!\!\!\ast} M_2) \triangleq \left\{ (V, \sigma_1^{\mathrm{ok}} \mathbin{\triangleleft\!\!\!\ast} \sigma_2^{\mathrm{ok}}, \sigma_1^{\mathrm{mat}} \mathbin{\triangleleft\!\!\!\ast} \sigma_2^{\mathrm{mat}}) \;\middle|\; \begin{array}{l} (V, \sigma_1^{\mathrm{ok}}, \sigma_1^{\mathrm{mat}}) \in \gamma(M_1) \text{ and} \\ (V, \sigma_2^{\mathrm{ok}}, \sigma_2^{\mathrm{mat}}) \in \gamma(M_2) \text{ and} \\ \text{exists } \sigma \text{ where } \sigma = (\sigma_1^{\mathrm{ok}} \ast \sigma_1^{\mathrm{mat}}) \mathbin{\triangleleft\!\!\!\ast} (\sigma_2^{\mathrm{ok}} \ast \sigma_2^{\mathrm{mat}}) \end{array} \right\}$$

Here, the concretizations of emp and ok are as in the reflection analysis: emp constrains both the almost type-consistent portion of the store and the materialized store to be empty, while ok allows any almost type-consistent store but requires no storage be materialized. The concretization of a single cell $\widehat{a} \mapsto \widehat{v}$ requires that storage for that cell exist in materialized store and requires the almost type-consistent store to be empty. The concretization of the standard separating conjunction of memories $M_1 \ast M_2$ is as in the reflection analysis: each of the components in the concretizations of the sub-heaps must be disjoint, as must their combination. Finally, and most importantly, the concretization of a gated separation of memories $M_1 \mathbin{\triangleleft\!\!\!\ast} M_2$ constrains the almost type-consistent portions of the concrete heaps to be gated separated, the materialized portions of the concrete heaps to be gate separated, and the standard conjunction of the heaps concretized from M_1 to be gated separated from that for those from M_2. That is, we require that all memory represented by M_1 is gate separated from the memory represented by M_2, regardless of whether it is in the almost type-consistent store or the materialized store.

The concretization of a value typing $\widehat{\Gamma}$ constrains both stores (almost type-consistent and materialized) and the valuation—again, this is analogous to value typings in the reflection analysis.

$\gamma : \mathsf{ValueTyping} \to \mathcal{P}(\mathsf{Valuation} \times \mathsf{Store} \times \mathsf{Store})$

$$\gamma(\widehat{\Gamma}) \triangleq \left\{ (V, \sigma^{\text{ok}}, \sigma^{\text{mat}}) \;\middle|\; \begin{array}{l} \text{for each } \widehat{v} : T \text{ in } \widehat{\Gamma} \\ \quad (\sigma^{\text{ok}}, \sigma^{\text{mat}}, V(\widehat{v})) \in \widehat{\gamma}(T) \end{array} \right\}$$

For each binding in the value typing, concrete value for the binding and the stores must be in the concretization $\widehat{(\gamma)}$ of the bound type. Note the decorated $\widehat{\gamma}$—this is the symbolic concretization of types (described below) rather than the concretization of types in the types domain given in Section 5.3.1.

The symbolic concretization $\widehat{\gamma}$ of a type in type-intertwined separation logic differs from the concretization of a types in the type domain in a similar fashion to base types in the reflective intertwined analysis: in the symbolic domain, the concretization yields both an almost type-inconsistent store and an materialized store:

$\widehat{\gamma} : \mathsf{Types} \rightarrow \mathcal{P}(\mathsf{Store} \times \mathsf{Store} \times \mathsf{Value})$

$$\widehat{\gamma}(\text{unit}) \triangleq \left\{ (\sigma^{\text{ok}}, \sigma^{\text{mat}}, v) \;\middle|\; v \text{ is } \langle\rangle \;\; \right\}$$

$$\widehat{\gamma}(\text{ref } T) \triangleq \left\{ (\sigma^{\text{ok}}, \sigma^{\text{mat}}, a) \;\middle|\; \begin{array}{l} \sigma^{\text{ok}} \ast \sigma^{\text{mat}}(a) = v \text{ and} \\ \quad \text{if } a \in \text{dom}(\sigma^{\text{ok}}) \text{ then } (\sigma^{\text{ok}}, \sigma^{\text{mat}}, v) \in \widehat{\gamma}(T) \end{array} \right\}$$

The concretization of the unit type yields the unit value $\langle\rangle$ and makes no constraint on either the almost type-inconsistent or the materialized heap. It is in the symbolic concretization of reference types that the difference between the materialized heap and the almost type-inconsistent ok heap is relevant. The concretization of a reference type 'ref T' yields an address that is the store (in either σ^{ok} or σ^{mat}) and, more importantly, if the storage for a is in σ^{ok} then the value and the two heaps are constrained by the concretization of the type T.

The concretization of a symbolic state Σ yields a set of stores:

$\gamma : (\mathsf{SymbolicState}) \rightarrow \mathcal{P}(\mathsf{Store})$

$$\gamma(M \mathrel{¦} \widehat{\Gamma}) \triangleq \left\{ \sigma^{\text{ok}} \ast \sigma^{\text{mat}}) \;\middle|\; \text{Exists } V \text{ where } (V, \sigma^{\text{ok}}, \sigma^{\text{mat}}) \in \gamma(M) \cap \gamma(\widehat{\Gamma}) \;\; \right\}$$

Here, the resultant concretized stores each consist of the combination of the almost type-consistent and materialized heaps, each of which are constrained by the state's symbolic memory and the value typing.

The concretization of a symbolic path is the union of its constituent paths:

$\gamma : (\mathsf{SymbolicPath}) \to \mathcal{P}(\mathsf{Store})$

$$\gamma(\Sigma) \triangleq \{\sigma \mid \sigma \in \gamma(\Sigma) \quad \}$$
$$\gamma(\Pi_1 \vee \Pi_2) \triangleq \gamma(\Pi_1) \cup \gamma(\Pi_2)$$
$$\gamma(\mathsf{false}) \triangleq \emptyset$$

Here, the concretization of a symbolic path consisting of a symbolic state is the concretization of that state. The concretization of the disjunction of two paths is the union of the concretizations of those paths, and the concretization of the false path is empty.

5.3.3 Static Semantics of Type-Intertwined Separation Logic

As we have seen, the key difference between traditional separation logic and type-intertwined separation logic is the presence of (1) the atomic formula ok representing the almost type-consistent heap and (2) the $\widehat{\Gamma}$, which determines what it means for a value to be consistent with its type.

We provide a system of inference rules for type-intertwined separation logic in Figure 5.9. Most of the inference rules of traditional separation logic (given in Figure 5.4)) can be mechanically converted to type-intertwined rules by extending the abstract state to consist of a pair $M \mathrel{\vdots} \widehat{\Gamma}$ of a memory M and a value typing $\widehat{\Gamma}$ rather than just a memory. Rather than provide those again here, we note that all but S-GatedFrame-Com can be modified to thread the additional state component in the usual way. Here, we focus on the additional complications required for type-intertwined reasoning: (1) ascribing types to newly-allocated storage, (2) the gated frame rule in the presence of a global ok summary; (3) materializing from and summarizing to ok with the possibility of gated separation; and (4) handoff to types.

$$\boxed{\vdash \{\Sigma\}\, c\, \{\Pi\}}$$

TI-Sep-Ascribe

$$\frac{\vdash \{K[\mathsf{ok} \mathbin{\triangleleft\!\!\!\ast} \widehat{a} \mapsto \widehat{v}] \mathrel{\text{¦}} \widehat{\Gamma}[\widehat{a} : \text{ref } T]\}\, c\, \{\Pi\}}{\vdash \{K[\mathsf{ok} \mathbin{\triangleleft\!\!\!\ast} \widehat{a} \mapsto \widehat{v}] \mathrel{\text{¦}} \widehat{\Gamma}\}\, c\, \{\Pi\}}$$

TI-Sep-GatedFrame

$$\frac{\vdash \{M_{\text{fore}} \mathrel{\text{¦}} \widehat{\Gamma}\}\, c\, \{M'_{\text{fore}} \mathrel{\text{¦}} \widehat{\Gamma}'\}}{\vdash \{M_{\text{fore}} \mathbin{\triangleleft\!\!\!\ast} M_{\text{aft}} \mathrel{\text{¦}} \widehat{\Gamma}\}\, c\, \{M'_{\text{fore}} \mathbin{\triangleleft\!\!\!\ast} M_{\text{aft}} \mathrel{\text{¦}} \widehat{\Gamma}'\}}$$

TI-Sep-Materialize

$$\frac{\begin{array}{c} \Sigma = K[\mathsf{ok} \ast M_1] \mathrel{\text{¦}} \widehat{\Gamma} \qquad \widehat{a} \in \underline{\text{addrs}}(M_1) \qquad \widehat{\Gamma}(\widehat{a}) = \text{ref } T \\ \Pi = \left(K[\mathsf{ok} \ast M_1 \ast \widehat{a} \mapsto \widehat{v}] \mathrel{\text{¦}} \widehat{\Gamma}' \vee \bigvee_{\widehat{y} \in \text{mayalias}_{\Sigma}(\widehat{a})} \Sigma|_{\widehat{a}=\widehat{y}} \right) \qquad \widehat{\Gamma}' = \widehat{\Gamma}[\widehat{v} : T] \qquad \widehat{v} \notin \Sigma \qquad \vdash \{\Pi\}\, c\, \{\Pi'\} \end{array}}{\vdash \{\Sigma\}\, c\, \{\Pi'\}}$$

TI-Sep-Summarize

$$\frac{\widehat{\Gamma}(\widehat{a}) = \text{ref } T \qquad \widehat{\Gamma}(\widehat{v}) = T \qquad \vdash \{K[\mathsf{ok} \ast M_1] \mathrel{\text{¦}} \widehat{\Gamma}\}\, c\, \{\Pi\}}{\vdash \{K[\mathsf{ok} \ast M_1 \ast \widehat{a} \mapsto \widehat{v}] \mathrel{\text{¦}} \widehat{\Gamma}\}\, c\, \{\Pi\}}$$

TI-Sep-Types-Handoff

$$\frac{\Gamma = \widehat{\Gamma} \circ M \qquad \Gamma \vdash c \qquad \widehat{\Gamma}' \circ M' = \Gamma \text{ where } M' \text{ is 1-1}}{\vdash \{\mathsf{ok} \ast M \mathrel{\text{¦}} \widehat{\Gamma}\}\, c\, \{\mathsf{ok} \ast M' \mathrel{\text{¦}} \widehat{\Gamma}'\}}$$

$$\boxed{\vdash \{\Pi\}\, c\, \{\Pi'\}}$$

TI-Sep-Cases

$$\frac{\vdash \{\Sigma_i\}\, c\, \{\Pi_i\} \qquad \text{for all } i}{\vdash \{\bigvee_i \Sigma_i\}\, c\, \{\bigvee_i \Pi_i\}}$$

Figure 5.9: Type-Intertwined Separation Logic.

Allocating and ascribing types to storage. The S-Alloc-Com for traditional separation logic allocates new gate-separated storage, but it does not provide a value typing for the fresh address. We allow a type-intertwined separation logic to **ascribe** any referenced type 'ref T' to an address that is gated-separated from the ok global summary, as described in TI-Sep-Ascribe. This rule could potentially apply more generally than to just newly-allocated storage—it says that if the analysis can determine that there are no direct pointers from the almost type-consistent heap summarized by ok into materialized storage, then the type of that storage can be safely changed without violating the expectations of storage in ok. Note that T does not have to be the type of the value $\widehat{v}$ currently stored at address $\widehat{a}$; the value typing merely expresses the promise that to be summarized (see below) into ok the cell must eventually contain a value of type T.

The gated frame rule with ok. The gated frame rule for type-intertwined separation logic (TI-Sep-GatedFrame) differs crucially from the gated version of the frame rule we described for non-type-intertwined separation logic (Section 5.2.2) in that it requires the gate separating the foregate M_{fore} from the framed-out aftgate M_{aft} to be at the top level of the spatial formula. This restriction is required because the value typing $\widehat{\Gamma}$ allows access (via materialization and handoff, described below) to the entire heap reachable from the almost type-consistent summary ok—which may be present in the foregate M_{fore}. For a gate at the top level, the local "dis-pointing" relationship between the foregate and aftgate implies global unreachability, which prevents type-intertwined interference in languages satisfying requirements of gate-frameability (Section 5.2.3). In practice, this limits framing out to newly allocated (rather than materialized) memory, although there may be mechanisms other than allocation to introduce gated separation.

Materializing and summarizing with gated framing. As we saw in Section 3.2 and Section 4.4, a type-intertwined analysis can materialize and summarize storage from the almost type-consistent summary ok to selectively violate and restore global type invariants and enable local, alias-aware reasoning. In the reflection symbolic analysis—which did not support framing out memory—we had to consider the possibility that the newly materialized storage could in fact alias with any existing already materialized storage. This possibility arose because type summaries of storage do not prevent accessing the same storage via different access paths. Framing out memory complicates materialization because we must consider potential aliasing with not just memory on the explicit, already-materialized heap but also memory that has been framed out. Fortunately, framing with gated separation—and a small tweak to the rule for materialization—can rule out this potential aliasing. We give this updated materialization rule in TI-Sep-Materialize. It says that we can materialize explicit storage for a symbolic address $\widehat{a}$ with type 'ref T' if we disjunctively account for all possible aliasing relations between $\widehat{a}$ and other addresses $\widehat{y}$ in the state Σ. In the disjunct where we add storage for $\widehat{a}$ to the memory, we also update the value typing $\widehat{\Gamma}$ to indicate that the fresh symbolic value $\widehat{v}$ has promised type T. With these disjuncts constructed, abstract execution can continue on a path-by-path basis (rule TI-Sep-Cases). Materialization in type-intertwined separation logic closely follows the M-Materialize auxiliary rule described for the reflective analysis presented in Section 4.4.3, although it is simpler here because the command language has neither objects nor dependent types. The crucial difference required for safe framing is the constraint (shown highlighted) that the materialized address must be somewhere in either the domain or range of M_1. This restriction ensures that the address $\widehat{a}$ could not possibly alias with memory that has already been framed out—otherwise the gated separation required for type-intertwined framing would not have held. Summarizing proceeds in the opposite direction (rule TI-Sep-Materialize). It requires that the summarized address $\widehat{a}$ be of reference type 'ref T' and that the value $\widehat{v}$ stored have promised type T. This is completely analogous to M-Summarize in Section 4.4.4.

Handoff The TI-Sep-Types-Handoff rule describes handoff to the type checker. This rule is analogous to the Sym-Type-Handoff from Section 4.4 except that here we consider a command language (rather than expressions) and we do not have dependent types. To switch to type checking, we require that the memory contain ok and that the non ok portions form a $*$-separated map from addresses to symbolic values. Then, we can type check the command in a type environment Γ constructed by composing the value typing $\widehat{\Gamma}$ with the memory. If the command type checks in that environment, then the resultant post-state can be described by a symbolic state consistent with the type environment and where the memory mapping is 1-1 (to make no assumptions about aliasing). The key thing to note about this rule is that it allows a switch to typechecking if the parts of the heap that are immediately type inconsistent have already been gate-framed out. This combination of framing and handoff enables checking of examples like that presented in Section 3.3, which requires a switch to types before the global type invariant has been restored for the entire heap.

5.4 Related Work

Perhaps the closest work related to gated separation is that by Petersen et al. [83] on a type theory for data layout and memory allocation. This approach employs an ordered type theory, based on ordered logic [85], in which bindings in a typing context can be neither dropped nor reordered—that is, like gated separating conjunction, their type system does not admit the arbitrary exchange rule. With this approach, they can reason about safe allocation and initialization of memory in the presence of a copying garbage collector. The key to this process is reasoning about newly allocated memory, which is initialized in an ephemeral region of memory called the frontier and then conceptually moved (by bumping of an allocation pointer) into the main heap. This reasoning is reminiscent of our approach to treating fresh memory as initially gate-separated from the rest of memory and then summarizing it into the almost-consistent ok heap—although we do not need to consider adjacency of storage, as they do.

Ahmed and Walker extend Hoare logic for a typed assembly language with an ordered logic for reasoning about typed stack allocation [4] that reasons about adjacency as well as separation and aliasing. Like our type-intertwined separation logic with gated separation, their logic contexts are trees (bunches [62, 79]) rather than lists. Their contexts allow nested alternation between ordered (non-commutative) and un-ordered (commutative) separators—similarly to our gated and traditional separating conjunction, respectively. They manage this tree structure with a similar decomposition into a context with a hole as we do in Section 5.1.2, but do not have to reason about whether an imperative update can violate tree-structured constraints (like we do with gated separation). Walker gives a broad introduction to the properties of substructural type systems in [104].

Chapter 6

Measuring Enforcement Windows with Symbolic Trace Interpretation

As described in Chapter 2, the motivation for type-intertwined separation logic came from the results of a series of experiments that I performed to investigate how developers ensure that their programs are free from null pointer exceptions. This chapter describes the measurement framework used in those experiments—symbolic trace interpretation—and proposes the philosophy of data-driven static analysis design that inspired our work on type-intertwined separation logic. The contents of this chapter originally appeared in my ISSTA 2012 paper [29] "Measuring Enforcement Windows with Symbolic Trace Interpretation: What Well-Behaved Programs Say", which was joint work with Bor-Yuh Evan Chang, Amer Diwan, and Jeremy G. Siek. It has has been lightly edited and reformatted.

6.1 Introduction

In the 1990s, a large program had hundreds of thousands of lines of code. By today's standards, such a program is tiny! For example, Windows Vista has a code base of 60 million lines of code created by ∼3,000 developers [12]. It is clear that no programmer can fully understand every line of code and how they relate to each other in such a large system. Rather, programmers rely on "isolation boundaries" following from modular design to reason about their code. These isolation boundaries do not always follow explicit modularization (e.g., methods, classes, and packages) but can be implicit (e.g., around groups of tightly coupled methods).

Static analysis tools, which help find bugs in software, can take advantage of isolation

```
1 o = new O(); ...
2 if (o != null) { ... } ...
3 if (o != null) {
4   ...
5   x = o.f;
6   ...
7 }
```

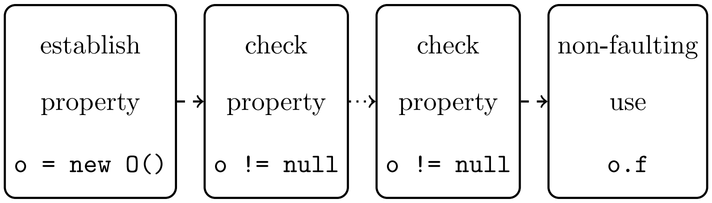
Figure 6.1: A potential validation scope.

boundaries to scale to real-world programs. Can we find and leverage the implicit isolation boundaries created by software developers to improve static analysis? To attack this question, we define the concepts of a **validation scope** and an **enforcement window** in this paper. We then create a framework for measuring enforcement windows with dynamic analyses. At a high level, a validation scope captures a **property-based** isolation boundary implied by the code itself, and an enforcement window is a dynamic approximation of a validation scope. We define these concepts in detail in the remainder of this introduction.

One of our key insights is that proving a property about a particular operation does not always require the entire program. In particular, we define a **validation scope** as a part of the code where if we reason operationally (e.g., directly by analyzing the code precisely), then we can prove a property of interest without any assumptions about its context. As an example, consider the Java fragment in Figure 6.1 and what it takes to validate that the read of `o.f` cannot dereference `null`. The highlighted code fragment between the null check on line 3 and the dereference `o.f` on line 5 (shaded and marked with vertical lines) may be a sufficient validation scope to prove that `o.f` does not dereference `null` (depending on what is on line 4).

Intuitively, a validation scope captures an implied isolation boundary with respect to a potential fault based on the **enforcements** inserted in the code. We use the term **enforcement** to refer generically to an operation that **establishes** or **checks** the property of interest (e.g., `o != null`). Validation scopes get at an important aspect of static analysis design and program reasoning: on one hand, a static analysis can leverage validation scopes to limit the precision use outside validation scopes, while on the other hand, a static analysis must be able to reason precisely enough inside the scope to capture the property of interest. In this paper, we propose techniques to identify potential validation scopes and ways to measure their “size” or “complexity” before designing a static analysis.

To do so, another key insight is that analyzing well-behaved executions can provide evidence for validation scopes. In particular, given a safety property and a **non-faulting** trace (i.e., one that does not violate the property of interest), there is an event that establishes the property, potentially followed by (redundant) checks that confirm that the property continues to hold, and finally ending with a non-faulting use as diagrammed below:



For example, a non-faulting object dereference (i.e., does not dereference null) is established by the object allocation and may be validated by any number of null checks before reaching the dereference site. We call such a sequence of establish, check, and use events an **enforcement window**. An enforcement window is a dynamic approximation in that we can begin to search for candidate validation scopes by mapping enforcement events back to source code locations. Our definition of an enforcement window is property independent—-all that needs to be defined for each property is what events count as an ”establish,” a ”check,” or a ”use.”

In this paper, we describe a measurement framework for enforcement windows and then measure enforcement windows for an example property—specifically, non-null dereference. One measurement of interest is the distance between a use (e.g., a dereference) and its closest enforcement (e.g., a null check or an allocation) for several different notions of “distance.” Intuitively, such a measure captures the “complexity” of the candidate validation scope from the closest enforcement to the use. As a simple example, consider the three-line Java fragment in Figure 6.2 where the methods corresponding to the called methods are shown inline, that is, path projected [65] (in grey backgrounds).

```
    1  id = new Id(); x = new X();
    2  x.setId(id);
            void setId(Id id) {
  2.1        assert id != null; this.id = id;
            }

    3  return x.getIdAsInt();

            int getIdAsInt() {
  3.1        if (this.id == null) { this.id = new Id(); }
  3.2        return this.getRawId();

                int getRawId() {
3.2.1            return this.id.raw;
                }

            }
```

Figure 6.2: Inlining depth as a metric for complexity.

Focusing on the `.raw` dereference at line 3.2.1, the enforcement window is as follows: (a) establish with the allocation of an `Id` at line 1 in the global context, (b) check at line 2.1 in `setId` (i.e., `id != null`), (c) check at line 3.1 in `getIdAsInt`, and (d) use at line 3.2.1 in `getRawId`. One interesting distance metric that we consider is the **inlining depth** needed to bring the path between the check and the use into the same method scope. In this case, there is an inlining depth of 1 between the last null check at line 3.1 in `getIdAsInt` and the dereference at line 3.2.1 in `getRawId`.

From the point of view of static analysis design, these dynamic measurements are interesting because they rule out insufficient designs. In the Java fragment and successful execution trace above, an inlining depth of 1 witnesses that a simple, conservative, **intra**procedural null deference analysis is insufficient and would necessarily result in a false alarm at the dereference site at line 3.2.1. We mean specifically that this analysis when analyzing `getRawId` has no precondition that it can assume about its context. Note that with these dynamic measurements, we get **necessary** conditions but not **sufficient** ones in that even after inlining `getRawId` into `getIdAsInt` a null dereference analysis may not be able to prove that the dereference site at line 3.2.1 is safe (perhaps because it is imprecise on aspects not captured by this particular measurement or because the dynamic analysis did not measure all program paths). We discuss why we chose dynamic over static analysis in Section 6.2.2 and consider this potential insufficiency further in Section 6.3.3.

From a software engineering perspective, our measurement framework also enables us to empirically support or refute widely-held intuitions about how programmers use enforcements in their code.

Overall, this work makes the following contributions:

- We introduce the notion of enforcement windows that enables us to rule out insufficient static analysis designs. We systematically examine choices in deciding, where, what, and how to measure enforcement windows, and we describe distance metrics that capture reasoning about both **control** and **data** (Section 6.2.2).

- We present a flexible framework for measuring enforcement windows dynamically (Section 6.3). A challenging requirement for these measurements is a way to get at static, source code notions with dynamic analysis. We address this challenge by applying symbolic reasoning techniques and propose **symbolic trace interpretation**, whose essence is an intertwined concrete-symbolic analysis state (Section 6.3.1). Taking these measurements dynamically rather than statically enables us to measure one aspect of analysis precision (e.g., context sensitivity) while factoring out others, such as imprecise heap reasoning (Section 6.2.2).

  Measuring enforcement windows in the presence of heap objects requires careful design and special mechanisms to scale to even modestly-sized benchmarks. We describe **piggybacked garbage collection**, which collects a "shadow heap" by instrumenting the collector of the concrete heap, and we propose **measurement update partitions** that capture ways to update groups of symbolic heap values simultaneously (Section 6.3.2).

- We study the extent to which our dynamic measurements of enforcement windows are sufficient from a static analysis perspective by measuring whether the check sites in our observed enforcement windows are static **bottlenecks** in the control-flow graph for their use sites (Section 6.3.3). We find that a significant portion (30% to 80%) of use sites are statically protected by their observed closest check sites, suggesting that these measured enforcement window distances are quite likely to indicate useful validation scopes.

- We apply our trace interpretation framework to study the evolution and distribution of enforcement windows for dereferences using metrics from four broad categories (Section 6.4). In particular, we measure how enforcement windows for dereferences change across bug fixes for `NullPointerException`s in Java. We find that (1) enforcement windows get shorter after bug fixes and (2) that longer enforcement windows are more likely to result in bugs. These findings provide empirical evidence supporting the commonly held but difficult to verify belief that programmers find it easier to reason locally than non-locally. We also find that (3) enforcement window sizes are remarkably stable over project lifetimes, even as code bases nearly double in size, and that (4) while measured enforcement windows are in general small, in some cases they are large along certain dimensions.

6.2 Overview and Metrics

In this section, we give an overview of our enforcement window measurement framework by following an example symbolic trace interpretation. Recall that our goal is to measure the "complexity" of candidate validation scopes that can potentially inform static analysis design or simply provide insights into how enforcements appear in code. We argue why symbolic reasoning on dynamic analysis is needed to get useful information by systematically laying out the various choices in deciding **where**, **what**, and **how** to measure. This discussion leads to metrics that we apply to get the measurement data presented in Section 6.4.

6.2.1 Preliminaries: Trace Instructions

Our measurement framework consists of two main components. The **trace collector** instruments Java bytecode to obtain a log of interesting events upon execution, a technique that is is fairly standard in dynamic analysis (e.g., [44]). The **trace interpreter** performs a symbolic interpretation of this log to obtain measurements of **enforcement distance**. We define an enforcement distance as some measurement between two events in an enforcement window (e.g., between establish, check, or use events). The log is essentially a sequence of instructions that records a "path slice" that we can symbolically reinterpret to obtain enforcement distances and consequently a view of how enforcements appear in the program. Crucially this symbolic interpretation enables us to extract a static, source-code view of enforcement from dynamic traces without introducing imprecision from a purely static approach (Choice 4 in Section 6.2.2).

Figure 6.3b shows the sequence of **trace instructions** that the trace collector emits during execution of the example source in Figure 6.3a. Ignore the boxed items for now. The purpose of separating the collector from the interpreter is to handle most of the complexity of Java's semantics in the collector. We can write a mostly generic collector that is customized to filter (and perhaps simplify) instructions for the properties of interest. Here, we show a trace language specialized to null-dereference analysis. For exposition, we use an operand-stack–based language like Java bytecode; that is, the local store is a stack of activation records where each activation record is a stack of values. There are no integer or numeric operations here because they can be filtered out for this example analysis.

Simply to explain the semantics of this trace language, we show a concrete (re)interpretation of trace instructions that (re)creates the states of interest that would be observed in the original execution (shown in the left column of boxed items). Ignore the right column of boxed items for now, we describe them in Section 6.2.2. Concrete states consist of a concrete stack of activation records on the left side of the $\shortparallel$ and a concrete heap on the right (i.e., *stack* $\shortparallel$ *heap*). An activation record (i.e., a value stack) is represented by a sequence of values separated by commas, while the $\triangleleft$ symbol is used to separate activation records. Stacks grow to the right (i.e., the rightmost element is the top of the stack). For example from point 3 to 4, we have pushed o'' onto the top of the current activation after executing an allocation instruction (`alloc`), while from point 4 to point 5, we have pushed a new activation record onto the activation stack after executing method call and method

```
x = new X(); x.c = new C(); d = new D();
c = x.getCIfOk(d); return c.f;

C getCIfOk(C ow) { return this.c != null ? this.c : ow; }
```

(a) source code

1	$\boxed{\cdot \shortparallel \cdot}$	$\boxed{\cdot \shortparallel \cdot \shortparallel \cdot}$
	alloc o'^{X}; dup; alloc o^{C}; †	
2	$\boxed{\mathsf{o}', \mathsf{o}', \mathsf{o} \shortparallel \cdot}$	$\boxed{\beta, \beta, \alpha \shortparallel \cdot \shortparallel \beta : \langle 1, 1\rangle, \alpha : \langle 1, 1\rangle}$
	putfield $\mathsf{o}'^{\mathsf{X}}.\mathsf{c}$;	
3	$\boxed{\mathsf{o}' \shortparallel (\mathsf{o}', \mathtt{X.c}) : \mathsf{o}}$	$\boxed{\beta \shortparallel (\mathsf{o}', \mathtt{X.c}) : \alpha \shortparallel \beta : \langle 1, 1\rangle, \alpha : \langle 1, 1\rangle}$
	alloc $\mathsf{o}''^{\mathsf{D}}$;	
4	$\boxed{\mathsf{o}', \mathsf{o}'' \shortparallel (\mathsf{o}', \mathtt{X.c}) : \mathsf{o}}$	$\boxed{\begin{array}{l}\beta, \gamma \shortparallel (\mathsf{o}', \mathtt{X.c}) : \alpha \shortparallel \\ \beta : \langle 1, 1\rangle, \alpha : \langle 1, 1\rangle, \gamma : \langle 1, 1\rangle\end{array}}$
	call $\mathsf{o}'^{\mathsf{X}}.\mathsf{n}$; enter $\mathsf{o}'^{\mathsf{X}}.\mathsf{n}$; swap; dupx;	
5	$\boxed{\cdot \triangleleft \mathsf{o}', \mathsf{o}'', \mathsf{o}' \shortparallel (\mathsf{o}', \mathtt{X.c}) : \mathsf{o}}$	$\boxed{\begin{array}{l}\cdot \triangleleft \beta', \gamma', \beta' \shortparallel (\mathsf{o}', \mathtt{X.c}) : \alpha \shortparallel \\ \alpha : \langle 1, 2\rangle, \beta' : \langle 2, 2\rangle, \gamma' : \langle 1, 2\rangle\end{array}}$
	getfield $\mathsf{o}'^{\mathsf{X}}.\mathsf{c}$; dup;	
6	$\boxed{\cdot \triangleleft \mathsf{o}', \mathsf{o}'', \mathsf{o}, \mathsf{o} \shortparallel (\mathsf{o}', \mathtt{X.c}) : \mathsf{o}}$	$\boxed{\begin{array}{l}\cdot \triangleleft \beta', \gamma', \alpha, \alpha \shortparallel (\mathsf{o}', \mathtt{X.c}) : \alpha \shortparallel \\ \alpha : \langle 1, 2\rangle, \beta' : \langle 2, 2\rangle, \gamma' : \langle 1, 2\rangle\end{array}}$
	nullcheck o; †	
7	$\boxed{\cdot \triangleleft \mathsf{o}', \mathsf{o}'', \mathsf{o} \shortparallel (\mathsf{o}', \mathtt{X.c}) : \mathsf{o}}$	$\boxed{\begin{array}{l}\cdot \triangleleft \beta', \gamma', \alpha \shortparallel (\mathsf{o}', \mathtt{X.c}) : \alpha \shortparallel \\ \alpha : \langle 2, 2\rangle, \beta' : \langle 2, 2\rangle, \gamma' : \langle 1, 2\rangle\end{array}}$
	exit $\mathsf{X}.\mathsf{n}$; returnfrom $\mathsf{X}.\mathsf{n}$;	
8	$\boxed{\mathsf{o} \shortparallel (\mathsf{o}', \mathtt{X.c}) : \mathsf{o}}$	$\boxed{\alpha \shortparallel (\mathsf{o}', \mathtt{X.c}) : \alpha \shortparallel \alpha : \langle 1, 2\rangle}$
	getfield $\mathsf{o}^{\mathsf{C}}.\mathsf{f}$; †	

(b) trace instructions

Figure 6.3: Concrete and symbolic trace interpretation of a short example. The left-hand–side of the figure is explained in Section 6.2.1, the right in Section 6.2.2.

enter instructions (`call` and `enter`). There are a few basic instructions that manipulate the value stack: `dup` duplicates the top value, `dupx` duplicates the top value while placing it under the top two values, and `swap` swaps the top two values. We write $\cdot$ for an empty state element (stack or heap). A heap is a map from object-field pairs to objects. For example, at point 3, we have the mapping $(\mathsf{o}', \mathtt{X.c})\!:\!\mathsf{o}$ in the heap, which means "field `X.c` of object o' contains the value o." The other instructions note a check for null (`nullcheck`), method exit and return (`exit`, `returnfrom`), and uses of fields (`getfield`, `putfield`).

Some instructions contain elements of the original execution state when the instruction was generated. For example, `alloc` o'^{X} at point 1 has as usual a type X but also an object identifier o' (e.g., address) of the allocated object. These pieces of the original concrete state serve to include concrete information for combined concrete-symbolic reasoning (e.g., somewhat similar to [49, 94]).

6.2.2 Measuring: Where, What, and How

Recall that an enforcement window is an establish-check-use sequence. Depending on the property of interest, particular trace instructions will correspond to establish, check, and use events. In the case of dereference reasoning, the **establish** is an `alloc`, followed by some number of `nullcheck` **check**s, and finally a `getfield`, `putfield`, or `call` **use** on the same object reference. For example, we have the establish-check-use sequence highlighted and marked by †s in Figure 6.3b (i.e., the sequence for the value dereferenced with `c.f` at the source-level).

The central question is given such a dynamic trace, where, what, and how can we measure to find candidate validation scopes with a static, source code notion of complexity. We devote the rest of this section to this question. Finding candidate validation scopes corresponds closely to **where** we measure (Choice 1). The measure of complexity is determined primarily by **what** we measure. We consider complexity in different dimensions (Choice 2) and what makes something more or less complex (Choice 3). One particularly interesting distance metric that we define is **inlining depth** alluded to earlier. A static, source code view is one that considers (all) other possible executions than the one observed, which typically requires some over-approximation of possible behavior. A key observation in this paper is that by controlling **how** we measure, we can model "loss of information" due to over approximation (Choices 5, 6, and 7). Such modeling necessitates intertwined concrete-symbolic reasoning and motivates **symbolic trace interpretation**.

Where to Measure: Defining the measurement points in an enforcement window.

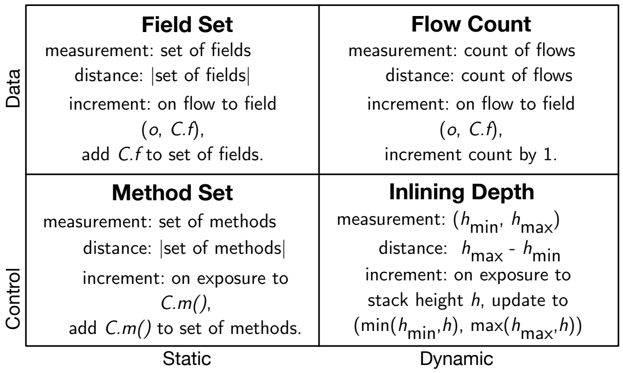
Choice 1 (Measurement Points): *In an enforcement window, which pairs of points are of interest?*

There are several potentially interesting points, any of which can be measured with our framework. In Section 6.4, we focus on uses and their closest check (or establish if there is no check). This distance captures the smallest validation scope needed to show that the use in this trace is non-faulting.

What to Measure: Defining the metric.

Choice 2 (Measurement Dimensions): *What kinds of events contribute to the complexity of a validation scope?*

We consider two orthogonal dimensions that we hypothesize affect validation complexity: control versus data reasoning. First, **control** reasoning is what code or statements would a developer have to reason about to make sure that a dereferenced value is not null. The events that we record for control reasoning are the methods that a value is **exposed** to as it travels from an enforcement to a use (i.e., an enforcement and a use in the same method has the minimum measurement). We use the term **expose** to refer generically to observing an event that updates a measurement. We chose methods as our atomic unit of distance because they capture a source code view of the program that is always preserved by compilation to bytecode, unlike control structures or statements. Second, **data** reasoning is the memory locations that the programmer must reason about to ensure that a dereference will not fault. For this dimension, we record the fields that a value flows through between an enforcement and a use. Discovering validation scopes with respect to data reasoning might help determine where coarse heap abstractions are sufficient and where they need to be more precise.



	Static	Dynamic
Data	**Field Set** measurement: set of fields distance: |set of fields| increment: on flow to field (o, $C.f$), add $C.f$ to set of fields.	**Flow Count** measurement: count of flows distance: count of flows increment: on flow to field (o, $C.f$), increment count by 1.
Control	**Method Set** measurement: set of methods distance: |set of methods| increment: on exposure to *C.m()*, add *C.m()* to set of methods.	**Inlining Depth** measurement: $(h_{\text{min}}, h_{\text{max}})$ distance: h_{max} - h_{min} increment: on exposure to stack height h, update to $(\text{min}(h_{\text{min}},h), \text{max}(h_{\text{max}},h))$

Figure 6.4: Distance metrics capturing combinations of control versus data reasoning and static versus dynamic reasoning.

Choice 3 (Increments of Measure): *How do interesting events (e.g. calls or field writes) increase measured distance?*

For the data and control reasoning dimensions identified in Choice 2, what events capture an increase in complexity?

In Section 6.4, we take measurements using four different distance metrics: Field Set, Flow Count, Method Set, and Inlining Depth. Inlining Depth is particularly interesting from a static analysis design perspective because it captures needed context-sensitivity—a standard concept. Thus, in Figure 6.3b, we use Inlining Depth as the example distance metric to illustrate symbolic trace interpretation.

To describe what is Inlining Depth and why we measure it, consider the source code of our running example in Figure 6.3a. The last enforcement for the use of `c.f` is the null check in the call to `x.getCIfOk(d)`. Thus, a validation scope for `c.f` must also include `getCIfOk`. We want to capture the additional power needed to reason across a method call. In particular, we want to measure the **inlining depth** that is needed to bring the path that the value takes from the enforcement to the use all into the same method (i.e., into the unit scope). Note that this measurement is different and more nuanced than simply counting the number of method calls in the dynamic trace between the enforcement and the use. Consider the fragment: `assert this.o != null; this.m1().m2(); this.o.f = 0;` where `m1` and `m2` are leaf methods (i.e., they do not make further calls), then the needed level of context is to check that o is not null is 1, not 2. In contrast to counting method calls in a trace, measuring Inlining Depth requires symbolic trace interpretation.

To see what needs to be measured for the inlining depth metric, consider the call tree shown in Figure 6.5. Each node represents a method, and each edge indicates a call from the source to the target node. The simple case is when the use is downwards along a call path (e.g., the enforcement is in m_0 and the use is in m_2), then the inlining depth is the length of the path between them (e.g., 2). The general case is that an enforcement happens in a previously called and returned method (e.g., the enforcement is in m_2 while the use is in m_4). The inlining depth is then the difference between the height of the shallowest method that the value has traveled through and that of the deepest method (e.g., m_0 and m_2, respectively, leading to a depth of 2). Thus, we measure a pair of integers $\langle h_{\text{min}}, h_{\text{max}} \rangle$ summarizing the lowest and highest activation stack height to which a value has been exposed since its last enforcement.

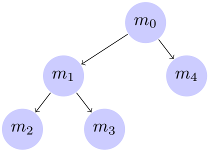


Figure 6.5: Call tree for inlining depth.

In Figure 6.3b, the right column of boxed items shows a symbolic interpretation with Inlining Depth. Consider the symbolic state at point 3. The state consists of three components, separated by ||. On the left is the symbolic stack. We use letters $\alpha, \beta, \ldots$ for symbolic values (i.e., symbolic object identities), which represent concrete values (i.e., concrete object identities). For the moment, however, we can view β and α as simply the names of o′ and o, respectively, in the symbolic world. In the middle, we have the symbolic heap; ignore this component for now, as we detail it under Choice 6. Finally, the rightmost component of the symbolic state associates measurements with symbolic object identities (e.g., $\beta : \langle 1, 1 \rangle$); that is, it tracks an event history summary with each value individually and independently. The domain of measurements is what would vary from metric to metric.

At point 2 in Figure 6.3b, both objects β and α have just been established as non-null (by an allocation), so we have that both $\beta\!:\langle 1,1\rangle$ and $\alpha\!:\langle 1,1\rangle$ since the height of the current activation stack is 1. At points 3 and 4, these facts do not change. The $\texttt{putfield o}'^{\texttt{X}}\texttt{.c}$ instruction pops arguments from the value stack and writes to the heap, and $\texttt{alloc o}''^{\texttt{D}}$ creates a new symbolic value γ with fact $\langle 1,1\rangle$. On a call, we set $h_{\max}$ to the new height if that height is greater than $h_{\max}$, as the value has now been exposed to a height one more call step away. So, for example, at point 5, $h_{\max}$ is incremented for α (i.e., $\alpha\!:\langle 1,2\rangle$).

The measurements for β' and γ' are discussed in the next choice (Choice 5). The measurement for α is the same until encountering the `nullcheck o` instruction, which is a null-check on α, and thus resets its measurement to the current stack height (i.e., $\alpha\!:\langle 2,2\rangle$ at point 7). The action for a return is analogous to the call, except that $h_{\min}$ is updated, as from point 7 to point 8 for α. The inlining depth (i.e., the distance measure of interest) is then given by $h_{\max} - h_{\min}$. If, for example, α were to be dereferenced at point 5 where $\alpha\!:\langle 1,2\rangle$, then the check-use distance for this inlining depth metric would be 1, as expected; if it were dereferenced at point 7 after the check, the distance would be 0.

The symbolic interpretation for our other metrics is analogous, but uses a different domain of measurements. We summarize these in Figure 6.4 where we classify the metrics along the data versus control dimension and also along a spectrum from more static to more dynamic. For example, the Flow Count metric is a rather dynamic, execution-based view that counts the number of copies from field to field from the check until the use. This metric counts all copies, even those between the same fields of different objects (e.g., between `o1.f` and `o2.f`). The Field Set metric instead counts only the number of distinct fields through which a value flows. Method Set is the control reasoning analogue that counts the number of distinct methods to which a value is exposed.

How to Measure: Defining measurements so that they relate to source code views.

Choice 4 (Static vs. Dynamic Analysis): *Should we measure enforcements windows with a static or a dynamic analysis?*

We use **dynamic** analysis to measure enforcement windows. An initially attractive alternative would be to do so **statically** because validation scopes are inherently a static, source code notion. But upon further inspection, we see that this approach is quite problematic. For one, a "fully precise" pointer analysis, which remains a difficult problem [57], is required to statically tie together establish-check-use sequences on an object. In particular, any imprecision in the static analysis could cloud what we find—which is especially problematic when the goal was to find validation scopes to rule out insufficient static analysis designs.

Dynamic analysis is attractive because, fundamentally, it is easier to lose precision than to get it in the first place. In Choices 5 and 6, we show how we use symbolic trace interpretation to selectively forget concrete information from a dynamic trace in order to selectively emulate how a static analysis would reason about a program. However, a potential disadvantage of any dynamic analysis is that its quality depends on how well the collected traces generalize to cover all possible executions (we evaluate this in Section 6.3.3).

Choice 5 (Applying Measurements to Objects): *How do we connect measurements to the object that they measure?*

The two most obvious choices seem wrong. Just keeping a measurement map from concrete object identities o to measurements corresponds to the questionable assumption that perfect aliasing information is available statically. Alternatively, keeping an abstract stack and heap of measurements like in a standard type system corresponds to assuming the static analysis is incapable of resolving any aliasing.

Instead, we measure over symbolic object identities that allow us to "lose" or "forget" aliasing information known dynamically in a controlled and selective manner. Specifically, two different symbolic object identities may represent the same concrete object (modeling lost aliasing information). At point 5 in Figure 6.3b, we use a fresh symbolic object β' for the receiver in the callee as opposed to reusing the value β from the caller. While both β' and β correspond to same concrete object o', we have chosen to forget this information in the symbolic state. In this case, we make this split to capture: (1) in any method, the receiver object `this` is known to be non-null, so from the prospective of the callee, the `call` instruction is a check on the receiver; but (2) from the prospective of the caller, the `call` is simply a use/dereference of the receiver. Thus, in our example, β', the receiver in the callee after the call, is summarized by $\langle 2, 2\rangle$ (i.e., checked in the callee). The parameter in the callee, γ', is also a fresh symbolic object identity where both γ and γ' correspond to the concrete object $\mathtt{o}''$. The measurement on γ' (i.e., $\langle 1, 2\rangle$) is derived from γ's, but the cloning means any check in the callee on $\mathtt{o}''$ is not seen by the caller, unless it is passed back in the return value or through the heap. This approach respects the implicit modularization implied by function boundaries—checks on a value escape a function only if the value itself does.

Choice 6 (Memory Model): *How is the memory modeled symbolically?*

In symbolic interpretation, the symbolic state is a model of the concrete state. In Figure 6.3b, we use an exact model of the activation stack except that the values are symbolic rather than concrete: that is, the identity of a called method is exactly known, but the receiver and parameters are interpreted symbolically. Similarly, each cell in our symbolic heap is identified exactly (by a concrete object identity and a field) but the **contents** of those cells are symbolic values. So, for example, at point 3 the symbolic heap maps field `X.c` of object $\mathtt{o}'$ to symbolic value α. Heap accessing instructions operate on combined symbolic and concrete values; the symbolic values come from the symbolic value stack, while the concrete values are explicitly incorporated into select trace instructions.

The symbolic interpretation of `getfield` $\mathtt{o}'^{\mathsf{X}}$`.c` from point 5 to 6 pops the symbolic value β' for the field owner from the value stack but then uses the **concrete** annotation $\mathtt{o}'$ to look up the symbolic value stored in the symbolic heap at field `X.c` of $\mathtt{o}'$, which is α, that is then pushed onto the stack. The interpretation of `putfield` is similar. Analogous to the modeling of copying actuals to formals discussed above under Choice 5, there is a choice in whether (a) to copy the symbolic value α from the heap to the stack or (b) to create a fresh symbolic value with a copy of the measurement. The former means a measurement update via the stack is reflected in the heap value and vice versa, while the latter "forgets" this aliasing relationship. In this case, we have chosen (a) to capture that enforcements on stack values obtained from instance variables (i.e., fields of `this`) (or vice versa) should seemingly apply in both places. Another reasonable option could choose (a) in some cases (e.g., only dereferences of fields of `this`) and (b) in other cases. There is no clear best choice regarding aliasing "remembering" and "forgetting," so importantly, our framework supports experimenting with different modeling decisions by switching between (a) and (b).

In essence, we record measurements on symbolic values but use concrete values to determine storage locations on the heap. Without the latter use of concrete values, the symbolic trace interpretation would itself need a precise static points-to analysis. Critically, this intertwining of concrete and symbolic modeling enables us to model source code reasoning in some respects while avoiding unrealistic static analysis imprecisions in other ways.

A significant implementation challenge is updating measurements for all heap-stored values (see Section 6.3.2). With Inlining Depth, for example, on every entry to and return from a method, we need to expose every value on the heap to the new activation stack height using the scheme laid out in Choice 3. So, for example, the measurements for α for the call between point 2 and point 5 change to reflect α's exposure to an activation stack with height 2. Similarly, the return from point 7 to point 8 exposes α to height 1.

Choice 7 (Measuring the Unknown): *How do values from uninstrumented library code contribute to check-use measurements?*

Some values will necessarily come from uninstrumentable code (e.g., libraries). We assign these values the measurement lib. In interpreting our measurements, we take the conservative viewpoint that a value returned from unknown code adds an unknown distance that must be viewed over-approximately as "infinite." Informally stated, unvalidated assumptions are made about the code outside of the validation scope, but unbounded inlining would be sufficient to validate those assumptions once the code is brought in. An alternative, optimistic approach would assign library values 0 distance indicating that libraries are understood through documentation of invariants and thus do not require reasoning about code at all.

6.3 Measurement Framework

In this section, we describe our symbolic trace interpretation framework (Section 6.3.1), discuss techniques for scaling our implementation of the symbolic heap (Section 6.3.2), and investigate the extent to which our dynamic measurements of enforcement windows are sufficient from a static analysis perspective (Section 6.3.3).

6.3.1 Symbolic Trace Interpretation

The key challenges addressed by this framework are (1) how to extract a more static view of an execution by forgetting run-time information in a principled way (see Section 6.2.2, Choice 5) and (2) how to meaningfully interact with uninstrumented library code. We accomplish (1) by using an intertwined concrete and symbolic state, associating information with symbolic values, and instantiating new symbolic values when we want to forget. With this approach, a single concrete value can be represented by multiple symbolic values. We address (2) by splitting method call and returns into separate instructions that captures the call or return event from the caller's and the callee's perspectives individually.

We first focus on describing our generic framework instantiated for measuring control reasoning using the inline depth metric as an example. An activation record $A ::= \cdot \mid A, \alpha$ consists of an operand stack with symbolic object identities; the symbol $\cdot$ indicates an empty stack. Then, we have a stack of activations $S ::= \cdot \mid S \triangleleft A \mid S \triangleleft \mathsf{unins}$, which consist of normal activations A but also uninstrumented activations unins. Informally, unins models some number of activations for uninstrumented methods. A heap $H ::= \cdot \mid H, (o, f) : \alpha$ is a finite map from a **concrete object, field pair** to the symbolic value stored in the field for that object. Observe that the heap is a mixed concrete-symbolic entity. A measurement map $\Gamma ::= \cdot \mid \Gamma, \alpha : t$ is a finite map from symbolic identities to the recorded measurement for that symbolic value, and a symbolic state $\Sigma ::= S \shortparallel H \shortparallel \Gamma$ is a triple of a stack of activations, a heap, and a measurement map. A measurement $t ::= \langle h_{\min}, h_{\max} \rangle \mid \mathsf{lib}$ can be either a known measurement or lib, indicating that the value came from uninstrumented

ALLOC

$$\frac{\alpha \notin \mathrm{dom}(\Gamma) \qquad t = \mathrm{enf}(S \triangleleft A \shortparallel H \shortparallel \Gamma)}{S \triangleleft A \shortparallel H \shortparallel \Gamma \vdash \texttt{alloc}\, o^{C} \Downarrow S \triangleleft A, \alpha \shortparallel H \shortparallel \Gamma, \alpha\!:\!t}$$

CALL-INS

$$\frac{\begin{array}{c}\beta', \alpha' \notin \mathrm{dom}(\Gamma) \\ S' = S \triangleleft A \triangleleft \beta', \alpha' \qquad \Sigma = S' \shortparallel H \shortparallel \Gamma \qquad \Gamma' = \Gamma,\ \mathrm{expose}(\Gamma|_{\mathrm{rng}(H)}, \Sigma),\ \beta'\!:\!\mathrm{enf}(\Sigma),\ \mathrm{expose}(\alpha'\!:\!\Gamma(\alpha), \Sigma)\end{array}}{S \triangleleft A, \beta, \alpha \shortparallel H \shortparallel \Gamma \vdash \texttt{call}_{\texttt{ins}}\, o^{C}.m \Downarrow S' \shortparallel H \shortparallel \Gamma'}$$

RETURNFROM-INS

$$\frac{S' = S \triangleleft A_1, \alpha \quad \Gamma' = \Gamma,\ \mathrm{expose}(\Gamma|_{\mathrm{rng}(H) \cup \{\alpha\}}, S' \shortparallel H \shortparallel \Gamma)}{S \triangleleft A_1 \triangleleft A_2, \alpha \shortparallel H \shortparallel \Gamma \vdash \texttt{returnfrom}_{\texttt{ins}}\, C.m \Downarrow S' \shortparallel H \shortparallel \Gamma'}$$

GETFIELD-INS

$$\frac{(o, C.f) \in \mathrm{dom}(H) \qquad \beta = H(o, C.f)}{S \triangleleft A, \alpha \shortparallel H \shortparallel \Gamma \vdash \texttt{getfield}\, o^{C}.f \Downarrow S \triangleleft A, \beta \shortparallel H \shortparallel \Gamma}$$

Figure 6.6: Symbolic trace interpretation for inlining depth.

code. When instantiated for the inlining depth metric, known measurements consist of a pair of integers $\langle h_{\min}, h_{\max} \rangle$ representing the minimum and maximum stack height to which the value has been exposed. The measurements are the only portion of the symbolic state that change from metric to metric. We write $\Gamma(\alpha)$ for looking up the measurement associated with symbolic object α in Γ and $\Gamma, \alpha\!:\!t$ for a map that either extends Γ with a binding for α or updates the binding of α to t if it exists. Similarly, $H(o, C.f)$ looks up a value at field $C.f$ of concrete object o in H and $H, (o, C.f)\!:\!\alpha$ extends it.

We define an interpretation judgment $\Sigma \vdash I \Downarrow \Sigma'$ in Figure 6.6 that states, "In state Σ, instruction I symbolically evaluates to Σ'." The trace instruction language is the same as in the example from Figure 6.3, except that we explicitly annotate `call` and `returnfrom` instructions with whether the called or returned-from method is instrumented (`ins`) or uninstrumented library code (`unins`). For completeness, we give the full trace language supplementally [30].

For non-null dereference analysis, an allocation is an establish event. Rule ALLOC pushes a fresh value α onto the stack with a measurement for an enforcement event (i.e., an establish or a check) in the current state. Under the inlining depth metric, this measurement has both $h_{\min}$ and $h_{\max}$ set to the current stack height. That is, we define $\mathrm{enf}(\Sigma) \overset{\text{def}}{=} \langle \mathrm{heightof}(S(\Sigma)), \mathrm{heightof}(S(\Sigma)) \rangle$

where the function $\mathrm{heightof}(S)$ gives the number of activations in stack S and $S(\Sigma)$ gives the stack component of the symbolic state Σ. Recall that the inlining depth for a measured exposure is given by $h_{\max} - h_{\min}$, so a use right after the allocation yields a 0 distance as intended. A `nullcheck` is essentially the same except that it updates the measurement for the object on the top of the stack (NULLCHECK rule elided here), as it is just another enforcement. For other properties, other instruction kinds may be identified as the enforcement events, but they have the same form: interpreting the semantics of the instruction along with asserting an enforcement in the measurements.

At a call to an instrumented method (rule CALL-INS), we create a fresh symbolic value to represent the receiver β' and assign it the enforcement measurement in current state. This constraint captures that the receiver is null-checked at this point from the callee's perspective (since `this` cannot be `null`) but it is not from the caller's viewpoint. Contrast this modeling with that for the parameter value α. It is assigned a new symbolic value in the callee α' so that checks in the callee do not automatically count in the caller. The measurements for that value are copied between the caller and the callee before exposing it to the new state in the callee. The $\mathrm{expose}(\alpha : t, \Sigma)$ function updates the measurement for object α to reflect exposure to a state Σ. Under the inlining depth metric, we define this as: $\mathrm{expose}(\alpha\!:\!\langle h_{\min}, h_{\max}\rangle, \Sigma) \stackrel{\mathrm{def}}{=} \alpha\!:\!\langle \min(h_{\min}, h), \max(h_{\max}, h)\rangle$ and $\mathrm{expose}(\alpha : \mathsf{lib}, \Sigma) \stackrel{\mathrm{def}}{=} \alpha : \mathsf{lib}$ where $h = \mathrm{heightof}(S(\Sigma))$. We lift expose to also apply to maps (i.e., $\mathrm{expose}(\Gamma, \Sigma)$). For control reasoning metrics, all measurements for values on the heap are also updated to reflect their exposure to a state on each call and return (i.e., $\mathrm{expose}(\Gamma|_{\mathrm{rng}(H)}, \Sigma)$). We write $\Gamma|_{\mathrm{rng}(H)}$ for the restriction of map Γ to mappings from symbolic values in the range of the heap H. Observe that this operation is prohibitively expensive to implement directly and motivates techniques described Section 6.3.2. On return (rule RETURNFROM-INS), the top activation is popped and the return value and the heap are exposed to the state.

The complexity of handling uninstrumented methods lies in transitions between instrumented and uninstrumented code. To detect transitions, we split a method call into two events: a `call` instruction, which is the event from the caller's perspective, and an `enter`, which is the event from the callee's perspective. When an instrumented method calls another instrumented method, then we see a `call` immediately followed by an `enter` as in Figure 6.3b. However, critically, this redundancy allows us to detect transitions between instrumented and uninstrumented code robustly. Specifically, we mark a call from instrumented code to an uninstrumented method by pushing an unins marker on to the stack. A call from uninstrumented code to an instrumented method is detected by an `enter` instruction while an unins marker is active. The interpretation of `enter` in this situation is to compensate for the lack of a `call` instruction right before it (and thus is analogous to rule CALL-INS). Method returns are similarly split into `exit` from the callee's perspective and `returnfrom` from the caller's perspective. The interpretation of `returnfrom` must make a similar compensation when it observes a return from uninstrumented code.

For control reasoning, getting and putting a field simply need to reflect the concrete semantics symbolically. Getting a field from an object pops the symbolic field owner off the stack and uses the **concrete** object identifier to look up the symbolic value stored for in that object's field in the heap (if it exists) and pushes it on the stack (GETFIELD-INS). Using concrete heap lookups enables us to factor out a potential source of unrealistic static analysis imprecisions. If the field has not been initialized, it pushes lib instead (as the assumption is that it was initialized in uninstrumented code). A `putfield` updates the symbolic heap to store a symbolic value from the stack in the field for the concrete object (rules are straightforward). For data reasoning, we would update measurements (i.e., apply exposures) on `getfield`s and `putfield`s instead of on `call`s and `returnfrom`s.

In this section, we have instantiated our measurement framework using the Inlining Depth metric. Using our Method Set metric is similar, except that the measurements are sets of method identities and exposing a value to a new state adds the method on the top of the stack to a measurement. How specifically our four metrics are instantiated in this framework is summarized in Figure 6.4.

6.3.2 Implementation: Dynamic Symbolic Heap

Two key challenges hide in the description of symbolic trace interpretation above. First, in defining the symbolic trace interpretation judgment (Figure 6.6), heaps H and measurement maps

Γ only grow. In essence, we assume that garbage is automatically collected from the symbolic heap (i.e., that objects on the heap disappear when they are no longer needed) and the measurement map. However, since the symbolic heap has no knowledge of heap operations in uninstrumented library code, there is no way the interpreter could ever safely garbage collect mappings in the symbolic heap. In our framework, we instrument the garbage collector running in the observed program to "piggyback" collecting an object in the symbolic heap when the object in the concrete heap is collected. Whenever the garbage collector frees a concrete object, the trace collector is signaled to emit a trace instruction telling the trace interpreter to remove that object from the symbolic heap. This "piggybacking" efficiently ensures that objects are only collected from the symbolic heap after they can no longer be used.

The second challenge to scalable symbolic trace interpretation involves updating the measurements for heap values on method calls and returns. In the CALL-INS and RETURNFROM-INS rules, we update every symbolic value on the heap to reflect exposure to a new control scope (i.e., $\text{expose}(\Gamma|_{\text{rng}(H)}, \Sigma)$). Naïvely iterating over the entire symbolic heap on each call and return is far too slow to be practical, even for relatively short programs.

To address this problem, we divide the symbolic values on the heap into **measurement update partitions** that help us update heap exposures more efficiently. We have two partition strategies: one that leverages a property of particular kinds of measurement metrics and one that is metric agnostic but more expensive.

For the Inlining Depth metric, we partition the symbolic values based on their $h_{\min}$ and $h_{\max}$ measurements. Then, on a method call, we only need to update those values whose $h_{\max}$ is the stack height before the call. Similarly, on a return, we need only update those symbolic values whose $h_{\min}$ measurements match the stack height before the return. These partitions are **prescriptive** in that using the measurements tells us exactly which symbolic values need to be updated, and fortunately, they are small enough to speed up interpretation of calls and returns drastically.

We also have a more general, heuristic approach to partition symbolic values on the heap based on how **recently** they were used (added to the heap, read from the heap, or dereferenced).

For example, in our implementation of the Method Set metric we keep a collection of up to 1000 "hot" symbolic values and update their measurements individually whenever they are exposed to a new state. The remainder of symbolic values on the heap are not updated individually on every call and return. Instead, we keep a single set summarizing the recent methods that all of these cold values have been exposed to. The key invariant that we maintain is that (1) the measurements (i.e., method exposure sets) for hot values are exactly what they would have been if we had traversed the entire heap on calls and returns and (2) the measurements for the cold values, unioned with the current summary set, are what they would have been in the naïve system. With this approach, if the program dereferences a hot value, it can record the measurement directly associated with the value. If it is cold, however, we have to first apply a lazy fixup and expose the value to each of the methods in the summarized sets. The direct measurements for that value now completely reflect what its measurements should be, so we safely move it to the hot collection. If the hot collection is full, we apply the summary set to **all** non-hot values in the heap (so that their measurements are now complete), reset the cold summary to be the empty method set, and mark all values as cold. At this point, again, the invariant holds. The essence is that we keep a fixup transformer that can be applied when a cold value gets used.

Both of these approaches make field accesses more expensive, but the savings from avoiding traversing the entire heap on calls and returns more than makes up for them.

6.3.3 Sufficiency for Static Analysis Design

One intended use of our enforcement window measurement framework is to help determine necessary conditions for static analysis design and, in particular, help designers decide at least how much scope their analysis needs to prove a property of interest. Our combined concrete-symbolic approach is well-suited to this task because it permits us to tease apart required scope from over-approximation in any abstract analysis domain. In essence it allows us to measure, for example, the window of code an analysis would need to examine if it was using exactly the right analysis abstraction. Even assuming such perfect reasoning, this measurement is an under-approximation

Table 6.1: Framework Sufficiency for Analysis Design.

		"Full"		"Recommended"	
Program	Interesting	False Dead	Bottle-necked	False Dead	Bottle-necked
antlr	1812	0	36%	330	35%
bloat	4424	0	32%	0	32%
chart	1219	34	48%	88	50%
fop	10164	0	78%	10044	43%
luindex	873	0	40%	290	51%
lusearch	661	0	56%	0	56%
pmd	830	0	66%	63	69%

for the validation scope that the hypothetical analysis would need because, as a dynamic analysis, our approach cannot measure the required scope for all possible paths in a given program. In this section, we investigate whether this under-approximation is **sufficient** in this sense: that is, to what extent a single execution discovers enough enforcement sites to determine a useful validation scope.

To do this, we first instantiated our measurement framework for the non-null dereference property to record the **location** of the closest enforcement (i.e., allocation or comparison to null) for each dereference. A dereference may have multiple such closest enforcements if it is called from different contexts. We then used the WALA framework[1] to run an interprocedural static analysis that examines each dynamically observed dereference site and verifies that all static paths in the control-flow graph to that site pass through at least one of the closest dynamically observed enforcements: if so, our approach is sufficient for that site.

The results of these experiments for a subset of the DaCapo benchmarks are shown in Table 6.1. Here we consider a dereference site "Interesting" if (1) it is executed in our dynamically observed run, (2) it does not dereference values from uninstrumented library code (i.e., a static analysis looking at only application code would have some hope of proving the dereference safe), and (3) it is not a dereference of `this` (which in Java cannot be null). WALA has some unsoundness in its handling of reflection, leading it to claim that some executed dereferences are not reachable. We ran WALA with two different reflection policies. "Full" makes a best effort to determine reflective

[1] T. J. Watson Libraries for Analysis (WALA), http://wala.sf.net/

method targets while “Recommended” (which was recommended to us by a WALA developer) optimistically assumes programmers’ casts after reflective instantiations are correct and uses these casts to determine the type of allocated objects. Because these policies are heuristic, they may falsely claim that some dynamically observed dereference sites are dead code. We give the number of these sites in the “False Dead” column.

The “Bottlenecked” column gives the percentage of interesting, statically reachable dereference sites for which all static paths to that site pass through a dynamically observed closest enforcement; that is, the closest enforcements are a **bottleneck** to reaching the use. The observation that a large percentage (30%–70%) of dereference sites are statically shown to flow through the dynamically observed enforcement sites gives us evidence that our approach finds candidate validation scopes that are likely to be useful: that is, that inferences gleaned from these these enforcement sites in one run (e.g., their typical enforcement distances) are likely to be representative for all possible runs of the program. Note that the non-minuscule bottlenecked percentages in Table 6.1 are significant: even under the very pessimistic assumptions that (1) we are allowed only one dynamic execution, and (2) we count only the last enforcement along that execution, our dynamic analysis frequently finds useful validation scopes. Our benchmarks range in size from ∼3,000 (antlr) to ∼25,000 (fop) methods. The bottlenecked percentage does not change much between the “Full” and “Recommended” configurations (except for fop), perhaps indicating an invariance property about enforcements across dereference sites.

6.4 Measurements

In this section, we apply our measurement framework to gain insights into how enforcements actually appear in code. We have two sets of experiments that both measure the distance between a use and its closest enforcement. The first evaluates distance metrics from Sections 6.2.2 and 6.3 with a case study of null pointer exception bugs that tests three hypotheses: (1) that programmers find it easier to reason across short enforcement distances than long ones, (2) that fixing bugs shortens these distances, and (3) that as code bases mature, programmers respond to increasing complexity with more defensive programming. The main challenge here was the laborious process of sifting through project issue queues and software repositories to find suitable bugs and then generating inputs that both exercise the buggy sites and remain valid across multiple versions of the projects. Our second set of experiments measures the distribution of enforcement distances over the DaCapo benchmark suite to characterize the size of potential validation scopes in typical programs.

6.4.1 Case Study: Bugs and Program Evolution

This case study covers two programs in depth: PMD, a "programming mistake detector" that analyzes source code to find style violations, and Lucene, a document indexing and search tool. We perform three experiments to test a hypothesis that programmers find it easier to reason across short distances than long ones. First, we investigate how enforcement distances change after programmers fix bugs. We hypothesize that fixed bugs are likely to exhibit shorter distances since the programmer must convince herself that the bug is, in fact, fixed. Second, we look at how buggy dereference sites differ from normal sites—if longer distances are harder to reason about, we would expect to see more bugs at longer sites. Finally, we hypothesize that as programs mature and grow more complicated, programmers will need to adopt more defensive strategies and thus enforce shorter distances, so we examine how these distances change over the lifetime of projects.

Benchmark Selection. To find buggy dereference benchmarks, we were constrained by the following requirements: (1) Projects must have source repositories to get versions of the code before and after a bug fix. (2) They must have a bug database with at least 20 reported `NullPointerException` bugs so that we had a reasonable chance of triggering a buggy dereference site. (3) We limited our search for benchmarks to non-GUI programs since instrumentation slows down execution enough to make analysis of interactive programs impractical. (4) We require representative inputs over which to run our benchmarks. These constraints led us to the DaCapo suite, though we looked broadly at several open source repositories. Based on the DaCapo `small` inputs, for PMD our inputs check a file from its own source base for a variety of style violations, while for Lucene we index short portions of Shakespeare poems and then search them for the term

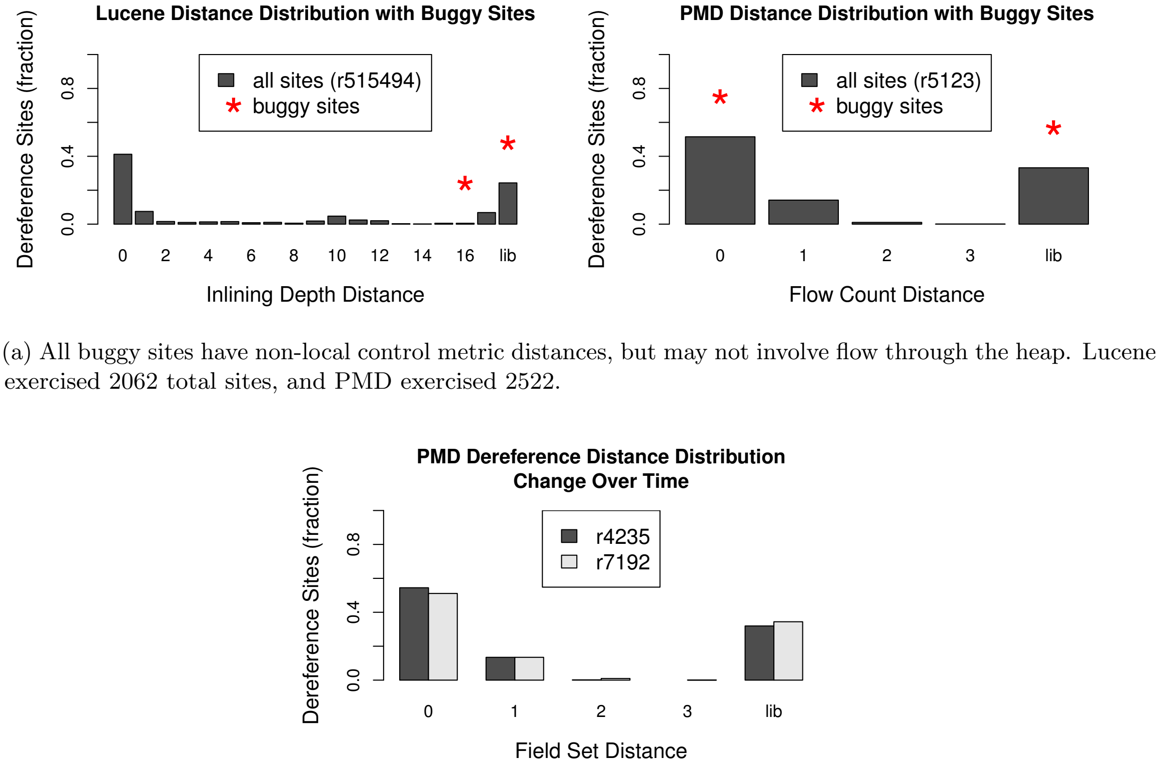
Table 6.2: Distances get shorter after bug fixes.

	Data Metric		Control Metric	
Issue	Flow Count	Field Set	Inlining Depth	Method Set
Lucene-825	lib → 0	lib → 0	lib → 0	lib → 1
Lucene-449	lib → 0	lib → 0	lib → 0	lib → 1
Lucene-174	lib → 0	lib → 0	lib → 0	lib → 1
Lucene-317	1 → 0	1 → 0	16 → 0	lib → lib
PMD-1425772	0 → 0	0 → 0	1 → 1	2 → 2
PMD-1529805	0 → 0	0 → 0	1 → 0	2 → 1
PMD-1552820	lib → 0	lib → 0	lib → 0	lib → 1
PMD-1728716	lib → lib	lib → lib	lib → lib	lib → lib

“death.”

Bug Selection. We searched each project’s bug report database for instances of the word “NullPointerException.” After filtering out unfixed bugs, we examined the reports to determine if they truly represented null pointer errors. For these candidates, we used the backtrace, patch date, patch author, mailing list comments, and repository logs to find the failing dereference and the source control revision numbers immediately before and after the fix. If the fix was applied across multiple commits, we used the latest revision. If the bug report spurred discovery of multiple related bugs, we only considered the original site. We removed the bugs where either revision did not exercise the buggy dereference site on the representative input. Although the site is exercised in all the remaining revisions, the bugs themselves do not manifest on the representative inputs and in some cases not all of the added code in the fixes is exercised. Overall, we obtained eight bug reports with before and after revisions on which to measure check-use distances.

Do Enforcement Distances Get Shorter After Bug Fixes? To determine whether enforcement distances get shorter after bug fixes, we annotated the buggy dereference sites for each of the bugs collected and interpreted them to collect the maximum distance at those sites before and after the fix. Table 6.2 shows how these distances changed after the programmer fixed the bug. Treating unknown values (e.g., from library code) as “infinite” distances, we make the following observations: (1) None of the distances get longer after a bug is fixed. The majority (five) get



(a) All buggy sites have non-local control metric distances, but may not involve flow through the heap. Lucene exercised 2062 total sites, and PMD exercised 2522.

(b) Distances change little over time, even as the number of sites grows from 1882 to 3205.

Figure 6.7: Studying reported buggy dereference sites and the code evolution using enforcement distances.

shorter, while three stay the same. (2) Of the distances that get shorter, most (four out of five) are “infinite” before the fix. (3) All of the distances that do get shorter go to the minimum possible distance under each metric; that is, the use and check are in the same method. For these buggy sites, the Flow Set and Flow Count distances are identical, although, as we show in Section 6.4.2, this is not always the case.

```
File dir = ...;
String[] fs = dir.list();
for (i = 0; i < fs.length; i++) {}
```

Figure 6.8: Misuse of the `java.io.File` API.

The nature of the bugs themselves are also quite interesting. Three of the Lucene bugs (825, 449, and 174) arise from related misuses of the `java.io.File` API to iterate through a directory (as shown in Figure 6.8). The developer fails to realize that `dir.list()` can return a null array if the program lacks privileges to read the directory, leading the dereference `fs.length` to raise a null pointer exception (similar to Figure 6.3a but with a mistaken assumption). The fixes for the three different bugs caused by this misunderstanding were also similar: the developer checks `files` for null and throws a more meaningful exception. In the fourth Lucene bug (317), a `lock` instance variable is set to null when threading is disabled, but the code calls `lock.unlock()` without checking to see if it is non-null.

```
ClassOrInterface p = node.getFirstParentOfType(ClassOrInterface.class);
if (p.isInterface()) {
  ...
}
```

Figure 6.9: Program evolution violates programmer expectation about a tree invariant.

For PMD, two of the bugs result from misuse of a utility function to query a node in the abstract syntax tree about its first ancestor of a given class (1425772, 1529805). In both cases, the programmer did not realize that such a parent may not exist and that the returned ancestor might be null (shown in Figure 6.9). Here the introduction of `enum` types in Java 5.0 broke the programmer's assumption that all nodes must have a containing class or interface. When the query returns null, the call to `p.isInterface()` throws a null pointer exception. A third PMD bug (1552820) arose from a similar query about a potentially missing child of a node. The final PMD bug (1728716) involved erroneously passing `null` to a string escape utility method. Although the bug was fixed, our analysis does not see any shortened distance because our inputs do not exercise the new check in the fix.

Overall in our case study we found that enforcement distances tend to get smaller after bug fixes. This shortening helps to validate our choice of distance metrics, since we would expect a high-quality metric to show shorter enforcement windows after a bug fix. Further, the fact that most bugs involve reasoning about larger (greater than minimum) distances and most bug fixes reduce the distance to the minimum indicates that programmers are comfortable reasoning locally (within a method) but are less capable of reasoning about non-local computation. Again, this is what we would expect, but now we have empirical evidence supporting this belief, gathered by examination of software artifacts.

Do Bugs Tend to Have Long Enforcement Distances? We investigate whether buggy dereferences have longer enforcement distances by comparing the distribution of distances for all dereference sites to that for buggy sites. Figure 6.7a shows the distribution of all dereference sites for Lucene under the Inlining Depth control metric and for PMD under the Flow Count data metric. Buggy sites are marked with stars. For these graphs the "all sites" distribution comes from the latest before-fix revision that we analyzed—since the "all sites" distribution does not change much over time (discussed below), these graphs are representative of how buggy sites compare to all sites. We give plots of the other metrics for both benchmarks supplementally [30]—they are visibly consistent with these representative graphs.

For the control-based metrics (Inlining Depth and Method Set), the fraction of of all dereference sites that require only local reasoning (i.e., minimum distance) is significant and remarkably consistent across benchmarks—about 40% of the total 4821 dereference sites measured. Yet **none** of the buggy dereference sites involve only local reasoning. This observation further contributes empirical

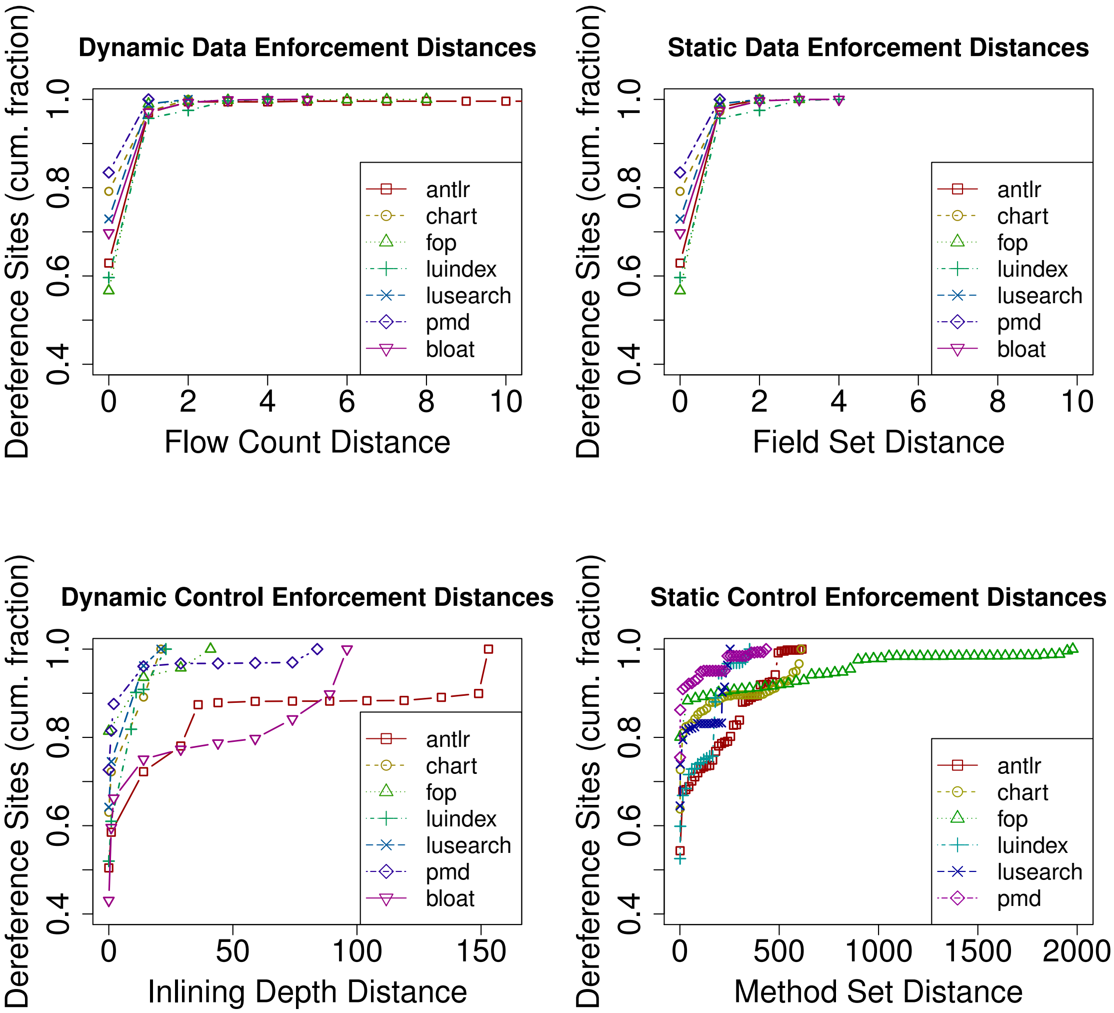


Figure 6.10: Distribution of dereference site measurements for DaCapo. Data enforcement distances are overwhelming short. Control distances can get very long for heap locations, suggesting non-operational heap reasoning (e.g., by encapsulation or invariants).

evidence that programmers are more comfortable reasoning locally than non-locally. The situation is not as clear-cut for data reasoning—half of the buggy sites for PMD involve a flow distance of 0 (that is, they do not involve the heap at all). These buggy sites have non-minimum measurements for both control-based metrics, possibly suggesting that these particular bugs resulted from faulty control reasoning alone. The key message from this study is that while a significant fraction of all dereference sites require only local control reasoning, buggy sites appear to be drawn from a different distribution tending towards non-local control reasoning.

How Do Enforcement Distances Change Over Time? To explore how enforcement distances change over time, we compare the distribution of distances for the first revision we analyzed to a more recent one, spanning five years for PMD and 18 months for Lucene. Figure 6.7b shows the fraction of dereference sites with a given maximum distance for PMD under the Field Set metric at the beginning and end of the span. The distribution barely shifts to slightly longer distances. This finding is quite surprising, as the sheer increase in code size (going from 1882 to 3205 executed dereference sites and from 361 to 693 executed methods) should bias towards longer distances. The results for Lucene, and for our other metrics, are exceedingly similar. For reference, they are available supplementally [30]. For Lucene, the growth in number of measured dereferences is even larger (going from 1389 to 4176). Perhaps our metrics are capturing properties not of programs but of programmers and we should expect to see similar results for more benchmarks.

6.4.2 Distribution of Enforcement Distances

To understand the distribution of enforcement distances across a set of real programs, we interpreted traces over the DaCapo benchmark suite's small inputs. We omit jython, hysqldb, and xalan because of limitations in how our instrumentation handles exceptions caught in uninstrumented code and eclipse because our instrumentation causes it to deadlock. Figure 6.10 shows the cumulative distribution of maximum dereference distances for the Flow Count, Field Set, and Inlining Depth metrics. The y-axis shows the fraction of dereference sites with a distance less than or equal to the value on the x-axis, so for example, for luindex under the Field Set metric, 60% of sites have a maximum dereference distance of 0, while around 97% have maximum distances of 2 or less. We omit the sites with unknown distances (i.e., from library code). Interestingly for antlr on the Flow Count metric, the cumulative fraction for antlr does not reach 1.0 until a distance of 97 (so the x-axis has been cut off prematurely to elide this outlier and expose the behavior at small distances). The graphs for Flow Count (i.e., dynamic data distance) and Field Set (i.e., static data distance) are identical for distance 0, reflecting the fact that that between 55% and 85% of sites do not require reasoning about the heap. Both go to nearly 100% by a distance of 4, although the dynamic metric, as we have seen, has a very long tail. This indicates that the number of data locations about which a programmer needs to reason to ensure that a dereference will succeed is generally small but may in rare cases be large.

```
saveField = o.field;
...// complicated recursive code
...// that may modify o.field
o.field = saveField;
```

Figure 6.11: Trading off data distance to reduce control distance.

The small (5) number of sites in antlr that have extremely long Flow Count (97) are very interesting. The values at these sites arise from repeated execution of the pattern shown in Figure 6.11. These sites also exhibit the highest Inlining Depth distance observed (153) over all of our benchmarks. It appears that the developer has made a deliberate decision to trade off extra data distance in order to avoid having to consider a large amount of control distance.

The situation for dynamic control (Inlining Depth metric) distances is markedly different than that for dynamic data. Although a large fraction of sites (70% to 95%) involve distances of less than 10, a significant fraction of sites show much higher distances. In the antlr benchmark, for example, around 12% of sites have distances greater than 45 and 8% have distances greater than 150. It is hard to imagine that programmers could reason operationally over such an inline depth. Recall that inlining depth speaks about an observed enforcement and its use along a call path; specifically an enforcement and a use in the same method separated by a long execution tree would still have distance 0. Instead, these large distances perhaps reflect the modeling in this metric that methods can modify **any** heap location. To examine this hypothesis further, we ran this inlining depth experiment except with heap modeling turned off (i.e., all reads from the heap are treated as unknowns and all writes to the heap are ignored), which in essence focuses the measurement to control distances of parameters. The result was that 95% of all sites had a distance of 3 or less, although both antlr and bloat had sites with maximum distances of 24 and 82 respectively. This provides some evidence for the somewhat unrealistic heap modeling hypothesis. For completeness, this plot is given supplementally [30]. In languages such as Java, type safety and encapsulation severely limit the heap locations that a given class or package can modify. An improved metric would perhaps take these features into account.

6.4.3 Threats to Validity

We have identified three principle threats to the validity to our conclusions: (1) *Benchmark selection:* We have chosen benchmarks that are easy to run under instrumentation and that have relatively stable interfaces (so as to allow us to use the same input over different versions of the program). This choice has led to a bias towards text processing tools. (2) *Bug selection:* We examine bugs reported in project databases, biasing our analysis towards bugs that are easier to report, which may have shorter enforcement distances. (3) *Metrics:* We have examined four of many possible different distance metrics. We discuss our reasons for choosing these metrics in Section 6.2, and the insights that we have obtained from the results discussed in this section has perhaps lessened this concern.

6.5 Related Work

The closest related work is perhaps Liang et al. [70], which measures dynamically whether particular heap abstractions would have been sufficient for race and deadlock detection analyses. They focus on evaluating the abstraction function and do not perform symbolic interpretation (i.e., they instead associate facts with concrete object identities). In contrast, our approach is abstraction-agnostic and is instead concerned with creating a framework to (a) rule out static analysis designs and (b) guess a scope (e.g., a code fragment) that may be sufficient to prove a property of interest. Livshits et al. [71] assume that bottlenecks in code enforce taint sanitization—in our work, we look to show that enforcements are bottlenecks. In contrast to work on augmenting symbolic execution with concrete information to perform directed testing or test case generation [18, 49, 94], we perform a symbolic analysis to understand source properties on a given concrete trace with an intertwined concrete-symbolic state. Dynamic invariant inference [40, 53] generalizes over observed dynamic executions to produce invariants and has been enriched with symbolic execution in DySy [35].

D'Ambros et al. [36] provide a comprehensive survey of artifact-based bug prediction metrics. Our work differs from these approaches in that we are not focused on predicting bugs per se but in understanding how enforcements are inserted to guard against faults. A large area of research studies programmers directly to see how they reason about programs (e.g., [67, 68]). With our measurements, we are not studying programmers but rather explaining empirical observations about enforcements with hypotheses about possible programmer behavior. These behaviors may be interesting to validate ethnographically.

Many have worked on null pointer error detection, both statically and dynamically. We are not specifically concerned in null pointer detection but see it as a property that naturally lends itself to the study of enforcement windows. Here, we mention a few pieces of work that make some relevant observations. Hovemeyer et al. [60] report that many null dereference bugs do not rely on heap invariants, but instead can be discovered with straightforward static data-flow analyses. Bond et al. [15] present **origin tracking**, an efficient run-time mechanism for tracing a null dereference back to the place where the null value was created.

6.6 Summary of Symbolic Trace Interpretation

We have identified two related concepts: **validation scopes** that are the code fragments needed to prove the absence of a fault and **enforcement windows** that are observed as **establish-check-use** sequences in non-faulting executions. The focus of this chapter has been on creating a framework and implementation for measuring enforcement windows that enable us to inform static analysis design and to gain insights into how enforcements appear in code. A novel aspect of this framework is the application of **symbolic trace interpretation** to selectively model limitations of static reasoning in a dynamic analysis. We have given an indication that finding enforcement windows can lead to useful validation scopes. Furthermore, we have provided empirical evidence to support some widely-held beliefs about software engineering.

We chose non-null dereference enforcement windows for a case study because (a) null dereferences faults are widely-known with many techniques targeted at eliminating them, and (b) there are clear syntactic constructs that indicate **establish-check-use** sequences for dereferences. Our framework and techniques should be more broadly applicable to other enforcements, for example, downcasts, where allocation is **establish**, `instanceof` is **check** and the downcast itself is **use**. We believe our approach holds promise to help analysis designers chose effective validation scopes for a variety of interesting safety properties.

Chapter 7

Conclusions and Future Work

In this dissertation, I describe type-intertwined separation logic, a static analysis that soundly combines two disparate approaches to reasoning about the mutable heap: types systems and separation logic. Each of these approaches is powerful in isolation. Traditional type systems take an alias-agnostic, global view of the heap that affords both fast verification and easy annotation of invariants holding over the entire program. Separation logic, in contrast, provides an alias-aware, local view of the heap in which invariants can vary at each program point. This work shows that these two approaches can be safely and efficiently combined in a manner that preserves the benefits of global reasoning for types systems and local reasoning for separation logic for invariants that hold almost everywhere.

The key contributions that make type-intertwined separation logic possible involve the **communication** and **preservation** of heap invariants across analysis boundaries. Type-consistent materialization communicates type invariants from the type system to the separation logic, allowing the analysis to both leverage and selectively violate global heap invariants derived by the type system. Type-consistent summarization ensures that these violations are restored before the separation logic switches to the type system, preserving the global type invariant. Similarly, gated separation strengthens separating conjunction to permit sound framing, preserving gate-separated invariants from type-intertwined interference—but there is no corresponding mechanism to communicate separation logic invariants to the type system.

An exciting possibility for future work would be to communicate these invariants by lifting separation logic spatial formulas into the type system—analogous to the lifting of types to the symbolic value typing in type-intertwined separation logic. This approach would allowing flow-insensitive, alias-aware types that could be materialized and summarized like any other. Combining these lifted-separation-logic types with type-intertwined analysis may be particularly fruitful for checking types in dynamic languages, such as JavaScript, in which programmers apply a global, type-like invariant that can only be precisely inferred with with local, alias-aware reasoning.

Bibliography

[1] clang: a C language family frontend for LLVM. `http://clang.llvm.org/` ; accessed 7-February-2014.

[2] The heartbleed bug. `http://http://www.heartbleed.com/` ; accessed 17-December-2014.

[3] Amal Ahmed, Matthew Fluet, and Greg Morrisett. L^3: A linear language with locations. Fundam. Inform., 77(4):397–449, 2007.

[4] Amal J. Ahmed and David Walker. The logical approach to stack typing. In Zhong Shao and Peter Lee, editors, Proceedings of TLDI'03: 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, New Orleans, Louisiana, USA, January 18, 2003, pages 74–85. ACM, 2003.

[5] Alexander Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In Ron Cytron and Rajiv Gupta, editors, PLDI, pages 129–140. ACM, 2003.

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

[7] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, International Conference on Integrated Formal Methods (IFM), volume 2999 of Lecture Notes in Computer Science, pages 1–20. Springer, 2004.

[8] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In Werner Damm and Holger Hermanns, editors, Conference on Computer-Aided Verification (CAV), volume 4590 of Lecture Notes in Computer Science, pages 178–192. Springer, 2007.

[9] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In Kwangkeun Yi, editor, Asian Symposium on Programming Languages and Systems (APLAS), volume 3780 of Lecture Notes in Computer Science, pages 52–68. Springer, 2005.

[10] Pamela Bhattacharya and Iulian Neamtiu. Assessing programming language impact on development and maintenance: a study on C and C++. In Taylor et al. [100], pages 171–180.

[11] Gavin M. Bierman, Andrew D. Gordon, Catalin Hritcu, and David E. Langworthy. Semantic subtyping with an SMT solver. J. Funct. Program., 22(1):31–105, 2012.

[12] Christian Bird, Nachiappan Nagappan, Premkumar T. Devanbu, Harald Gall, and Brendan Murphy. Does distributed development affect software quality? An empirical case study of Windows Vista. In International Conference on Software Engineering (ICSE), pages 518–528. IEEE, 2009.

[13] 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 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), pages 169–190, New York, NY, USA, October 2006. ACM Press.

[14] 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 Taylor et al. [100], pages 241–250.

[15] Michael D. Bond, Nicholas Nethercote, Stephen W. Kent, Samuel Z. Guyer, and Kathryn S. McKinley. Tracking bad apples: Reporting the origin of null and undefined value errors. In Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), pages 405–422, 2007.

[16] Mathias Braux and Jacques Noyé. Towards partially evaluating reflection in Java. In Julia L. Lawall, editor, PEPM, pages 2–11. ACM, 2000.

[17] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, Symposium on Operating Systems Design and Implementation (OSDI), pages 209–224. USENIX Association, 2008.

[18] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, Symposium on Operating Systems Design and Implementation (OSDI), pages 209–224. USENIX Association, 2008.

[19] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In Zhong Shao and Benjamin C. Pierce, editors, Symposium on Principles of Programming Languages (POPL), pages 289–300. ACM, 2009.

[20] Luca Cardelli. Type systems. In Allen B. Tucker, editor, The Computer Science and Engineering Handbook, pages 2208–2236. CRC Press, 1997.

[21] Bor-Yuh Evan Chang, Adam J. Chlipala, George C. Necula, and Robert R. Schneck. Type-based verification of assembly language for compiler debugging. In J. Gregory Morrisett and Manuel Fähndrich, editors, Workshop on Types in Language Design and Implementation (TLDI), pages 91–102. ACM, 2005.

[22] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In George C. Necula and Philip Wadler, editors, Symposium on Principles of Programming Languages (POPL), pages 247–260. ACM, 2008.

[23] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In Radhia Cousot, editor, SAS, volume 2694 of Lecture Notes in Computer Science, pages 1–18. Springer, 2003.

[24] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for JavaScript. In Gary T. Leavens and Matthew B. Dwyer, editors, OOPSLA, pages 587–606. ACM, 2012.

[25] Ravi Chugh, Patrick Maxim Rondon, and Ranjit Jhala. Nested refinements: a logic for duck typing. In John Field and Michael Hicks, editors, Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012, pages 231–244. ACM, 2012.

[26] Computer Emergency Response Team Coordination Center. CERT Advisory CA-2001-19 “Code Red” worm exploiting buffer overflow in IIS indexing service DLL. `http://www.cert.org/advisories/CA-2001-19.html`, July 2001.

[27] Jeremy Condit, Matthew Harren, Zachary R. Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In Rocco De Nicola, editor, ESOP, volume 4421 of Lecture Notes in Computer Science, pages 520–535. Springer, 2007.

[28] Devin Coughlin and Bor-Yuh Evan Chang. Fissile type analysis: modular checking of almost everywhere invariants. In Suresh Jagannathan and Peter Sewell, editors, The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’14, San Diego, CA, USA, January 20-21, 2014, pages 73–86. ACM, 2014.

[29] 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 Mats Per Erik Heimdahl and Zhendong Su, editors, International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012, pages 276–286. ACM, 2012.

[30] Devin Coughlin, Bor-Yuh Evan Chang, Amer Diwan, and Jeremy G. Siek. Measuring enforcement windows with symbolic trace interpretation: What well-behaved programs say (extended version). Technical Report CU-CS-1093-12, CU-Boulder, 2012.

[31] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Symposium on Principles of Programming Languages (POPL), pages 238–252, 1977.

[32] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Symposium on Principles of Programming Languages (POPL), pages 269–282, 1979.

[33] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In Symposium on Principles of Programming Languages (POPL), pages 84–96, 1978.

[34] Arlen Cox, Bor-Yuh Evan Chang, and Sriram Sankaranarayanan. Quic graphs: Relational invariant generation for containers. In European Conference on Object-Oriented Programming (ECOOP), pages 401–425, 2013.

[35] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. DySy: dynamic symbolic execution for invariant inference. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, International Conference on Software Engineering (ICSE), pages 281–290. ACM, 2008.

[36] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In IEEE Working Conference on Mining Software Repositories (MSR), pages 31–41, May 2010.

[37] Manuvir Das. Unification-based pointer analysis with directional assignments. In Conference on Programming Language Design and Implementation (PLDI), pages 35–46, 2000.

[38] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In Holger Hermanns and Jens Palsberg, editors, Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 3920 of Lecture Notes in Computer Science, pages 287–302. Springer, 2006.

[39] Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J. Summers. A unified framework for verification techniques for object invariants. In ECOOP, pages 412–437, 2008.

[40] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. IEEE Trans. Software Eng., 27(2):99–123, 2001.

[41] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In Conference on Programming Language Design and Implementation (PLDI), pages 13–24, 2002.

[42] Cormac Flanagan. Hybrid type checking. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, Symposium on Principles of Programming Languages (POPL), pages 245–256. ACM, 2006.

[43] Cormac Flanagan. Hybrid type checking. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, POPL, pages 245–256. ACM, 2006.

[44] Cormac Flanagan and Stephen N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In Workshop on Program Analysis for Software Tools and Engineering (PASTE), pages 1–8, New York, NY, USA, 2010. ACM.

[45] Jeffrey S. Foster, Tachio Terauchi, and Alexander Aiken. Flow-sensitive type qualifiers. In Jens Knoop and Laurie J. Hendren, editors, Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002, pages 1–12. ACM, 2002.

[46] Tim Freeman and Frank Pfenning. Refinement types for ML. In David S. Wise, editor, Conference on Programming Language Design and Implementation (PLDI), pages 268–277. ACM, 1991.

[47] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In OOPSLA, OOPSLA '09, New York, NY, USA, 2009. ACM.

[48] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, Conference on Programming Language Design and Implementation (PLDI), pages 213–223. ACM, 2005.

[49] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, Conference on Programming Language Design and Implementation (PLDI), pages 213–223. ACM, 2005.

[50] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In NDSS. The Internet Society, 2008.

[51] Bhargav S. Gulavani, Supratik Chakraborty, Ganesan Ramalingam, and Aditya V. Nori. Bottom-up shape analysis. In Static Analysis Symposium (SAS), pages 188–204, 2009.

[52] Bolei Guo, Neil Vachharajani, and David I. August. Shape analysis with inductive recursion synthesis. In Jeanne Ferrante and Kathryn S. McKinley, editors, Conference on Programming Language Design and Implementation (PLDI), pages 256–265. ACM, 2007.

[53] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In International Conference on Software Engineering (ICSE), pages 291–301. ACM, 2002.

[54] William R. Harris, Sriram Sankaranarayanan, Franjo Ivancic, and Aarti Gupta. Program analysis via satisfiability modulo path programs. In Hermenegildo and Palsberg [56], pages 71–82.

[55] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In Symposium on Principles of Programming Languages (POPL), pages 58–70, 2002.

[56] Manuel V. Hermenegildo and Jens Palsberg, editors. Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010. ACM, 2010.

[57] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In Workshop on Program Analysis for Software Tools and Engineering (PASTE), pages 54–61. ACM, 2001.

[58] C. A. R. Hoare. An axiomatic basis for computer programming. Commun. ACM, 12(10):576–580, 1969.

[59] Eric Holk, Ryan Newton, Jeremy G. Siek, and Andrew Lumsdaine. Region-based memory management for GPU programming languages: enabling rich data structures on a spartan host. In Andrew P. Black and Todd D. Millstein, editors, Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014, pages 141–155. ACM, 2014.

[60] David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In Workshop on Program Analysis for Software Tools and Engineering (PASTE), pages 13–19, 2005.

[61] Apple Inc. Xcode Download and Resources - Apple Developer. `https://developer.apple.com/xcode/` ; accessed 7-February-2014.

[62] 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.

[63] Michael Karr. Affine relationships among variables of a program. Acta Inf., 6:133–151, 1976.

[64] Yit Phang Khoo, Bor-Yuh Evan Chang, and Jeffrey S. Foster. Mixing type checking and symbolic execution. In Benjamin G. Zorn and Alexander Aiken, editors, Conference on Programming Language Design and Implementation (PLDI), pages 436–447. ACM, 2010.

[65] Yit Phang Khoo, Jeffrey S. Foster, Michael Hicks, and Vibha Sazawal. Path projection for user-centered static analysis tools. In Workshop on Program Analysis for Software Tools and Engineering (PASTE), pages 57–63, 2008.

[66] James C. King. Symbolic execution and program testing. Commun. ACM, 19(7):385–394, 1976.

[67] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Trans. Softw. Eng., 32:971–987, December 2006.

[68] Thomas D. LaToza, David Garlan, James D. Herbsleb, and Brad A. Myers. Program comprehension as fact finding. In European Software Engineering Conference (ESEC) held jointly with Symposium on Foundations of Software Engineering (FSE), pages 361–370, New York, NY, USA, 2007. ACM.

[69] Vincent Laviron, Bor-Yuh Evan Chang, and Xavier Rival. Separating shape graphs. In Andrew D. Gordon, editor, European Symposium on Programming (ESOP), volume 6012 of Lecture Notes in Computer Science, pages 387–406. Springer, 2010.

[70] Percy Liang, Omer Tripp, Mayur Naik, and Mooly Sagiv. A dynamic evaluation of the precision of static heap abstractions. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), pages 411–427. ACM, 2010.

[71] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: Specification inference for explicit information flow problems. In PLDI, pages 75–86, New York, NY, USA, 2009. ACM.

[72] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for Java. In Kwangkeun Yi, editor, APLAS, volume 3780 of Lecture Notes in Computer Science, pages 139–160. Springer, 2005.

[73] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In Hermenegildo and Palsberg [56], pages 211–222.

[74] Robin Milner. A theory of type polymorphism in programming. J. Comput. Syst. Sci., 17(3):348–375, 1978.

[75] Antoine Miné. The octagon abstract domain. Higher-Order and Symbolic Computation, 19(1):31–100, 2006.

[76] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. A case study of open source software development: the apache server. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf, editors, Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000., pages 263–272. ACM, 2000.

[77] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford-Chen, and Nicholas Weaver. Inside the slammer worm. IEEE Security & Privacy, 1(4):33–39, 2003.

[78] National Institute of Standards and Technology. Software errors cost U.S. economy $59.5 billion annually, NIST News Release 2002-10, June 2002.

[79] Peter W. O'Hearn. On bunched typing. J. Funct. Program., 13(4):747–796, 2003.

[80] Peter W. O'Hearn. Resources, concurrency, and local reasoning. Theor. Comput. Sci., 375(1-3):271–307, 2007.

[81] Matthew J. Parkinson. Local Reasoning for Java. PhD thesis, University of Cambridge, Computer Laboratory, 2005.

[82] Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In Jens Palsberg and Martín Abadi, editors, Symposium on Principles of Programming Languages (POPL), pages 247–258. ACM, 2005.

[83] Leaf Petersen, Robert Harper, Karl Crary, and Frank Pfenning. A type theory for memory allocation and data layout. In Alex Aiken and Greg Morrisett, editors, Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003, pages 172–184. ACM, 2003.

[84] Geoffrey Phipps. Comparing observed bug and productivity rates for java and C++. Softw., Pract. Exper., 29(4):345–358, 1999.

[85] Jeff Polakow. Ordered Linear Logic and Applications. PhD thesis, Carnegie Mellon University, 2001. Technical report CMU-CS-01-152.

[86] Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. Semantics and types for objects with first-class member names. In International Workshop on Foundations of Object-Oriented Languages (FOOL), 2012.

[87] Phillip Porras, Hassen Saidi, and Vinod Yegneswaran. An analysis of Conficker's logic and rendezvous points. Technical report, SRI International, March 2009.

[88] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In Symposium on Logic in Computer Science (LICS), pages 55–74. IEEE Computer Society, 2002.

[89] Patrick Maxim Rondon, Alexander Bakst, Ming Kawaguchi, and Ranjit Jhala. Csolve: Verifying C with liquid types. In P. Madhusudan and Sanjit A. Seshia, editors, Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings, volume 7358 of Lecture Notes in Computer Science, pages 744–750. Springer, 2012.

[90] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In Rajiv Gupta and Saman P. Amarasinghe, editors, PLDI, pages 159–169. ACM, 2008.

[91] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Low-level liquid types. In POPL, pages 131–144, 2010.

[92] 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.

[93] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In Michel Wermelinger and Harald Gall, editors, European Software Engineering Conference (ESEC) held jointly with Symposium on Foundations of Software Engineering (FSE), pages 263–272. ACM, 2005.

[94] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In Michel Wermelinger and Harald Gall, editors, European Software Engineering Conference (ESEC) held jointly with Symposium on Foundations of Software Engineering (FSE), pages 263–272. ACM, 2005.

[95] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, Program Flow Analysis: Theory and Applications, chapter 7, pages 189–233. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[96] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In Scheme and Functional Programming Workshop, pages 81–92, 2006.

[97] Bjarne Steensgaard. Points-to analysis in almost linear time. In Symposium on Principles of Programming Languages (POPL), pages 32–41, 1996.

[98] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Software Eng., 12(1):157–171, 1986.

[99] Ross Tate, Juan Chen, and Chris Hawblitzel. Inferable object-oriented typed assembly language. In PLDI, PLDI '10, New York, NY, USA, 2010. ACM.

[100] Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors. Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011. ACM, 2011.

[101] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In POPL, pages 395–406, 2008.

[102] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin, editors, Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994, pages 188–201. ACM Press, 1994.

[103] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective taint analysis of web applications. In Michael Hind and Amer Diwan, editors, PLDI, pages 87–97. ACM, 2009.

[104] David Walker. Substructural type systems. In Benjamin C. Pierce, editor, Advanced Topics in Types and Programming Languages. MIT Press, Boston, 2002.

[105] Hongwei Xi. Imperative programming with dependent types. In LICS, pages 375–387. IEEE Computer Society, 2000.

[106] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In Andrew W. Appel and Alex Aiken, editors, POPL, pages 214–227. ACM, 1999.

[107] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In David Garlan, editor, SIGSOFT '96, Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering, San Francisco, California, USA, October 16-18, 1996, pages 81–92. ACM, 1996.

[108] Michael Zhivich and Robert K. Cunningham. The real cost of software errors. IEEE Security & Privacy, 7(2):87–90, 2009.

Appendix A

Additional Proofs for Fissile Type Analysis

A.1 Lemmas Concerning Abstract State

While standard subtyping relates the concretizations of types within the same domain, subtyping under substitution relates concretizations in different domains. But before we can formally describe the soundness conditions for subtyping under subsitution, we must give semantic meaning to transformations between the concretizations (and thus describe soundess for types under substitution).

Lemma 11 (Meaning of Types Under Substitution):

(a) $\gamma(T^{\mathrm{L}}\ [\theta]) = \left\{ (E, H, v) \;\middle|\; (E \circ \theta, H, v) \in \gamma(T^{\mathrm{L}}) \right\}$

(b) If $\mathrm{fv}(T^{\mathrm{S}}) \subseteq \mathrm{dom}(\tilde{\theta}^{-1})$ *then*

$$\gamma(T^{\mathrm{S}}\ [\tilde{\theta}^{-1}]) \subseteq \left\{ (E, H, v) \;\middle|\; (E \circ \tilde{\theta}^{-1}, H, \cdot, v) \in \gamma(T^{\mathrm{S}}) \right\}$$

(c) If $\mathrm{fv}(T^{\mathrm{L}}) \subseteq \mathrm{dom}(\tilde{\theta})$ *then*

(i) $\left\{ (V, H^{\mathrm{ok}}, H^{\mathrm{mat}}, v) \in \gamma(T^{\mathrm{L}}\ [\tilde{\theta}]) \;\middle|\; H^{\mathrm{mat}} = \cdot \right\} \subseteq$
$\left\{ (V, H^{\mathrm{ok}}, H^{\mathrm{mat}}, v) \;\middle|\; (V \circ \tilde{\theta}, H^{\mathrm{ok}}, v) \in \gamma(T^{\mathrm{L}}) \right\}$

(ii) $\left\{ (V, H^{\mathrm{ok}}, H^{\mathrm{mat}}, v) \;\middle|\; \begin{array}{l} H^{\mathrm{mat}} = \cdot \textit{ and exists } E \supseteq V \circ \tilde{\theta} \\ \textit{such that } (E, H^{\mathrm{ok}}, v) \in \gamma(T^{\mathrm{L}}) \end{array} \right\} \subseteq \gamma(T^{\mathrm{L}}\ [\tilde{\theta}])$

Proof. By definitions and Lemma 3. □

We rely on this lemma to show the soundness of subtyping under substitution (Lemma 4) in Section 4.4.2.

We also note some structural properties of environments and valuations: (1) the concretization of type environments permits weakening in local environments and (2) the concretization of symbolic states permits weakening in valuations.

Lemma 12 (Weakening):

(1) If $(E, H) \in \gamma(\Gamma^{\mathrm{L}})$ *and* $E' \supseteq E$ *then* $(E', H) \in \gamma(\Gamma^{\mathrm{L}})$.

(2) If $(V, H^{\mathrm{ok}}, H^{\mathrm{mat}}) \in \gamma(\widetilde{E}, \widetilde{\Sigma})$ *and* $V' \supseteq V$ *then* $(V', H^{\mathrm{ok}}, H^{\mathrm{mat}}) \in \gamma(\widetilde{E}, \widetilde{\Sigma})$.

Proof. Straight-forward unrolling and re-rolling of definitions. □

In essence, this property says that refinements must behave in a local manner; that is, they constrain only the parts of the environment or valuation that they refer to. We rely on this property throughout the proof.

A.2 Lemmas Concerning Concrete Execution

Before we can prove full type-intertwined soundness for Fissile, we first need several small lemmas concerning single concrete execution rules. Essentially, these are factored out arguments that we rely on in the body of the main proof (Theorem 1).

Lemma 13 (The Concrete Heap Only Grows): *If* $E \vdash [H]\, e\, [H' \downarrow v]$ *then for all* $a : \langle o, B\rangle$ *in* H *there exists an* o' *such that* $H'(a) = \langle o', B\rangle$.

Proof. By induction on the derivation of $E \vdash [H]\, e\, [H' \downarrow v]$. The only interesting cases are E-Object-Literal and E-Write-Field. □

This lemma is a trivial property of concrete execution; it says that as the program executes, addresses are only added to the concrete heap and never removed. This lemma is required to prove the soundness of type checking of method calls.

Lemma 14 (Evolving Type-Consistent Heaps): *If* $(E, H, v) \in \gamma(T_1^{\mathrm{L}})$ *and*

(i) for all $a : \langle o, B_a \rangle$ *in* H *we have* $(H', a) \in \gamma(B_a)$ *and also there exists an* o' *such that* $H'(a) = \langle o', B_a \rangle$ *and*

(ii) for all $a : \langle o, B_a \rangle$ *in* H' *it is the case that* $(H', a) \in \gamma(B_a)$

then $(E, H', v) \in \gamma(T_1^{\mathrm{L}})$.

Proof. Follows from definition of concretization of type environments. □

This lemma essentially says that if a concrete environment is in the concretization of a type environment and the heap changes in such a way that (i) it does not remove any addresses and (ii) any added or mutated objects are type consistent, then pair of the concrete environment heap and the new heap will still be in the concretization of the type environment. We use this lemma in the **E-Reflective-Call** case of Theorem 1 to show the heap that results from executing the body of a method is still compatible with the type environment of the caller.

We now consider another special-purpose lemma, covering the effect of field writes in the concretization of symbolic types.

Lemma 15 (Mutation of Materialized Heap):
If $o' = o[f : v]$ *and* $(V, H^{\mathrm{ok}}, H^{\mathrm{mat}} * a : \langle o, B_a \rangle, v) \in \gamma(T^{\mathrm{S}})$ *then*
$(V, H^{\mathrm{ok}}, H^{\mathrm{mat}} * a : \langle o', B_a \rangle, v) \in \gamma(T^{\mathrm{S}})$.

Proof. By cases for refinements and by induction on the structure of base types for base types. □

This lemma says that field writes to materialized storage will not cause the type of a value to change. We use this lemma in the **E-Write-Field** case of Theorem 1.

Our final special-purpose lemma is used in the **E-Object-Literal** case of Theorem 1. It says that adding a newly allocated object to the heap will not change the type of any values.

Lemma 16 (Field Type Allocation Lemma):

If $B_o = \{\overline{\text{var } f : T_f},\ \overline{\text{def } m(\overline{p : T_p}) \to B_{\text{ret}}}\}$ *and* $(H, o) \in \gamma(\overline{f : T_f})$ *and* $o(m) \in \gamma(B_o, (\overline{p : T_p}) \to B_{\text{ret}})$ *for all methods* m *and* $a \notin dom(H)$ *and* $H' = H[a : \langle o, B_o \rangle$ *then*

(1) If $(H, o, v) \in \gamma(T^{\text{F}})$ *then* $(H', o, v) \in \gamma(T^{\text{F}})$

(2) If $(E, H, v) \in \gamma(T^{\text{L}})$ *then* $(E, H', v) \in \gamma(T^{\text{L}})$

Proof. By cases for refinements and by induction on the structure of base types for base types. □

A.3 Details of the Proof of Intertwined Soundness Fissile Type Analysis

In this section, I give the details of the main proof of soundness for Fissile type analysis.

Theorem 1 (Soundness of Fissile Type Analysis)**.**
If $E \vdash [H]\, e\, [r]$ *then*

(1) If $\Gamma^{\text{L}} \vdash e : T^{\text{L}}$ *and* $(E, H) \in \gamma(\Gamma^{\text{L}})$ *then* $r = H' \downarrow v'$ *where* $(E, H') \in \gamma(\Gamma^{\text{L}})$ *and* $(E, H', v') \in \gamma(T^{\text{L}})$*; and*

(2) If $\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e\, \{\widetilde{P}\}$ *and* $(E, H) \in \gamma(\widetilde{E}, \widetilde{\Sigma})$ *then* $r = H' \downarrow v'$ *where* $(E, H', v') \in \gamma(\widetilde{E}, \widetilde{P})$*; and*

(3) If $\widetilde{E} \vdash \{\widetilde{P}\}\, e\, \{\widetilde{P}'\}$ *and* $(E, H, v) \in \gamma(\widetilde{E}, \widetilde{P})$ *then* $r = H' \downarrow v'$ *where* $(E, H', v') \in \gamma(\widetilde{E}, \widetilde{P}')$*.*

Proof. By induction on the derivation of $E \vdash [H]\, e\, [r]$.

Here we give details only for concrete execution rules that exercise the core, non-standard, parts of our analysis. We present the cases for branching (to illustrate soundness of disjunctive symbolic execution), sequencing (to demonstrate sound threading of state), field writes (for imperative updates), reflective method call (to illustrate reflection call safety and modular checking), and object literal creation (modular checking).

Case E-Branch-Left By assumption, we have that

$$\frac{\mathcal{E} :: E \vdash [H]\, e_1\, [r]}{E \vdash [H]\, e_1 \,[]\, e_2\, [r]}\ \textsc{E-Branch-Left}$$

For this case, it suffices to show that:

(a) If $\Gamma^{\mathrm{L}} \vdash e_1 \mathbin{[\!]} e_2 : T^{\mathrm{L}}$ and $(E, H) \in \gamma(\Gamma^{\mathrm{L}})$ then $r = H' \downarrow v_1$ where $(E, H') \in \gamma(\Gamma^{\mathrm{L}})$ and $(E, H', v_1) \in \gamma(T^{\mathrm{L}})$; and

(b) If $\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_1 \mathbin{[\!]} e_2 \,\{\widetilde{P}\}$ and $(E, H) \in \gamma(\widetilde{E}, \widetilde{\Sigma})$ then $r = H' \downarrow v_1$ where $(E, H', v_1) \in \gamma(\widetilde{E}, \widetilde{P})$; and

(c) If $\widetilde{E} \vdash \{\widetilde{P}\}\, e_1 \mathbin{[\!]} e_2 \,\{\widetilde{P}'\}$ and exists v where $(E, H, v) \in \gamma(\widetilde{E}, \widetilde{P})$ then $r = H' \downarrow v_1$ where $(E, H', v_1) \in \gamma(\widetilde{E}, \widetilde{P}')$.

We will show these properties by an inner simultaneous induction on the derivation of $\Gamma^{\mathrm{L}} \vdash e_1 \mathbin{[\!]} e_2 : T^{\mathrm{L}}$, the derivation of $\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_1 \mathbin{[\!]} e_2 \,\{\widetilde{P}\}$, and the derivation of $\widetilde{E} \vdash \{\widetilde{P}\}\, e_1 \mathbin{[\!]} e_2 \,\{\widetilde{P}\}$. For this inner induction, there are seven potential cases for the last rule applied to an expression of the form $e_1 \mathbin{[\!]} e_2$: T-Branch, Sym-Branch, Sym-Materialize, Sym-Summarize, T-Sym-Handoff, Sym-Type-Handoff, and Sym-Cases.

Case T-Branch By assumption, we have

$$\textsc{T-Branch}\quad \frac{\Gamma^{\mathrm{L}} \vdash e_1 : T^{\mathrm{L}} \qquad \Gamma^{\mathrm{L}} \vdash e_2 : T^{\mathrm{L}}}{\Gamma^{\mathrm{L}} \vdash e_1 \mathbin{[\!]} e_2 : T^{\mathrm{L}}}$$

Suppose $(E, H) \in \gamma(\Gamma^{\mathrm{L}})$. Since $\Gamma \vdash e_1 : T^{\mathrm{L}}$, we can apply the (outer) induction hypothesis for $\mathcal{E}$, yielding $r = H' \downarrow v_1$ where $(E, H') \in \gamma(\Gamma^{\mathrm{L}})$ and $(E, H', v_1) \in \gamma(T^{\mathrm{L}})$, as desired.

Case Sym-Branch By assumption, we have

$$\textsc{Sym-Branch}\quad \frac{\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_1 \,\{\widetilde{P}_1\} \qquad \widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_2 \,\{\widetilde{P}_2\}}{\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_1 \mathbin{[\!]} e_2 \,\{\widetilde{P}_1 \vee \widetilde{P}_2\}}$$

Suppose $(E, H) \in \gamma(\widetilde{E}, \widetilde{\Sigma})$. Since $\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_1 \,\{\widetilde{P}_1\}$, we can apply the (outer) induction hypothesis for $\mathcal{E}$, yielding $r = H' \downarrow v_1$ where $(E, H', v_1) \in \gamma(\widetilde{E}, \widetilde{P}_1)$ and thus $(E, H', v_1) \in \gamma(\widetilde{E}, \widetilde{P}_1 \vee \widetilde{P}_2)$, as desired.

Case Sym-Materialize By assumption, we have

$$\textsc{Sym-Materialize}\quad \frac{\widetilde{\Sigma} \xrightarrow{\mathsf{materialize}} \bigvee_i \widetilde{\Sigma}'_i \qquad \mathcal{S}_i :: \widetilde{E} \vdash \{\widetilde{\Sigma}'_i\}\, e_1 \,[\!]\, e_2 \,\{\widetilde{P}_i\} \qquad \text{for all } i}{\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_1 \,[\!]\, e_2 \,\{\bigvee_i \widetilde{P}_i\}}$$

Suppose $(E, H) \in \gamma(\widetilde{E}, \widetilde{\Sigma})$. Then by the soundness of materialization (Lemma 7), $(E, H) \in \gamma(\widetilde{E}, \bigvee_i \widetilde{\Sigma}'_i)$ and thus $(E, H) \in \gamma(\widetilde{E}, \widetilde{\Sigma}'_i)$ for some i. Applying the (inner) induction hypothesis for $\mathcal{S}_i$ yields $r = H' \downarrow v_1$ where $(E, H', v_1) \in \gamma(\widetilde{E}, \widetilde{P}_i)$ and thus $(E, H', v_1) \in \gamma(\widetilde{E}, \bigvee_i \widetilde{P}_i)$, as desired.

Case Sym-Summarize By assumption, we have

$$\textsc{Sym-Summarize}\quad \frac{\widetilde{\Sigma} \xrightarrow{\mathsf{summarize}} \widetilde{\Sigma}' \qquad \mathcal{S} :: \widetilde{E} \vdash \{\widetilde{\Sigma}'\}\, e_1 \,[\!]\, e_2 \,\{\widetilde{P}\}}{\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_1 \,[\!]\, e_2 \,\{\widetilde{P}\}}$$

Suppose $(E, H) \in \gamma(\widetilde{E}, \widetilde{\Sigma})$. Then by the soundness of summarization (Lemma 9), $(E, H) \in \gamma(\widetilde{E}, \widetilde{\Sigma}')$ Applying the (inner) induction hypothesis for $\mathcal{S}$ yields $r = H' \downarrow v_1$ where $(E, H', v_1) \in \gamma(\widetilde{E}, \widetilde{P})$, as desired.

Case T-Sym-Handoff By assumption, we have

$$\textsc{T-Symbolic-Handoff}\quad \frac{\begin{array}{c}\Gamma^{\mathrm{L}} \xrightarrow{\mathsf{symbolize}} \widetilde{\Gamma}, \widetilde{E} \qquad \mathcal{S} :: \widetilde{E} \vdash \{\widetilde{\Gamma}, \mathsf{ok}\}\, e_1 \,[\!]\, e_2 \,\{\bigvee_i (\widetilde{\Gamma}_i, \mathsf{ok}) \downarrow \widetilde{x}_i\} \\ \widetilde{\Gamma}_i, \widetilde{E} \xrightarrow{\mathsf{typeify}} \Gamma^{\mathrm{L}} \qquad \widetilde{\Gamma}_i \vdash \widetilde{\Gamma}_i(\widetilde{x}_i) <: T^{\mathrm{L}}[\widetilde{E}] \qquad \text{for all } i\end{array}}{\Gamma^{\mathrm{L}} \vdash e_1 \,[\!]\, e_2 : T^{\mathrm{L}}}$$

Suppose $(E, H) \in \gamma(\Gamma^{\mathrm{L}})$. We have $\Gamma^{\mathrm{L}} \xrightarrow{\mathsf{symbolize}} \widetilde{\Gamma}, \widetilde{E}$ and thus by the soundness of local environment symbolization (Lemma 5) $(E, H) \in \gamma(\widetilde{E}, (\widetilde{\Gamma}, \mathsf{ok}))$. Then, applying the (inner) induction hypothesis for $\mathcal{S}$ yields $r = H' \downarrow v_1$ where $(E, H', v_1) \in \gamma(\widetilde{E}, \bigvee_i (\widetilde{\Gamma}_i, \mathsf{ok}) \downarrow \widetilde{x}_i)$. From this, there exists some i such that $(E, H', v_1) \in \gamma(\widetilde{E}, (\widetilde{\Gamma}_i, \mathsf{ok}) \downarrow \widetilde{x}_i)$.

We wish to show that (i) $(E, H') \in \gamma(\Gamma^{\mathrm{L}})$ and (ii) $(E, H', v) \in \gamma(T^{\mathrm{L}})$.

For (i), since $(E, H', v_1) \in \gamma(\widetilde{E}, (\widetilde{\Gamma}_i, \mathsf{ok}) \downarrow \widetilde{x}_i)$ we have $(E, H') \in \gamma(\widetilde{E}, (\widetilde{\Gamma}_i, \mathsf{ok}))$ by the definition of concretization of symbolic state. But $\widetilde{\Gamma}_i, \widetilde{E} \xrightarrow{\mathsf{typeify}} \Gamma^{\mathrm{L}}$, so by the soundness of local environment typeification (Lemma 5), $(E, H') \in \gamma(\Gamma^{\mathrm{L}})$, as desired.

To show (ii), since (again) we have $(E, H', v_1) \in \gamma(\widetilde{E}, (\widetilde{\Gamma}_i, \mathsf{ok}) \downarrow \widetilde{x}_i)$, there exists valuation V where $V(\widetilde{x}_i) = v_1$ and $(V, E) \in \gamma(\widetilde{E})$ and $(V, H', \cdot) \in \gamma(\widetilde{\Gamma}_i)$. So $(H', \cdot, v_1) \in \gamma(\widetilde{\Gamma}_i(\widetilde{x}_i))$. Further, since $\widetilde{\Gamma}_i \vdash \widetilde{\Gamma}_i(\widetilde{x}_i) <: T^{\mathrm{L}}[\widetilde{E}]$, by the soundness of symbolic subtyping (Lemma 1), we have $(H', \cdot, v_1) \in \gamma(T^{\mathrm{L}}[\widetilde{E}])$. Then, by Lemma 11 $(V \circ \widetilde{E}, H', v_1) \in \gamma(T^{\mathrm{L}})$. But $E \supseteq V \circ \widetilde{E}$, so by weakening of local environments (Lemma 12), $(E, H', v_1) \in \gamma(T^{\mathrm{L}})$, as desired.

Case Sym-Type-Handoff By assumption, we have

$$\textsc{Sym-Type-Handoff}\quad \frac{\widetilde{\Gamma}, \widetilde{E} \ \overrightarrow{\mathsf{typeify}}\ \Gamma^{\mathrm{L}} \qquad \mathcal{T} :: \Gamma^{\mathrm{L}} \vdash e_1 \mathbin{[\!]} e_2 : T^{\mathrm{L}} \qquad \Gamma^{\mathrm{L}} \ \overrightarrow{\mathsf{symbolize}}\ \widetilde{\Gamma}', \widetilde{E} \qquad \widetilde{z} \notin \mathrm{dom}(\widetilde{\Gamma}')}{\widetilde{E} \vdash \{\widetilde{\Gamma}, \mathsf{ok}\}\, e_1 \mathbin{[\!]} e_2\, \{\widetilde{\Gamma}'[\widetilde{z} : T^{\mathrm{L}}[\widetilde{E}]], \mathsf{ok} \downarrow \widetilde{z}\}}$$

Suppose $(E, H) \in \gamma(\widetilde{E}, (\widetilde{\Gamma}, \mathsf{ok})$. Then, since $\widetilde{\Gamma}, \widetilde{E} \ \overrightarrow{\mathsf{typeify}}\ \Gamma^{\mathrm{L}}$, we have $(E, H) \in \gamma(\Gamma^{\mathrm{L}})$ by the soundness of local environment typeification (Lemma 5). Applying the (inner) inductive hypothesis for $\mathcal{T}$ yields $r = H' \downarrow v_1$ where $(E, H', v_1) \in \gamma(T^{\mathrm{L}})$ and $(E, H') \in \gamma(\Gamma^{\mathrm{L}})$. Then, by the soundness of local environment symbolization (Lemma 5), we have $(E, H') \in \gamma(\widetilde{E}, (\widetilde{\Gamma}', \mathsf{ok}))$. Therefore, there exists a valuation V where $(V, E) \in \gamma(\widetilde{E})$ and $(V, H', \cdot) \in \gamma(\widetilde{\Gamma}')$ and $(V, H', \cdot) \in \gamma(\mathsf{ok})$.

Now, since $(E, H', v_1) \in \gamma(T^{\mathrm{L}})$ and $(E \supseteq V \circ \widetilde{E})$, we can apply Lemma 4, yielding $(V, H', \cdot, v_1) \in \gamma(T^{\mathrm{L}}[\widetilde{E}])$.

Let $V'(\widetilde{x}) = \begin{cases} v_1 & \text{if } \widetilde{x} = \widetilde{z} \\ V(\widetilde{x}) & \text{otherwise} \end{cases}$

By weakening of valuations (Lemma 12), we have $(V', E) \in \gamma(\widetilde{E})$ and $(V', H', \cdot) \in \gamma(\widetilde{\Gamma}')$ and $(V', H', \cdot) \in \gamma(\mathsf{ok})$. It remains to show that $(V', H', \cdot) \in \gamma(\widetilde{\Gamma}'[\widetilde{z} : T^{\mathrm{L}}[\widetilde{E}]])$. Fix $\widetilde{x} : T^{\mathrm{S}}$ in $\widetilde{\Gamma}'[\widetilde{z} : T^{\mathrm{L}}[\widetilde{E}]]$. There are two cases: if $\widetilde{x} = \widetilde{z}$ then $(V', H', \cdot, V(\widetilde{x})) \in \gamma(\widetilde{\Gamma}'[\widetilde{z} : T^{\mathrm{L}}[\widetilde{E}]](\widetilde{x}))$ since $V(\widetilde{x}) = V(\widetilde{z}) = v_1$ and $\widetilde{\Gamma}'[\widetilde{z} : T^{\mathrm{L}}[\widetilde{E}]](\widetilde{x}) = T^{\mathrm{L}}[\widetilde{E}]$. In the second case, $\widetilde{x} \neq \widetilde{z}$ and thus $\widetilde{x} \in \mathrm{dom}(\widetilde{\Gamma}')$. But $(V', H', \cdot) \in \gamma(\widetilde{\Gamma}')$, so $(V', H', \cdot, V(\widetilde{x}) \in \gamma(\widetilde{\Gamma}'[\widetilde{z} : T^{\mathrm{L}}[\widetilde{E}]](\widetilde{x}))$. Since $\widetilde{x} : T^{\mathrm{S}}$ was chosen arbitrarily, $(V', H', \cdot) \in \gamma(\widetilde{\Gamma}'[\widetilde{z} : T^{\mathrm{L}}[\widetilde{E}]])$ and thus $(E, H', v_1) \in \gamma(\widetilde{E}, \widetilde{\Gamma}'[\widetilde{z} : T^{\mathrm{L}}[\widetilde{E}]], \mathsf{ok} \downarrow \widetilde{z})$, as desired.

Case Sym-Cases By assumption, we have

$$\dfrac{\mathcal{S}_i :: \widetilde{E} \vdash \{\widetilde{\Sigma}_i\}\, e_1 \mathbin{[\!]} e_2\, \{\widetilde{P}_i\} \qquad \text{for all } i}{\widetilde{E} \vdash \{\bigvee_i \widetilde{\Sigma}_i \downarrow \widetilde{x}_i\}\, e_1 \mathbin{[\!]} e_2\, \{\bigvee_i \widetilde{P}_i\}}\ \textsc{Sym-Cases}$$

Suppose exists v where $(E, H, v) \in \gamma(\bigvee_i \widetilde{\Sigma}_i \downarrow \widetilde{x}_i)$. Then for some i, $(E, H, v) \in \gamma(\widetilde{\Sigma}_i \downarrow \widetilde{x}_i)$ and thus $(E, H) \in \gamma(\widetilde{\Sigma}_i)$. We can then apply the (inner) inductive hypothesis for $\mathcal{S}_i$, yielding $r = H' \downarrow v_1$ where $(E, H', v_1) \in \gamma(\widetilde{P}_i)$ and thus $(E, H', v_1) \in \gamma(\bigvee_i \widetilde{P}_i)$, as desired.

Case E-Branch-Right Resuming the outer induction, this is nearly identical to Case E-Branch-Right.

Case E-Seq

By assumption, we have

$$\dfrac{\mathcal{E}_1 :: E \vdash [H]\, e_1\, [H' \downarrow v_1] \qquad \mathcal{E}_2 :: E \vdash [H']\, e_2\, [r]}{E \vdash [H]\, e_1 \,;\, e_2\, [r]}\ \textsc{E-Seq}$$

For this case, it suffices to show that:

(a) If $\Gamma^{\mathrm{L}} \vdash e_1 \,;\, e_2 : T^{\mathrm{L}}$ and $(E, H) \in \gamma(\Gamma^{\mathrm{L}})$ then $r = H'' \downarrow v_2$ where $(E, H'') \in \gamma(\Gamma^{\mathrm{L}})$ and $(E, H'', v_2) \in \gamma(T^{\mathrm{L}})$; and

(b) If $\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_1 \,;\, e_2\, \{\widetilde{P}\}$ and $(E, H) \in \gamma(\widetilde{E}, \widetilde{\Sigma})$ then $r = H'' \downarrow v_2$ where $(E, H'', v_2) \in \gamma(\widetilde{E}, \widetilde{P})$; and

(c) If $\widetilde{E} \vdash \{\widetilde{P}\}\, e_1 \,;\, e_2\, \{\widetilde{P}'\}$ and exists v where $(E, H', v) \in \gamma(\widetilde{E}, \widetilde{P})$ then $r = H'' \downarrow v_2$ where $(E, H'', v_2) \in \gamma(\widetilde{E}, \widetilde{P}')$.

Similar to **Case E-Branch-Left**, we will show this by an inner simultaneous induction on the derivation of $\Gamma^{\mathrm{L}} \vdash e_1 \,;\, e_2 : T^{\mathrm{L}}$, the derivation of $\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_1 \,;\, e_2\, \{\widetilde{P}\}$, and the derivation of $\widetilde{E} \vdash \{\widetilde{P}\}\, e_1 \,;\, e_2\, \{\widetilde{P}\}$.

For this inner induction, there are seven potential cases for the last rule applied to an expression of the form $e_1 \mathbin{[\!]} e_2$: T-Seq, Sym-Seq, Sym-Materialize, Sym-Summarize, T-Sym-Handoff, Sym-Type-Handoff, and Sym-Cases.

The arguments for Sym-Materialize, Sym-Summarize, T-Sym-Handoff, Sym-Type-Handoff, and Sym-Cases are identical to those given in **Case E-Branch-Left** (note that those arguments do not refer to the outer induction hypothesis and also do not refer to the rule-specific sub-structure of the expression or the state components). Here, we give only the two syntax-directed cases (T-Seq, Sym-Seq).

Case T-Seq By assumption, we have

$$\textsc{T-Seq}\quad \frac{\Gamma^{\mathrm{L}} \vdash e_1 : T_1^{\mathrm{L}} \qquad \Gamma^{\mathrm{L}} \vdash e_2 : T_2^{\mathrm{L}}}{\Gamma^{\mathrm{L}} \vdash e_1 \,;\, e_2 : T_2^{\mathrm{L}}}$$

Suppose $(E, H) \in \gamma(\Gamma^{\mathrm{L}})$. Since $\Gamma \vdash e_1 : T^{\mathrm{L}}$, we can apply the outer induction hypothesis for $\mathcal{E}_1$, yielding $(E, H') \in \gamma(\Gamma^{\mathrm{L}})$. With this, the outer induction hypothesis for $\mathcal{E}_2$ yields $r = H'' \downarrow v_2$ with $(E, H'') \in \gamma(\Gamma^{\mathrm{L}})$ and $(E, H'', v_2) \in \gamma(T_1^{\mathrm{L}})$, as desired.

Case Sym-Seq By assumption, we have

$$\textsc{Sym-Seq}\quad \frac{\mathcal{S} :: \widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_1 \,\{\widetilde{P}'\} \qquad \mathcal{P} :: \widetilde{E} \vdash \{\widetilde{P}'\}\, e_2 \,\{\widetilde{P}''\}}{\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_1 \,;\, e_2 \,\{\widetilde{P}''\}}$$

Suppose $(E, H) \in \gamma(\widetilde{E}, \widetilde{\Sigma})$. Then by the outer inductive hypothesis for $\mathcal{S}$, we have $(E, H', v_1) \in \gamma(\widetilde{P}')$. Then by the inner inductive hypothesis for $\mathcal{P}$, we have $r = H'' \downarrow v_2$ with $(E, H'', v_2) \in \gamma(\widetilde{E}, \widetilde{P}')$, as desired.

Case E-Write-Field

By assumption, we have

$$\textsc{E-Write-Field}\quad \frac{a = E(x) \qquad v = E(y) \qquad \langle o, B\rangle = H(a) \qquad o' = o[f : v] \qquad H' = H[a : \langle o', B\rangle]}{E \vdash [H]\, x.f = y\, [r]}$$

where $r = H' \downarrow v$.

For this case, it suffices to show that:

(a) If $\Gamma^{\mathrm{L}} \vdash x.f = y : T^{\mathrm{L}}$ and $(E, H) \in \gamma(\Gamma^{\mathrm{L}})$ then $r = H' \downarrow v$ where $(E, H') \in \gamma(\Gamma^{\mathrm{L}})$ and $(E, H', v) \in \gamma(T^{\mathrm{L}})$; and

(b) If $\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, x.f = y\, \{\widetilde{P}\}$ and $(E, H) \in \gamma(\widetilde{E}, \widetilde{\Sigma})$ then $r = H' \downarrow v$ where $(E, H', v) \in \gamma(\widetilde{E}, \widetilde{P})$; and

(c) If $\widetilde{E} \vdash \{\widetilde{P}\}\, x.f = y\, \{\widetilde{P}'\}$ and exists v where $(E, H, v) \in \gamma(\widetilde{E}, \widetilde{P})$ then $r = H' \downarrow v$ where $(E, H', v) \in \gamma(\widetilde{E}, \widetilde{P}')$.

Again, we will show this by a simultaneous inner induction. For this inner induction, the only interesting new case is Sym-Field-Write. We skip T-Field-Write since this is essentially the same as that from the Deputy type system [27].

Case Sym-Write-Field By assumption, we have

$$\frac{\textsc{Sym-Write-Field}}{\widetilde{E} \vdash \{\widetilde{\Gamma}, \widetilde{H}\}\, x.f := y\, \{\widetilde{\Gamma}, \widetilde{H}\big[\widetilde{E}(x) : \big(\widetilde{H}(\widetilde{E}(x))[f : \widetilde{E}(y)]\big)\big] \downarrow \widetilde{E}(y)\}}$$

Suppose $(E, H) \in \gamma(\widetilde{E}, (\widetilde{\Gamma}, \widetilde{H}))$. Without loss of generality, let $\widetilde{a} = \widetilde{E}(x)$, $\widetilde{y} = \widetilde{E}(y)$, $\tilde{o} = \widetilde{H}(\widetilde{a})$, $\widetilde{H}^{\text{rest}} * \widetilde{a} : \tilde{o}$. Let $\tilde{o}' = \tilde{o}[f : \widetilde{y}]$ and $\widetilde{H}' = \widetilde{H}^{\text{rest}} * \widetilde{a} : \tilde{o}'$.

Then, there exists a valuation V, and heaps H^{ok} and H^{mat} such that:

(A1) $(V, E) \in \gamma(\widetilde{E})$

(A2) $(V, H^{\text{ok}}, H^{\text{mat}}) \in \gamma(\widetilde{H}^{\text{rest}} * \widetilde{a} : \tilde{o})$,

(A3) $(V, H^{\text{ok}}, H^{\text{mat}}) \in \gamma(\widetilde{\Gamma})$, and

(A4) $H = H^{\text{ok}} * H^{\text{mat}}$.

Because $a = E(x)$ and $v = E(y)$, by (A1) we have $V(\widetilde{a}) = a$ and $V(\widetilde{y}) = v$. By (A2) we have $H^{\text{mat}} = {H^{\text{mat}}}_{\text{rest}} * a : \langle o, B\rangle$ where $o \in \gamma(\tilde{o})$. But $H = H^{\text{ok}} * {H^{\text{mat}}}_{\text{rest}} * a : \langle o, B\rangle$ (A4), so $H' = H^{\text{ok}} * {H^{\text{mat}}}_{\text{rest}} * a : \langle o', B\rangle$.

We wish to show that $(E, H', v) \in \gamma(\widetilde{E}, (\widetilde{\Gamma}, \widetilde{H}') \downarrow \widetilde{y})$, so we must show:

(W1) $(V, E) \in \gamma(\widetilde{E})$

(W2) $(V, H^{\text{ok}}, {H^{\text{mat}}}_{\text{rest}} * a : \langle o', B\rangle) \in \gamma(\widetilde{H}^{\text{rest}} * \widetilde{a} : \tilde{o}')$, (W3) $(V, H^{\mathrm{ok}}, {H^{\mathrm{mat}}}_{\mathrm{rest}} * a : \langle o', B\rangle) \in \gamma(\widetilde{\Gamma})$, and

(W4) $H' = H^{\mathrm{ok}} * {H^{\mathrm{mat}}}_{\mathrm{rest}} * a : \langle o', B\rangle$.

We have (W1) by (A1) and have already shown (W4). For (W2), since $o' = o[f : v]$, $V(\widetilde{y}) = v$, and $\tilde{o}' = \tilde{o}[f : \widetilde{y}]$, we have $o' \in \gamma(\tilde{o}')$ and thus $(V, H^{\mathrm{ok}}, {H^{\mathrm{mat}}}_{\mathrm{rest}} * a : \langle o', B\rangle) \in \gamma(\widetilde{H}^{\mathrm{rest}} * \widetilde{a} : \tilde{o}')$

For (W3), choose $\widetilde{x} : T^{\mathrm{S}}$ in $\widetilde{\Gamma}$. By (A3) we have $(V, H^{\mathrm{ok}}, {H^{\mathrm{mat}}}_{\mathrm{rest}} * a : \langle o, B\rangle, V(\widetilde{x}) \in \gamma(\widetilde{\Gamma})$. Then by Lemma 15 and the definition of concretization of symbolic types we have $(V, H^{\mathrm{ok}}, {H^{\mathrm{mat}}}_{\mathrm{rest}} * a : \langle o', B\rangle, V(\widetilde{x}) \in \gamma(\widetilde{\Gamma})$ So we have (W4): $(V, H^{\mathrm{ok}}, {H^{\mathrm{mat}}}_{\mathrm{rest}} * a : \langle o', B\rangle) \in \gamma(\widetilde{\Gamma})$ and thus $(E, H', v) \in \gamma(\widetilde{E}, (\widetilde{\Gamma}, \widetilde{H}') \downarrow \widetilde{y})$, as desired.

Case E-Call-Refl

By assumption, we have

E-Call-Refl

$$\frac{\begin{array}{c} m = E(y) \\ \langle o, B\rangle = H(E(z)) \qquad e = o(m) \qquad E' = \overline{p : E(x)}, \mathrm{self} : E(z) \qquad \mathcal{E} :: E' \vdash [H]\, e\, [H' \downarrow v] \end{array}}{E \vdash [H]\, z.[y](\overline{x})\, [r]}$$

For this case, it suffices to show that:

(a) If $\Gamma^{\mathrm{L}} \vdash z.[y](\overline{x}) : T^{\mathrm{L}}$ and $(E, H) \in \gamma(\Gamma^{\mathrm{L}})$ then $r = H' \downarrow v$ where $(E, H') \in \gamma(\Gamma^{\mathrm{L}})$ and $(E, H', v) \in \gamma(T^{\mathrm{L}})$; and

(b) If $\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, z.[y](\overline{x})\, \{\widetilde{P}\}$ and $(E, H) \in \gamma(\widetilde{E}, \widetilde{\Sigma})$ then $r = H' \downarrow v$ where $(E, H', v) \in \gamma(\widetilde{E}, \widetilde{P})$; and

(c) If $\widetilde{E} \vdash \{\widetilde{P}\}\, z.[y](\overline{x})\, \{\widetilde{P}'\}$ and exists v where $(E, H, v) \in \gamma(\widetilde{E}, \widetilde{P})$ then $r = H' \downarrow v$ where $(E, H', v) \in \gamma(\widetilde{E}, \widetilde{P}')$.

Again, we will show this by a simultaneous inner induction. For this inner induction, the only new case is T-Reflective-Method-Call.

Case T-Reflective-Method-Call By assumption, we have

$$\frac{\Gamma \;<:_{[\overline{p \mapsto x}, \text{self}:z]}\; \overline{p : T_p}, \text{self} : \Gamma(z) \qquad \Gamma(z) = \{\cdots\} \upharpoonright \textsf{respondsTo}\; y(\overline{p : T_p}) {\rightarrow} B_{\text{ret}}}{\Gamma \vdash z.[y](\overline{x}) : B_{\text{ret}} \upharpoonright}\;\textsc{T-Reflective-Method-Call}$$

Assume $(E, H) \in \gamma(\Gamma)$. Since $\Gamma \;<:_{[\overline{p \mapsto x}, \text{self}:z]}\; \overline{p : T_p}, \text{self} : \Gamma(z)$, by Lemma 4 we have $(E \circ (\overline{p \mapsto x}, \text{self} : z), H) \in \gamma(\overline{p : T_p}, \text{self} : \Gamma(z))$. But $E \circ (\overline{p \mapsto x}, \text{self} : z) = \overline{p : E(x), \text{self} : E(z)} = E'$, so $(E', H) \in \gamma(\overline{p : T_p}, \text{self} : \Gamma(z))$.

But since $(E, H) \in \gamma(\Gamma)$, $(E, H, E(z)) \in \gamma(\Gamma(z))$ and so

(a) the base type of $\Gamma(z)$ is B; and

(b) $(E, H, E(z)) \in \gamma(\textsf{respondsTo}\; y(\overline{p : T_p}) {\rightarrow} B_{\text{ret}})$.

From (a) we have $(E', H) \in \gamma(\overline{p : T_p}, \text{self} : B \upharpoonright)$.

From (b), by the definition of concretization for respondsTo refinements, $o(E(y)) \in \gamma(B, (\overline{p : T_p}) {\rightarrow} B_{\text{ret}})$. Since $E(y) = m$, and $e = o(m)$, $e \in \gamma(B, (\overline{p : T_p}) {\rightarrow} B_{\text{ret}})$. By the definition of concretization for method type signatures, $\overline{p : T_p}, \text{self} : B \upharpoonright \vdash e : B_{\text{ret}} \upharpoonright$.

We can thus apply the outer inductive hypothesis for $\mathcal{E}$, yielding $r = H' \downarrow v$ where $(E', H') \in \gamma(\overline{p : T_p}, \text{self} : B \upharpoonright)$ and $(E', H', v) \in \gamma(B_{\text{ret}} \upharpoonright)$

We want to show that (1) $(E, H') \in \gamma(\Gamma)$ and (2) $(E, H', v) \in \gamma(B_{\text{ret}} \upharpoonright)$

For (1), choose $u : T$ in Γ. Since $(E, H) \in \gamma(\Gamma)$, we have $(E, H, E(v)) \in \gamma(T)$. We want $(E, H', E(v)) \in \gamma(T)$. Since $(E', H') \in \gamma(\overline{p : T_p}, \text{self} : B \upharpoonright)$, by the definition of concretization of type environments, we have $(H', a) \in \gamma(B_a)$ for all $a : \langle o_a, B_a \rangle$ in H'. And since $E' \vdash [H]\, e\, [H' \downarrow v]$, by Lemma 13, we have for all $a : \langle o_a, B_a \rangle$ in H there exists an object o'_a such that $H'(a) = \langle o'_a, B_a \rangle$. With these two conditions, we can apply Lemma 14, yielding $(E, H', E(v)) \in \gamma(T)$. Since $u : T$ was chosen arbitrarily, we have $(E, H') \in \gamma(\Gamma)$

For (2), we note that $(E', H', v) \in \gamma(B_{\text{ret}} \upharpoonright)$. But $B_{\text{ret}} \upharpoonright$ has no refinements, so it cannot constrain a concrete environment and thus $(E, H', v) \in \gamma(B_{\text{ret}} \upharpoonright)$, as desired.

Case E-Object-Literal

By assumption, we have

$$\textsc{E-Object-Literal}$$

$$\dfrac{\begin{array}{c} o = \overline{[f : E(x_f)]}\,\overline{[m : e]} \\ B = \{\overline{\text{var } f : T_f},\ \ \overline{\text{def } m(\overline{p : T_p}) \to B_{\text{ret}}}\} \qquad a \notin \text{dom}(H) \qquad H' = H[a : \langle o, B\rangle] \end{array}}{E \vdash [H]\,\{\overline{\text{var } f : T_{\text{f}} \ =\ x_f},\ \ \overline{\text{def } m(\overline{p : T_p}) : B_{\text{ret}} \ =\ e}\}\,[r]}$$

where $r = H' \downarrow a$.

For this case, it suffices to show that:

(a) If $\Gamma^{\mathrm{L}} \vdash \{\overline{\text{var } f : T_{\text{f}} \ =\ x_f},\ \ \overline{\text{def } m(\overline{p : T_p}) : B_{\text{ret}} \ =\ e}\} : T^{\mathrm{L}}$ and $(E, H) \in \gamma(\Gamma^{\mathrm{L}})$ then $r = H' \downarrow a$ where $(E, H') \in \gamma(\Gamma^{\mathrm{L}})$ and $(E, H', a) \in \gamma(T^{\mathrm{L}})$; and

(b) If $\widetilde{E} \vdash \{\widetilde{\Sigma}\}\,\{\overline{\text{var } f : T_{\text{f}} \ =\ x_f},\ \ \overline{\text{def } m(\overline{p : T_p}) : B_{\text{ret}} \ =\ e}\}\,\{\widetilde{P}\}$ and $(E, H) \in \gamma(\widetilde{E}, \widetilde{\Sigma})$ then $r = H' \downarrow a$ where $(E, H', a) \in \gamma(\widetilde{E}, \widetilde{P})$; and

(c) If $\widetilde{E} \vdash \{\widetilde{P}\}\,\{\overline{\text{var } f : T_{\text{f}} \ =\ x_f},\ \ \overline{\text{def } m(\overline{p : T_p}) : B_{\text{ret}} \ =\ e}\}\,\{\widetilde{P}'\}$ and exists v where $(E, H, v) \in \gamma(\widetilde{E}, \widetilde{P})$ then $r = H' \downarrow a$ where $(E, H', a) \in \gamma(\widetilde{E}, \widetilde{P}')$.

Again, we will show this by a simultaneous inner induction. For this inner induction, the only new case is T-Object-Literal.

Case T-Object-Literal By assumption, we have

$$\textsc{T-Object-Literal}$$

$$\dfrac{\begin{array}{c} \Gamma \ <:_{\overline{[f : x_f]}}\ \overline{f : T_f} \\ \overline{p : T_p}, \text{self} : B \upharpoonright\ \vdash\ e\ :\ B_{\text{ret}} \upharpoonright \qquad \text{for all methods} \qquad B = \{\overline{\text{var } f : T_f},\ \ \overline{\text{def } m(\overline{p : T_p}) \to B_{\text{ret}}}\} \end{array}}{\Gamma \vdash \{\overline{\text{var } f : T_{\text{f}} \ =\ x_f},\ \ \overline{\text{def } m(\overline{p : T_p}) : B_{\text{ret}} \ =\ e}\}\ :\ B \upharpoonright}$$

Assume $(E, H) \in \gamma(\Gamma)$. We wish to show that (1) $(E, H') \in \gamma(\Gamma)$ and (2) $(E, H', a) \in \gamma(B \upharpoonright)$.

The key property to show is that $(H', a) \in \gamma(B)$. There requires showing that there exists o' where

(W1) $H'(a) = \langle o', B\rangle$ and

(W2) $o'(m) \in \gamma(B, (\overline{p : T_p}) \to B_{\text{ret}})$ for all methods m

(W3) $(H', o') \in \gamma(\overline{f : T_f})$.

Let $o' = o$. We have $H' = H * H[a : \langle o, B\rangle]$, so (W1) holds. Since $\overline{p : T_p}, \text{self} : B \upharpoonright \vdash e : B_{\text{ret}} \upharpoonright$ for all method m in B, (W2) holds. For (W3), since (E, H) and $\Gamma <:_{[\overline{f{:}x_f}]} \overline{f : T_f}$, by Lemma 4 we have $(H, E \circ \overline{f : x_f}) \in \widetilde{\gamma}(\overline{f : T_f})$. But $E \circ \overline{f : x_f} = \overline{[f : E(x_f)]}$, which are exactly the fields of o, so $(H, o) \in \gamma(\overline{f : T_f})$.

We want to show $(H', o) \in \gamma(\overline{f : T_f})$. So choose $f : \gamma(T_f^{\text{F}})$ in $\overline{f : T_f}$. Since $(H, o) \in \gamma(\overline{f : T_f})$, we have $(H, o, o(f) \in \gamma(T_f)$. Then by Lemma 16, $(H', o, o(f) \in \gamma(T_f)$. Since $f : \gamma(T_f^{\text{F}})$ was chosen arbitrarily, we have $(H', o) \in \gamma(\overline{f : T_f})$. and so (W3) holds and thus $(H', a) \in \gamma(B)$.

To show (1) we must must show that $(H', a) \in \gamma(B)$ (which we just showed) and $(E, H') \in \gamma(\Gamma)$. Fix $x : \gamma(T_x^{\text{L}})$ in Γ. Then, since $(E, H) \in \gamma(\Gamma)$ we have $(E, H, E(x)) \in \gamma(T_x^{\text{L}})$. Then by Lemma 16 we have $(E, H', E(x)) \in \gamma(T_x^{\text{L}})$. Since $x : \gamma(T_x^{\text{L}})$ was chosen arbitrarily, $(E, H') \in \gamma(\Gamma)$, as desired.

For (2), we have $(H', a) \in \gamma(B)$. Since Since $B \upharpoonright$ has no refinements, $(E, H', a) \in \gamma(B \upharpoonright)$, as desired.

Case E-Call-Refl-Lookup-Err

By assumption, we have

$$\begin{array}{l}\textsc{E-Call-Refl-Lookup-Err}\\ \dfrac{m = E(y) \qquad \langle o, B\rangle = H(E(z)) \qquad m \notin \text{dom}(o)}{E \vdash [H]\, z.[y](\overline{x})\, [r]}\end{array}$$

where $r = \mathsf{err}$.

For this case, it suffices to show that:

(a) If $\Gamma^{\text{L}} \vdash z.[y](\overline{x}) : T^{\text{L}}$ and $(E, H) \in \gamma(\Gamma^{\text{L}})$ then $r = H' \downarrow a$ where $(E, H') \in \gamma(\Gamma^{\text{L}})$ and $(E, H', v' \in \gamma(T^{\text{L}})$; and

(b) If $\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, z.[y](\overline{x})\, \{\widetilde{P}\}$ and $(E, H) \in \gamma(\widetilde{E}, \widetilde{\Sigma})$ then $r = H' \downarrow a$ where $(E, H', v') \in \gamma(\widetilde{E}, \widetilde{P})$; and

(c) If $\widetilde{E} \vdash \{\widetilde{P}\}\, z.[y](\overline{x})\, \{\widetilde{P'}\}$ and exists v where $(E, H, v') \in \gamma(\widetilde{E}, \widetilde{P})$ then $r = H' \downarrow a$ where $(E, H', v') \in \gamma(\widetilde{E}, \widetilde{P'})$.

That is, we will show that r cannot be err, and thus derive a contradiction. We will (as usual), prove this with a simultaneous inner induction. Note that the conditions (a), (b), and (c) above are structural the same (except for the identity of the command) as those we have used in the other evaluation cases. The only different case is

Case T-Reflective-Method-Call By assumption, we have

$$\textsc{T-Reflective-Method-Call}$$
$$\frac{\Gamma <:_{[\overline{p \mapsto x}, \mathrm{self}:z]} \overline{p : T_p}, \mathrm{self} : \Gamma(z) \qquad \Gamma(z) = \{\cdots\} \upharpoonright \mathsf{respondsTo}\ y(\overline{p : T_p}) \to B_{\mathrm{ret}}}{\Gamma \vdash z.[y](\overline{x}) \ :\ B_{\mathrm{ret}} \upharpoonright}$$

Assume $(E, H) \in \gamma(\Gamma)$. Then $(E, H, E(z)) \in \gamma(\Gamma(z))$. But $\gamma(\Gamma(z))$ has refinement $\mathsf{respondsTo}\ y(\overline{p : T_p}) \to B_{\mathrm{ret}}$, so $(E, H, E(z)) \in \gamma(\mathsf{respondsTo}\ y(\overline{p : T_p}) \to B_{\mathrm{ret}})$. So, by the definition of concretization for respondsTo refinements, $o(E(y)) \in \gamma(B, (\overline{p : T_p}) \to B_{\mathrm{ret}})$. Since $E(y) = m$ we have $e = o(m)$. But this violates the assumption that $m \notin \mathrm{dom}(o)$, so we can assert $(E, H') \in \gamma(\Gamma^{\mathrm{L}})$ and $(E, H', v') \in \gamma(T^{\mathrm{L}})$, as desired.

Case E-Write-Field-Deref-Err

By assumption, we have

$$\textsc{E-Write-Field-Deref-Err}$$
$$\frac{v = E(x) \qquad v \notin \mathrm{dom}(H)}{E \vdash [H]\, x.f = y\, [r]}$$

where $r = \mathsf{err}$.

For this case, it suffices to show that:

(a) If $\Gamma^{\mathrm{L}} \vdash x.f = y \ :\ T^{\mathrm{L}}$ and $(E, H) \in \gamma(\Gamma^{\mathrm{L}})$ then $r = H' \downarrow a$ where $(E, H') \in \gamma(\Gamma^{\mathrm{L}})$ and $(E, H', v') \in \gamma(T^{\mathrm{L}})$; and

(b) If $\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, x.f = y\, \{\widetilde{P}\}$ and $(E, H) \in \gamma(\widetilde{E}, \widetilde{\Sigma})$ then $r = H' \downarrow a$ where $(E, H', v') \in \gamma(\widetilde{E}, \widetilde{P})$; and

(c) If $\widetilde{E} \vdash \{\widetilde{P}\}\, x.f = y\, \{\widetilde{P'}\}$ and exists v where $(E, H, v) \in \gamma(\widetilde{E}, \widetilde{P})$ then $r = H' \downarrow a$ where $(E, H', v') \in \gamma(\widetilde{E}, \widetilde{P'})$.

We show this by simultaneous induction on the typing and symbolic execution judgments. Again, we consider only the symbolic field write, since the type field write is similar to Deputy.

Case Sym-Write-Field By assumption, we have

$$\textsc{Sym-Write-Field}\quad \frac{}{\widetilde{E} \vdash \{\widetilde{\Gamma}, \widetilde{H}\}\, x.f := y\, \{\widetilde{\Gamma}, \widetilde{H}\big[\widetilde{E}(x) : \big(\widetilde{H}(\widetilde{E}(x))[f : \widetilde{E}(y)]\big)\big] \downarrow \widetilde{E}(y)\}}$$

Suppose $(E, H) \in \gamma(\widetilde{E}, (\widetilde{\Gamma}, \widetilde{H}))$. Without loss of generality, let $\widetilde{v} = \widetilde{E}(x)$ and note that the application of Sym-Write-Field requires that $\widetilde{v} \in \mathrm{dom}(\widetilde{H})$.

By the definition of concretization of symbolic state, Then, there exists a valuation V, and heaps H^{ok} and H^{mat} such that:

(A1) $(V, E) \in \gamma(\widetilde{E})$

(A2) $(V, H^{\mathrm{ok}}, H^{\mathrm{mat}}) \in \gamma(\widetilde{H})$,

(A3) $H = H^{\mathrm{ok}} * H^{\mathrm{mat}}$.

Because $v = E(x)$, by (A1) we have $V(\widetilde{v}) = v$. Since $\widetilde{v} \in \mathrm{dom}(\widetilde{H})$, by (A2) we have $v \in \mathrm{dom}(H^{\mathrm{mat}})$. But $H = H^{\mathrm{ok}} * H^{\mathrm{mat}}$, so $v \in \mathrm{dom}(H)$. But the application of E-Write-Field-Deref-Err requires that $v \notin \mathrm{dom}(H)$. This is a contradiction, so we can assert $r = H' \downarrow v'$ where $(E, H', v') \in \gamma(\widetilde{E}, \widetilde{P})$, as desired.

Case E-Seq-Err-1 By assumption, we have

$$\textsc{E-Seq-Err-1}\quad \frac{E \vdash [H]\, e_1\, [r_1]}{E \vdash [H]\, e_1 \ ;\ e_2\, [r]}$$

where $r_1 = \mathsf{err}$ and $r = \mathsf{err}$.

For this case, it suffices to show that:

(a) If $\Gamma^{\mathrm{L}} \vdash e_1 \,;\, e_2 : T^{\mathrm{L}}$ and $(E, H) \in \gamma(\Gamma^{\mathrm{L}})$ then $r = H'' \downarrow v_2$ where $(E, H'') \in \gamma(\Gamma^{\mathrm{L}})$ and $(E, H'', v_2) \in \gamma(T^{\mathrm{L}})$; and

(b) If $\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_1 \,;\, e_2 \,\{\widetilde{P}\}$ and $(E, H) \in \gamma(\widetilde{E}, \widetilde{\Sigma})$ then $r = H'' \downarrow v_2$ where $(E, H'', v_2) \in \gamma(\widetilde{E}, \widetilde{P})$; and

(c) If $\widetilde{E} \vdash \{\widetilde{P}\}\, e_1 \,;\, e_2 \,\{\widetilde{P}'\}$ and exists v where $(E, H', v) \in \gamma(\widetilde{E}, \widetilde{P})$ then $r = H'' \downarrow v_2$ where $(E, H'', v_2) \in \gamma(\widetilde{E}, \widetilde{P}')$.

The two interesting cases are: T-Seq and Sym-Seq.

Case T-Seq By assumption, we have

$$\textsc{T-Seq}\quad \frac{\Gamma^{\mathrm{L}} \vdash e_1 : T_1^{\mathrm{L}} \qquad \Gamma^{\mathrm{L}} \vdash e_2 : T_2^{\mathrm{L}}}{\Gamma^{\mathrm{L}} \vdash e_1 \,;\, e_2 : T_2^{\mathrm{L}}}$$

Suppose $(E, H) \in \gamma(\Gamma^{\mathrm{L}})$. Since $\Gamma \vdash e_1 : T^{\mathrm{L}}$, we can apply the outer induction hypothesis for $\mathcal{E}_1$, yielding $r_1 = (E, H', v_1)$ where $(E, H') \in \gamma(\Gamma^{\mathrm{L}})$. But, by assumption, $r_1 = \mathsf{err}$. This is a contradiction, so we can vacuously assert $r = H'' \downarrow v_2$ where $(E, H'') \in \gamma(\Gamma^{\mathrm{L}})$ and $(E, H'', v_2) \in \gamma(T^{\mathrm{L}})$, as required.

Case Sym-Seq By assumption, we have

$$\textsc{Sym-Seq}\quad \frac{\mathcal{S} :: \widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_1 \,\{\widetilde{P}'\} \qquad \mathcal{P} :: \widetilde{E} \vdash \{\widetilde{P}'\}\, e_2 \,\{\widetilde{P}''\}}{\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e_1 \,;\, e_2 \,\{\widetilde{P}''\}}$$

Suppose $(E, H) \in \gamma(\widetilde{E}, \widetilde{\Sigma})$. Then by the outer inductive hypothesis for $\mathcal{S}$, we have $r_1 = (E, H', v_1)$ where (E, H') where $(E, H', v_1) \in \gamma(\widetilde{P}')$. But, by assumption, $r_1 = \mathsf{err}$. This is a contradiction, so we can vacuously assert $r = H'' \downarrow v_2$ where $(E, H'', v_2) \in \gamma(\widetilde{E}, \widetilde{P})$.

□