Fissile Type Analysis: Modular Checking of Almost Everywhere Invariants

Devin Coughlin Bor-Yuh Evan Chang
University of Colorado Boulder
{devin.coughlin,evan.chang}@colorado.edu

Abstract

We present a generic analysis approach to the *imperative relationship update problem*, in which destructive updates temporarily violate a global invariant of interest. Such invariants can be conveniently and concisely specified with dependent refinement types, which are efficient to check flow-insensitively. Unfortunately, while traditional flow-insensitive type checking is fast, it is inapplicable when the desired invariants can be temporarily broken. To overcome this limitation, past works have directly ratcheted up the complexity of the type analysis and associated type invariants, leading to inefficient analysis and verbose specifications. In contrast, we propose a *generic lifting* of modular refinement type analyses with a symbolic analysis to efficiently and effectively check concise invariants that hold *almost everywhere*. The result is an efficient, highly modular flow-insensitive type analysis to *optimistically* check the preservation of global relationship invariants that can fall back to a precise, disjunctive symbolic analysis when the optimistic assumption is violated. This technique permits programmers to temporarily break and then re-establish relationship invariants—a flexibility that is crucial for checking relationships in real-world, imperative languages. A significant challenge is selectively violating the global type consistency invariant over heap locations, which we achieve via *almost type-consistent heaps*. To evaluate our approach, we have encoded the problem of verifying the safety of reflective method calls in dynamic languages as a refinement type checking problem. Our analysis is capable of validating reflective call safety at interactive speeds on commonly-used Objective-C libraries and applications.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords almost everywhere invariants; dependent refinement types; almost type-consistent heaps; reflection; Objective-C

1. Introduction

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* [38], which have been applied extensively to, for example, checking array bounds [12, 31, 32, 37]. Refinement 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. Reflective method call is a dynamic language feature that enables programmers to invoke a method via a run-time string value called a *selector*. In many languages, these calls have become commonplace in libraries as a mechanism to decouple client and framework code. 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.

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).

The imperative relationship update problem. A significant challenge in checking relationship invariants in dynamic, imperative programming languages is reasoning precisely about destructive updates of related storage locations, such as local variables on the stack and object fields on the heap. To illustrate this issue, consider an object `o` with an (object) invariant specifying a relationship between two fields `o.f` and `o.g`. For example, `o.f` could point to an object that should respond to a selector stored in `o.g`. We can then use this invariant to show that a reflective method call on `o.f` using selector `o.g` is safe. Again, there are many other interesting properties that require specification of relationships (e.g., `o.f` is an array with an in-bounds index stored in `o.g`, or `o.f` is a monitor object guarded by the lock stored in `o.g`).

Now consider a call `o.updateRelation(p)` to a simple method `updateRelation` that updates fields `o.f` and `o.g` with the corresponding fields from the argument `p`:

def `updateRelation(x) =` ❶ **self**`.f = x.f;` ❷ **self**`.g = x.g;` ❸

where three program points ❶ to ❸ are marked explicitly. Suppose that fields `p.f` and `p.g` of object `p` have the corresponding relationship with each other as those fields of `o` (e.g., `p.f` responds to `p.g`). Then it is clear that after a call to `o.updateRelation(p)`,

POPL '14, January 22–24, 2014, San Diego, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2544-8/14/01…$15.00.
http://dx.doi.org/10.1145/2535838.2535855

the relationship invariant between `o.f` and `o.g` should still hold. This property seems obvious, but two issues make it challenging to verify: (1) a standard, flow-insensitive type analysis by itself is insufficient and (2) sound symbolic reasoning about potential aliasing requires an expensive disjunctive analysis.

Issue 1 arises because the first assignment violates the type refinement on **self**`.f`. In our reflective call safety example, after the assignment, the new **self**`.f` (holding the value passed in `x.f`) might not respond to the old **self**`.g`. Switching the order of assignments does not help: then after the first assignment, the old **self**`.f` might not respond to the new **self**`.g`.

Problems arising from imperative relationship updates have been studied in the context of array bounds checking [12, 32, 37]. To deal with Issue 1, these type systems either require simultaneous relationship updates [12] or drop flow-insensitivity altogether and move to a flow-sensitive treatment of typing [32, 37]. In contrast to a flow-insensitive invariant, the type of a location in a flow-sensitive treatment is not fixed globally and is permitted to vary.

We argue that for the relationship update problem, computing new types at every program point is *overly pessimistic*. Although the ideal flow-insensitive type invariant imposed by the refinement on the **self**`.f` field is broken at program point ❷, it is restored almost immediately at point ❸. While the blocks of code between violations and restorations (e.g., points ❶ to ❸) require at least flow-sensitivity, a fast flow-insensitive analysis can still be effective in checking that the relationship invariant is preserved everywhere else.

Issue 2 arises in the above example because we must reason about (potentially) two concrete objects: the object pointed to by **self** and the (possibly same) object pointed to by `x`. While this code satisfies the desired property regardless of whether or not **self** and `x` are aliases, the analysis must consider both cases. Consider a slight variant where, for example, an assignment **self**`.f = null` occurs before point ❶. A sound analysis must detect a bug in this variant (when **self** and `cb` are aliases).

Prior approaches (e.g., [1, 2, 18, 32]) enable strong updates in some situations by reasoning about concrete objects one-at-a-time. While one-at-a-time reasoning is sound, it is insufficiently precise for even this unextraordinary example. In particular, a disjunctive analysis (e.g., a path-sensitive analysis) must reason about two cases: (1) where there is one concrete object—**self** and `x` are aliases—and (2) where there are two objects—**self** and `x` are not aliases.

We also argue that directly augmenting a type system to handle temporary invariant violations or other flow-analysis issues is *overly specific*. The imperative relationship update problem applies to checking just about any property that requires relational invariants. In contrast to the aforementioned works, the essence of our approach is to mix a property-specific refinement type system with a generic, symbolic flow analysis to handle temporary violations of the type invariant. The symbolic analysis performs execution-based reasoning and decouples issues like flow- and path-sensitivity from the particular type abstraction of interest.

Fissile type analysis. We propose flow-insensitive storage location (FISSILE) type analysis—a framework for enriching a dependent refinement type system to tolerate temporary violations via a generic, symbolic, separation logic-based, flow analysis. The result is an analysis that soundly checks a type invariant that holds almost everywhere. When the fast, flow-insensitive analysis detects an invariant violation, FISSILE type analysis *splits* the type environment (which relates storage locations to storage locations) into two components: (1) relationships between symbolic values (i.e., refinement types lifted to values) and (2) the locations where those values are stored (i.e., a symbolic memory). The symbolic analysis permits *bounded violation* of the storage location property and takes over until the type invariant is restored, at which point the fast flow-insensitive analysis resumes. As illustrated in the above example, we must support bounded relationship violations not only among local variables but also among a number of heap locations. Supporting bounded violations of heap locations is a significant challenge.

When applied to relationships that hold almost everywhere, FISSILE type analysis addresses the imperative relationship update problem while still maintaining two key benefits of flow-insensitivity:

1. It is nearly as fast as a flow-insensitive analysis (5,000 to 38,000 lines of code per second—see Section 4), since it begins with the *optimistic* assumption that the invariant holds everywhere and then only has to reason precisely about the relatively few program locations in which any relationships are violated.
2. It permits the vast majority (99.95%) of method type signatures to be flow-insensitive. Flow-insensitive type signatures do not need to specify the effect of the method on the heap. Contrast this to a modular, flow-sensitive analysis that has summaries specifying the effects on all methods, in essence, to rule out the *pessimistic* assumption that any callee could violate the invariant.

In summary, we make the following contributions:

- We describe the FISSILE type analysis framework for intertwined fast type and precise symbolic analysis of almost flow-insensitive storage location properties and instantiate it to check reflective call safety. Our framework is built on the observation that flow-insensitive type invariants on mutable storage locations really serve two roles: specifying facts about the values stored in them and constraining what values are allowed to be stored in them. We define two generic operations to *handoff* between the type and symbolic analyses that essentially either decouples or relinks these roles (Section 3.2).
- We introduce the notion of *almost type-consistent heaps* that (at the concrete level) splits the overall heap into two regions: one explicit region where locations are permitted to be type inconsistent and one implicit region where locations are type consistent up to encountering a location that is potentially inconsistent. We then define type-consistent materialization and summarization operations that permit transitioning an arbitrary number of locations with any connectivity relation between the two regions. With materialized locations, we thus enable strong update reasoning in our disjunctive symbolic analysis. These capabilities are critical for supporting bounded relationship violations among an *arbitrarily* bounded number of heap locations (Sections 2.2, 3.3, and 3.4).
- A key challenge in verifying re-establishment of violated refinement relationships is that the code that breaks a relationship may be in one module while the storage for that relationship is in another. We introduce symbolic summaries that enable programmers to specify encapsulated relationship storage and thus retain modular checking for *cross-module bounded violations* (Sections 2.3 and 3.5).
- We evaluate the effectiveness of our approach to checking reflective call safety on a suite of Objective-C libraries and applications and on code snippets posted by inexperienced developers on public forums (Section 4).

2. Overview

In this section, we present an example that illustrates the main challenges in permitting temporary violations of type consistency, particularly with respect to heap-allocated objects. Our examples are drawn from verifying reflective call safety in real-world Objective-C code, which require such temporary violations to be able to use simple, global type invariants. 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.

```
1  @interface Button
2  - (void)drawState:(String * ⦇↾in{'Up', 'Down'}⦈ )state {
3    String *m = ...
4    CustomImage *image = ...
5    m = ["draw" append:state];
6    [image setDelegate:self selector:m];
7    [image draw];
8  }
9  - (void)drawUp { ... }
10 - (void)drawDown { ... }
11 @end
12 @interface CustomImage {
13   Object * ⦇↾respondsTo selector()→void⦈ delegate; String *selector;
14 }
15 - (void)setDelegate:(Object * ⦇↾respondsTo s()→void⦈ )d
16            selector:(String *)s {
17   self->delegate = d;
18   self->selector = s;
19 }
20 - (void)draw { [self->delegate performSelector:self->selector]}; }
21 @end
```

Figure 1. Verifying reflective call safety requires knowing responds-to relationships between delegates and selectors.

The example in Figure 1, adapted from the ShortcutRecorder library, demonstrates how programmers use reflective method call to avoid boilerplate code and to decouple components. 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 infix notation (Smalltalk-inspired syntax). 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 20—the delegate and selector pair form, in essence, a callback. 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 program is reflection-safe, we use refinement types [19, 20, 31] 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 20. 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->delegate` responds to `self->selector`. There is an unexpressed invariant that requires that for every instance of `CustomImage`, the object stored in the `delegate` field must respond to the selector stored in the `selector` field. We capture this invariant by applying the **respondsTo** selector()→`void` refinement to the `delegate` field at line 13. This refinement expresses the desired *relationship* between the `delegate` field and the `selector` field. The method signature ()→`void` states that the `selector` field holds the name of a method that takes zero parameters with return type `void`. This expresses an intuitive invariant that, unfortunately, does not quite hold everywhere. Still, our framework is capable of using this *almost* everywhere invariant to check that the required relationships hold when needed—we revisit this point in Section 2.1.

Working backward, we see that the `setDelegate:selector:` method updates the `delegate` and `selector` 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 `d` to require that it responds to `s`. Any time the method is called, the first argument must respond to the second. Thus at the call to `setDelegate:selector:` at line 6, we 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 (whereas 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 the refinement that limits strings to one of a set of string constants (i.e., a union of singletons). For simplicity in presentation, our only refinement for string invariants is **in**, although more complex string reasoning is possible.

2.1 Insufficient: Flow-Insensitive Typing of Refinements

Restricting values stored in variables, parameters, and fields with **respondsTo** relationships and **in** refinements captures the essential invariants needed to verify reflective call safety. Here, we show why a flow-insensitive type analysis with these refinements is insufficient in the presence of imperative updates. We focus on the most interesting case: updates to fields of heap-allocated objects.

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. As a notational convenience, we write them without the bound variable and assume the bound variable is used as the first argument of all atomic relations, writing for example, Int $\restriction \geq 0$ instead of $\{\, v : \mathrm{Int} \mid v \geq 0 \,\}$.

Subtyping with refinement types. We assume the refinement type system of interest comes equipped with a subtyping judgment $\Gamma \vdash T_1 <: T_2$ that is a static over-approximation of semantic inclusion (i.e., under a context 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"`. Since `Button *` is a subtype of `Object *` in the base Objective-C type system and `Button` has both a `drawUp` and a `drawDown` method, `Button *` is a subtype of `Object *` ↾**respondsTo** m *in this environment*. If, for example, Γ(m) instead had refinement **in** {'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 [12] (although extended to handle subtyping). Here, we focus on why flow-insensitive typing raises an alarm after the field assignment `self->delegate = d` at line 17 (see Section 3 for more details on how checking proceeds). To check this 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

$$\begin{aligned}\Gamma_a \quad = \quad & \Gamma[\mathsf{d} : \texttt{Object *} \upharpoonright \textbf{respondsTo}\,\mathsf{s}] \\ & [\mathsf{delegate} : \texttt{Object *} \upharpoonright \textbf{respondsTo}\,\mathsf{selector}]\end{aligned}$$

for some Γ and where we explicitly show the two **respondsTo** refinements (for presentation, we use the field names as the fresh locals). Checking the assignment, we end up with the subtyping check $\Gamma_a \vdash \Gamma_a(\mathsf{d}) <: \Gamma_a(\mathsf{delegate})$, which expresses the invariant that the relationship between delegate and selector should be preserved across this assignment. Note that while this subtyping check is what is prescribed for assignment in a standard, non-dependent type system, the weakest-preconditions–based approach ensures that this same check is also required (and thus also fails) for the next line (line 18) where `selector` is updated. Although these two assignments are each unsafe in isolation, considered in combination, they are safe. The first assignment breaks the invariant that the `delegate` field 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 **respondsTo** 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.

2.2 Tolerating Violation of Relationship Refinements

As we previewed in Section 1, we argue that moving to a flow-sensitive treatment of typing is *too pessimistic*. A flow-sensitive type analysis drops the global constraints on mutable storage locations that enable concise specification. Instead it focuses on tracking facts on values as they flow through the program, which then require more verbose summaries. Other possible alternatives are to change the programming language to disallow temporary violations (e.g., by requiring simultaneous updates) or to somehow generalize the type invariant to capture the temporary violations of the simpler invariant; we argue that these approaches are *too specific*.

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. To set up our notion of *almost type-consistent* heaps, 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 the value stored in it *immediately* without pointer dereferences violates a type constraint or 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:

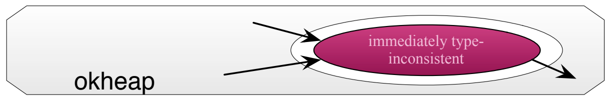
Premise 1 *All* of the heap is type-consistent *most* of the time.

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. Under this premise, these periods of violation are bounded in execution—and short enough that the path explosion from precise symbolic analysis is manageable.

To enable the intertwining of a type and a symbolic analysis, we require two conceptually inverse operations, symbolization and typeification, that are applied when switching between the two kinds of analyses. *Symbolization* splits a type environment Γ (which expresses relationship constraints between storage locations) into a symbolic fact map $\widetilde{\Gamma}$ and a symbolic memory $\widetilde{M}$. The fact map expresses relationships between symbolic *values* (i.e., refinement types lifted to the symbolic domain) and the memory indicates where those values are stored (symbolic local variables and heap). So, for example, splitting a type environment Γ with $[\mathsf{d} : \texttt{Object *} \upharpoonright \textbf{respondsTo}\,\mathsf{s}]$ will result in a fact map $\widetilde{\Gamma}$ with fact $[\widetilde{v_1} : \texttt{Object *} \upharpoonright \textbf{respondsTo}\,\widetilde{v_2}]$ and a memory $\widetilde{M}$ which maps d to $\widetilde{v_1}$ and s to $\widetilde{v_2}$. Here $\widetilde{v_1}$ and $\widetilde{v_2}$ are arbitrary symbolic names for values initially stored in d and s upon symbolization. In order for a symbolization to be sound, the symbolized fact map must be "no stronger" than the original type environment. A type environment Γ is not directly comparable to a symbolic fact map $\widetilde{\Gamma}$, so we capture the required relationship with a "subtyping under substitution" judgment $\Gamma <:_{\tilde{\theta}^{-1}} \widetilde{\Gamma}$ that, in essence, converts the symbolic fact map to a type environment before performing the comparison. We perform this conversion with a substitution $\tilde{\theta}^{-1}$, constructed from the symbolic memory, that maps symbolic values to the local variables that store them. *Typeification* fuses $\widetilde{\Gamma}$ and $\widetilde{M}$ back into a type environment Γ. In this case, the type environment under the forward memory substitution must be no stronger than the fact map: $\widetilde{\Gamma} <:_{\tilde{\theta}} \Gamma$. We describe these relations fully in Section 3.3.

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 intuitively below:



In the illustration, the dark node represents one location that is immediately type-inconsistent, while the light rectangle around it represents the almost type-consistent region. Note that there may be pointers (shown as arrows) to the this location where such objects are not type-consistent but only transitively type-inconsistent. In the analysis, the locations in the almost type-consistent region are summarized and represented by an atomic assertion okheap, while the possibly immediately type-inconsistent locations are materialized and explicitly given in the symbolic memory $\widetilde{M}$.

During symbolic analysis, we can materialize explicit storage locations from the almost type-consistent region okheap and thus update them to hold values that do not match their expected types (i.e., to become immediately type-inconsistent). When the materialized storage is again consistent with its expected types, we can summarize it back into okheap. When the heap consists *entirely* of okheap, we know that the entire heap is fully type-consistent and we can return to type checking. Given this mechanism, we can easily allow for more than one materialization at a time as long as we account for possible aliasing with already materialized locations—Premise 2 suggests that this explosion is also manageable. As an aside, we see okheap as a specialization of separating implication $-\!\!*$ from separation logic [30] for type-consistency. Materializing from okheap corresponds to removing a location from its concretization and introducing an implication awaiting type-consistency of the location (intuitively, the hole in the above illustration). Summarizing into okheap corresponds to putting a location into its concretization and eliminating an implication (see Section 3.3).

2.3 Modular Checking and Symbolic Summaries

Flow-insensitive type signatures are effective for achieving method-level modularity, but they do not summarize the effect that a method has on the heap. Such heap effect summaries are needed when the code that breaks and restores a relationship invariant is spread between multiple methods. Consider again the temporary invariant violation at line 17 in Figure 1, but suppose that instead of updating the delegate and selector fields directly, the programmer used the accessor idiom with setDelegate: and setSelector: methods:

```
17 [self setDelegate:d];
18 [self setSelector:s];
```

Each of these methods in isolation break the relationship invariant, but they are safe when used in combination. Flow-insensitive type signatures specify neither alias information nor the effect of the method on the heap—a richer specification is needed to check such cross-method relationship violations.

Rather than complicate the type analysis with flow-sensitive method type signatures or effect annotations, we enrich the symbolic analysis with the ability to summarize methods via pre- and post-conditions describing the structure of the heap (in separation logic [30]). For example, the symbolic summary for the setDelegate: method with a parameter named p would have input heap $\mathsf{self} \mapsto \{\mathsf{delegate} : -\}$ and output heap $\mathsf{self} \mapsto \{\mathsf{delegate} : \mathsf{p}\}$. This summary says that (1) regardless of its contents on method entry, on method exit the delegate field contains the passed in the parameter and (2) the method does not touch any fields other than delegate. When symbolically executing the caller of one of these methods, we can modularly apply the summary, as long we ensure conformance to the summary when checking the method body (Section 3.5).

Uniquely, we need symbolic method summaries only as a rare escape hatch because of an intertwined approach: within symbolic analysis, we can apply a nested type analysis with a different local type invariant. This enables using flow-insensitive method types within symbolic analysis and other ways to leverage symbolic reasoning outside of the type system (see Section 3.5). On one hand, flow-insensitive type signatures are imprecise but simple to express and fast to check. On the other, symbolic method summaries can precisely describe method heap effects but are complex to express and slow to check. We therefore can obtain the same precision as a fully symbolic analysis but still take advantage of the simpler, more efficient type analysis where the global type invariant holds. In contrast to flow-sensitive approaches (e.g., [32]), we do not need heap effect summaries for all methods—but rather only for those very few (0.05%) that leave a relationship invariant violated.

3. Storage Type and Symbolic Value Analysis

In this section, we provide an example-driven description of how FISSILE type analysis intertwines flow-insensitive type analysis and path-sensitive symbolic analysis to modularly check refinement relationships between storage locations. For reference, we give the full details of the type and symbolic analyses in our TR [13]; here we focus on the core contributions of our framework and illustrate them via an instantiation to check reflective call safety.

Preliminaries: Syntax of language and types. We describe FISSILE type analysis over a core imperative programming language of expressions e with objects and reflective method call. 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 $\{\overline{\mathrm{var}\, f : T \;=\; e}, \overline{\mathrm{def}\, m(\overline{p : T_p}) : B_{\mathrm{ret}}[S] \;=\; e}\}$. 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. Methods can also be optionally annotated with symbolic summaries S, which we revisit in Section 3.5. 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 @ 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 $e_1 \,[\!]\, e_2$. 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 and can be reflected in the symbolic analysis in the usual way (i.e., by strengthening the symbolic state with the guard condition). Finally, we have two method call forms: one for direct calls $z.m(\overline{x})$ and one for reflective calls $z.[y](\overline{x})$. 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 $\{\overline{\mathrm{var}\, f : T_f}, \; \overline{\mathrm{def}\, m(\overline{p : T_p}) {\rightarrow} B_{\mathrm{ret}}}\}$ with a set of refinements R, which are interpreted conjunctively.

The framework is parametrized by the language of refinements R needed to specify the invariants of interest, and refinements should be parametric with respect to the syntactic class of identifiers ι. 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 3.2), respectively. Because types include refinements, types are parametrized as well, written T^{L}, T^{F}, or T^{S}, as are type environments Γ.

3.1 Instantiation: The Reflection Checking Type Refinement

To verify reflective call safety, we have seen that the key property is the '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 symbolic fact, it says that an object must respond to the value named by ι. But as a type invariant on storage locations, the refinement also constrains *both* the storage location on which the refinement is applied and the storage location named by ι to hold a responder and a correspondingly valid selector for it. We also need some refinements on string values, such as the union of singletons: 'in $\{c_1, \ldots, c_n\}$' which says the value is one of the following (string) literals $c_1, \ldots, c_n$.

To type expressions, we use the standard typing judgment form $\Gamma \vdash e : T$ that says in a type environment Γ, expression e has type T. A type environment Γ is a finite map from 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 emphasizes that Γ is a flow-insensitive invariant.

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

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

Figure 2. Flow-insensitive typing for reflective method calls. The respondsTo refinement is checked on reflective dispatch.

We provide the full type system for reflective call safety in our TR [13]—here we focus on the rules needed to check reflective calls given the desired type invariant (Figure 2). The T-Reflective-Method-Call rule for checking a reflective method is itself not complicated: we require that the responder object z has a refinement guaranteeing that it responds to the selector y with method type signature $(\overline{p : T_p}) \to B_{\mathrm{ret}}$. The arguments to call are checked against the specified types of the parameters via $\Gamma <:_{[\overline{p{:}x},\mathrm{self}:z]} \overline{p : T_p}, \mathrm{self} : \Gamma(z)$. 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 $B_{\mathrm{ret}} \restriction$.

The respondsTo refinement is introduced via subtyping. The subtyping judgment $\Gamma \vdash T <: T'$ says that in typing environment Γ, type T implies type T'. The Sub-Obj-RespondsTo-Refl subtyping rule combines information from base object types and the environment to introduce respondsTo. This rule says that for any location x 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 x with that signature. We also have subtyping rules expressing component-wise implication of refinements, conjunctive weakening of the set of refinements, and disjunctive weakening of in refinements. 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 [31] and also replace the type rules for string operations with an off-the-shelf string solver.

Another Instantiation: Typing for array refinements. The Fissile type analysis framework is parametrized by the language of refinements and a decision procedure for 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) hasLength, which indicates that an array has the specified (non-zero) length and (2) 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 indexedBy x. We 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 $x < y$:

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

We 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 $\phi_1 \vdash_{\mathrm{SMT}} \phi_2$ 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.

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

Figure 3. The symbolic analysis state splits type environments into types lifted to values and the locations where values are stored.

3.2 Symbolic Analysis State and Handoff

The type analysis 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 to a path-sensitive symbolic analysis that continues until the constraints again hold.

We now walk through a modified version of the example from Section 2 to describe the key components of this switch: (1) we describe our symbolic analysis state and how we convert ("symbolize") a type environment to a symbolic state; (2) we describe type-consistent materialization and summarization from the almost type-consistent heap in Section 3.3; (3) we describe the proofs of soundness for handoff, materialization/summarization, and our overall analysis in Section 3.4; and (4) we explore the interaction between modular symbolic analysis and the almost type-consistent heap in Section 3.5.

Symbolic analysis state. In the symbolic analysis, we split a type environment Γ into a symbolic environment $\widetilde{E}$ and a symbolic state $\widetilde{\Sigma}$ (Figure 3). 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 $\ast$ notation borrowed from separation logic [30] to state that we refer to disjoint storage locations.

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 $\widetilde{H}_1 \ast \widetilde{H}_2$. Lastly and most importantly, a sub-heap can be okheap, 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 2.2, is the key mechanism for soundly transitioning between the type and symbolic analyses.

Handoff from type checking to symbolic execution. Consider the formal language version of the callback example from Section 2.2:

T-SYM-HANDOFF

$$\frac{\Gamma \ \overrightarrow{\mathsf{symbolize}}\, \widetilde{\Gamma}, \widetilde{E} \qquad \widetilde{E} \vdash \{\widetilde{\Gamma}, \mathsf{okheap}\}\, e\, \{\bigvee_i (\widetilde{\Gamma}_i, \mathsf{okheap}) \downarrow \widetilde{x}_i\} \qquad \widetilde{\Gamma}_i, \widetilde{E} \ \overrightarrow{\mathsf{typeify}}\, \Gamma \qquad \widetilde{\Gamma}_i \vdash \widetilde{\Gamma}_i(\widetilde{x}_i) <: T[\widetilde{E}] \qquad \text{for all } i}{\Gamma \vdash e : T}$$

C-STACK-SYMBOLIZE

$$\frac{\widetilde{E} \text{ is 1-1} \qquad \Gamma <:_{\widetilde{E}^{-1}} \widetilde{\Gamma}}{\Gamma \ \overrightarrow{\mathsf{symbolize}}\, \widetilde{\Gamma}, \widetilde{E}}$$

C-OBJECT-SYMBOLIZE

$$\frac{\Gamma^{\mathrm{F}} = \text{fieldtypes(B)} \qquad \tilde{o} \text{ is 1-1} \qquad \Gamma^{\mathrm{F}} <:_{\tilde{o}^{-1}} \widetilde{\Gamma}}{B \ \overrightarrow{\mathsf{symbolize}}\, \widetilde{\Gamma}, \tilde{o}}$$

M-MATERIALIZE

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

SYM-WRITE-FIELD

$$\frac{}{\widetilde{E} \vdash \{\widetilde{\Gamma}, \widetilde{H}\}\, x.f := y\, \{\widetilde{\Gamma}, \widetilde{H}\big[\widetilde{E}(x) : \big(\widetilde{H}(\widetilde{E}(x))[f : \widetilde{E}(y)]\big)\big] \downarrow \widetilde{E}(y)\}}$$

M-SUMMARIZE

$$\frac{\mathsf{okheap} \in \widetilde{H} \qquad \widetilde{\Gamma}(\widetilde{a}) = B \upharpoonright \cdots \qquad \widetilde{\Gamma}, \tilde{o} \ \overrightarrow{\mathsf{typeify}}\, B}{\widetilde{\Gamma}, (\widetilde{H} * \widetilde{a} : \tilde{o}) \ \overrightarrow{\mathsf{summarize}}\, \widetilde{\Gamma}, \widetilde{H}}$$

C-OBJECT-TYPEIFY

$$\frac{\widetilde{\Gamma} <:_{\tilde{o}} \text{fieldtypes}(B)}{\widetilde{\Gamma}, \tilde{o} \ \overrightarrow{\mathsf{typeify}}\, B}$$

SUB-TYPES-FACTS

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

C-STACK-TYPEIFY

$$\frac{\widetilde{\Gamma} <:_{\widetilde{E}} \Gamma}{\widetilde{\Gamma}, \widetilde{E} \ \overrightarrow{\mathsf{typeify}}\, \Gamma}$$

SUB-FACTS-TYPES

$$\frac{\widetilde{\Gamma} \vdash \widetilde{\Gamma}(\tilde{\theta}(x)) <: T[\tilde{\theta}] \qquad \text{for all } x : T \in \Gamma}{\widetilde{\Gamma} <:_{\tilde{\theta}} \Gamma}$$

Figure 4. Selected rules demonstrating how FISSILE type analysis switches to a symbolic analysis to tolerate bounded violations.

```
1  { var del: {} ↾ respondsTo sel, var sel: Str,
2    def update(d: {} ↾ respondsTo s, s: Str): Unit =
3      self.del := d;
4      self.sel := s  }
```

Here the `update` method updates the `del` and `sel` fields in sequence. Recall that the assignment at line 3 breaks the type invariant and the assignment at line 4 restores it. We illustrate the core operations (Figure 4) behind FISSILE type 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 3 and so will switch to symbolic execution. To do so, it will (1) "symbolize" a suitable symbolic analysis state from the type environment, (2) symbolically execute the two field writes, and (3) attempt to "typeify" the resultant symbolic analysis state back to the original type environment. If this succeeds, then the type analysis can continue.

The T-SYM-HANDOFF rule formalizes this process. It 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 and symbolically executing e in that state. The judgment form $\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e\, \{\widetilde{P}\}$ 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 state-value pairs $\widetilde{P}$ on output. Here the input facts and environment are obtained from Γ (via the $\Gamma \ \overrightarrow{\mathsf{symbolize}}\, \widetilde{\Gamma}, \widetilde{E}$ judgment form), while the input heap is initialized to a fully type-consistent heap okheap. After symbolic execution, the resultant symbolic state must be consistent with ("typeify to") the original Γ, and the symbolic facts about the resulting symbolic value must be consistent with the inferred type T of the expression. Note that here we lift the subtyping judgment $\cdot \vdash \cdot <: \cdot$ to symbolic types in the expected way. 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. Both the initially symbolized heap *and* the finally typeified heap must consist solely of okheap—the heap is assumed consistent on entry and must be restored on exit. 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. We will discuss restoration in detail later—first we describe symbolization.

Symbolizing type environments to symbolic states. 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 above at line 3, immediately before the first write:

$$\Gamma = [\mathsf{d} : \{\} \upharpoonright \mathsf{respondsTo}\ \mathsf{s}][\mathsf{s} : \mathsf{Str}][\mathsf{self} : \mathrm{T_{Image}}]$$

where $\mathrm{T_{Image}} = \{\mathsf{var\ del} : \{\} \upharpoonright \mathsf{respondsTo\ sel}, \mathsf{var\ sel} : \mathsf{Str}\}$. We can symbolize this environment to create a symbolic environment where $\widetilde{E} = [\mathsf{d} : \widetilde{\mathsf{d}}][\mathsf{s} : \widetilde{\mathsf{s}}][\mathsf{self} : \widetilde{\mathsf{self}}]$. Here we have created fresh symbolic names to represent the values stored on the stack: $\widetilde{\mathsf{d}}$ is the name of the value stored in local d, $\widetilde{\mathsf{s}}$ 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:

$$\widetilde{\Gamma} = [\widetilde{\mathsf{d}} : \{\} \upharpoonright \mathsf{respondsTo}\ \widetilde{\mathsf{s}}][\widetilde{\mathsf{s}} : \mathsf{Str}][\widetilde{\mathsf{self}} : \mathrm{T_{Image}}]$$

Note that the refinement on $\widetilde{\mathsf{d}}$ refers to symbolic value $\widetilde{\mathsf{s}}$ 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 $\mathrm{T_{Image}}$ 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 $\mathrm{T_{Image}}$ when summarized back into okheap.

We formalize type environment symbolization in rule C-STACK-SYMBOLIZE, 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 $B \upharpoonright R^{\mathrm{L}}$ 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 okheap (i.e., the almost type-consistent region). As we detail next, with this interpretation, our analysis *materializes* storage for objects from okheap *on demand*, which is not only more efficient but is required in the presence of recursion.

3.3 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 3. 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 okheap, 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 okheap. Before the field write at line 3 can proceed, the analysis must first materialize storage for the T_{Image} object pointed to by self to get:

$$\widetilde{H} = \mathsf{okheap} * \widetilde{\mathsf{self}} \mapsto \{\mathsf{del} : \widetilde{\mathsf{del}}, \mathsf{sel} : \widetilde{\mathsf{sel}}\}$$

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

We formalize type-consistent materialization with the C-OBJECT-SYMBOLIZE and M-MATERIALIZE rules. 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 okheap, 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 (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 $\mathsf{mayalias}_{\widetilde{\Sigma}}(\widetilde{a})$ 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 [26, 28] of objects.

Symbolic execution. With the symbolization and materialization complete, the analysis now executes the field writes at lines 3 and 4. 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 $\widetilde{H}$. We give the rest of our symbolic executions rules in our TR [13]—they are as expected. Unlike traditional symbolic analysis, our mixed approach can soundly ensure termination by falling back to type checking. In practice, we switch to types at the end of loop bodies to cut back edges and cut recursion with method summaries.

Summarizing symbolic objects back into types. After execution of the field writes, the symbolic heap at line 4 is:

$$\widetilde{H} = \mathsf{okheap} * \widetilde{\mathsf{self}} \mapsto \{\mathsf{del} : \widetilde{\mathsf{d}}, \mathsf{sel} : \widetilde{\mathsf{s}}\}.$$

That is, the fields now contain the values passed in as parameters. But recall that $\widetilde{\Gamma}(\widetilde{\mathsf{d}}) = \{\} \restriction \mathsf{respondsTo}\ \widetilde{\mathsf{s}}$ and $\widetilde{\Gamma}(\widetilde{\mathsf{self}}) = T_{\text{Image}}$. In this state, the value stored in field del again responds to the value stored in field sel—the flow-insensitive type invariant (T_{Image}) promised by $\widetilde{\Gamma}(\widetilde{\mathsf{self}})$ again holds—and thus the object can be safely summarized back into the okheap. We describe this process in rule M-SUMMARIZE, 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 $\widetilde{H} = \mathsf{okheap}$), the checker can end the handoff to symbolic analysis and resume type checking (back to rule T-SYM-HANDOFF) as long as the symbolic locals are consistent with the original Γ for all symbolic paths (rule C-STACK-TYPEIFY) and the returned symbolic values have facts consistent with the return type of the expression.

3.4 Concretization and Soundness

An important concern for materialization and summarization is whether information is transferred soundly between the type analysis and the symbolic analysis (i.e., we have a sound reduced product [15]). In particular, materialization "pulls" information from the heap type invariant on demand during symbolic execution and then permits temporary violations of the global heap invariant in some locations. We take an abstract interpretation-based approach [14] 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 describe, via concretization, the different meanings of object types and their associated reachable heaps in the two analyses. Further, we present the properties of these concretizations that are key to proving soundness of handoff and materialization/summarization and also state a theorem of intertwined analysis soundness. We provide complete concretization functions and a full proof of soundness in our TR [13].

Concretization. A concrete state consists of a concrete environment E, mapping variables x to values v, and a concrete heap H, which is a finite map from addresses a to concrete objects o bundled with their allocated base types B. We overload $*$ to indicate both the disjoint heap union of two concrete subheaps and the separating conjunction of two static symbolic subheaps.

Concretization functions give meaning to abstract constructs by describing how they constrain the set of possible values and states. As is standard, we write γ for concretization and overload it for all constructs, except base types and field types. Object base types and field types, crucially, have different meanings in the type domain and symbolic domain. To disambiguate, we write $\gamma(B)$ for concretization in the type analysis and $\widetilde{\gamma}(B)$ for the symbolic analysis.

Concretization in the type analysis. In the type analysis, the concretization of an object type is fairly standard: it constrains

$$\gamma(B) \triangleq \left\{ (H,a) \;\middle|\; \begin{array}{l} \text{exists } o \text{ where } H(a) = \langle o, B\rangle \text{ and} \\ \quad ① \text{ for all methods } m \\ \quad\quad o(m) \in \gamma(B, \overline{(p:T_p)} \to B_{\text{ret}}) \text{ and} \\ \quad ② \text{ for all fields } f \\ \quad\quad (H, o, o(f)) \in \gamma(T_f^{\text{F}}). \end{array} \right\}$$

$$\gamma(B \restriction R_1^{\text{F}}, \cdots, R_n^{\text{F}}) \triangleq \left\{ (H,o,v) \;\middle|\; \begin{array}{l} (H,v) \in \gamma(B) \text{ and} \\ \text{for all refinements } R_i \\ \quad (H,o,v) \in \gamma(R_i^{\text{F}}) \end{array} \right\}$$

(a) Types domain

$$\widetilde{\gamma}(B) \triangleq \left\{ (H^{\text{ok}}, H^{\text{mat}}, a) \;\middle|\; \begin{array}{l} \text{exists } o \text{ where } H^{\text{ok}} * H^{\text{mat}}(a) = \langle o, B\rangle \text{ and} \\ \quad ① \text{ for all methods } m \\ \quad\quad o(m) \in \gamma(B, \overline{(p:T_p)} \to B_{\text{ret}}) \text{ and} \\ \quad ② \text{ for all fields } f \\ \quad\quad f \in \text{dom}(o) \text{ and if } a \in \text{dom}(H^{\text{ok}}) \text{ then} \\ \quad\quad\quad (H, o, o(f)) \in \widetilde{\gamma}(T_f^{\text{F}}). \end{array} \right\}$$

$$\widetilde{\gamma}(B \restriction R_1^{\text{F}}, \cdots, R_n^{\text{F}}) \triangleq \left\{ (H^{\text{ok}}, H^{\text{mat}}, v) \;\middle|\; \begin{array}{l} (H,v) \in \widetilde{\gamma}(B) \text{ and} \\ \text{for all refinements } R_i \\ \quad (H^{\text{ok}} * H^{\text{mat}}, o, v) \in \gamma(R_i^{\text{F}}) \end{array} \right\}$$

(b) Symbolic domain

Figure 5. Concretization of an object base type $B = \{\overline{\text{var } f : T_f}, \ \overline{\text{def } m(\overline{p:T_p}) \to B_{\text{ret}}}\}$ in the types and symbolic domains. In the symbolic domain, values stored in the fields of an object in H^{ok} must not be immediately type-inconsistent; values stored in the fields of an object in H^{mat} are not constrained. The shaded region highlights the key difference between the concretizations.

not only the value but also the entire heap reachable from that object. As we show in Figure 5a, the concretization $\gamma(B)$ of an object type $B = \{\overline{\text{var } f : T_f}, \ \overline{\text{def } m(\overline{p:T_p}) \to B_{\text{ret}}}\}$ yields a set of pairs of heaps and values (addresses). The concretization requires that the address point to an object o which ① has suitable method implementations (i.e., constrained by the concretization of the method signature $\overline{(p:T_p)} \to B_{\text{ret}}$ on an object of type B) for each method m in the object type and ② has suitable field values for each declared field f of type T_f^{F}. We adorn the type with its storage class, F, to make clear that it is a field type dependent on other fields.

The key property of concretization in the type domain is that the concretization of a field type $T^{\text{F}} = B \restriction R_1^{\text{F}}, \cdots, R_n^{\text{F}}$ is mutually inductively defined with that for base types and thus *constrains the entire heap reachable from that field*, in addition to the other fields of the object. This concretization yields a heap-object-value triple where the heap and value are constrained by the concretization of the base type and the entire triple is constrained by the concretization of each of the field refinements R_i^{F}. Because these refinements are dependent refinements, they may constrain *other* fields of the object in addition to the value of the field in question. For example, a heap-object-value triple in the concretization of a field refinement $\mathsf{respondsTo}\ g$ would constrain the g field of the object to store a string with name m that is a valid method for the (potentially different) object pointed to by the value. The concretization of a type environment $\gamma(\Gamma)$ is a set of concrete environment-heap (E,H) pairs where the values stored in local variables are consistent with the declared types and reachable heaps from the type bindings in Γ.

Concretization in the symbolic analysis. In contrast to the type analysis, in the symbolic world part of the heap may be explicitly materialized—and thus immediately type-inconsistent, leaving the rest of the heap almost type-consistent. To capture this difference, the symbolic concretization of an object type yields *two* heaps: H^{ok} and H^{mat}, corresponding to a non-deterministic choice of which objects are in the almost type-consistent heap and which objects are materialized and thus may have field values differing from their declared types. We write $\widetilde{\gamma}(B)$ here to emphasize concretization of base types in the symbolic domain. The shaded region of Figure 5b illustrates the key difference: in the symbolic domain, an object's fields are only constrained by the concretization of the field types if the object is in H^{ok}. If the object is in H^{mat}, the fields are guaranteed to exist, but the values stored in them are not constrained. Because concretization of object types is defined inductively, the concretization can "opt out" of type constraints for the reachable subheap at each field dereference, depending on the partitioning of the full heap into H^{ok} and H^{mat}. Crucially, this definition of concretization permits pointers from H^{ok} to H^{mat} and vice-versa. Note that regardless of which heap an object resides in, its method implementations are still constrained by their signatures.

The concretization $\gamma(\widetilde{E}, \widetilde{\Sigma})$ of a symbolic state $\widetilde{\Sigma} = (\widetilde{\Gamma}, \widetilde{H})$ with respect to a symbolic environment $\widetilde{E}$ yields a environment-heap pair (E,H). There must exist a valuation $V : \mathsf{SymVal} \to \mathsf{Values}$ mapping symbolic values to concrete values and a partitioning of the heap $H = H^{\text{ok}} * H^{\text{mat}}$ such that the concretizations of $\widetilde{E}$, $\widetilde{\Gamma}$, and $\widetilde{H}$ all agree upon. Symbolic fact maps $\widetilde{\Gamma}$ and heaps $\widetilde{H}$ both concretize to a set of valuation-heap-heap tuples where emp in the symbolic heap requires both H^{ok} and H^{mat} be empty whereas okheap requires that H^{mat} be empty but H^{ok} can be any heap. The singleton heap formula concretizes to a singleton in H^{mat} and $*$ is the heap disjoint union in both H^{ok} and H^{mat} A symbolic path is a disjunction of singleton paths $\widetilde{P} = \widetilde{\Sigma} \downarrow \widetilde{x}$; its concretization is similar to that of a symbolic state, except that it yields a triple (E,H,v) where $V(\widetilde{x}) = v$.

Soundness. The soundness of FISSILE type analysis—and in particular of handoff and materialization/summarization—depends on the following key properties of concretization.

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 H^{ok}), the meaning of base types in the symbolic domain is the same as the meaning of base types in the type domain:

Lemma 1 (Equivalence of Typed and Symbolic Base Types)**.** $\gamma(B) = \ \{(H,v) \mid (H,\cdot,v) \in \widetilde{\gamma}(B) \ \}$

We rely on this result to show that that the T-SYM-HANDOFF and SYM-TYPE-HANDOFF (Section 3.5) rules are sound.

To show soundness of the M-MATERIALIZE rule, we rely on a property about the meaning of base types in the symbolic domain:

Lemma 2 (Type-Consistent Materialization for Types)**.** If $(H^{\text{ok}} * a : \langle o_a, B_a\rangle, H^{\text{mat}}, v) \in \widetilde{\gamma}(B)$ then $(H^{\text{ok}}, H^{\text{mat}} * a : \langle o_a, B_a\rangle, v) \in \widetilde{\gamma}(B)$.

This lemma considers a value v of type B and a heap containing an object o_a of allocated type B_a stored at address a (informally, a is in v's reachable heap, otherwise it is uninteresting). If there is one concretization of B where a is in the almost type-consistent heap H^{ok}, then the configuration with a moved to the materialized H^{mat} is also in its concretization. In essence, moving the storage for a to the materialized heap will not cause the type of v to change from the perspective of the symbolic analysis.

Sym-Type-Handoff

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

Sym-Method-Call-Summary

$$\frac{\begin{array}{c}(\overline{p})[h/h'] : p^{\mathrm{ret}} = \mathrm{summary}(\widetilde{\Gamma}(\widetilde{E}(x)),m)\\ \widetilde{H} \vdash h \circledast \widetilde{H}^{\mathrm{fr}} \backslash \tilde{\theta}^{\mathrm{h}} \qquad \tilde{\theta} = \tilde{\theta}^{\mathrm{h}} \uplus [\overline{p : \widetilde{E}(y)}]\end{array}}{\widetilde{E} \vdash \{\widetilde{\Gamma},\widetilde{H}\}\, x.m(\overline{y})\, \{\widetilde{\Gamma},(\widetilde{H}^{\mathrm{fr}} \ast h'[\tilde{\theta}]) \downarrow \tilde{\theta}(p^{\mathrm{ret}}))\}}$$

Figure 6. Symbolic execution rules to switch to the typed analysis and to apply symbolic summaries.

A related lemma describes summarization:

Lemma 3 (Soundness of Type-Consistent Summarization for States)**.** If $\widetilde{\Gamma},\tilde{o}\ \overrightarrow{\mathsf{typeify}}\ B$ and $\mathsf{okheap} \in \widetilde{H}$ and $\widetilde{\Gamma}(\widetilde{a}) = B\ \upharpoonright\ \cdots$ then for all $\widetilde{E}$ we have $\gamma(\widetilde{E},(\widetilde{\Gamma},\widetilde{H} \ast \widetilde{a} : \tilde{o})) \subseteq \gamma(\widetilde{E},(\widetilde{\Gamma},\widetilde{H}))$.

That is, if the symbolic execution determines that a materialized symbolic object is not immediately type inconsistent, then it can summarize the symbolic storage for that object back into okheap (rule M-Summarize) without unsoundly dropping concrete states.

We rely on the above lemmas to prove soundness of the intertwined analysis:

Theorem 1. Fissile *Type Analysis is sound. That is, 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* $(E,H',v') \in \gamma(\widetilde{E},\widetilde{P})$.

This is a fairly standard statement of soundness of for both analyses. Here, $E \vdash [H]\, e\, [r]$ says that in a concrete environment E and with a concrete heap H, an expression e big-step evaluates to a result r. For types, if the initial concrete state is described by the static type environment and the expression type checks, then the expression evaluates to a heap-value pair (that is, not to an error) where the final state still conforms to the type environment and the value conforms to the static type. Similarly, the symbolic analysis soundly over-approximates the concrete execution and rules out a faulting error. The proof proceeds by induction on the derivation of concrete execution and covers an additional judgment form describing generalized path-to-path symbolic analysis—we provide full details in our TR [13].

3.5 Fully-Intertwined Type-Symbolic Analysis

As described in Section 3.3, the type analysis can hand off checking of an expression to the symbolic analysis. We also would like to perform the opposite handoff from symbolic to type (i.e., apply type analysis within symbolic analysis). For example, we would like do this handoff when we encounter a method call during symbolic analysis to enable modular analysis with only type specifications.

Fortunately, we can employ the same consistency mechanisms to allow handoff in the other direction, that is, to switch from symbolic analysis to type checking (rule Sym-Type-Handoff in Figure 6). The key constraint is that the entire heap must be fully type-consistent (okheap). In general, this requirement means summarization should have been applied so that no locations are materialized. The entire state is checked type consistent via "typeify" to Γ before type checking, and then symbolic analysis can resume by "symbolize"-ing a new symbolic state from Γ and adding an assumption from the type of the expression.

Suppose this symbolic analysis is the context of an outer flow-insensitive type analysis. Observe that type environment Γ is derived solely from the current symbolic execution state and could be more precise than the outer flow-insensitive invariant. A common scenario is to leverage symbolic reasoning about guard conditions in an `if` but switch to type analysis inside the `if` body with a stronger invariant without needing to the make any changes to the type analysis (cf. occurrence typing [35]). For the reflective call safety client, the "checked delegate" idiom is quite common where a reflective call is performed after checking if an object responds to a particular string—`respondsToSelector:` in Objective-C.

Symbolic summaries for cross-module bounded violations. On the other hand, we have seen a handful of cases where a further precision refinement is needed: when programmers break relationship refinements across module boundaries, such as when they abstract the heap with getters and setters. To see how this is a problem, consider the slightly modified version of the update function in which the direct field access is replaced the following setter functions:

```
  def setDel(d)[del : −/del : d] : − = self.del := d
  def setSel(s)[sel : −/sel : s] : − = self.sel := s
```

This small change exacerbates the problem of checking relationship refinement safety because now the invariant violation crosses module (function) boundaries and thus cannot go to a fully type-consistent heap on method call. To support this scenario, we enable the programmer to supply additional checked annotations, symbolic method summaries of the form $(\overline{p})[h/h'] : p^{\mathrm{ret}}$, that permit symbolic checking across module boundaries. Here $\overline{p}$ is a sequence of method parameter names, h and h' are summaries of the input heap and output heap respectively, and p^{ret} specifies the method return value. The input summary gives the form of the part of the heap the method will operate upon (the *footprint*) and gives names to the values stored in the fields of the input heap. The output summary gives the form of the heap after the method has finished executing, in terms of the names given in the input heap and the parameters. For example, the method summary annotation at line 1 says that whatever the `del` field stores before `setDel` is called, after the method is executed the field stores the value from the `d` parameter. Since the method is a setter, the return value is irrelevant. The annotation for `setSel` is analogous. Applying the summary for `setDel` leaves the heap in an inconsistent state, but then the summary for `setSel` restores it.

With these annotations, checking the calls to the setters is analogous to checking the field writes as before, except that rather than applying the symbolic transfer function for field writes, we apply the method summary. We formalize this in the Sym-Method-Call-Summary rule in Figure 6. The auxiliary function $\mathrm{summary}(T,m)$ looks up the summary of a method m on a static type T. The judgment $\widetilde{H} \vdash h \circledast \widetilde{H}^{\mathrm{fr}} \backslash \tilde{\theta}^{\mathrm{h}}$ splits the heap $\widetilde{H}$ into a footprint (specified by h) and the left-over frame ($\widetilde{H}^{\mathrm{fr}}$) that the method is guaranteed not to touch. This frame inference also produces a symbolic map $\tilde{\theta}^{\mathrm{h}}$ that captures the symbolic variables that the parameters in the h match to during the splitting. We then apply this map (combined with the mapping from parameter names to the symbolic values passed in as parameters) to the output summary h' to get the footprint on method exit and add it back to the frame to get the entire heap on method exit. We also consult this map $\tilde{\theta}$ to look up the value returned as specified by the summary. We write $\uplus$ for disjoint union of maps where the result is undefined if the union is not a map. The frame inference is a simple application of subtraction [3] but makes this rule quite flexible. The developer can choose to refuse access to okheap in a symbolic summary (as above) to provide a stronger guarantee to the method's callers or request it to get a stronger assumption for checking the method implementation.

We note that the particular kind of symbolic summary that we describe above (essentially, standard pre-/post-conditions in separation logic) is not the most interesting point. Other symbolic analysis techniques could be applied, such as context-sensitive, nonmodular reasoning. Rather, we argue that the interest lies in that heavier-weight symbolic analysis machinery can be applied when needed without the cost of applying it everywhere, all the time.

Checking Method Implementations. For modular checking, we type check each method implementation separately according to its type signature (as is standard). We guarantee that the implementation of a symbolically-summarized method conforms to its summary with our symbolic analysis infrastructure, although the summary checking technique is orthogonal to that for checking relationship refinements. One could substitute a different approach (e.g. abstract interpretation or interactive proofs) if desired.

4. 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.

Table 1 describes our prototype's performance on a benchmark suite of 6 libraries and 3 large applications. 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—something that would be challenging for a purely symbolic analysis. We discuss these results throughout the rest of this section.

The developer cost to add modular reflection checking. To seed potential type errors, we first annotated 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, for each benchmark we report the total number of developer annotations required, as well as the average number required per reflective call site, to give an indication of how much work it would be for developers to modularly check their use of reflection (column "Total Annotations"). All annotations are *checked*—they emit a static type error if their requirements are not met. Based on this column, 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 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. We then compared the number of static type errors reported by each (columns "FI Type Errors" and "FISSILE Type Errors"). FISSILE type analysis sometimes significantly reduces 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 (like, what has been 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 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 2.2): (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. As Table 1 shows, the number of times the analysis switches to symbolic execution and back (column "Successful Symbolic Sections") is quite low, even for large programs—Premise 1 appears to hold empirically. The maximum number of simultaneous materializations (column "Max. Mats.) 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. 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 (column "Analysis Time") 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.

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

Benchmark	Lines of Code	Refl. Call Sites	Methods	Total Annotations / Per Refl. Site	Symbolic Annotations / Per Refl. Site	Check Sites	FI Type Errors	FISSILE Type Errors (% Reduced)	Successful Symbolic Sections	Max. Mats.	Analysis Time (Rate)
OAuth	1248	7	92	5 / 0.71	0 / 0.00	11	7	2 (- 71%)	7	1	0.24s (5.3 kloc/s)
SCRecorder	2716	12	200	9 / 0.75	4 / 0.33	15	2	0 (-100%)	2	2	0.28s (10.8 kloc/s)
ZipKit	3301	28	165	0 / 0.00	0 / 0.00	28	0	0 (–)	0	0	0.10s (33.0 kloc/s)
Sparkle	5290	40	320	0 / 0.00	0 / 0.00	40	4	1 (- 75%)	3	1	0.67s (7.9 kloc/s)
ASIHTTP	13565	68	707	2 / 0.03	2 / 0.03	68	50	10 (- 80%)	59	2	0.50s (27.2 kloc/s)
OmniFrwks	160769	192	7611	49 / 0.26	2 / 0.01	259	82	74 (- 10%)	9	1	4.25s (37.8 kloc/s)
Vienna	37348	186	2261	24 / 0.13	4 / 0.02	207	59	38 (- 36%)	28	2	2.79s (13.4 kloc/s)
Skim	60211	207	3010	7 / 0.03	0 / 0.00	212	43	43 (- 0%)	0	0	2.49s (24.1 kloc/s)
Adium	176632	587	8723	40 / 0.07	0 / 0.00	648	87	70 (- 20%)	17	1	8.79s (20.1 kloc/s)
Combined	461080	1327	23089	136 / 0.10	12 / 0.01	1488	334	238 (- 29%)	125	2	20.09s (23.0 kloc/s)

Table 1. The "Lines of Code" count includes project headers but excludes comments and whitespace; "Methods" indicates the total number of methods; "Refl. Call Sites" gives the number of calls to system library methods that perform reflection, either directly or as part of some other operation; "Total Annotations" lists the total number of annotations and the average number of annotations required per reflective callsite; "Symbolic Annotations" lists gives the number of symbolic summaries required; "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; "Successful Symbolic Sections" gives the number of times our analysis successfully switched from type checking to symbolic execution and back again; "Max. Mats." gives the maximum number of materialized objects ever present in the explicit heap (this includes unsuccessful symbolic sections); and "Analysis Time" indicates the speed of our analysis on each benchmark, in both absolute terms and in lines of code per second. Our benchmarks include OAuth, which performs OAuth Consumer authentication; SCRecorder, which records custom keyboard shortcuts and is the source of our motivating example in Section 2; ZipKit, which reads and writes compressed archives; Sparkle, a widely-used automatic updater; ASIHTTP, which performs web services calls; and the OmniFrwks, which provide base functionality to the widely used OmniGraffle application; Vienna, an RSS newsreader; Skim, a PDF reader; and Adium, an instant message chat client. 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.

analysis detects this error and issues an alarm. We reported the bug to the developers; they acknowledged it as a bug and fixed it (see `https://github.com/ViennaRSS/vienna-rss/pull/85`).

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. 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.

5. Related Work

Dependent refinement types [20, 38] 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 [34, 37] 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 [31], of which CSOLVE [32] 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 [12] and support arbitrary materialization with a disjunctive symbolic analysis, as opposed to proving non-aliasing for one materialization (e.g., [1, 2, 18]). DJS [11] 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.

There has been a recent explosion in techniques (e.g., [7, 22]) that have significantly improved the effectiveness of symbolic execution. The SMPP approach [24] 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. Our approach of switching between type checking and symbolic execution is similar to the MIX system [25] 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. The notion of temporary violations of an invariant is also reminiscent of the large body of work on object invariants (see [17] 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.

On materialized heap locations, our symbolic analysis works over separation logic [3, 30] formulas. We define an on-demand materialization [33] that is universal in separation-logic–based analyzers [4, 9, 16]. However, our materialization operator pulls out heap cells that are summarized and validated *independently* using a refinement type analysis. Bi-abductive shape analyses [8, 23] 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.

Most prior work on reflection analysis has focused on whole-program *resolution*: determining, at a reflective site, what method is called (either statically [6, 10, 36] or dynamically [5, 21]).

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. [29] describe a type system that modularly checks reflection safety by combining occurrence typing [35] 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. [27] *assume* reflection safety and leverage this assumption to improve precision of callgraph construction.

6. Conclusion

We have described FISSILE type analysis, which intertwines fast type analysis over storage locations and precise symbolic analysis over values to efficiently, effectively, and modularly prove relationship properties. We have evaluated FISSILE using an interesting safety property—reflective method call safety. The key technical enabler for our analysis is *materialization from an almost type-consistent heap*. On a benchmark suite consisting of commonly-used Objective-C libraries (6) and applications (3), we find that our approach is capable of finding confirmed bugs in both production and beginner code. It has a balanced annotation burden that is negligible for clients of reflection and moderate (up to 0.75 annotations per reflective call site) for reflective interfaces.

FISSILE type analysis starts with the optimistic assumption that global flow-insensitive relationship invariants hold almost everywhere—it only has to use precise and expensive reasoning for those few program locations that violate a relationship. Contrast this to traditional modular flow-sensitive analyses, which require *all* methods to specify their effect on the heap to rule out the pessimistic assumption that relationship invariants could be violated anywhere. Our approach permits the vast majority (99.95%) of methods to avoid any annotations related to heap effects. Checking most of the program flow-insensitively allows FISSILE to validate reflective method call safety at interactive speeds (5 to 38 kloc/s with an overall rate of 23.0 kloc/s over our benchmark suite).

Acknowledgments

We thank Sriram Sankaranarayanan, Amer Diwan, Jeremy Siek, the CUPLV group, and Ranjit Jhala for insightful discussions, as well as the anonymous reviewers for their helpful comments. This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-1218208 and CCF-1055066.

References

[1] A. Ahmed, M. Fluet, and G. Morrisett. L^3: A linear language with locations. *Fundam. Inform.*, 77(4), 2007.

[2] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *PLDI*, 2003.

[3] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, 2005.

[4] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, 2007.

[5] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming Reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*, 2011.

[6] M. Braux and J. Noyé. Towards partially evaluating reflection in Java. In *PEPM*, 2000.

[7] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[8] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.

[9] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, 2008.

[10] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *SAS*, 2003.

[11] R. Chugh, D. Herman, and R. Jhala. Dependent types for JavaScript. In *OOPSLA*, 2012.

[12] J. Condit, M. Harren, Z. R. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *ESOP*, 2007.

[13] D. Coughlin and B.-Y. E. Chang. Fissile Type Analysis: Modular checking of almost everywhere invariants (extended version), 2013.

[14] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[15] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.

[16] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.

[17] S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In *ECOOP*, 2008.

[18] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, 2002.

[19] C. Flanagan. Hybrid type checking. In *POPL*, 2006.

[20] T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, 1991.

[21] M. Furr, J.-h. D. An, and J. S. Foster. Profile-guided static typing for dynamic scripting languages. In *OOPSLA*, 2009.

[22] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.

[23] B. S. Gulavani, S. Chakraborty, G. Ramalingam, and A. V. Nori. Bottom-up shape analysis. In *SAS*, 2009.

[24] W. R. Harris, S. Sankaranarayanan, F. Ivancic, and A. Gupta. Program analysis via satisfiability modulo path programs. In *POPL*, 2010.

[25] Y. P. Khoo, B.-Y. E. Chang, and J. S. Foster. Mixing type checking and symbolic execution. In *PLDI*, 2010.

[26] V. Laviron, B.-Y. E. Chang, and X. Rival. Separating shape graphs. In *ESOP*, 2010.

[27] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In *APLAS*, 2005.

[28] M. J. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, Computer Laboratory, 2005.

[29] J. G. Politz, A. Guha, and S. Krishnamurthi. Semantics and types for objects with first-class member names. In *FOOL*, 2012.

[30] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[31] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.

[32] P. M. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL*, 2010.

[33] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1), 1998.

[34] R. Tate, J. Chen, and C. Hawblitzel. Inferable object-oriented typed assembly language. In *PLDI*, 2010.

[35] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *POPL*, 2008.

[36] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective taint analysis of web applications. In *PLDI*, 2009.

[37] H. Xi. Imperative programming with dependent types. In *LICS*, 2000.

[38] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, 1999.