Combined Static and Dynamic Mutability Analysis

Shay Artzi Adam Kieżun David Glasser Michael D. Ernst
MIT Computer Science and Artificial Intelligence Laboratory
{artzi,akiezun,glasser,mernst}@csail.mit.edu

Abstract

Knowing which method parameters may be mutated during a method's execution is useful for many software engineering tasks. We present an approach to discovering parameter immutability, in which several lightweight, scalable analyses are combined in stages, with each stage refining the overall result. The resulting analysis is scalable and combines the strengths of its component analyses. As one of the component analyses, we present a novel, dynamic mutability analysis and show how its results can be improved by random input generation. Experimental results on programs of up to 185 kLOC show that, compared to previous approaches, our approach increases both scalability and overall accuracy.

1. Introduction

Knowing which method parameters are accessed in a read-only way, and which ones may be mutated, is useful in many software engineering tasks, such as modeling [8], verification [54], compiler optimizations [11, 47], program transformations such as refactoring [20], test input generation [2], regression oracle creation [32, 59], invariant detection [18], specification mining [13], program slicing [58, 52], and program comprehension [17].

Previous work on mutability has employed static analysis techniques to detect *immutable* parameters. Static analysis approximations can lead to weak results and computing better approximations affects scalability. Dynamic analyses offer an attractive complement to static approaches, both in not using approximations and in detecting *mutable* parameters.

This paper presents an approach to the mutability problem that combines the strengths of static and dynamic analyses to create an analysis that is both accurate and scalable. In our approach, different analyses are combined in stages, forming a "pipeline", with each stage refining the overall result. The result is an analysis that is more accurate and more scalable than previously developed techniques.

This paper makes the following contributions:

- The first formal definition of reference immutability that takes into account parameter aliasing.
- A staged analysis approach for discovering parameter mutability. The idea of staged analyses is not new, but a staged approach has not been investigated in the context of mutability analysis. Our staged approach is unusual in that it combines static and dynamic stages and it explicitly represents analysis imprecision. The framework is sound, but an unsound analysis may be used as a component, and we examine the tradeoffs involved.
- Mutability analyses. The primary contribution is a novel, dynamic analysis that scales well, yields accurate results (it has a sound mode as well as optional heuristics), and complements existing analyses. We extend the dynamic analysis with random input generation, which improves the analysis results by increasing code coverage.
- Evaluation. We have implemented our framework and analyses for Java, and we investigate the costs and benefits of various sound and unsound techniques, including both our own and that of Sălcianu and Rinard [48]. Our results show that a well-designed collection of fast, simple analyses can outperform a sophisticated analysis in both scalability and accuracy.

The remainder of this paper is organized as follows. Section 2 describes the problem of inferring parameter mutability and illustrates it on an example. Section 3 presents our staged mutability analysis. Sections 4 and 5 describe the dynamic and static mutability analyses that we developed as components in the staged analysis. Section 6 describes the experimental evaluation. Section 7 surveys related work, and Section 8 concludes.

2. Parameter Reference Immutability

The goal of parameter mutability analysis is the classification of each method parameter (including the receiver) as either reference-mutable or reference-immutable.

Appendix A formally defines reference (im)mutability. Informally, reference immutability guarantees that a given reference is not used to modify its referent. Parameter *p* of method *m* is reference-*mutable* if there exists an execution of *m* in which *p* is *used* to mutate the state of the object pointed to by *p*. Parameter *p* is said to be *used* in a mutation, if the left hand side of the mutating assignment can be replaced with a series of executed field accesses from *p*. (Array access are treated analogously throughout this paper.) If no such execution exists, the parameter *p* is reference-*immutable*. The state of an object *o* consists of the values of *o*'s primitive

```
1 class C {
2   public C next;
3 }
4
5 class Main {
6  void modifyParam1(C p1, boolean doIt) {
7     if (doIt) {
8        p1.next = null;
9     }
10 }
11
12 void modifyParam1Indirectly(C p2, boolean doIt) {
13    modifyParam1(p2, doIt);
14 }
15
16 void modifyAll(C p3, C p4, C p5, C p6, boolean doIt) {
17    p4.next = p3;
18    C c = p5.next;
19    c.next = null;
20    modifyParam1Indirectly(p6, doIt);
21 }
22
23 void modifyParam2Indirectly(C p7, C p8) {
24    modifyParam1(p8, true);
25 }
26 }
```

Figure 1. Example code that illustrates our staged approach to parameter immutability. All non-primitive parameters other than `p7` are *mutable*.

fields (e.g., `int`, `float`) and the states of all objects pointed to by *o*'s non-primitive fields. The mutation may occur in *m* itself or in any method that *m* (transitively) calls.

Reference immutability may be combined with aliasing information at each call site to determine whether a specific object passed as a parameter may change [6, 48]. If the object is unreachable from any *mutable* parameter, then the call will not change it.

2.1 Example

In the code in Figure 1, parameter `p7` is reference-*immutable*, and all non-`boolean` parameters other than `p7` are reference-*mutable*, because there exists an execution of their declaring method such that the object pointed to by the parameter reference is modified *via the reference*.

***immutable* parameters**:

- `p7` is reference-*immutable*. An object passed to `p7` might be mutated by an execution of method `modifyParam2Indirectly`; for example in a call where `p7` and `p8` are aliased, therefore `p7` is object-*mutable*. However, no execution of the method can cause a mutation via parameter `p7`, hence parameter `p7` is reference-*immutable*.

***mutable* parameters**:

- `p1` may be directly modified in `modifyParam1` (line 8).
- `p2` is passed to `modifyParam1`, in which it may be mutated.
- `p3` is *mutable* because the state of the object passed to `p3` can get modified on line 19 via `p3`. This can happen because `p4` and `p5` might be aliased; for example, in the call `modifyAll(x1, x2, x2, x3, false)`. In this case, the reference to `p3` is copied into `c` and then used to perform a modification on line 19.
- `p4` is directly modified in `modifyAll` (line 17). Note that line 17 does *not* modify `p3`, `p5`, or `p6` because the mutation occurs via reference `p4`. In this paper, we are concerned with *reference*-(im)mutability rather than *object-(im)mutability* and thus the reference via which the modification happens is significant.
- `p5` is *mutable* because line 19 modifies `p5.next.next`.
- `p6` is passed to `modifyParam1Indirectly`, in which it may be mutated.
- `p8` is passed to `modifyParam1`, in which it may be mutated.

Our dynamic and static analyses complement each other to classify parameters into *mutable* and *immutable*, in the following steps:

1. Initially, all parameters are *unknown*.
2. A mostly flow-insensitive, intra-procedural static analysis classifies `p1`, `p4` and `p5` as *mutable*. The analysis classifies `p7` as *immutable*—there is no direct mutation in the method and the parameter is not used in a method call.
3. An inter-procedural static analysis propagates the current classification along the call-graph and classifies `p2`, `p6`, and `p8` as *mutable*.
4. Dynamic analysis needs an example execution in order to classify parameters. If the following main method

```
1 void main() {
2   modifyAll(x1, x2, x2, x3, false);
3 }
```

is supplied, the dynamic analysis will classify `p3` as *mutable* (the other parameters are left *unknown*).

Our staged analysis correctly classifies all parameters in Figure 1. However, this example poses difficulties for purely static or purely dynamic techniques. On the one hand, static techniques have difficulties correctly classifying `p3`. This is because, to avoid over-conservativeness, static analyses often assume that on entry to a method all parameters are fully un-aliased, i.e., point to disjoint parts of the heap. In our example, this assumption may lead such analyses to incorrectly classify `p3` as *immutable* (in fact, Sălcianu uses a similar example to illustrate the unsoundness of his analysis [47, p.78]). On the other hand, dynamic analyses are limited to a specific execution and only consider modifications that happen during that execution. In our example, a purely dynamic technique may incorrectly classify `p2` and `p6` as *immutable* because during the execution of `main`, those parameters are not modified.

3. Staged Mutability Analysis

In our approach, mutability analyses are combined in stages, forming a "pipeline". The input to the first stage is the initial classification of all parameters (typically, all *unknown*, though parameters declared in the standard libraries may

be pre-classified). Each stage of the pipeline refines the results computed by the previous stage by classifying some *unknown* parameters. Once a parameter is classified as *immutable* or *immutable*, further stages do not change the classification. The output of the last stage is the final classification, in which some parameters may remain *unknown*.

Combining mutability analyses can yield an analysis that has better accuracy than any of the components. For example, a static analysis can cover the whole program code, while a dynamic analysis can conclusively prove the presence of a mutation.

Combining analyses in pipelines also has performance benefits—a component analysis in a pipeline may ignore code for which all parameters have been classified as *mutable* or *immutable*. This can permit the use of techniques that would be too computationally expensive if applied to an entire program.

The problem of mutability inference is undecidable, so no analysis can be both sound and complete. An analysis is *i-sound* if it never classifies a *mutable* parameter as *immutable*. An analysis is *m-sound* if it never classifies an *immutable* parameter as *mutable*. An analysis is *complete* if it classifies every parameter as either *mutable* or *immutable*.

In our staged approach, analyses may explicitly represent their incompleteness using the *unknown* classification. Thus, an analysis result classifies parameters into three groups: *mutable*, *immutable*, and *unknown*. Previous work that used only two output classifications [44, 42] loses information by conflating parameters/methods that are known to be mutable with those where analysis approximations prevent definitive classification.

Different clients of mutability analyses have different requirements. For example, using immutability for compiler optimizations requires an i-sound analyses, while using immutability in test generation benefits from more complete classification and can tolerate some classification mistakes. To address the needs of different mutability analysis contexts, the analyses presented in this paper can be combined in pipelines with different properties. For example, the client of the analysis can create an i-sound analysis by combining only i-sound components (all of our analyses have i-sound variations), while clients who desire more complete analyses may use i-unsound components as well.

4. Dynamic Mutability Analysis

Our dynamic mutability analysis observes the program's execution and classifies as *mutable* those method parameters that are used to mutate objects. The algorithm is m-sound: it classifies a parameter as *mutable* only when the parameter is mutated. The algorithm is also i-sound: it classifies all remaining parameters as *unknown*. Section 4.1 gives the idea behind the algorithm, and Section 4.2 describes an optimized implementation.

To improve the analysis results, we developed several heuristics (Section 4.3). Each heuristic carries a different risk of unsoundness. However, most are shown to be accurate in our experiments. The analysis has an iterative variation with random input generation (Section 4.4) that improves analysis precision and run-time.

4.1 Conceptual Algorithm

During program execution, the dynamic analysis tags each reference in the running program with the set of all formal parameters (from any method invocation on the call stack) whose fields were directly or indirectly accessed to obtain the reference. When a reference is side-effected (i.e., used as right-hand-side in a field-write), all formal parameters in its set are classified as mutable. The analysis tags references, not objects, because more than one reference can point to the same object. Primitives need not be tagged, as they are immutable.

The algorithm for detecting mutable parameters is given by a set of data-flow rules. The rules track mutations to each parameter. Next, we present those rules informally. The rules are formalized in Appendix A.

1. On method entry, the algorithm adds each formal parameter (that is classified as *unknown*) to the parameter set of the corresponding actual parameter reference.
2. On method exit, the algorithm removes all parameters for the current invocation from the parameter sets of all references in the program.
3. Assignments, including pseudo-assignments for parameter passing and return values, propagate the parameter sets unchanged.
4. Field accesses also propagate the sets unchanged: the set of parameters for `x.f` is the same as that of `x`.
5. For a field write `x.f = v`, the algorithm classifies as *mutable* all parameters in the parameter set of `x`.

The algorithm as presented so far has a significant runtime cost—maintaining reference tag sets for all references is computationally expensive. The next section presents an alternative algorithm that we implemented.

4.2 Dynamic Analysis Algorithm

To overcome the performance problem of the algorithm in Section 4.1, we developed an alternative algorithm that does not maintain parameter reference tags and is, nevertheless, i-sound and m-sound. The alternative algorithm is, however, less complete—it classifies fewer parameters. In the alternative algorithm, parameter *p* of method *m* is classified as *mutable* if: (i) the transitive state of the object that *p* points to changes during the execution of *m*, and (ii) *p* is not aliased to any other parameter of *m*. Part (ii) is critical for maintaining m-soundness of the algorithm—without part (ii), parameters may be wrongly classified as *mutable* when they are aliased to a *mutable* parameter during the execution (but are not, themselves, *mutable*).

The example code in Figure 1 illustrates the difference between the conceptual algorithm presented in Section 4.1 and the algorithm presented in this section. When method `main` executes, it calls `modifyAll`. The conceptual algorithm based on the definition (correctly) classifies all non-`boolean` parameters (except `p7` and `p8`) as *mutable*. The alternative algorithm leaves `p4` and `p5` as *unknown*—when the modification to the referent object happens (line 17), parameters `p4` and `p5` are aliased. Note that the inter-procedural static analysis (Section 5.1) compensates for the incompleteness of the dynamic analysis in this case and correctly classifies `p4` and `p5` as *mutable*.

The algorithm permits an efficient implementation: when method *m* is called during the program's execution, the analysis computes the set *reach*(*m*, *p*) of objects that are transitively reachable from each parameter *p* via field references. When the program writes to a field in object *o*, the analysis finds all parameters *p* of methods that are currently on the call stack. For each such parameter *p*, if $o \in reach(m, p)$ and *p* is not aliased to other parameters of *m*, then the analysis classifies *p* as *mutable*. The algorithm checks aliasing by verifying emptiness of intersection of reachable subheaps (ignoring immutable objects, such as boxed primitives, which may be shared).

The implementation of the dynamic analysis is straightforward: the analyzed code is executed and instrumented at load-time. The analysis works online, i.e., in tandem with the target program, without creating a trace file. Our implementation includes the following three optimizations, which together improve the run time by over 30×: (a) the analysis determines object reachability by maintaining and traversing its own data structure that mirrors the heap; this is faster than using reflection, (b) the analysis computes the set of reachable objects lazily, when a modification occurs, and (c) the analysis caches the set of objects transitively reachable from every object, invalidating it when one of the objects in the set is modified.

4.3 Dynamic Analysis Heuristics

The dynamic analysis algorithm described in Sections 4.1 and 4.2 is m-sound—a parameter is classified as *mutable* only if it is modified during execution. The recall (see Section 6) of the algorithm can be greatly improved by using heuristics. The heuristics allow the algorithm to take advantage of the *absence* of parameter modifications and of the classification results computed by previous stages in the analysis pipeline.

Using the heuristics may potentially introduce i-unsoundness or m-unsoundness to the analysis results, but in practice, they cause few misclassifications (see Section 6.4.5). We developed the following heuristics:

(**A**) **Classifying parameters as *immutable* at the end of the analysis.** This heuristic classifies as *immutable* all (*unknown*) parameters that satisfy conditions that are set by the client of the analysis. In our framework, the heuristic classifies as *immutable* a parameter *p* declared in method *m* if *p* was not modified, *m* was executed at least *N* times, and the executions achieved block coverage of at least *t*%. Higher values of the threshold *N* or *t* increase i-soundness but decrease completeness; see Section 6.4.5.

The intuition behind this heuristic is that, if a method executed multiple times, and the executions covered a large part of the method, and the parameter was not modified during any of those executions, then the parameter may in fact be *immutable*. This heuristic is m-sound but i-unsound. In our experiments, this heuristic greatly improved recall and was not a significant source of mistakes.

(**B**) **Using current mutability classification.** This heuristic classifies a parameter as *mutable* if the object to which the parameter points is passed in a method invocation to a formal parameter that is already classified as *mutable* (by a previous or the current analysis). That is, the heuristic does not wait for the actual modification of the object but assumes that the object will be modified if it is passed to a *mutable* position. The heuristic enables not tracking the object in the new method invocation, which improves analysis performance.

The intuition behind this heuristic is that if an object is passed as an argument to a parameter that is known to be *mutable*, then it is likely that the object will be modified during the call. The heuristic is i-sound but m-unsound. In our experiments, this heuristic improved recall and run time of the analysis and caused few misclassifications.

(**C**) **Classifying aliased mutated parameters.** This heuristic classifies a parameter *p* as *mutable* if the object that *p* points to is modified, regardless of whether the modification happened through an alias to *p* or through the reference *p* itself. For example, if parameters *a* and *b* happen to point to the same object *o*, and *o* is modified, then this heuristic will classify both *a* and *b* as *mutable*, even if it the modification is only done using the formal parameter's reference to *a*.

The heuristic is i-sound but m-unsound. In our experiments, using this heuristic improved the results in terms of recall, without causing any misclassifications.

4.4 Using Randomly Generated Inputs

The dynamic mutability analysis requires an example execution. Random generation [36] of method calls can complement (or even replace) an execution provided by a user, for instance by increasing coverage.

Using only randomly generated execution has benefits for a dynamic analysis. First, the analysis that uses random executions may able to explore parts of the program that the user-supplied execution may not reach. Second, the analysis becomes fully-automated and requires only the program's code—the user need not provide a representative execution. Third, each of the generated random inputs may be executed immediately—this allows the client of the analysis to stop generating inputs when the client is satisfied with the results of the analysis computed so far. Forth, the client of the

analysis may direct the input generator towards methods for which the results are incomplete. In contrast, by using a user-provided execution, the client does not have such a fine-grained control.

Our generator gives a higher selection probability to methods with *unknown* parameters and methods that have not yet been executed by other dynamic analyses in the pipeline. By default, the number of generated method calls is $max(5000, \#methodsInProgram)$.

Generation of random inputs is iterative. After the dynamic analysis has classified some parameters, it makes sense to propagate that information (see Section 5.3) and to re-focus random input generation on the remaining *unknown* parameters. Such re-focusing iterations continue as long as at least 1% of the remaining *unknown* parameters are classified (the threshold is user-settable).

5. Static Mutability Analysis

This section describes a simple, scalable static mutability analysis. It consists of two phases: S, an intraprocedural analysis that classifies as (*im*)*mutable* parameters (never) affected by field writes within the procedure itself (Section 5.2), and P, an interprocedural analysis that propagates mutability information between method parameters (Section 5.3). P may be executed at any point in an analysis pipeline after S has been run, and may be run multiple times (interleaving with other analyses). S and P both rely on a coarse intraprocedural pointer analysis that calculates the parameters pointed to by each local variable (Section 5.1).

5.1 Intraprocedural Points-To Analysis

The analysis must determine which parameters can be pointed to by each expression (without loss of generality, we assume three-address SSA form and consider only local variables). We use a coarse, scalable, intraprocedural, flow-insensitive, 1-level field-sensitive, points-to analysis.

The points-to analysis calculates, for each local variable l, a set $P_0(l)$ of parameters whose state l can point to directly and a set $P(l)$ of parameters whose state l can point to directly or transitively. The points-to analysis has "overestimate" and "underestimate" varieties; they differ in how method calls are treated (see below).

For each local variable l and parameter p, the analysis calculates a distance map $D(l, p)$ from the fields of object l to a non-negative integer or ∞. $D(l, p)(f)$ represents the number of dereferences that can be applied to l starting with a dereference of the field f to find an object pointed to (possibly indirectly) by p. Each map $D(l, p)$ is either strictly positive everywhere or is zero everywhere. Suppose l directly references p or some object transitively pointed to by p; then $D(l, p)(f) = 0$ for all f. As another example, suppose $l.f.g.h = p.x$; then $D(l, p)(f) = 3$. The distance map D makes the analysis field-sensitive, but only at the first layer of dereferencing; we found that this provided satisfactory results.

The points-to analysis computes $D(l, p)$ via a fixpoint computation on each method. At the beginning of the computation, $D(p, p)(f) = 0$, and $D(l, p)(f) = \infty$ for all other l and p. Due to space constraints, we give the flavor of the dataflow rules with a few examples:

- A field dereference $l_1 = l_2.f$ updates

$$\begin{aligned} \forall g : D(l_1, p)(g) &\leftarrow \min(D(l_1, p)(g), D(l_2, p)(f) - 1) \\ D(l_2, p)(f) &\leftarrow \min(D(l_2, p)(f), \min_g D(l_1, p)(g) + 1) \end{aligned}$$

- A field assignment $l_1.f = l_2$ updates

$$\begin{aligned} D(l_1, p)(f) &\leftarrow \min(D(l_1, p)(f), \min_g D(l_2, p)(g) + 1) \\ \forall g : D(l_2, p)(g) &\leftarrow \min(D(l_2, p)(g), D(l_1, p)(f) - 1) \end{aligned}$$

- Method calls are handled either by assuming they create no aliasing (creating an underestimate of the true points-to sets) or by assuming they might alias all of their parameters together (for an overestimate). If an underestimate is desired, no values of $D(l, p)(f)$ are updated. For an overestimate, let S be the set of all locals used in the statement (including receiver and return value); for each $l \in S$ and each parameter p, set $D(l, p)(f) \leftarrow \min_{l' \in S, g} D(l', p)(g)$.

After the computation reaches a fixpoint, it sets

$$P(l) = \{p \mid \exists f : D(l, p)(f) \neq \infty\}$$
$$P_0(l) = \{p \mid \forall f : D(l, p)(f) = 0\}$$

5.2 Intraprocedural Phase: S

The intraprocedural phase first calculates the "overestimate" points-to analysis described in Section 5.1.

The analysis marks as *mutable* some parameters that are currently marked as *unknown*: For each mutation $l_1.f = l_2$, the analysis marks all elements of $P_0(l_1)$ as *mutable*. Because of infeasible paths, and because its pointer analysis is an overestimate, S is not m-sound.

Next, S marks as *immutable* some parameters that are currently *unknown*. The analysis computes a "leaked set" L of locals, consisting of all arguments (including receivers) in all method invocations and any local assigned to a static field (in a statement of the form `Global.field = local`). The analysis then marks as *immutable* all *unknown* parameters that are not in the set $\cup_{l \in L} P(l)$.

S never marks any parameter as *immutable* if the parameter can be referred to in a mutation or escape to another method body. However, S as presented so far is not i-sound, because of it does not account for all possible aliasing relationships; for an example (that is also misclassified by J), see Figure 10.

We correct this problem with a sound version, denoted SS, that is just like S except that it does not classify any parameter as *immutable* unless it can classify all parameters as *immutable*.

5.3 Interprocedural Propagation Phase: P

The interprocedural propagation phase refines the current parameter classification by propagating both mutability and immutability information through the call graph. Given an i-sound input classification and a precise call-graph, propagation is i-sound.

Because propagation ignores the bodies of methods, the P phase is sound only if the method bodies have already been analyzed. It is intended to be run only after the S phase of Section 5.1 has already be run. However, it can be run multiple times (with other analyses in between).

5.3.1 Binding Multi-Graph

The propagation uses a variant (that accounts for pointer data structures) of the *binding multi-graph* (BMG) [12]. Each node is a method parameter `m.p`. An edge from `m1.p1` to `m2.p2` exists iff `m1` calls `m2`, passing as position `p2` part of `p1`'s state (either `p1` or an object that may be transitively pointed-to by `p1`).

We create a BMG by generating a call-graph and translating each method call edge into a set of parameter dependency edges, using the sets $P(l)$ described in Section 5.1 to tell which parameters correspond to which locals.

Our algorithm is parameterized by a call-graph construction algorithm. Our experiments used CHA [14]—the simplest and least precise call-graph construction algorithm offered by Soot. In the future, we want to investigate using more precise but still scalable algorithms, such as RTA [3] (available in Soot, but containing bugs that prevented us from using it), or those proposed by Tip and Palsberg [53] (not implemented in Soot).

The true BMG is not computable, because determining perfect aliasing and call information is undecidable. Our analysis uses an under-approximation (i.e., it contains a subset of edges of the ideal graph) and an over-approximation (i.e., it contains a superset of edges of the ideal graph) to the BMG as safe approximations for determining mutable and immutable parameters, respectively. As the over-approximated BMG, our implementation uses the *fully-aliased* BMG, which is created with an overestimating points-to analysis which assumes that method calls introduce aliasings between *all* parameters. As the under-approximated BMG, our implementation uses the *un-aliased* BMG, which is created with an underestimating points-to analysis which assumes that method calls introduce *no* aliasings between parameters. More precise approximations could be computed by a more complex points-to analysis.

To construct the under-approximation of the true BMG, propagation needs a call-graph that is an under-approximation of the real call-graph. However, most existing call-graph construction algorithms [14, 16, 3, 53] create an over-approximation. Therefore, our implementation uses the same call-graph for building the un- and fully-aliased BMGs. In our experiments, this never caused misclassification of parameters as *immutable*, and only a minimal number of parameters misclassified as *mutable*.

program	size (LOC)	classes	parameters		
			all	non-trivial	inspected
jolden	6,215	56	705	470	470
sat4j	15,081	122	1,499	1,136	118
tinysql	32,149	119	2,408	1,708	206
htmlparser	64,019	158	2,270	1,738	82
ejc	107,371	320	9,641	7,936	7,936
daikon	185,267	842	16,781	13,319	73
Total	410,102	1,617	33,304	26,307	8,885

Figure 2. Subject programs.

5.3.2 Propagation Algorithm

Propagation refines the parameter classification in 2 phases.

The **mutability propagation** classifies as *mutable* all the *unknown* parameters that can reach in the under-approximated BMG (can flow to in the program) a parameter that is classified as *mutable*. Using an over-approximation to the BMG would be unsound because spurious edges may lead propagation to incorrectly classify parameters as mutable.

The **immutability propagation** phase classifies additional parameters as *immutable*. This phase uses a fix point computation: in each step, the analysis classifies as *immutable* all *unknown* parameters that have no *mutable* or *unknown* successors (callees) in the over-approximated BMG. Using an under-approximation to the BMG would be unsound because if an edge is missing in the BMG, the analysis may classify a parameter as *immutable* even though the parameter is really mutable. This is because the parameter may be missing, in the BMG, a *mutable* successor.

6. Evaluation

We experimentally evaluated 192 combinations of mutability analyses, comparing the results with each other and with a manually computed (and inspected) correct classification of parameters. Our results indicate that staged mutability analysis can be accurate, scalable, and useful.

6.1 Methodology and Measurements

We computed mutability for 6 open-source subject programs (see Figure 2). When an example input was needed (e.g., for a dynamic analysis), we ran each subject program on a single input.

- **jolden**[1] is a benchmark suite of 10 small programs. As the example input, we used the `main` method and arguments that were included with the benchmarks. We included these programs primarily to permit comparison with Sălcianu's evaluation [48].

[1] `http://www-ali.cs.umass.edu/DaCapo/benchmarks.html`

- **sat4j**[2] is a SAT solver. We used a file with an unsatisfiable formula as the example input.
- **tinysql**[3] is a minimal SQL engine. We used the program's test suite as the example input.
- **htmlparser**[4] is a real-time parser for HTML. We used our research group's webpage as the example input.
- **ejc**[5] is the Eclipse Java compiler. We used one Java file as the example input.
- **daikon**[6] is an invariant detector. We used the StackAr test case from its distribution as the example input.

As the input to the first analysis in the pipeline, we used a pre-computed (manually created) classification for all parameters in the standard Java libraries. Callbacks from the library code to the client code (e.g., `toString()`, `hashCode()`) were analyzed under the closed world assumption in which all of the subject programs were included. The pre-computed classification was created once, and reused many times in all the experiments. Another benefit of using this classification is that it covers otherwise un-analyzable code, such as native calls.

We measured the results only for non-trivial parameters declared in the application. That is, we ignored parameters declared in external or JDK libraries, and ignored all parameters with a primitive, boxed primitive, or `String` type.

To measure the accuracy of each mutability analysis, we determined the correct classification (*mutable* or *immutable*) for 8,885 parameters: all of jolden and ejc, and 5 randomly-selected classes from each of the other programs. To find the correct classification, we first ran every tool available to us (including our analysis pipelines, Sălcianu's tool, and the Javarifier [55] type inference tool for Javari). Then, we manually determined the correct classification for every parameter where any two tool results differed, or where only one tool completed successfully.

Figure 3 and the tables in Section 6.4 present precision and recall results, computed as follows:

$$\begin{aligned} \text{i-precision} &= \frac{ii}{ii+im} \\ \text{i-recall} &= \frac{ii}{ii+ui+mi} \\ \text{m-precision} &= \frac{mm}{mm+mi} \\ \text{m-recall} &= \frac{mm}{mm+um+im} \end{aligned}$$

where *ii* is the number of immutable parameters that are correctly classified, and *mi* is the number of immutable parameters incorrectly classified as *mutable* (similarly, *ui*). Similarly, for mutable parameters, we have *mm*, *im* and *um*. i-precision is measure of soundness: it counts how often the analysis is correct when it classifies a parameter as *immutable*. i-recall is measure of completeness: it counts how many immutable parameters are marked as such by the analysis. m-precision and m-recall are similarly defined. An i-sound analysis has i-precision of 1.0, and an m-sound analysis has m-precision of 1.0. Ideally, both precision and recall should be 1.0, but this is not feasible: there is always a trade-off between analysis precision and recall.

[2] `http://www.sat4j.org/`

[3] `http://sourceforge.net/projects/tinysql`

[4] `http://htmlparser.sourceforge.net/`

[5] `http://www.eclipse.org/`

[6] `http://pag.csail.mit.edu/daikon/`

6.2 Evaluated Analyses

Our experiments evaluate pipeline analyses composed of analyses described in Section 3. X-Y-Z denotes a staged analysis in which component analysis X is followed by component analysis Y and then by component analysis Z.

Our experiments use the following component analyses:

- S is the intraprocedural static analysis (Section 5.2).
- SS is the sound intraprocedural static analysis (Section 5.2).
- P is the interprocedural static propagation (Section 5.3).
- D is the dynamic analysis (Section 4).
- DH is D, augmented with all the heuristics described in Section 4.3. DA, DB, and DC are D, augmented with just one of the heuristics.
- DRH is DH enhanced with random input generation (Section 4.4); likewise for DRA, etc.
- J is Sălcianu and Rinard's state-of-the-art static analysis JPPA [48]. It never classifies parameters as *mutable*—only *immutable* and *unknown*.
- JMH is J, augmented in two ways. First, JMH uses a `main` method that contains calls to all the public methods in the subject program [42]; J only analyzes methods that are reachable from `main`, limiting its code coverage and thus recall. Second, JMH heuristically classifies as *mutable* any parameter for which J provides an explanation of a potential modification; J has m-precision and m-recall of 0.

6.3 Results

Figure 3 compares the accuracy of a selected set of mutability analyses with which we experimented.

Different analyses are appropriate in different situations, but the pipeline with the highest overall precision and recall was S-P-DRH-P. It dominates Sălcianu's [48] state-of-the-art analysis, J. For every subject program, the staged mutability analysis, combined of static and dynamic phases, achieves equal i-precision and better i-recall, and much better m-recall and m-precision, because J never classifies parameters as *mutable*. The staged analysis is also considerably more scalable.

In certain applications, i-soundness is a critical property. We evaluated i-sound versions of our analyses (see Section 6.4.6), and Figure 3 shows the results for SS-P-DBC-P, the best-performing i-sound staged analysis.

Prog.	Analysis	i-recall	i-precision	m-recall	m-precision
ejc	J	0.593	0.999	0.000	0.000
	JMH	0.734	0.998	0.691	0.941
	JMH-S-P-DRH-P	0.939	0.997	0.944	0.951
	S-P	0.777	1.000	0.904	0.971
	S-P-DRH-P	0.928	0.996	0.907	0.971
	SS-P-DRBC-P	0.781	1.000	0.915	0.956
jolden	J	0.894	1.000	0.000	0.000
	JMH	0.985	1.000	0.660	0.955
	JMH-S-P-DRH-P	0.989	0.996	0.990	0.970
	S-P	0.829	1.000	0.907	1.000
	S-P-DRH-P	0.973	1.000	1.000	0.970
	SS-P-DRBC-P	0.829	1.000	1.000	0.924
daikon	J	0.750	1.000	0.000	0.000
	JMH	-	-	-	-
	JMH-S-P-DRH-P	-	-	-	-
	S-P	0.636	1.000	0.931	0.844
	S-P-DRH-P	0.750	1.000	0.931	0.844
	SS-P-DRBC-P	0.705	1.000	0.931	0.844
tiny+sat+html	J	-	-	-	-
	JMH	-	-	-	-
	JMH-S-P-DRH-P	-	-	-	-
	S-P	0.836	1.000	0.863	0.965
	S-P-DRH-P	0.968	0.984	0.947	0.957
	SS-P-DRBC-P	0.836	1.000	0.863	0.953

Figure 3. Mutability analyses on subject programs. Subjects tinysql, sat4j and htmlparser are presented jointly as the last group, marked as tiny+sat+html. Empty cells mean that the analysis aborted with an error.

6.4 Discussion of Results

We experimented with six programs and 192 different analysis pipelines. This section discusses the important observations that stem from the results of our experiments. Each sub-section discusses one observation that is supported by a table listing representative pipelines illustrating the observation. The tables in this section present results for ejc. Results for other programs were similar. However, for smaller programs all analyses did better and the differences in results were not as pronounced.

6.4.1 Interprocedural Propagation

Running interprocedural propagation (P in the tables) is always beneficial, as the following table shows on representative pipelines.

Analysis	i-recall	i-precision	m-recall	m-precision
S	0.563	1.000	0.299	0.998
S-P	0.777	1.000	0.904	0.971
S-P-DRH	0.922	0.996	0.906	0.971
S-P-DRH-P	0.928	0.996	0.907	0.971
DRH	0.540	0.715	0.144	0.987
DRH-P	0.940	0.776	0.663	0.988

Propagation may decrease m-precision but in our experiments, the decrease was never larger than 0.03. In the experiments, propagation always increased all other statistics (sometimes significantly). For example, the table shows that propagation increased i-recall from 0.563 in S to 0.777 in S-P and it increased m-recall from 0.299 in S to 0.904 in S-P. Moreover, since almost all of the run-time cost of propagation lies in the call-graph construction, only the first execution incurs notable run-time cost on the analysis pipeline; subsequent executions of propagation are fast. Therefore, most pipelines presented in the sequel have P stages executed after every other analysis.

6.4.2 Combining Static and Dynamic Analysis

Combining static and dynamic analysis in either order is helpful—the two types of analysis are complementary.

Analysis	i-recall	i-precision	m-recall	m-precision
S-P	0.777	1.000	0.904	0.971
S-P-DRH	0.922	0.996	0.906	0.971
S-P-DRH-S-P	0.928	0.996	0.907	0.971
DRH	0.540	0.715	0.144	0.987
DRH-S-P	0.939	0.812	0.722	0.981
DRH-S-P-DRH	0.943	0.813	0.722	0.981

For best results, the static stage should precede the dynamic stage. Pipeline S-P-DRH, in which the static stage precedes the dynamic stage, achieved significantly better i-precision and m-recall than DRH-S-P, with only marginally lower i-recall and m-precision.

Repeating executions of static or dynamic analyses bring no substantial further improvement. For example, S-P-DRH-S-P (i.e., static-dynamic-static) achieves the same results as S-P-DRH (i.e., static-dynamic). Similarly, DRH-S-P-DRH (i.e., dynamic-static-dynamic) only marginally improves i-recall over DRH-S-P (i.e., dynamic-static).

6.4.3 Comparing Static Stages

In a staged mutability analysis, using a more complex static analysis does not bring much benefit. We experimented with replacing our lightweight interprocedural static analysis with J, Sălcianu's heavyweight static analysis.

Analysis	i-recall	i-precision	m-recall	m-precision
J-DRH-P	0.973	0.787	0.664	0.998
JMH-DRH-P	0.939	0.922	0.878	0.949
JMH-S-P-DRH-P	0.939	0.997	0.944	0.951
S-P-DRH-P	0.928	0.996	0.907	0.971

S-P-DRH-P outperforms JMH-DRH-P with respect to 3 of 4 statistics, including i-precision (see Section 6.4.6). Combining the two static analyses improves recall—JMH-S-P-DRH-P has better i-recall than S-P-DRH-P and better m-recall than JMH-DRH-P. This shows that the two kinds of static analysis are complementary.

6.4.4 Randomly Generated Inputs in Dynamic Analysis

Using randomly generated inputs to the dynamic analysis (DRH) achieves better results than using a user-supplied execution (DH). We also considered pipelines that use both types of executions.

Analysis	i-recall	i-precision	m-recall	m-precision
S-P-DH	0.827	0.984	0.911	0.961
S-P-DH-P-DRH	0.917	0.984	0.915	0.958
S-P-DRH	0.922	0.996	0.906	0.971
S-P-DRH-P-DH	0.932	0.983	0.912	0.970

Pipeline S-P-DRH achieves better results than S-P-DH with respect to i-precision, i-recall and m-precision (with marginally lower m-recall). Using both kinds of executions can have different effects. For instance, S-P-DH-P-DRH has better results than S-P-DH, but S-P-DRH-P-DH has a lower i-precision with a small gain in i-recall and m-recall over S-P-DRH-P-DH.

The surprising finding that randomly generated code is as effective as using an example execution suggests that other dynamic analyses (e.g., race detection [49, 35], invariant detection [18], inference of abstract types [21], and heap type inference [39]) might also benefit from replacing example executions with random executions.

6.4.5 Dynamic Analysis Heuristics

By exhaustive evaluation, we determined that each of the heuristics is beneficial. A pipeline with DRH achieves notably higher i-recall and only slightly lower i-precision than a pipeline with DR (which uses no heuristics). This section indicates the unique contribution of each heuristic, by removing it from the full set (because some heuristics may have overlapping benefits). For consistency with other tables in this section, we present the results for ejc; however, the effects of heuristics were more pronounced on other benchmarks.

Heuristic **A** (evaluated by the DRBC line) has the greatest effect; removing this heuristic significantly lowers i-recall (as compared to S-P-DRH-P, which includes all heuristics.) However, because the heuristic is i-unsound, removing it increases i-precision, albeit only by 0.004. Heuristic **B** (the DRAC line) increases both i-recall and i-precision, and improves performance by 10%. Heuristic **C** (the DRAB line) is primarily a performance optimization. Including this heuristic results in a 30% performance improvement and a small increase to m-recall.

Analysis	i-recall	i-precision	m-recall	m-precision
S-P-DR-P	0.777	1.000	0.905	0.971
S-P-DRH-P	0.928	0.996	0.907	0.971
S-P-DRBC-P	0.777	1.000	0.906	0.971
S-P-DRAC-P	0.927	0.995	0.905	0.971
S-P-DRAB-P	0.928	0.996	0.906	0.971

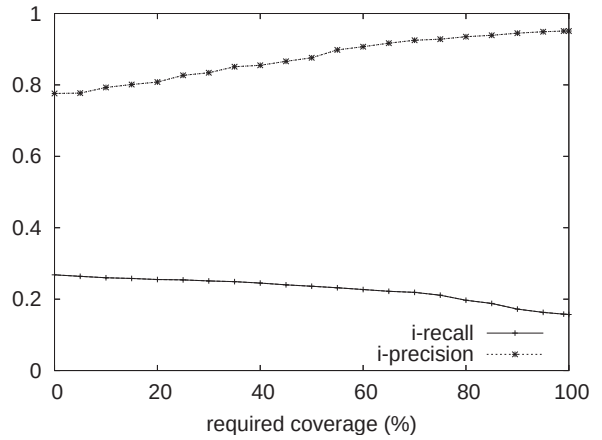

Figure 4. Relation between i-precision, i-recall and the coverage threshold in dynamic analysis heuristic **A**. The presented results are for the dynamic analysis D on the ejc subject program.

Heuristic **A** is parameterized by a coverage threshold *t*. Higher values of the threshold classify fewer parameters as *immutable*, increasing i-precision but decreasing i-recall. Figure 4 shows this relation on results for D on ejc (the dependency still exists, but is less pronounced, on other subjects and pipelines). The heuristic is m-sound, so it has no effect on m-precision. The threshold value may affect m-recall (if the analysis incorrectly classifies a *mutable* parameter), but, in our experiments, we have not observed this.

6.4.6 i-sound Analysis Pipelines

An i-sound mutability analysis never incorrectly classifies a parameter as *immutable*. All our component analyses have i-sound variations and composing i-sound analyses yields an i-sound staged analyses. We evaluated i-sound versions of the staged analyses

Analysis	i-recall	i-precision	m-recall	m-precision
SS	0.454	1.000	0.299	0.998
SS-P	0.777	1.000	0.904	0.971
SS-P-DRBC	0.777	1.000	0.906	0.971
SS-P-DBC	0.777	1.000	0.912	0.959

SS is the i-sound version of the intra-procedural static analysis S. Not surprisingly, the i-sound pipelines achieve lower i-recall than i-unsound pipelines presented in Figure 3 (which presents the results for SS-P-DRBC-P for all subjects). For clients for whom i-soundness is critical, this may be an acceptable trade-off. In contrast to our analyses, J is not i-sound [47], although it did achieve very high i-precision (see Figure 3).

6.5 Scalability

Figure 5 shows run times of analyses on daikon (185 kLOC, which is larger than previous evaluations of mutability analyses [44, 42, 48]). The experiments were run using a quad-

Analysis	total time (s)	last component (s)
J	5586	5586
JM	-	-
JMH	-	-
SS	167	167
SS-P	564	397
S	167	167
S-P	564	397
S-P-DH	859	295
S-P-DH-P	869	10
S-P-DRH	1484	920
S-P-DRH-P	1493	9

Figure 5. Run time, in seconds, of analyses on daikon, the largest of analyzed programs: both the cumulative time and the time for the last component analysis in the pipeline. Empty cells indicate that the analysis aborted with an error (JM denotes J executed on a `main` that includes calls to all public methods in the application).

core AMD Opteron 64-bit 4×1.8GHz machine with 4GB of RAM, running Debian Linux and Sun HotSpot 64-bit Server VM 1.5.0_09-b01. Staged mutability analysis scales to large code-bases and runs in about a quarter the time of Sălcianu's analysis (i.e., J in Figure 5). JMH, the augmented version of J, aborted with an error (the error was not due to the heuristic—JM also aborted with an error).

The figure overstates the cost of both the P and DRH stages, due to limitations of our implementation. First, the major cost of propagation (P) is computing the call graph, which can be reused later in the same pipeline. According to Sălcianu, J's RTA [3] call graph construction algorithm takes seconds, and our tool takes two orders of magnitude longer to perform CHA [14] (a less precise algorithm) using Soot [57]. Use of a more optimized implementation could greatly reduce the cost of propagation. Second, the DRH step iterates many times, each time performing load-time instrumentation and other tasks that could be cached; without this repeated work, DRH can be much faster than DH. These implementation fixes would save between 50% and 70% of the total S-P-DRH-P time.

However, the figure also overstates the cost of J; in the experiments, Sălcianu's analysis analyzed the whole JDK library on every execution, while our analysis was able to reuse a pre-computed analysis result.

Note that SS-P (Section 6.4.6) runs, on daikon, an order of magnitude faster than J (or even better, if differences in call graph construction are discounted). Moreover, SS-P is i-sound, while J is i-unsound. Finally, SS-P has high m-recall and m-precision, while J has 0 m-recall and m-precision.

6.6 Application: Test Input Generation

In addition to evaluating the accuracy of mutability analysis, we evaluated how much the computed immutability information helps a client analysis. We experimented with

analysis	nodes	ratio	edges	ratio	time (s)	ratio
jolden + ejc + daikon						
no immutability	444,729	1.00	624,767	1.00	6,703	1.00
J	131,425	3.83	210,354	2.97	4,626	1.44
S-P-DRH-P	124,601	3.57	201,327	3.10	4,271	1.56
htmlparser + tinysql + sat4j						
no immutability	48,529	1.00	68,402	1.00	215	1.00
J	-	-	-	-	-	-
S-P-DRH-P	8,254	5.88	13,047	5.24	90	2.38

Figure 6. Palulu [2] model size and model generation time, when assisted by immutability classifications. The numbers are sums over indicated subject programs. Smaller models are better. Also shown are improvement ratios over no immutability information (the "ratio" columns). Empty cells indicate that the analysis aborted with an error.

Palulu [2], a system that generates models for model-based testing. The model is a directed graph that describes permitted sequences of method calls. The model can be pruned (without changing the state space it describes) by removing calls that do not mutate specific parameters, because non-mutating calls are not useful in constructing new test inputs. A smaller model permits a systematic test generator to explore the state space more quickly, or a random test generator to explore more of the state space.

We ran Palulu on our subject programs using no immutability information, and using immutability information computed by J and by S-P-DRH-P. Figure 6 shows the number of nodes and edges in the generated model graph, and the time Palulu took to generate the model (not counting the immutability analysis). Mutability information permitted Palulu to run faster and to generate smaller models.

7. Related Work

Section 7.1 discusses previous work that discovers immutability (for example, determines when a parameter is never modified during execution). Section 7.2 discusses previous work that checks or enforces mutability annotations written by the programmer (or inserted by a tool).

7.1 Discovering Mutability

There is a rich history of research in analyzing programs to determine what mutations may occur. Early work [4, 12] considered pointer-free languages, such as Fortran. In such a language, aliases are induced only by reference parameter passing, and aliases persist until the procedure returns. MOD analysis determines which of the reference parameters, and which global variables, are assigned by the body of a procedure. Our analysis shares similar data structures and approach, but handles pointers and object-oriented programs and incorporates field-sensitivity, among other differences.

Subsequent research, often called side effect analysis, addressed aliasing in languages containing pointers. An update `r.f = v` has the potential to modify any object that might be

referred to by `r`. An alias analysis can determine the possible referents of pointers and thus the possible side effects. (An alias or class analysis also aids in call graph construction for object-oriented programs, by indicating the type of receivers and so disambiguating virtual calls.) This work indicates which which aliased locations might also be mutated — often reporting results in terms of the number of locations (typically, an allocation site in the program) that may be referenced [27] — but less often indicates what other variables in the program might also refer to that site. More relevantly, it does not answer reference immutability questions regarding what references might be used to perform a mutation; ours is the first analysis to do so. A follow-on alias or escape analysis can be used to strengthen reference immutability to object immutability [6].

New alias/class analyses lead to improved side effect analyses [46, 42]. Landi et al. [28] improve the precision of previous work by using program-point-specific aliasing information. Ryder et al. [46] compare the flow-sensitive algorithm [28] with a flow-insensitive one that yields a single alias result that is valid throughout the program. The flow-sensitive version is more precise but slower and unscalable, and the flow-insensitive version provides adequate precision for certain applications. Milanova et al. [33] provide a yet more precise algorithm via an object-sensitive, flow-insensitive points-to analysis that analyzes a method separately for each of the objects on which the method is invoked. Object sensitivity outperforms Andersen's context-insensitive analysis [1, 43]. Rountev [42] compares RTA to a context-sensitive points-to analysis for call graph construction; the latter found only one more side-effect-free method than the former, out of a total of 40. Rountev's experimental results suggest that sophisticated pointer analysis may not be necessary to achieve good results. (This mirrors other work questioning the usefulness of highly complex pointer analysis [45, 22].) We, too, compared a sophisticated analysis (Sălcianu's) to a simpler one (ours) and found the simpler one competitive.

Side-effect analysis [9, 44, 33, 42, 48, 47] originated in the compiler community and has focused on i-sound analyses. Our work investigates other tradeoffs and other uses for the immutability information. Specifically, differently from previous research, our work (1) computes both *mutable* and *immutable* classifications, (2) trades off soundness and precision to improve overall accuracy, (3) combines dynamic and static stages, (4) includes a novel dynamic mutability analysis, and (5) permits an analysis to explicitly represent its imprecision.

Preliminary results of using side effect analysis for optimization — an application that requires an i-sound analysis — show modest speedups. Le et al. [29] report speedups of 3–5% for a coarse CHA analysis, and only 1% more for a finer points-to analysis. Clausen [11] reports an average 4% speedup, using a CHA-like side effect analysis in which each field is marked as side-effected or not. Razamahefa [41] reports an average 6% speedup for loop invariant code motion in an inlining JIT Le et al. [29] summarize their own and related work as follows: "Although precision of the underlying analyses tends to have large effects on static counts of optimization opportunities, the effects on dynamic behavior are much smaller; even simple analyses provide most of the improvement."

The research that is most related to ours is that of Rountev [42] and Sălcianu [48, 47]. Both are static analyses for determining side-effect-free methods. Like ours and every practical mutability analyses of which we are aware, they combine a pointer analysis, a local (intra-procedural) analysis to determine "immediate" side effects, and inter-procedural propagation to determine transitive side effects.

Sălcianu defines a side-effect-free method as one that does not modify any heap cell that existed when the method was called. Rountev use a more restricted definition that prohibits a side-effect-free method from creating and returning a new object, or creating and using a temporary object. Sălcianu's analysis can compute per-parameter mutability information in addition to per-method side effect information. (A method is side-effect-free if it modifies neither its parameters nor the global state, which is an implicit parameter.) Rountev's coarser analysis results are one reason that we cannot compare directly to his implementation. Rountev applies his analysis to program fragments by creating an artificial `main` routine that calls all methods of interest; we adopted this approach in augmenting `J` (see Section 6).

Sălcianu's [48, 47] analysis uses a complex pointer analysis. Its flow-insensitive method summary represents in a special way objects allocated by the current method invocation, so a side-effect-free method may perform side effects on a newly-allocated objects. Like ours, Sălcianu's analysis handles code that it does not have access to, such as native methods, by using manually prepared annotations. Sălcianu describes an algorithm for computing object immutability and proves it sound, but his implementation computes reference immutability and contains some minor unsoundness. We evaluated our analyses, which also compute reference immutability, against Sălcianu's implementation (Section 5). In the experiments, our staged analyses achieve comparable or better accuracy and scaled better.

Work by Porat et al. [40, 5] infers class immutability for global (static) variables in Java's `rt.jar`, thus indicating the extent to which immutability can be found in practice; the work also addresses sealing/encapsulation.

7.2 Specifying and Checking Mutability

To specify and enforce immutability, a programming language is augmented to include tool-checked mutability annotations.

Type and effect systems [31, 24] allow specifying side-effects of functions. The Java Modeling Language (JML) [8] allows specifying pure methods (i.e., methods that have

no side effects on any of their parameters or the global state), but it has only weak support for checking these specifications. Other approaches that allow specifying immutability annotations are data groups [26, 30] and ownership types [10]. The Splint [19] tool statically checks user-provided type mutability annotations.

Language extensions that provide reference immutability by enhancing the type system include Islands [23], Flexible Alias Protection [34], C++ `const` [51], ModeJava [50], JAC [25], Capabilities [7], Javari [6, 56], Universes [15], and IGJ [60]. Most of those solutions aim to provide transitive reference-immutability (C++ `const` and Boyland's Capabilities are non-transitive). Appendix A contains, to the best of our knowledge, the first formal definition of transitive reference-immutability.

To compare our immutability approach with that of Javari, a reference-immutability extension to Java, we used a pre-release of the Javarifier type inference tool [55] to annotate the jolden programs with Javari annotations. Our analysis agreed with Javari-annotated code on 97.4% of parameters. The other 2.6% reflect differences in the immutability definitions between the two approaches. For example, Javari's type system requires that method overriding preserve immutability of the receiver parameter, while our definition (and Sălcianu's [48]) allows this immutability to vary between the overriding and overridden method. In this case, our definition is more expressive. In another example, Javari's definition is more expressive: Javari allows immutable arrays of mutable elements, while our definition requires transitive immutability and treats array elements as fields.

Object immutability is a stronger property than reference immutability: it guarantees that a particular value is never modified, even through aliased parameters. Reference mutability, together with an alias or escape analysis, is enough to establish object immutability [6]. Pechtchanski [37] allows the user to annotate his code with object immutability annotation and employs a combination of static and dynamic analysis to detect where those annotations are violated. The IGJ language [60] supports both reference and object immutability via a type system based on Java generics.

8. Conclusion

We have described a staged mutability analysis framework for Java, along with a set of component analyses that can be plugged into the analysis. The framework permits combinations of mutability analyses, including static and dynamic techniques. The framework explicitly represents analysis imprecision, and this makes it possible to compute both immutable and mutable parameters. Our component analyses take advantage of this feature of the framework.

Our dynamic analysis is novel, to the best of our knowledge; at run time, it marks parameters as mutable based on mutations of objects. We presented a series of heuristics, optimizations, and enhancements that make it practical. For example, iterative random test input generation appears competitive with user-supplied sample executions. Our static analysis reports both *immutable* and *mutable* parameters, and it demonstrates that a simple, scalable analysis can perform at a par with much more heavyweight and sophisticated static analyses. Combining the lightweight static and dynamic analyses yields a combined analysis with many of the positive features of both, including both scalability and accuracy.

Our evaluation includes many different combinations of staged analysis, in both sound and unsound varieties. This evaluation sheds insight into both the complexity of the problem and the sorts of analyses that can be effectively applied to it. We also show how the results of the mutability analysis can improve a client analysis.

References

[1] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

[2] Shay Artzi, Michael D. Ernst, Adam Kieżun, Carlos Pacheco, and Jeff H. Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *M-TOOS*, October 2006.

[3] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA*, pages 324–341, October 1996.

[4] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *POPL*, pages 29–41, January 1979.

[5] Marina Biberstein, Joseph Gil, and Sara Porat. Sealing, encapsulation, and mutability. In *ECOOP*, pages 28–52, June 2001.

[6] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *OOPSLA*, pages 35–49, October 2004.

[7] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP*, pages 2–27, June 2001.

[8] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, June 2005.

[9] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, pages 232–245, January 1993.

[10] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, pages 292–310, October 2002.

[11] Lars R. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045, 1997.

[12] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI*, pages 57–66, June 1988.

[13] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with ADABU. In *WODA*, pages 17–24, May 2006.

[14] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP*, pages 77–101, August 1995.

[15] Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

[16] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *OOPSLA*, pages 292–305, October 1996.

[17] José Javier Dolado, Mark Harman, Mari Carmen Otero, and Lin Hu. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE TSE*, 29(7):665–670, July 2003.

[18] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, February 2001.

[19] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Softw.*, 19(1):42–51, 2002.

[20] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[21] Philip Jia Guo. A scalable mixed-level approach to dynamic analysis of C and C++ programs. Master's thesis, MIT Dept. of EECS, May 5, 2006.

[22] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, pages 54–61, June 2001.

[23] John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA*, pages 271–285, October 1991.

[24] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *POPL*, pages 303–310, January 1991.

[25] Günter Kniesel and Dirk Theisen. JAC — access right based encapsulation for Java. *Software: Practice and Experience*, 31(6):555–576, 2001.

[26] Viktor Kuncak and Rustan Leino. In-place refinement for effect checking. In *Workshop on Automated Verification of Infinite-State Systems*, April 2003.

[27] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *PLDI*, pages 235–248, San Francisco, Calif., June 1992.

[28] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *PLDI*, pages 56–67, June 1993.

[29] Anatole Le, Ondřej Lhoták, and Laurie Hendren. Using interprocedural side-effect information in JIT optimizations. In *Compiler Construction 2005*, pages 287–304, April 2005.

[30] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *PLDI*, pages 246–257, June 2002.

[31] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *POPL*, pages 47–57, January 1988.

[32] Leonardo Mariani and Mauro Pezzè. Behavior capture and test: Automated analysis of component integration. In *ICECCS*, pages 292–301, June 2005.

[33] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ISSTA*, pages 1–11, July 2002.

[34] James Noble, John Potter, David Holmes, and Jan Vitek. Flexible alias protection. In *ECOOP*, pages 158–185, July 1998.

[35] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *PPOPP*, pages 167–178, July 2003.

[36] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE*, May 2007.

[37] Igor Pechtchanski and Vivek Sarkar. Immutability specification and its applications. In *Java Grande*, pages 202–211, November 2002.

[38] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[39] Marina Polishchuk, Ben Liblit, and Chloë Schulze. Dynamic heap type inference for program understanding and debugging. In *POPL*, January 2007.

[40] Sara Porat, Marina Biberstein, Larry Koved, and Bilba Mendelson. Automatic detection of immutable fields in Java. In *CASCON*, November 2000.

[41] Chrislain Razamahefa. A study of side-effect analyses for Java. Master's thesis, School of Computer Science, McGill University, Montreal, December 1999.

[42] Atanas Rountev. Precise identification of side-effect-free methods in Java. In *ICSM*, pages 82–91, September 2004.

[43] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for java based on annotated constraints. In *OOPSLA*, pages 43–55, October 2001.

[44] Atanas Rountev and Barbara G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *CC*, pages 20–36, April 2001.

[45] Erik Ruf. Context-insensitive alias analysis reconsidered. In *PLDI*, pages 13–22, June 1995.

[46] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, and Rita Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM TOPLAS*, 23(2):105–186, March 2001.

[47] Alexandru Sălcianu. *Pointer analysis for Java programs: Novel techniques and applications*. PhD thesis, MIT Dept. of EECS, September 2006.

[48] Alexandru Sălcianu and Martin C. Rinard. Purity and side-effect analysis for Java programs. In *VMCAI*, pages 199–215, January 2005.

[49] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *SOSP*, pages 27–37, December 1997.

[50] Mats Skoglund and Tobias Wrigstad. A mode system for read-only references in Java. In *FTfJP*, June 2001.

[51] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Boston, MA, special edition, 2000.

[52] Frank Tip. A survey of program slicing techniques. Report CS-R9438, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1994.

[53] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA*, pages 281–293, October 2000.

[54] Oksana Tkachuk and Matthew B. Dwyer. Adapting side effects analysis for modular program model checking. In *ESEC/FSE*, pages 188–197, September 2003.
[55] Matthew S. Tschantz. Javari: Adding reference immutability to Java. Master's thesis, MIT Dept. of EECS, August 2006.
[56] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, October 2005.
[57] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java bytecode optimization framework. In *CASCON*, pages 125–135, November 1999.
[58] Mark Weiser. Program slicing. *IEEE TSE*, SE-10(4):352–357, July 1984.
[59] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *ECOOP*, pages 380–403, July 2006.
[60] Yoav Zibin, Alex Potanin, Shay Artzi, Adam Kieżun, and Michael D. Ernst. Object and reference immutability using Java generics. Technical Report MIT-CSAIL-TR-2007-018, MIT CSAIL, March 16, 2007.

A. Parameter Mutability Definition

A formal definition of parameter reference mutability is a pre-requisite to verifying an algorithm or tool. Previous research [56, 60, 25, 34, 50, 15] defines reference-mutability informally: a reference is *mutable* if there exists an execution in which the reference is used to change the state of its referent object. However, previous work left the term "used" undefined. A formal definition of parameter reference mutability is non-trivial since a reference, or references obtained from it by a series of field accesses, may be stored in a variable or passed as an argument to a function, and used in performing a modification later during the execution of the function.

Intuitively, a reference r is *used* in a mutation if the mutation happens to an object via r or via a reference that was obtained via a series of dereferences from r. Section A.1 formalizes this intuition. Section A.2 illustrates the definition via examples.

A.1 Formal Definition

We proceed to define parameter mutability as follows. First, we define a core language for which we build the mutability definition. Second, we augment the term evaluation rules (i.e., the operational semantics) to additionally compute whether the parameter reference is used in a mutation. Third, we formally define reference-mutability (Definition 3).

We define reference-mutability in the context of λUnit-Ref-Mut, an augmented version of λUnit-Ref [38, pp. 166–167], a core language of untyped lambda-calculus and references. The modified language captures all essential features that affect mutability.

Figure 7 presents the syntax of λUnit-Ref-Mut. Changes from λUnit-Ref are shaded. The main changes are:

t	::=		*terms* :
		x	*variable*
		t t	*application*
		ref t	*reference creation*
		!t	*dereference*
		t := t	*assignment*
		v	*value*
v	::=		*values* :
		$\langle \lambda$x.t , $\bot\rangle$	*abstraction value*
		$\langle\lambda_c$x.t, $\bot\rangle$	*abstraction value with checked parameter*
		$\langle$ unit , $\bot\rangle$	*constant* unit
		$\langle$ l , $d\rangle$	*store location*
d	::=		*distances* :
		N	*distance*
		$\bot$	*undefined distance*
μ	::=		*stores* :
		$\varnothing$	*empty store*
		μ, l = v	*location binding*

Figure 7. Syntax of λUnit-Ref-Mut. Changes from λUnit-Ref [38] are shaded.

- One function in the program is marked as λ_c (*checked* abstraction). This is the function that declares the parameter whose mutability is being defined (the *checked parameter*).
- A value in λUnit-Ref-Mut is a pair containing the corresponding value from λUnit-Ref (i.e., abstraction, constant, or location) and a distance, which is a lifted natural number.

The checked parameter's location has a distance of 0. A location has a non-negative distance n if it was obtained (during execution) by dereferencing the checked parameter n times. Other locations (that were not obtained from the checked parameter), and all non-locations (which cannot be modified), have a distance of $\bot$. The distance for a value can be thought of as the length of a chain of executed dereferences from the checked parameter to the given value. Any location with a non-$\bot$ distance aliases part of the checked parameter's state.

In λUnit-Ref-Mut, evaluation maintains the distances associated with each value. The checked parameter is *mutable* if a location that has a non-$\bot$ distance (was reached by a series of dereferences from the parameter) is assigned.

The reduction rules for λUnit-Ref-Mut are shown in Figure 8. Each reduction rule is a relationship, $\mathtt{t} \mid \mu \longrightarrow \mathtt{t}' \mid \mu'$,

where $\mathtt{t}$ with a store μ reduces to $\mathtt{t}'$ with a store μ'. For simplicity, the rules account for only one checked parameter in the program. There are five changes to the semantics of λUnit-Ref:

E-AppAbs When an abstraction $f = \lambda_c \mathtt{x}.\mathtt{t}$ is applied to a location $\langle \mathtt{l}, d \rangle$, the parameter x is substituted in the resulting expression with a pair containing the original location and a distance of 0. This indicates that if l is modified, the modification to the value passed to f will occur at a distance of 0 dereferences from that value.
If λ_c is applied to a non-location (E-AppAbs2), or a regular (non-checked) λ is applied to any value (E-AppAbs1), then the distance of the value is unchanged.

E-RefV When a new reference $\mathtt{l}$ to an existing value $\mathtt{v}$ is created, the value is put in the store with its distance. Since the newly created location is not reachable from the checked parameter (or any other location), its distance is $\bot$.

E-DerefLoc When a location $\langle \mathtt{l}, \bot \rangle$ is dereferenced, the obtained value retains its original distance, since $\mathtt{l}$ was not reached from the parameter (rule E-DerefLoc1).
When a location $\langle \mathtt{l}_1, d_1 \rangle$ is dereferenced, the resulting location $\mathtt{l}_2$ is obtained using d_1+1 dereferences. (Location $\mathtt{l}_2$ may have already been obtained from the parameter using a different path of length $d_2 \neq \bot$, before it was put in the store. In that case, we could choose either $d_1 + 1$ or d_2. This choice has no effect, since a mutation only depends on whether or not the distance is $\bot$.)

E-Assign During evaluation of an assignment to a location that was not reached from the parameter ($\bot$ distance in E-Assign), the store is updated and the computed value is $\langle \text{unit}, \bot \rangle$ since the distance of unit is always $\bot$.

E-AssignError The rule E-AssignError stops the execution when a location with a non-$\bot$ distance is mutated (changing the state of the checked parameter).

We define reference-immutable and reference-mutable parameters:

Definition 1. (contains execution) Term e_f contains an *execution* of checked abstraction $f \equiv \lambda_c x.t$ iff

- f is a sub-term of e_f,
- f is the only checked abstraction in e_f.
- f is applied during the evaluation of e_f according to the rules of Figure 8.

Definition 2. (modification) Parameter p of a checked abstraction $f \equiv \lambda_c p.t$ is *used in a modification* if during the evaluation of e_f, an application of f evaluates to error:modification using the rules of Figure 8.

Definition 3. (parameter reference (im)mutability) Parameter p of a checked abstraction $f \equiv \lambda_c p.t$ is *reference-mutable* if there exists a term e_f containing an execution of f such that p is used in a modification during the evaluation of e_f. Otherwise, p is *reference-immutable*.

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1' \mid \mu'}{\mathtt{t}_1\ \mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2 \mid \mu'} \quad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_2' \mid \mu'}{\mathtt{v}_1\ \mathtt{t}_2 \mid \mu \longrightarrow \mathtt{v}_1\ \mathtt{t}_2' \mid \mu'} \quad \text{(E-App2)}$$

$$(\lambda x.\mathtt{t}_{12})\ \mathtt{v}_2 \mid \mu \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \mid \mu \quad \text{(E-AppAbs1)}$$

$$\frac{\mathtt{v}_2 \text{ not location}}{(\lambda_c x.\mathtt{t}_{12})\ \mathtt{v}_2 \mid \mu \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \mid \mu} \quad \text{(E-AppAbs2)}$$

$$\frac{\mathtt{v}_2 = \langle \mathtt{l}_2, d \rangle}{(\lambda_c x.\mathtt{t}_{12})\ \mathtt{v}_2 \mid \mu \longrightarrow [\mathtt{x} \mapsto \langle \mathtt{l}_2, 0 \rangle]\mathtt{t}_{12} \mid \mu} \quad \text{(E-AppAbs3)}$$

$$\frac{\mathtt{l} \notin \mathrm{dom}(\mu)}{\mathrm{ref}\ \mathtt{v}_1 \mid \mu \longrightarrow \langle\, \mathtt{l}\ , \bot \rangle \mid (\mu, \mathtt{l} \mapsto \mathtt{v}_1)} \quad \text{(E-RefV)}$$

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1' \mid \mu' t}{\mathrm{ref}\ \mathtt{t}_1 \mid \mu \longrightarrow \mathrm{ref}\ \mathtt{t}_1' \mid \mu'} \quad \text{(E-Ref)}$$

$$\frac{\mu(\mathtt{l}) = \mathtt{v}}{!\langle\, \mathtt{l}\ , \bot \rangle \mid \mu \longrightarrow \mathtt{v} \mid \mu} \quad \text{(E-DerefLoc1)}$$

$$\frac{\mu(\mathtt{l}_1) = \langle \mathtt{l}_2, d_2 \rangle \qquad d_1 \neq \bot}{!\langle \mathtt{l}_1, d_1 \rangle \mid \mu \longrightarrow \langle \mathtt{l}_2, d_1 + 1 \rangle \mid \mu} \quad \text{(E-DerefLoc2)}$$

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1' \mid \mu'}{!\mathtt{t}_1 \mid \mu \longrightarrow !\mathtt{t}_1' \mid \mu'} \quad \text{(E-Deref)}$$

$$\langle\, \mathtt{l}\ , \bot \rangle := \mathtt{v}_2 \mid \mu \longrightarrow \langle\, \mathrm{unit}\ , \bot \rangle \mid [\mathtt{l} \mapsto \mathtt{v}_2]\mu \quad \text{(E-Assign)}$$

$$\frac{d \neq \bot}{\langle \mathtt{l}, d \rangle := \mathtt{v}_2 \mid \mu \longrightarrow \text{error:modification}} \quad \text{(E-AssignError)}$$

$$\frac{\mathtt{t}_1 \mid \mu \longrightarrow \mathtt{t}_1' \mid \mu'}{\mathtt{t}_1 := \mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_1' := \mathtt{t}_2 \mid \mu'} \quad \text{(E-Assign1)}$$

$$\frac{\mathtt{t}_2 \mid \mu \longrightarrow \mathtt{t}_2' \mid \mu'}{\mathtt{v}_1 := \mathtt{t}_2 \mid \mu \longrightarrow \mathtt{v}_1 := \mathtt{t}_2' \mid \mu'} \quad \text{(E-Assign2)}$$

Figure 8. Operational semantics (evaluation rules) for λUnit-Ref-Mut. Changes from λUnit-Ref [38] are shaded.

To show correctness of the reference immutability definition, we prove the following lemma.

Lemma 4. A parameter x of abstraction $f = \lambda_c x.t$ is *mutable* iff there exists a term e_f containing an execution of f, such that the following conditions are met:

- **(c1)** e_f evaluates to $\langle \mathtt{l}', d' \rangle := v$, $d' \neq \bot$.

- **(c2)** Let $\langle \mathtt{l}, d \rangle$ be the argument in the last application of f in e_f then $\mathtt{l}'$ was obtained using a series of d' dereferences from $\mathtt{l}$.

Proof. $\Longrightarrow$ If x is *mutable* then by definitions 1, 2, and 3 we get the following facts:

- **(i)** There exists a term e_f containing an execution of f.
- **(ii)** f is a sub-term of e_f.
- **(iii)** f is the only checked abstraction in e_f.
- **(iv)** f is applied during the evaluation of e_f.
- **(v)** e_f evaluates to error:modification.

By rule E-AssignError and fact **(v)** the term

$$\langle \mathtt{l}', d' \rangle := \mathtt{v},\ d' \neq \bot \tag{1}$$

is the last to be reduced in the evaluation of e_f (satisfying condition **(c1)**).

Since a location is created with a $\bot$ distance (E-RefV), and a $\bot$ distance can only change in rule E-AppAbs3, it follows from (1) that rule E-AppAbs3 must have been applied in the execution of e_f. Since f is the only checked abstraction in e_f (fact**(iii)**), it follows that f is applied to a location; let $\langle \mathtt{l}, d \rangle$ be the last such location.

Condition **(c2)** is proven by induction. When $d' = 0$ then $\mathtt{l}' = \mathtt{l}$ by rule E-AppAbs3. Assume that the value $\langle \mathtt{l}'', d' - 1 \rangle$ was reached by a series of $d' - 1$ dereferences from $\mathtt{l}$. The only way to get value $\langle \mathtt{l}', d' \rangle$ is by applying rule E-DerefLoc2 on the expression

$$!\langle \mathtt{l}'', d' - 1 \rangle. \tag{2}$$

The antecedent of the rule E-DerefLoc2 is $\mu(\mathtt{l}'') = \langle \mathtt{l}', d' \rangle$. Therefore $\mathtt{l}'$ is reached by one dereference operation from $\mathtt{l}''$ proving condition **(c2)**.

$\Longleftarrow$ This direction follows immediately from Definition 3 and the fact the $\langle \mathtt{l}', d' \rangle := \mathtt{v},\ d \neq \bot$ evaluates to error:modification (E-AssignError).

□

A.2 Examples

We illustrate the formal definition of parameter reference mutability on the example functions in Figures 9 and 10.

Function `f1` (Figure 10)

In function `f1`, references `p2` and `p3` are reference-*mutable* (line 2 modifies `p2`, and line 4 modifies `p3.next`). Reference `p1` is also reference-*mutable*: when `p2` and `p3` are aliased, for example in the call `f1(x, y, y)`, the state of the object passed to `p1` is modified on line 4 using a series of dereferences from `p1`.

Figure 9 demonstrates that parameter `p1` of function `f1` is reference-*mutable*. Function `f1` is converted to λUnit-Ref-Mut: field accessed are replaced with location dereferencing and multiple function parameters are supplied by currying. The top-most abstraction is the checked abstraction, i.e., λ_c. An execution e_{f1} is selected such that it shows `p1`'s mutability. This execution corresponds to the call `f1(x, y, y)`. The execution e_{f1} is evaluated using the set of rules in Figure 8. Figure 9 shows the evaluation. The evaluation finishes with error:modification, which demonstrates that `p1` is *mutable*. In each step, the rule in the "rule" column is applied to the underlined redex in the expression on the same row, resulting in the expression, store, and distance shown in the next row.

Function `f2` (Figure 9)

In function `f2`, reference `p4` is clearly *mutable* because line 2 modifies `p4.next`. However, reference `p5` is reference-*immutable*—it is never used to make any modification to an object during the execution of `f2`. The parameter `p5` is immutable *despite* the fact that the *object* passed to `p5` may be mutated, e.g., when parameters are aliased in the call `f2(x, x)`. Our definition is concerned with *reference* mutability, which, together with aliasing information, may be used to compute object mutability. In the example of function `f2`, the information that parameter `p5` is reference-immutable can be combined with information about `p4` and `p5` being aliased in the call `f2(x, x)` to determine that, in that call, both objects may be modified.

Figure 10 demonstrates that parameter `p5` of function `f2` is not reference-*mutable* in the call `f2(x, x)` (i.e., when parameters are aliased). The execution e_{f2} is evaluated. The evaluation finishes without error and computes value $\langle \text{unit}, \bot \rangle$, which shows that, in this execution, `p5` was not used to modify a location.

```
1  void f1(C p1, C p2, C p3) {
2   p2.next = p1;
3   C local = p3.next;
4   local.next = null;
5  }
```

$f1 \equiv \lambda_c \mathtt{p}_1.\lambda \mathtt{p}_2.\lambda \mathtt{p}_3.(\lambda \mathtt{x}.(\lambda \mathtt{v}.!v := \langle \text{unit}, \bot\rangle)(!p_3))(!p_2 := p_1)$

$e_{f1} \equiv (\lambda y.(\lambda x.f1\ x\ y\ y)(\text{ref } \langle \text{unit}, \bot\rangle)(\text{ref ref } \langle \text{unit}, \bot\rangle)$

step	expression	store	rule
1	$(\lambda y.(\lambda x.f1\ x\ y\ y)\underline{(\text{ref } \langle \text{unit}, \bot\rangle)(\text{ref ref } \langle \text{unit}, \bot\rangle)}$	$\varnothing$	E-RefV
2	$\underline{(\lambda y.(\lambda x.f1\ x\ y\ y))}\ \underline{\langle \mathtt{l}_x, \bot\rangle\ \langle \mathtt{l}_y, \bot\rangle}$	$\{(\mathtt{l}_x, \langle \text{unit}, \bot\rangle), (\mathtt{l}_y, \langle \mathtt{l}'_y, \bot\rangle), (\mathtt{l}'_y, \langle \text{unit}, \bot\rangle)\}$	E-AppAbs2
3	$\underline{f1\ \langle \mathtt{l}_x, \bot\rangle}\ \langle \mathtt{l}_y, \bot\rangle\ \langle \mathtt{l}_y, \bot\rangle$	...	E-AppAbs3
4	$(\underline{\lambda \mathtt{p}_2.\lambda \mathtt{p}_3}.(\lambda \mathtt{x}.(\lambda \mathtt{v}.!v := \langle \text{unit}, \bot\rangle)(!p_3))(!p_2 := \langle \mathtt{l}_x, 0\rangle))\underline{\langle \mathtt{l}_y, \bot\rangle\ \langle \mathtt{l}_y, \bot\rangle}$	...	E-AppAbs2
5	$(\lambda \mathtt{x}.(\lambda \mathtt{v}.!v := \langle \text{unit}, \bot\rangle)\underline{(\mathtt{l}_y, \bot\rangle)}(\underline{!\langle l_y, \bot\rangle} := \langle \mathtt{l}_x, 0\rangle)$	...	E-DerefLoc1
6	$(\lambda \mathtt{x}.(\lambda \mathtt{v}.!v := \langle \text{unit}, \bot\rangle)(!\langle \mathtt{l}_y, \bot\rangle)\underline{(\langle l'_y, \bot\rangle := \langle \mathtt{l}_x, 1\rangle)}$	...	E-Assign
7	$(\underline{\lambda \mathtt{x}}.(\lambda \mathtt{v}.!v := \langle \text{unit}, \bot\rangle)\langle !l_y, \bot\rangle)\underline{\langle \text{unit}, \bot\rangle}$	$\{(\mathtt{l}_x, \langle \text{unit}, \bot\rangle), (\mathtt{l}_y, \langle \mathtt{l}'_y, \bot\rangle), (\mathtt{l}'_y, \langle \mathtt{l}_x, 0\rangle)\}$	E-AppAbs1
8	$(\lambda \mathtt{v}.!v := \langle \text{unit}, \bot\rangle)\ \underline{!\langle \mathtt{l}_y, \bot\rangle}$	...	E-DerefLoc1
9	$(\underline{\lambda \mathtt{v}}.!v := \langle \text{unit}, \bot\rangle)\ \underline{\langle \mathtt{l}'_y, \bot\rangle}$	...	E-AppAbs2
10	$\underline{!\langle \mathtt{l}'_y, \bot\rangle} := \langle \text{unit}, \bot\rangle$	...	E-DerefLoc1
11	$\underline{\langle \mathtt{l}_x, 0\rangle := \langle \text{unit}, \bot\rangle}$	...	E-AssignError
12	error:modification	...	

Figure 9. Call `f1(x, y, y)`, converted to λUnit-Ref-Mut (with $\mathtt{p}_1$ as checked parameter) and evaluated using rules in Figure 8. Evaluation finishes with error:modification, which means that parameter `p1` is used in a mutation and thus reference-*mutable*. $f1$ is the converted function, e_{f1} is the call to `f1`, and the figure shows the evaluation.

```
1  void f2(C p4, C p5) {
2   p4.next = null;
3  }
```

$f2 \equiv \lambda_c \mathtt{p}_5.\lambda \mathtt{p}_4.!p_4 := \langle \text{unit}, \bot\rangle$

$e_{f2} \equiv (\lambda x.f2\ x\ x)(\text{ref ref ref } \langle \text{unit}, \bot\rangle)$

step	expression	store	rule
1	$(\lambda x.f2\ x\ x)\underline{(\text{ref ref ref } \langle \text{unit}, \bot\rangle)}$	$\varnothing$	E-RefV
2	$(\underline{\lambda x}.f2\ x\ x)\ \underline{\langle \mathtt{l}_x, \bot\rangle}$	$\{(\mathtt{l}_x, \langle \mathtt{l}'_x, \bot\rangle), (\mathtt{l}'_x, \langle \mathtt{l}''_x, \bot\rangle), (\mathtt{l}''_x, \langle \text{unit}, \bot\rangle)\}$	E-AppAbs2
3	$\underline{f\ \langle \mathtt{l}_x, \bot\rangle}\ \langle \mathtt{l}_x, \bot\rangle$	...	E-AppAbs3
4	$(\underline{\lambda \mathtt{p}_4}.!\mathtt{p}_4 := \langle \text{unit}, \bot\rangle)\ \underline{\langle \mathtt{l}_x, \bot\rangle}$	...	E-AppAbs2
5	$\underline{!\langle \mathtt{l}_x, \bot\rangle} := \langle \text{unit}, \bot\rangle$	...	E-DerefLoc1
8	$\underline{\langle \mathtt{l}'_x, \bot\rangle := \langle \text{unit}, \bot\rangle}$	...	E-Assign
9	$\langle \text{unit}, \bot\rangle$	$\{(\mathtt{l}_x, \langle \mathtt{l}'_x, \bot\rangle), (\mathtt{l}'_x, \langle \text{unit}, \bot\rangle), (\mathtt{l}''_x, \langle \text{unit}, \bot\rangle)\}$	

Figure 10. Call `f2(x, x)`, converted to λUnit-Ref-Mut (with `p5` as checked parameter) and evaluated using rules in Figure 8. Evaluation does not finish with error:modification, which means that parameter `p5` is *not* mutated in this execution. $f2$ is the converted function, e_{f2} shows the call to `f2`, and the figure shows the evaluation.