CiFi: Versatile Analysis of Class and Field Immutability

1st Tobias Roth
Software Technology
Technische Universität
Darmstadt, Germany
roth@cs.tu-darmstadt.de

2nd Dominik Helm
Software Technology
Technische Universität
Darmstadt, Germany
helm@cs.tu-darmstadt.de

3rd Michael Reif
Software Technology
Technische Universität
Darmstadt, Germany
mi.reif.mr@gmail.com

4th Mira Mezini
Software Technology
Technische Universität
Darmstadt, Germany
mezini@cs.tu-darmstadt.de

Abstract—**Reasoning about immutability is important for preventing bugs, e.g., in multi-threaded software. So far, static analysis to infer immutability properties has mostly focused on individual objects and references. Reasoning about fields and entire classes, while significantly simpler, has gained less attention. Even a consistently used terminology is missing, which makes it difficult to implement analyses that rely on immutability information. We propose a model for class and field immutability that unifies terminology for immutability flavors considered by previous work and covers new levels of immutability to handle lazy initialization and immutability dependent on generic type parameters. Using the OPAL static analysis framework, we implement CiFi, a set of modular, collaborating analyses for different flavors of immutability, inferring the properties defined in our model. Additionally, we propose a benchmark of representative test cases for class and field immutability. We use the benchmark to showcase CiFi's precision and recall in comparison to state of the art and use CiFi to study the prevalence of immutability in real-world libraries, showcasing the practical quality and relevance of our model.**

Index Terms—**class and field immutability, static analysis**

I. INTRODUCTION

Immutability is the property of a program element stating that it is unchangeable or not changed after its creation [1]. The immutability property is important for program correctness and security: Immutable data structures are not prone to race conditions in multi-threaded applications [2]–[4]. Immutable values are less prone to security issues, hence recommended by the *Secure Coding Guidelines for Java SE* [5]. Some APIs, like Java's `Map` interface, assume objects, used as keys, not to be mutated[1] [6]. Immutability is also a prerequisite for precisely deriving other properties, e.g., method purity [7], [8].

In this paper, we focus on immutability of classes and fields. Previous research on immutability has often focused on individual objects and references [8]–[12]. However, it has been argued [13], [14] that focusing on classes and fields simplifies the implementation of systems that enforce immutability restrictions[2] and their usage by developers.

We address the following limitations of the state of the art in checking and enforcing class and field immutability.

First, existing approaches address only individual specific levels of immutability. For instance, with their `final`, resp. `val`

annotations, the Java and Scala programming languages support a weak level of immutability called *non-assignability* [1], [8].

Coblenz et al. [14] and Porat et al. [15] deal only with *transitive immutability*, where every value referred to directly or transitively by a *transitively immutable* class or field is immutable. Nelson et al. [16], on the other hand, deal with *non-transitive* immutability of fields, where *non-transitive immutability* only guarantees that the respective field is non-assignable. However, none of the approaches handles both *transitive* and *non-transitive immutability*.

Second, existing approaches do not properly cover common programming patterns, as we elaborate in Section II. Examples of programming patterns that are not properly handled are lazy initialization and generic type parameters often found in collections and collection-like classes (e.g. `java.util.Optional`). With lazy initialization (cf. Listing 1), a field is assigned only when it is accessed for the first time. Here, the field cannot just be restricted to assignments in the class' constructor. In turn, care has to be taken to ensure that really only a single initialization can be performed. Also, it has to be ensured that the field cannot be observed before its initialization, as observing different values before and after initialization contradicts the guarantees that immutability aims to provide. Generic classes require special treatment, too, as their immutability can depend on the immutability of their type parameters. In Listing 2, the immutability of class `Generic` depends on the type parameter `T` used for the final field `t`.

```
1  class C {
2      private Object object;
3      public synchronized Object getObject() {
4          if (object==null)
5              object = new Object();
6          return object;
7  } }
```

Listing 1. Thread-safe Lazy Initialization Example

```
1  class Generic<T> {
2      private final T t;
3      public Gen(T t){ this.t = t; }
4  }
```

Listing 2. Dependently Immutable Class Example

Finally, we lack a common model that provides unified terminology for different levels of class and field immutability.

---

[1]Mutations that don't effect `equals()` comparisons are allowed.
[2]Immutability restrictions can be, e.g., in the form of annotations.

For instance, *deep* respectively *shallow* immutability are used [1], or just immutability [17] to refer to the same concepts as *(non)-transitive immutability*. Hence, we need a unified model that not only considers trivial cases, like final fields with immutable types, but also common programming patterns such as lazy initialization and generic classes.

The work presented here addresses the above limitations. First, we define a model for class and field immutability that incorporates all relevant levels of immutability and precisely defines their meaning and relations, thus establishing a consistent terminology. Second, based on the model, we define CiFi, a set of modular, independent, collaborating static analyses to infer the different levels of immutability for fields and classes, including entire class hierarchies. CiFi uses our OPAL framework [18] and can be used to reason about codebases and their immutability guarantees directly by developers or by further analyses.

We evaluate our work along several dimensions. First, we demonstrate the expressiveness of the proposed model by categorizing it along the classification system for immutability support proposed by Coblenz et al. [13]. Second, we evaluate CiFi against *CiFi-Bench*, a set of handcrafted test cases annotated with immutability properties. To the best of our knowledge, such a benchmark did not exist yet – *CiFi-Bench* can be used to guide and test other analyses of class and field immutability. We use it to evaluate Glacier, the state of the art in class- and field-immutability enforcement, to compare its precision and recall with those of CiFi. Finally, we investigate the extent to which immutability flavors (e.g., class and field immutability) and levels (e.g., mutable, or (non-)transitively immutable) defined in our model are found in real-world libraries. In short, we show that CiFi (i) precisely identifies important immutability patterns, while soundly over-approximating remaining edge cases, (ii) thereby clearly outperforms Glacier, and (iii) identifies significant amounts of immutable data in real-world libraries.

To recap, our contributions are:

- A literature survey on the definitions and terminology used for class and field immutability (Section II).
- A comprehensive, fine-grained lattice-based model of all relevant levels of class and field immutability (Section III).
- CiFi, a set of modular, collaborating static analyses that infer the properties defined in the model (Section IV).
- A handcrafted benchmark to serve as a ground truth for class and field immutability analyses (Section V-B1).
- An extensive evaluation of CiFi based on the benchmark, real-world libraries, and the state of the art (Section V).

We discuss threats to validity in Section VI, present further related work regarding object and reference immutability in Section VII, and conclude the paper in Section VIII.

II. STATE OF THE ART

We survey prior work on different levels of class and field immutability. In lack of an existing consistent terminology, we use the original names for the considered levels.

Weak levels of immutability enforcement have been part of programming language design since decades. In Scala, fields can be declared with the keyword `val` which corresponds to Java's `final` modifier. These constructs prevent the field from being reassigned, but give no guarantee that the object referenced by the field is immutable. With *case classes* in Scala and *Records* [19] introduced in Java 16, these languages also offer classes that store data in fields that, implicitly, cannot be reassigned. However, mutable objects can be assigned to them. To sum up, while the above language features underline the importance of immutability, they enforce only weak guarantees that other authors call *non-assignability* [1], [8].

Potanin et al. [1] introduce the terms *shallow* and *deep immutability* to distinguish between non-assignable fields referring to mutable objects or arrays (*shallow*) and non-assignable fields (transitively) referring to objects or arrays that cannot be mutated either (*deep*). Listing 3 illustrates both cases. The final (non-assignable) field `s` refers to a `java.util.String` (known to be deeply immutable), thus, `s` is *deeply immutable*. In contrast, the public field `iArr` refers to an array that can be mutated outside its class (arrays are mutable), thus, `iArr` is *shallowly immutable*.

```
1  public final String s = "string"; // deeply immutable
2  public final int[] iArr = {42}; // shallowly immutable
```

Listing 3. Deep/Shallow Immutability Example

Coblenz et al. [13] use different terms for the same immutability concepts, namely *non-transitive* for *shallow* and *transitive* for *deep*. Their Glacier [14] system uses annotations for Java classes and fields, with `@Immutable` enforcing *transitive immutability* and `@MaybeMutable` stating that a field or class is not guaranteed to be transitively immutable. Gordon et al. [2] call transitively immutable fields just *immutable*, whereas Nelson et al. [16] use the term *immutable* for `final` fields, i.e., fields only guaranteed to be non-transitively immutable.

Glacier has no direct support for non-assignability or non-transitive immutability, arguing that non-transitive immutability provides only weak guarantees [13]. In order for a class `C` to be `@Immutable` in Glacier, (a) all fields of `C` must be transitively immutable and may only be assigned in the class' constructors, and (b) `C` must have only `@Immutable` subclasses. Because of (a), *Glacier* cannot handle cases where fields are assigned outside a constructor, e.g., in lazy initialization. For generic classes annotated as `@Immutable`, *Glacier* enforces that type parameters are instantiated with `@Immutable` types. This is overly conservative, as type parameters do not necessarily influence a class' immutability. Also, it prevents generic immutable classes such as immutable collections from being annotated `@Immutable` if they are used to store mutable or non-transitively immutable data.

Porat et al. [15] propose an inter-procedural data-flow analysis to detect transitively immutable classes and fields in Java. According to their definition, a field is immutable if its value or referee is not mutated after being assigned in the static initializer or constructor. Like Glacier, this restrictive immutability definition cannot handle lazy initialization. A

class is said to be *immutable*, if all of its non-static fields are immutable. The approach was implemented and evaluated on the Java Development Kit (JDK) 1.2 (released in 1998); thus, it lacks support for newer features of Java, e.g., generics.

Kjolstad et al. [17] use the term *immutable* for classes that have only *transitively immutable* instance fields. Their refactoring tool *Immutator* transforms mutable classes to *immutable* ones in order to benefit from the guarantees provided by transitive immutability. To ensure that all fields are initialized in the constructor, Immutator adds two new constructors: A public one without parameters initializes all fields with a default value and a private one taking an initialization parameter for each field. Finally, it rewrites all methods mutating the transitive state into factory methods and all client methods such that they access class instances in an immutable way. Immutator makes transformed classes `final` to prohibit mutable subclasses. Thus, the refactoring is limited to classes without subclasses. With fields made `final`, lazy initialization is not possible. Also, *Immutator* does not handle generic classes.

> ■ *The survey of the state of the art in analyzing class and field immutability reveals that we lack a consistent terminology for class and field immutability. While some authors use* deep *and* shallow, *others use* transitive *and* non-transitive. *Still others use* immutable *with different meanings.*
> ■ *None of the existing approaches can simultaneously handle both non-transitive and transitive immutability. Also, none of the presented approaches can recognize lazy initialization and properly handle immutability of generic classes.*
> ■ *Each approach focuses on a fixed composition of immutability flavors, e.g., class and field immutability, and a single level—most often transitive immutability—and it is not possible for client analyses to get information for other immutability flavors or levels.*

III. Model

We present our unified model of immutability properties for fields, classes, and types along with their order and relations structured in lattices. While they are simple chains only, this is in line with the OPAL framework's [18] terminology and requirements. We exemplify the properties with Java code snippets, but the model can be used for any object-oriented language.

A. Field Assignability

Potanin et al. [1] define assignability to indicate whether a static or instance field is or can be reassigned after it is initialized. We extend on that, defining several levels of assignability which we elaborate below. Their order is illustrated in Fig. 1.

1) (Effectively) Non-Assignable Fields: Fields can explicitly be enforced to be *non-assignable*, e.g., using Java's `final` keyword, or can be *effectively non-assignable* because there is no reassignment present and none can be added through other code that is not analyzed.

Definition 1: A field is *non-assignable* if it is only assigned once and cannot be reassigned.

Definition 2: A field is *effectively non-assignable* if it cannot be observed with different values.

This distinction allows to find fields that are not yet enforced to be non-assignable but could be made so. Examples for both cases are given in Listing 4. The field `imm` (Line 2) is `final` and, thus, it is only assigned once (during the execution of the implicit constructor). As a result, it cannot be reassigned. Similarly, the field `effImm` (Line 3) is initialized only once and is never reassigned again. As `effImm` is declared `private`, no code outside of class `C` can assign to it, thus rendering it *effectively non-assignable*.

```
1  class C {
2      private final int imm = 42;
3      private int effImm = 42;
4  }
```

Listing 4. (Effectively) Non-Assignable Fields

2) Lazily Initialized Fields: This is a common pattern used to avoid the cost of computing or storing a value if it is never accessed, while performing the computation only once if it is accessed repeatedly. It is often implemented by a field accessible only through a single method that only computes and stores the value if the field still has its default value.

An example of a lazily initialized field was given in Listing 1. As the field `object` is `private`, no other code can access the field except through the method `getObject`. This method will initialize the field `object` only if its value is still `null`. As the method is `synchronized`, it is guaranteed that the field is only initialized once, even in the presence of multi-threaded execution. Without the `synchronized` annotation, the field `object` could be assigned to more than once. This happens if concurrent threads each see `object` at its default state (`null`) in Line 4 before any of them performs the assignment in Line 5. In this case, each thread may assign a different instance to `object`, with only the last assignment being persistent. Yet, for programs known to be single-threaded, one can still provide a valuable guarantee. For this reason, we define two properties related to lazy initialization: *lazily initialized* (Definition 3) and *unsafely lazily initialized* (Definition 4):

Definition 3: A field is *lazily initialized*, if its lifetime can be divided into two distinct phases: During the first phase, no accesses to the referenced value are made except to check whether the field must be transferred to the second phase. During the second phase, the field is effectively non-assignable.

Definition 4: A field is *unsafely lazily initialized* if, as long as only one thread accesses it, its lifetime can be divided into two distinct phases: During the first phase, no accesses to the referenced value are made except to check whether the field must be transferred to the second phase. During the second phase, the field is effectively non-assignable.

3) Assignable Fields: Assignable is the top (least precise) value of the field-assignability lattice:

Definition 5: A field is *assignable* if none of the previous definitions apply.
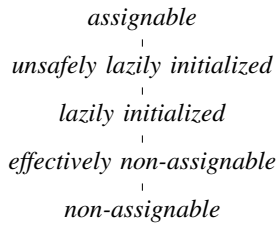
assignable

unsafely lazily initialized

lazily initialized

effectively non-assignable

non-assignable

Fig. 1. Field-Assignability Lattice

mutable

non-transitively immutable

dependently immutable

transitively immutable

Fig. 2. Immutability Lattice

Fig. 1 gives the lattice order of the previously defined levels of assignability based on the provided guarantees: while non-assignable fields cannot be assigned outside the constructor, effectively non-assignable fields could be reassigned but provably are not. In turn, (unsafely) lazily initialized fields are reassigned once. However, they can be observed before they are initialized only by the check for the default value.

B. Field Immutability

Field immutability combines *assignability* of a given static or instance field f with the immutability of f's value. Our lattice for field immutability is shown in Fig. 2. The top (least precise) value of the field-immutability lattice is *mutable*:

Definition 6: A field is *mutable* if and only if it is assignable.

For the purpose of this definition, we treat an *unsafely lazily initialized* field as assignable if it is unknown whether multiple threads might access the field.

If a field f is not assignable, its immutability depends on the immutability of the values f can potentially refer to. Primitive values are always immutable. Array values are mutable (Java has no concept of immutable arrays), hence, the immutability of a field f that refers to an array arr depends on whether arr or any of its elements is actually mutated or could be mutated by unknown code. The same applies to objects values, too. However, unlike arrays, for some objects it is possible to infer whether such mutation is actually possible either by inspecting the static type of f or by analyzing the potential runtime types of objects that f may refer to. Line 2 in Listing 5 illustrates a non-assignable field that refers to an array that is not and cannot be mutated. Line 3 illustrates a non-assignable field of type `java.lang.String`—known to be immutable.

```
1  class C {
2    private final int[] iArr = new int[]{ 1, 2, 3, 4 };
3    private final String finalString = "final string";
4  }
```
Listing 5. Field Immutability Example

Our immutability lattice distinguishes between *transitively immutable* and *non-transitively immutable* fields:

Definition 7: A field is *transitively immutable* if it is not assignable and no object (or array) that can transitively be reached through the field can ever be mutated.

Definition 8: A field is *non-transitively immutable* if it is not assignable, but objects (or arrays) transitively reachable through the field might be mutated.
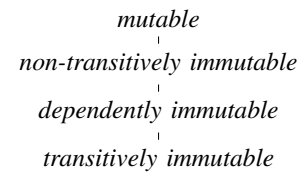
Finally, we define the level *dependently immutable*, which models the effect of generic types on immutability. A field with a generic type T (i.e., `private final T t;`) that is not assignable (including *unsafely lazily initialized* only if it is known that only one thread accesses the field) can either be *transitively* or *non-transitively immutable* depending on the concrete runtime type of T. Thus, we say that such a field is *dependently immutable*. The property *dependently immutable* is—as generic types are—parameterized over all types that influence the reference's immutability, e.g., above generically typed field `t` is said to be *dependently immutable for T*.

Definition 9: A field is *dependently immutable* if it is not assignable and the (transitive) immutability of the referenced object depends on at least one generic type parameter.

C. Class and Type Immutability

To determine whether a field is transitively immutable or not, we need information about class and type immutability. Class immutability takes the same values as field immutability, i.e., the ones given in Fig. 2 and is defined through field immutability as follows:

Definition 10: The immutability of a class is the least upper bound (join) of the immutability of all of its instance fields, respecting specialization of generic types for dependently immutable fields.

As a corollary, class immutability is the least upper bound (join) of the immutability of all possible instances of that class (because the instance fields' immutability is determined by the immutability of their values, which make up the state of the class' instances). Not all instances of a class necessarily have the same immutability property. The following factors can lead to a more precise immutability of a particular instance in comparison to the immutability of its class:

Firstly, while some instance field f of a class C may, in general, not be effectively non-assignable, it may provably not be assigned to for a particular instance o. This is, e.g., the case, if no method that assigns to f ever gets invoked on o. Secondly, during the creation of a particular instance o of a generic class, type parameters can be substituted by concrete types. This determines whether dependently immutable fields of o are actually transitively or non-transitively immutable. Finally, while the declared type of a field f might not be transitively immutable, the concrete object assigned to f can be, in which case f becomes transitively immutable after assignment. Thus, an instance of a class with fields that are not transitively immutable can still be transitively immutable depending on how

Escape

↑

Field Assignability ← **Field Immutability**

Type Immutability → **Class Immutability** | →dependency
→cyclic dep.
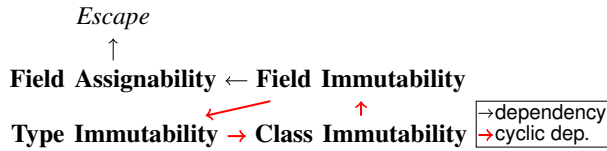
Fig. 3.   Analysis Dependencies

it is created. This is illustrated in Listing 6. Depending on the constructor used, the field `nonTransitive` in Line 2 can be assigned either a `MutableClass` or an `ImmutableClass` instance. While an instance of `C` created with the first constructor is *non-transitively immutable*, one created with the second constructor is *transitively immutable*.

```
1  class C {
2    private final Object nonTransitive;
3    public C(MutableClass mc) { nonTransitive = mc; }
4    public C(ImmutableClass ic) { nonTransitive = ic; }
5  }
```

Listing 6.   Immutability Dependent on Constructor

It is often useful to determine the immutability at the level of types, e.g., to quickly determine whether a field of a given static type can be transitively immutable. In object-oriented languages, a type is either populated by one class (of the same name as the type) and all of its (potential) subclasses or by an interface (of the same name as the type) and all of its implementing classes. Type immutability is defined through class immutability and also uses the lattice from Fig. 2.

Definition 11: The immutability of a type is the least upper bound of the immutability of all classes populating that type.

As a corollary, the type of a `final` class has the same immutability as the class. Depending on the analysis scenario, the set of potential subclasses may not be known completely, e.g., when analyzing an extensible library; in this case the type must conservatively be considered to be mutable [15], [20].

IV. CiFi: Analysis Implementation

CiFi implements the presented model as a set of collaborating modular analyses for field assignability and for field, class, and type immutability based on our static analyses framework OPAL [21]. OPAL enables the composition of decoupled interdependent static analyses that collaborate via a blackboard architecture [18] model for fixed-point computations. The results of CiFi can be used to derive immutability guarantees introduced in Section I or to reveal their possible absence.

A. Overall Architecture of CiFi

Fig. 3 shows the dependencies between CiFi's analyses. Field immutability depends on field assignability and type immutability. The latter depends on class immutability, which, in turn, depends on field immutability. The analysis for field assignability depends on an escape analysis for determining effective non-assignability of fields. This analysis is not shown in bold font as we used an escape analysis provided by OPAL.

As indicated by the red arrows in Fig. 3, there is a circular dependency of analyses. Thanks to OPAL's blackboard architecture and fixed-point solver, analyses, including cyclically dependent ones, execute in an interleaved way, even if otherwise autonomous. Thus, despite the cycle, our analyses can profit from the intermediate results of each other. This simplifies the implementation of CiFi's analyses and enables to easily exchange their implementation or add further analyses for trading off precision, soundness, and performance.

B. Field-Assignability Analysis

The field-assignability analysis is based on the respective lattice (cf. Fig. 1) and is a prerequisite for the field-immutability analysis. We omit a discussion of more trivial aspects and focus on handling assignments outside of constructors. Simplified pseudocode for handling lazy initialization is shown in Listing 7. The analysis checks whether an initialization is only performed after a default-value check (e.g., `null` in case of objects) has succeeded (Line 2). To determine thread safety, the analysis checks whether the initialization is performed in a synchronized method or a block synchronized on the object holding the field (Line 3). Furthermore, the analysis ensures that even if exceptions are thrown within the lazy initialization method, either the field is guaranteed to be written before its value is returned, or its value is not returned at all (Line 4).

```
1  fun isFieldLazilyInitialized(field):
2    if(initializationNotWithinDefaultValueCheck(field) ||
3       initializationNotSynchronized(field) ||
4       exceptionsLeakUninitializedField(field)) false
5    else true
```

Listing 7.   Lazy Initialization Recognition (Pseudocode)

Additionally, CiFi is able to recognize fields that are assigned only on freshly created instances before they can be accessed elsewhere. For this purpose CiFi checks that the instance does not escape before it is returned. This pattern, illustrated in Listing 8, is often used to implement the `clone` method.

```
1  class C {
2    private int i;
3    public C clone(){
4      C c = new C();
5      c.i = i;
6      return c;
7    }
8  }
```

Listing 8.   Clone Pattern

C. Field-Immutability Analysis

The field-immutability analysis combines results from analyses for field assignability and class and type immutability. Its logic is sketched in Listing 9. It always considers assignable fields mutable (Line 2). For all other fields, it checks whether all objects assigned to the field can be identified (Line 3). If this is the case, the join of the respective class immutability properties is computed and used (Line 4), otherwise the immutability for the field's static type is checked (Line 5).

```
1  fun getFieldImmutability(field):
2    if (isFieldAssignable(field)) Mutable
3    else if (canAllAssignedObjectsBeIdentified(field))
4      join(getAssignedObjects(field).map(_.getClassImmutability))
5    else if (getTypeImmutability(field)==TransitivelyImmutable)
6      TransitivelyImmutable
7    else if (hasGenericType(field)) // Dependent Immutablity
8      if (onlyTransitivelyImmutableTypeParams(field))
9        TransitivelyImmutable
10     else if (hasANotTransitivelyImmutableTypeParam(field))
11       NonTransitivelyImmutable
12     else DependentlyImmutable
13   else NonTransitivelyImmutable
```

Listing 9. Field Immutability Analysis (Pseudocode)

The analysis recognizes dependently immutable fields using information from the field's *Signature* attribute in the Java Bytecode. If the *Signature* attribute contains generic type parameters, the field might be *dependently immutable* (Line 7). In this case, it is checked whether all generic type parameters are instantiated with transitively immutable types (Line 8); if this is the case, the field is transitively immutable. It is next checked whether at least one generic type parameter was instantiated with a type that is non-transitively immutable or mutable (Line 10). In this case, the field is non-transitively immutable. If neither of the latter two cases applies, the field is dependently immutable (Line 12).

This handling of fields with generic types is exemplified in Listing 10. First note that class `GC` is *dependently immutable for T* because of its generically typed field `genericField` (Line 2). For field `gcTransitive` in Line 6, the single generic type parameter is instantiated with the transitively immutable type `java.lang.Integer`. Thus, `gcTransitive` is also transitively immutable. In turn, the generic type parameter of field `gcMutable` in Line 7 is instantiated with the (presumably mutable) type `MutableClass`. Thus, `gcMutable` is only non-transitively immutable. Finally, for field `gcGeneric` in Line 8, the generic type parameter is instantiated with another generic type parameter, `T`. Thus, `gcGeneric` is dependently immutable.

```
1  final class GC<T> {
2    private final T genericField;
3    public GC(T value){ this.genericField = value; }
4  }
5  class C<T> {
6    private final GC<Integer> gcTransitive;
7    private final GC<MutableClass> gcMutable;
8    private final GC<T> gcGeneric;
9    [...]
10 }
```

Listing 10. Dependent Immutability

D. Class- and Type-Immutability Analysis

The class-immutability analysis of a class `C` joins the immutability of `C`'s parent class and the immutability of the instance fields declared in `C` (cf. Definition 10). Simplified pseudocode of its logic is shown in Listing 11. Note that interfaces implemented by `C` do not have to be considered as they cannot contain instance fields. As analyzing `java.lang.String`'s

immutability is far from trivial, CiFi is configured to treat it as transitively immutable. This is in line with other immutability analyses (e.g., [15]) that are configured similarly. Also, we do not consider specialization of generic type parameters.

```
1  fun getClassImmutability(class):
2    classImm = getClassImmutability(getSuperClass(class))
3    for field in class.instanceFields
4      fieldImm = getFieldImmutability(field)
5      if (fieldImm > classImm) classImm = fieldImm
6    classImm
```

Listing 11. Class Immutability Analysis (Pseudocode)

The type-immutability analysis' logic is sketched in Listing 12. It follows the definition of *type immutability* in Definition 11, joining the individual classes' immutability properties while taking into consideration whether the analysis is performed in a closed- or open-world scenario (Line 2).

```
1  fun getTypeImmutability(class):
2    if (isExtensible(class)) return Mutable
3    typeImm = getClassImmutability(class)
4    for subclass in class.allSubclasses
5      classImm = getClassImmutability(subclass)
6      if (classImm > typeImm) typeImm = classImm
7    typeImm
```

Listing 12. Type Immutability Analysis (Pseudocode)

While other tools usually support only one (cf. [13]), CiFi lets users configure either open- or closed-world assumption. Under open-world assumption, it assumes classes can be added to all packages except for subpackages of `java`[3] and all non-final classes can be extended. Under closed-world assumption, it assumes no classes can be added to existing packages and existing classes cannot be extended. However, `public` fields and methods are assumed to be accessible.

E. Threats To Soundness

CiFi does not consider any field access by means of reflection, `sun.misc.Unsafe`, or native methods calls. Such accesses, potentially anywhere in the program, cannot reliably be linked to specific fields. Consciously omitting such features in order to improve precision is called *soundiness* by Livshits et al. [22]. Doing so is in line with other state-of-the-art static immutability analyses; e.g., Porat et al. [15] do not consider native code and "dynamic effects resulting from reflection" in their class- and field-immutability analyses.

V. EVALUATION

Our evaluation[4] targets the following research questions:

RQ1 How expressive is our model relative to the classification of immutability facets defined by Coblenz et al. [13]?

RQ2 How precisely and soundily does CiFi fulfill our model?

RQ3 How does CiFi compare to the state of the art[5] with respect to the previous question?

---

[3]The classloader usually prohibits adding new classes to these packages.
[4]For reproducibility: https://doi.org/10.5281/zenodo.5227231
[5]We consider Glacier [14] (cf. Section II) as the state-of-the-art approach in enforcing class and field immutability.

RQ4 Does our model reflect immutability facets in the real world?

The rationale for these questions is as follows. Once we demonstrate the model's conceptual quality (**RQ1**) and check that CiFi precisely and soundly implements it (**RQ2**, **RQ3**), we use CiFi to assess the model's practical quality (**RQ4**).

A. Expressiveness of the Model

For **RQ1**, we use the system that Coblenz et al. [13] proposed for classifying mutability restrictions along several dimensions.

a) Type of Restriction: Our model considers the immutability of fields, classes and types, not just read-only restrictions on individual references. This provides stronger guarantees for developers [13]. We also consider assignability for fields. We do not consider ownership of objects, which we discuss in Section VII together with read-only restrictions.

b) Scope: Our model focuses on class immutability, which Coblenz et al. [13] point out to be frequently needed.

c) Transitivity: We consider both transitive and non-transitive immutability. This enables a more fine-grained view compared to systems surveyed by Coblenz et al.

d) Initialization: We do not support explicit relaxing of restrictions during initialization. However, our definition of lazy initialization also encompasses delayed initialization, if fields are assigned only once, also enabling cyclic data structures.

e) Abstract vs. Concrete State: We consider the set of all instance fields of an object, i.e., its *concrete state*. Immutability can also be defined on *abstract state* [9], [11][6], encoded by annotations. Assuming such annotations are available, our model can be applied to abstract state, too.

f) Backward Compatibility: Our approach performs static analysis to infer immutability; it does not require developers to use specific language features or annotations. Thus, it is soundy regardless of potentially unknown code interfacing with the analyzed software when used with an open-world assumption. If the analyzed program cannot be extended, a closed-world assumption can be used to uncover more immutability.

g) Enforcement: We infer immutability instead of enforcing it, but do provide static guarantees on immutability. Static enforcement may burden developers if they have to annotate all relevant program constructs [13]. This concern does not apply to our automated inference.

h) Polymorphism: Handling mutable and immutable parameters of functions is not applicable to our approach that infers actual immutability instead of enforcing restrictions.

> ■ *Our model is more expressive than approaches surveyed by Coblenz et al. [13] without completely covering the described design space. To balance expressiveness with usability [14], we focus on fields, classes, and types, which improves usability [13], [14]. Yet, the model can be easily extended, e.g., with object or reference immutability. Such extensions are well-supported by CiFi's inference approach (no annotations) and its modular architecture, enabling to plug-and-play analyses depending on what results are considered relevant.*

---

[6]Excluding some non-essential state, e.g., fields used for caching.

B. Precision and Recall

A ground truth is needed to validate precision and recall of CiFi and other analyses w.r.t. our model. To the best of our knowledge, no benchmark for class and field immutability exists that could be annotated with our model's properties. Thus, we handcrafted *CiFi-Bench*.

1) Benchmark: CiFi-Bench[7] includes a total of more than 470 test cases for all immutability levels defined in our model, organized into 13 categories:

- **Assignability:** different (effectively) (non-)assignable fields including clone pattern (counter)examples.
- **General:** simple cases, e.g., static fields, interfaces, trivially transitively immutable and mutable classes.
- **Known Types**
  - **Single:** cases where a single concrete object is assigned to a field, yielding stronger immutability guarantees than possible to infer from the field's static type.
  - **Multiple:** cases where different objects can be assigned to a field and stronger immutability guarantees can be inferred than possible from the field's static type, including cases where concrete objects or only their types are known.
- **Generic**
  - **Simple:** cases of immutability in combination with generic types, i.e., dependent immutability.
  - **Extended:** advanced usages of generics such as multiple nested generic types and generic types with bounds.
- **Arrays**
  - **Non-Transitive:** cases with mutable arrays resulting in non-transitively immutable fields.
  - **Transitive:** cases with arrays that cannot be or are not mutated, resulting in transitively immutable fields.
- **Lazy Initialization**
  - **Arrays:** cases of lazy initialization of array typed fields.
  - **Objects:** cases of thread-safely as well as unsafely lazily initialized fields with object types.
  - **Primitive Types:** lazy initialization without synchronization which can be thread-safe for primitive types.
  - **Scala Lazy val:** an example modeled after Scala 2.12's implementation of `lazy val` [23].
- **String:** a class with two fields modeled after the shared `char` array and `hashCode` method of `java.lang.String`.

We annotated fields, classes, and types with the respective assignability and immutability properties as expected with an open-world assumption.

2) Results for CiFi: For each test case tc, CiFi either produces the precise value annotated in tc (in five categories), or a soundy over-approximation, i.e., a value further up in the respective lattice, which is less precise than possible but can be soundly used by further analyses/optimizations. CiFi did not produce any unsound results, i.e., values further down in the lattice. In some more detail, the results are as follows:

---

[7]https://github.com/opalj/CiFi-Benchmark

- CiFi inferred immutability properties precisely for the categories: *Assignability*, *General*, *Known Types*, *Generic/Simple*, *Arrays/Non-Transitive*, *Lazy Initialization/Arrays*, and *Lazy Initialization/Objects*.
- In category *Generic/Extended*, CiFi soundly over-approximates some complex test cases such as doubly nested generic classes (`Gen<Gen<T>>`), generic cases with bounds, and more complex lazy initialization patterns than the one we described in Section IV-B. For doubly nested generics, the approximation is not *mutable*, but *non-transitively immutable*, retaining some precision. Generic classes with bounds are soundly over-approximated as *dependently immutable*.
- In *Arrays/Transitive*, CiFi soundly over-approximates all tests to *non-transitively immutable*, and in *Lazy Initialization/Primitive Types* to *unsafely lazily initialized* or *assignable*. All test cases in *Lazy Initialization/Scala Lazy Val* and *String* are soundly over-approximated to *assignable*, except for the field referring to the final `char` array in the category *String* which is soundly over-approximated to *non-transitively immutable*.

> ■ *CiFi matches the annotated properties of the benchmark either precisely or soundly over-approximates them. The observed over-approximations are due to missing support for the respective features. Leaving complex features out of scope when the expected benefit is small is in line with other immutability analyses [15]. Handling these complex features precisely would not lead to considerably more immutability being recognized, as they represent rare corner cases. Handling each of these corner cases would only prevent few over-approximations.*

C. Comparison with Glacier

To answer **RQ3**, we run Glacier [14], the state-of-the-art tool for enforcing class and field immutability on *CiFi-Bench*. As Glacier only considers transitive immutability, we can only evaluate it w.r.t. this level of immutability. Hence, we annotated all classes and fields of *CiFi-Bench* with Glacier's `@Immutable` annotation. We consider Glacier to pass a test if either of the following holds: (a) it does not output an error for transitively immutable fields and classes, (b) it outputs such an error for fields and classes that are not transitively immutable (since Glacier, does not handle non-transitive or dependent immutability, respective fields have to be considered mutable). The results for each category are as follows.

- For category *Known Types/Multiple*, Glacier can enforce transitive immutability.
- For two cases in *General* resp. *Known Types/Single*, Glacier produces unsound results. First, Glacier treats both `@Immutable` and `@MaybeMutable` classes as subtypes of `java.lang.Object`. Thus, a mutable object can be assigned to an `@Immutable` field of type `Object`. Second, while Glacier prohibits assignments to fields outside of the constructor, it does not check whether a field being assigned in a constructor belongs to

the object being constructed. Thus, `@Immutable` fields can be mutated while constructing other objects. Both cases are shown in Listing 13.

```
1   @Immutable class C {
2       @Immutable private Object o;
3       public C(C parent, Object o){ parent.o = o; }
4   }
```

Listing 13. Glacier Unsoundness Example

- Glacier was unsound in three *Assignability* tests. Two are again due to `@MaybeMutable` being a subtype of `java.lang.Object`, but the third one revealed another issue: Glacier ignores compound-assignment operators like `+=`. Thus, primitive or `java.lang.String` fields can be mutated outside of constructors. In CiFi, such omissions are less likely to occur accidentally as it analyzes bytecode. Additionally, Glacier could not handle the clone pattern cases properly because of assignments outside of constructors.
- Glacier passed all test cases in *Generic* as it enforces that only `@Immutable` types are used for type parameters of `@Immutable` classes and only `@Immutable` classes can extend `@Immutable` classes. But this means that Glacier cannot handle dependent immutability, which results in lost opportunities for being more precise.
- In category *Arrays*, non-transitively immutable fields are handled correctly. Some transitively immutable fields are also enforced correctly, but require four annotations: `@Immutable int @Immutable[] arr = new @Immutable int @Immutable[5];` Glacier cannot enforce transitive immutability where array elements are not mutated, despite not being `@Immutable`.
- In category *Lazy Initialization*, Glacier cannot enforce transitive immutability due to its rule that in `@Immutable` classes, fields may only be assigned in constructors.
- In category *String*, Glacier handles the case concerning the `char` array shared between identical strings precisely, but it cannot enforce immutability for the lazily initialized field caching the `hashCode` method's result.

> ■ *Glacier can only recognize transitive immutability compared to CiFi's fine-grained immutability results.*
> ■ *Glacier shows three cases of unsoundness.*
> ■ *While Glacier strictly enforces transitive immutability for generics, including nested and bounded generic types, it lacks the flexibility of dependent immutability to allow generic classes to be treated differently depending on whether they are instantiated with transitively immutable types or not. Additionally, Glacier does not handle lazy initialization.*
> ■ *To recap, CiFi is more soundy and often more precise than Glacier without requiring manual effort for annotations. As a result, CiFi can be applied easily to existing codebases and third-party code, even if source code is not available.*

D. Immutability Prevalence

To answer **RQ4**, we analyzed the following libraries: Open-JDK 1.8.0_292, Google Guava 30.1.1, Eclipse Collections

TABLE I
LIBRARY RESULTS ASSIGNABILITY (OPEN WORLD)

Library	ass.	unsafe l. i.	l. i.	eff. non ass.	non ass.	$\sum$
OpenJDK	26 684	351	189	8 615	58 115	93 954
Eclipse	2 380	0	0	30	11 478	13 888
Guava	656	12	0	4	3 215	3 887
Apache	275	18	0	4	652	949
Scala	1 249	0	0	4	5 373	6 626

ass. = assignable, l. i. = lazily initialized, eff. = effectively

TABLE III
LIBRARY RESULTS ASSIGNABILITY (CLOSED WORLD)

Library	ass.	unsafe l. i.	l. i.	eff. non ass.	non ass.	$\sum$
OpenJDK	22 885	435	198	12 321	58 115	93 954
Eclipse	2 380	0	0	30	11 478	13 888
Guava	598	30	0	44	3 215	3 887
Apache	269	21	0	7	652	949
Scala	1 249	0	0	4	5 373	6 626

ass. = assignable, l. i. = lazily initialized, eff. = effectively

TABLE II
LIBRARY RESULTS IMMUTABILITY (OPEN WORLD)

Library	Analysis	mutable	non-tra.	dep.	tra.	$\sum$	time (s)
OpenJDK	Field	27 035	23 004	78	43 837	93 954	5.47 (6.17)
	Class	12 398	4 259	27	5 393	22 077	
	Type	20 155	1 475	6	3 203	24 839	
Eclipse	Field	2 380	7 620	142	3 746	13 888	1.56 (2.72)
	Class	883	4 410	61	2 247	7 601	
	Type	6 186	364	41	1 057	7 648	
Guava	Field	668	1 995	35	1 189	3 887	1.06 (1.79)
	Class	636	785	17	721	2 159	
	Type	1 697	195	9	391	2 292	
Apache	Fields	293	360	18	278	949	0.81 (1.66)
	Class	262	147	7	69	485	
	Type	424	49	1	50	524	
Scala	Field	1 249	3 433	344	1 600	6 626	1.57 (5.50)
	Class	490	2 109	74	1 150	3 823	
	Type	3 331	661	60	430	4 482	

dep. = dependently immutable, tra. = transitively immutable

TABLE IV
LIBRARY RESULTS IMMUTABILITY (CLOSED WORLD)

Library	Analysis	mutable	non-tra.	dep.	tra.	$\sum$	time (s)
OpenJDK	Field	23 320	24 269	80	46 285	93 954	7.61 (8.58)
	Class	11 573	4 741	31	5 732	22 077	
	Type	13 378	5 225	35	6 201	24 839	
Eclipse	Field	2 380	7 595	142	3 771	13 888	1.92 (3.27)
	Class	883	4 397	61	2 260	7 601	
	Type	950	4 552	60	2 086	7 648	
Guava	Field	628	1 931	36	1 292	3 887	1.58 (2.35)
	Class	633	773	18	735	2 159	
	Type	715	848	20	709	2 292	
Apache	Fields	290	353	18	288	949	1.17 (1.98)
	Class	262	142	9	72	485	
	Type	294	146	9	75	524	
Scala	Field	1 249	3 314	359	1 704	6 626	2.48 (6.53)
	Class	490	2 064	96	1 173	3 823	
	Type	770	2 196	134	1 382	4 482	

dep. = dependently immutable, tra. = transitively immutable

10.4, Apache Commons Collections 4.4.4., and Scala 2.12.10. We performed the evaluation on a server with two AMD(R) EPYC(R) 7542 CPUs (32 cores / 64 threads each) @ 2.90 GHz and 512 GB RAM. For runtimes, we report the median of 15 executions as runtimes of OPAL vary significantly. CiFi was run using OpenJDK 1.8.0_292, Scala 2.12.13, and the Scala build tool sbt 1.4.6 with 32 GB of heap memory. In this experiment, we applied an open-world assumption. To ease analysis, OPAL replaces `invokedynamic` bytecode instructions with synthetic fields and classes that are also included in the result figures. The number of fields having the respective levels of assignability and the total number of analyzed fields are shown in Table I. Results for the field-, class-, and type-immutability analyses are given in Table II, listing the number of entities with respective levels of immutability, total count and execution time for all analyses combined. Total runtime including preparatory steps, e.g., loading the libraries' files is given in parentheses. While the numbers for types include interfaces, those for classes do not (interfaces don't contain potentially mutable state).

The results provide empirical evidence that most of the immutability properties defined in Section III are prevalent in real-world libraries. Even if absolute numbers for dependent immutability appear to be low, one has to consider that generic classes are often widely used collections and thus can have a significant impact. We found several hundreds of safely

and unsafely lazily initialized fields in the JDK and some in Guava and Apache Commons Collections, but none in Eclipse Collections. We studied the latter library's source code and indeed Eclipse Collections seems not to use any lazy initialization at all. CiFi does not (yet) handle Scala's `lazy val`, but lazy initialization is a prominent feature of the Scala language, too. We can also see that all libraries have significant quantities of *(effectively) non-assignable* fields; OpenJDK has about 46% of transitively immutable fields, while the other libraries have mostly non-transitively immutable fields. All libraries also have significant shares of (non-)transitively and dependently immutable classes, ranging from 43% to 88%. To recap, the results presented so far signify the relevance of our immutability model in practice. The properties of the model prevail despite the fact that CiFi over-approximates the model in several cases (cf. **RQ2**) and that it was executed with a conservative open-world assumption. To investigate the effect of the latter, we re-executed CiFi on the same libraries with the same setup but with a closed-world assumption. Results for field assignability are given in Table III and for the other analyses in Table IV. Comparing to the open-world scenario (cf. Tables I and II), we make the following observations.

First, the number of types with stronger immutability guarantees increases significantly. This is to be expected, as no subclasses can be added in the closed-world scenario. Second, the impact on the number of fields and classes found to

exhibit different levels of assignability and immutability is minimal. Differences are most significant for OpenJDK, where 14.2% of formerly assignable and 13.7% of formerly mutable fields and 6.7% of formerly mutable classes exhibit stronger guarantees for assignability or immutability, respectively. The increased number of types with stronger immutability guarantees does not proportionally influence field immutability due to the high percentage of fields with primitive types or type `java.lang.String` (e.g, > 50% in OpenJDK). Third, the runtime increased by 23% up to 58%. This is because in an open-world scenario, we avoid performing expensive computations, e.g., for extensible types or for protected non-final fields in extensible classes, which are just *mutable*.

> ■ *All immutability levels and flavors of our model are prevalent in real-world libraries. This means that (a) the definitions in our model reflect immutability in practice and (b) the versatile inference of CiFi is needed to consider fine-grained levels and diverse flavors of immutability.*
> ■ *Except for type immutability, applying an open-world assumption does not seem to significantly reduce precision while consuming significantly less computation time. Thus, it may be beneficial to use an open-world assumption even if all program code is available. CiFi gives users the flexibility to choose between an open- and a closed-world assumption.*

VI. Threats to Validity

An internal validity threat arises if *CiFi-Bench* does not cover relevant aspects of class and/or field immutability, or if its test cases are annotated incorrectly. To mitigate this threat, tests were created by one author based on the literature survey and checked by a second author. All authors have years of experience in static analysis and immutability research.

An external threat to validity arises if the libraries used for evaluation are not representative of real-world immutability. However, we chose well-known libraries of significant size that include a significant number of data structures many of which are documented to be immutable. The Scala standard library also provides an insight into immutability for Java Virtual Machine bytecode not compiled from Java source code.

VII. Related Work

In this section, we discuss approaches to object and reference immutability. They are related but not the main focus of this work. We surveyed approaches that, like our work, address class and field immutability in Section II.

Haack et al. [4] distinguish *observational* and *state-based* immutability. Observational immutability describes that an observer is not able to see any difference in an object at any two points in time after its initialization. State-based immutability describes that the internal state of an object does not change at all. Like in our model, for state-based immutability, the distinction is made between transitive and non-transitive immutability. Haack et al. express their belief that observational immutability is more intuitive while state-based immutability is better-suited for static analysis. This is in line with our approach which also considers state-based immutability.

Potanin et al. [1] distinguish between *abstractly immutable* objects that may change their internal representation while preserving their semantics as visible by their clients and *representationally immutable* objects that never change their internal representation. This corresponds to observational and state-based immutability as used by Haack et al.

Zibin et al. [11] enforce transitive immutability of fields that belong to an object's abstract state with their language extension *Immutability Generic Java (IGJ)* that uses Java generics to describe the immutability of a class through an additional type parameter (*Mutable*, *Immutable*, or *ReadOnly*).

Ownership Immutability Generic Java (OIGJ) [24] by Zibin et al. uses ownership to enforce object immutability. As only an object's owner can mutate it, it is easy to check for mutations if the owner is known. Leino et al. [25] also use ownership to *freeze* any object at any time during program execution. When an object is frozen, its owner is changed to be the *freezer object*. As that object is not exposed to the rest of the program, and as changing fields requires ownership, the frozen object becomes immutable and cannot be unfrozen again. This applies to objects owned transitively by the frozen object as well.

For references, the *readonly* property has been studied extensively [8]–[12], [26]–[28]. Tschantz and Ernst use it in the *Javari* type system [9]. Through a *readonly* reference, the referenced object and all transitively referenced objects belonging to the abstract state of the referenced object cannot be mutated, while they may still be mutated through other references. Thus, *readonly* is different from the *transitive immutability* property – the latter requires the referenced object, including all transitively referenced objects, to be immutable through any reference. Additionally, a *romaybe* modifier expresses polymorphic immutability of references, i.e., whether the reference returned by a method is mutable or not depends on the context in which the method is accessed and whether the object referred to by this reference, also transitively, is mutated or not. That is, a method may return a potentially mutable but not yet escaped object as *romaybe*, allowing the caller to treat it as immutable or mutate it. To support lazy initialization, it is possible to exclude lazy initialized fields from the abstract state in Javari (cf. [8]). Gordon et al. [2] describe a similar concept to *readonly*, but use the term *readable* instead.

Huang et al. use *Javari* as a basis for their type system *ReIm* and their immutability and purity analysis *ReImInfer* [8]. However, they use *polyread* instead of *romaybe*. Additionally, while *Javari*'s `readonly` modifier refers to the abstract state, here `readonly` applies to the concrete state of the referenced object, i.e., it includes all fields and referenced objects.

Milanova and Dong [29] build upon *ReIm* to infer and check object immutability by combining a reference immutability analysis with escape analysis. They consider transitive immutability, enforcing that no transitively referenced values, objects, or arrays of an `immutable` object are mutated. They also address delayed object initialization with their `endorse` modifier for statements. This results in the analysis ignoring the

statement's effects on immutability, which is, e.g., necessary to support circular initialization. With the *unstrict* block, Gordon et al. [2] present a similar approach.

Quinonez et al. [10] find it "tedious and error-prone" to manually add modifiers like `readonly` to existing code bases. They propose to infer them automatically with *Javarifier*, which can also infer *Javari*'s modifiers for arrays and their values as well as for the type parameters of generic classes.

Boyland [30] cautioned against adding *readonly* to the Java language because its transitive rule would be too restrictive while it cannot prevent harmful observational exposure, i.e., the state of a mutable field can be seen via a readonly reference while it can be modified through another reference. This leads to problems, e.g., in multi-threading contexts or when a client expects a non-mutable object. Our model is in line with Boyland and considers the immutability of an entire class rather than the immutability through a given reference. This avoids harmful observational exposure because a transitively immutable class has only transitively immutable instances.

VIII. Conclusion

We proposed a comprehensive, fine-grained lattice model for field assignability and for field, class, and type immutability. Based on a literature survey, the model unifies the terminology of the research area, which has so far been used inconsistently. Unlike the state of the art, the model distinguishes between these different flavors of immutability and provides levels of immutability to represent relevant aspects such as lazily initialized fields and dependent immutability for generic classes. As we have shown, our model covers a wider range of immutability than previous models. Accompanying this model, we provide *CiFi-Bench*, a handcrafted set of test cases to serve as a ground truth for class- and field-immutability analyses. We introduced CiFi, a set of modular analyses for each of the immutability flavors of our model. We used *CiFi-Bench* to showcase CiFi's precision and recall, then used CiFi to study the prevalence of immutability in real-world libraries.

In future work, we plan to investigate possibilities to increase CiFi's precision further without degrading its runtime performance disproportionately. This may include the ability to precisely find more lazy initialization patterns and additional support for generic type parameters, e.g., regarding their instantiation or their statically provided bounds. It is also possible to extend CiFi with further modular analyses, e.g., for object or reference immutability.

Acknowledgments

References

[1] A. Potanin, J. Östlund, Y. Zibin, and M. D. Ernst, "Immutability," in *Aliasing in Object-Oriented Programming*. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 233–269.

[2] C. S. Gordon, M. Parkinson, J. Parsons, A. Bromfield, and J. Duffy, "Uniqueness and Reference Immutability for Safe Parallelism," Microsoft Research, Tech. Rep. MSR-TR-2012-79, October 2012.

[3] P. Helland, "Immutability changes everything," *Communications of the ACM*, vol. 59, no. 1, pp. 64–70, 2015.

[4] C. Haack, E. Poll, and A. Schubert, "Immutable objects in Java," in *Programming Languages and Systems*, ser. ESOP'07. Springer Berlin Heidelberg, 2007, pp. 347–362.

[5] Oracle. (2020, Sep.) Secure Coding Guidelines for Java SE. Oracle. https://www.oracle.com/java/technologies/javase/seccodeguide.html.

[6] Oracle. (2021) Map (Java SE 16 & JDK 16). https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Map.html. Oracle.

[7] D. Helm, F. Kübler, M. Eichberg, M. Reif, and M. Mezini, "A Unified Lattice Model and Framework for Purity Analyses," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE'18. New York, NY, USA: ACM, 2018, pp. 340–350.

[8] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst, "Reim & ReImInfer: Checking and Inference of Reference Immutability and Method Purity," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA'12. New York, NY, USA: ACM, 2012, pp. 879–896.

[9] M. S. Tschantz and M. D. Ernst, "Javari: Adding reference immutability to Java," in *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, San Diego, CA, USA, October 18–20, 2005, pp. 211–230.

[10] J. Quinonez, M. S. Tschantz, and M. D. Ernst, "Inference of reference immutability," in *ECOOP 2008 — Object-Oriented Programming, 22nd European Conference*, ser. ECOOP'08, Paphos, Cyprus, Jul. 2008, pp. 616–641.

[11] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kie—un, and M. D. Ernst, "Object and Reference Immutability Using Java Generics," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE'07. New York, NY, USA: ACM, 2007, pp. 75–84.

[12] J. T. Boyland, J. Noble, and W. Retert, "Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only," in *ECOOP*, 2001.

[13] M. Coblenz, J. Sunshine, J. Aldrich, B. Myers, S. Weber, and F. Shull, "Exploring language support for immutability," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 736–747.

[14] M. Coblenz, W. Nelson, J. Aldrich, B. Myers, and J. Sunshine, "Glacier: Transitive class immutability for Java," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 496–506.

[15] S. Porat, M. Biberstein, L. Koved, and B. Mendelson, "Automatic detection of immutable fields in Java." in *CASCON*, 2000, p. 10.

[16] S. Nelson, D. J. Pearce, and J. Noble, "Profiling field initialisation in Java," in *International Conference on Runtime Verification*. Springer, 2012, pp. 292–307.

[17] F. B. Kjolstad, D. Dig, G. Acevedo, and M. Snir, "Refactoring for immutability," University of Illinois, Tech. Rep., 2010.

[18] D. Helm, F. Kübler, M. Reif, M. Eichberg, and M. Mezini, "Modular collaborative program analysis in OPAL," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE'20, 2020, pp. 184–196.

[19] G. Bierman. (2021, Mar) JEP 395: Records. Oracle. https://openjdk.java.net/jeps/395.

[20] M. Reif, M. Eichberg, B. Hermann, J. Lerch, and M. Mezini, "Call graph construction for Java libraries," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ISSTA'16, 2016, pp. 474–486.

[21] M. Eichberg, F. Kübler, D. Helm, M. Reif, G. Salvaneschi, and M. Mezini, "Lattice Based Modularization of Static Analyses," in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ser. ISSTA '18. New York, NY, USA: ACM, 2018, pp. 113–118.

[22] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: A manifesto," *Communications of the ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[23] A. Prokopec, D. Petrashko, M. Garcia, J. Zaugg, H. Plociniczak, V. Klang, and M. Odersky. (2021, Apr) SIP-20 - Improved Lazy Vals Initialization. https://docs.scala-lang.org/sips/improved-lazy-val-initialization.html.

[24] Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst, "Ownership and Immutability in Generic Java," *SIGPLAN Not.*, vol. 45, no. 10, pp. 598–617, Oct. 2010.

[25] K. R. M. Leino, P. Müller, and A. Wallenburg, "Flexible immutability with frozen objects," in *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 2008, pp. 192–208.

[26] A. Birka and M. D. Ernst, "A practical type system and language for reference immutability," *ACM SIGPLAN Notices*, vol. 39, no. 10, pp. 35–49, 2004.

[27] W. Dietl and P. Müller, "Universes: Lightweight Ownership for JML." *Journal of Object Technology*, vol. 4, no. 8, pp. 5–32, 2005.

[28] G. Kniesel and D. Theisen, "JAC—access right based encapsulation for Java," *Software: Practice and Experience*, vol. 31, no. 6, pp. 555–576, 2001.

[29] A. Milanova and Y. Dong, "Inference and Checking of Object Immutability," in *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, ser. PPPJ'16. New York, NY, USA: ACM, 2016, pp. 6:1–6:12.

[30] J. Boyland, "Why we should not add readonly to Java (yet)." *Journal of Object Technology*, vol. 5, no. 5, pp. 5–29, 2006.