Transitive, Abstract, and Class Polymorphic Immutability

AOSEN XIONG, University of Waterloo, Canada
YUDI BAI, University of Waterloo, Canada
HAIFENG SHI, University of Waterloo, Canada
LIAN SUN, University of Waterloo, Canada
MIER TA, University of Waterloo, Canada
WERNER DIETL, University of Waterloo, Canada

State mutations can often lead to silent program errors, including broken invariants and security vulnerabilities. Object-oriented languages offer basic mechanisms to prevent mutation; however, enforcing desired guarantees remains challenging. Two such guarantees are *transitive immutability*, which disallows mutation of all objects reachable from a reference, and *abstract immutability*, which permits controlled mutation of otherwise immutable objects. Furthermore, introducing readonly references to support subtype polymorphism often complicates the soundness of the type system. The integration of immutability into a class hierarchy introduces challenges, primarily manifesting as duplicated code between mutable and immutable variants.

We present Precise Immutability for Classes and Objects (PICO), a type system that enforces transitive abstract immutability with readonly references. PICO introduces novel viewpoint adaptation rules to achieve transitivity. These rules prevent unsoundness caused by mutable and immutable cross-type aliasing, a long-standing issue for systems combining immutability and assignability. Additionally, PICO formally defines the abstract state, which allows developers to permit mutation for selected parts of the object graph. PICO provides four state-preservation guarantees within a single system by selecting corresponding viewpoint adaptation rules: abstract-, concrete-, readonly-, and transitive-state preservation. Finally, the system supports safe *class mutability polymorphism*: one class can express both mutable and immutable uses, avoiding duplicate mutable/immutable class variants while also enabling backward-compatible retrofitting of existing hierarchies.

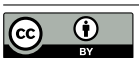
We formalize PICO and prove its type soundness and four state-preservation guarantees in the Rocq proof assistant. We also implement a type checker for Java using the Checker Framework. We evaluate this implementation on the Java Collections Framework in OpenJDK 17 and other benchmarks, covering approximately 26,000 non-comment lines of code. The results demonstrate that PICO effectively enforces immutability guarantees and can successfully retrofit existing libraries without duplicating code.

CCS Concepts: • **Software and its engineering** → *Object oriented languages*; *Software reliability*; • **Theory of computation** → **Program semantics**; **Type theory**; *Program analysis*.

Additional Key Words and Phrases: Readonly references, Side effects, Type systems, Mechanized proof

ACM Reference Format:
Aosen Xiong, Yudi Bai, Haifeng Shi, Lian Sun, Mier Ta, and Werner Dietl. 2026. Transitive, Abstract, and Class Polymorphic Immutability. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 359 (October 2026), 41 pages. https://doi.org/10.1145/3839491

Authors' Contact Information: Aosen Xiong, aosen.xiong@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada; Yudi Bai, yudi.bai@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada; Haifeng Shi, h223shi@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada; Lian Sun, l82sun@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada; Mier Ta, m2ta@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada; Werner Dietl, wdietl@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada.



ACM 2475-1421/2026/10-ART359
https://doi.org/10.1145/3839491

1 Introduction

Unintended state mutations are a common source of software defects, leading to unexpected behaviour [4, 39], security vulnerabilities [34], and concurrency issues [21, 44]. Immutability ensures that an object's state remains unchanged after its creation. It offers strong guarantees about the object's behaviour and is useful for documentation, API design [49], and program verification [11, 13, 29]. Transitive immutability, where all objects reachable from a reference are immutable, is desirable compared to shallow immutability, which only ensures that the object's immediate fields cannot be modified. However, most mainstream object-oriented (OO) programming languages, while providing modifiers such as Java's *final* and Kotlin's and Scala's *val* to prevent reassignment, cannot statically enforce transitive immutability. Consequently, developers fall back on programming guidelines, such as Effective Java's recommendation [4] to "Minimize Mutability" by making all fields *final* and prohibiting class extension. Although this approach achieves immutability, it restricts inheritance and limits code reuse and patterns such as lazy initialization. Recent OO languages propose value classes and objects [1, 43] to remedy the situation, but transitive immutability is still not guaranteed, as these constructs only enforce shallow immutability and do not prevent value objects from holding references to mutable objects. This absence of transitive immutability often leads to subtle, hard-to-debug errors. For example, an immutable `Person` may still expose a mutable nested `Address`:

```
Person person = new Person("Sherlock", // Assume Person is immutable
    new Address("221B Baker Street", "London", "NW1 6XE"));
// Mutates transitive state!
person.getAddress().setStreet("222B Baker Street");
```

Even if `Person` is intended to be immutable, mutating its nested `Address` still changes `Person`'s transitive state. Programmers need a guarantee of transitive immutability, not only shallow immutability.

Object-immutability systems such as IGJ and OIGJ introduce readonly references for subtype polymorphism: a readonly reference can point to either a mutable or an immutable object [51, 52]. A readonly reference prevents mutation through that reference but does not prevent another mutable alias from modifying the same object.

Type qualifiers [19] augment an existing type system with additional properties, such as nullability, interning, and tainting. A few frameworks [2, 12, 37] have been developed to introduce pluggable type checking [6] into OO languages. Several type qualifier systems have been proposed to add immutability to OO languages, including Javari [48], Jimuva [23], IGJ [51], OIGJ [52], L42 [20], ReIm [24], Constrictor [28], and others [39]. However, existing systems have several limitations. We discuss these limitations and address the challenges in this work.

Challenge 1: *Sound Transitive Immutability with Readonly References and Assignability.*

Transitivity is usually achieved by viewpoint adaptation [13], where the result type is determined by adapting the declared field type from the receiver's type. To have transitivity, if the receiver is mutable/immutable, the result type should be mutable/immutable. But if the receiver is readonly, should the result type be readonly? Suppose a readonly reference `ro` points to an immutable object `imm`, so a field with an immutable type, such as `imm.f`, should refer only to immutable objects. If the field is declared assignable, meaning its slot can normally be reassigned independently of the receiver's mutability, and a field access through `ro` is typed as readonly, an assignment such as `ro.f = mut` can capture a mutable object in `imm.f`—immutable and mutable references refer to the same object, breaking type soundness. This issue, known as cross-type aliasing, must be avoided to ensure soundness.

Challenge 2: *Abstract Immutability.*

While transitive immutability is desired in many scenarios, controlled mutation is necessary for performance optimizations, such as lazy initialization and caching. Immutability is often relaxed to allow selected mutations, and an object's immutability is defined based on its abstract state [48], where selected parts of the object graph may permit mutation. Formally defining the abstract state and the guarantees this relaxed immutability provides is challenging.

Challenge 3: *Class Mutability Polymorphism.*

Integrating immutability into a class hierarchy introduces structural challenges, making it difficult to introduce immutable types without duplicating the entire class hierarchy. Also, existing widely used libraries, such as the Java Collections Framework (JCF), throw runtime exceptions when mutations occur on immutable collections. Retrofitting mutability in such a design is challenging for practical immutability systems. Languages such as Scala [42] and Kotlin[1] address related concerns by separating collection interfaces according to mutability. This approach is effective, but it introduces additional code due to separate interfaces and a converter to convert mutable objects to immutable ones. Rust [33] uses ownership and borrowing to have stronger properties, which is difficult to adopt for OO languages' legacy codebases because it requires significant code refactoring to a different programming style.

We solve these three challenges in PICO. While our system guarantees immutability, it intentionally leaves object aliasing unrestricted; concepts such as ownership and uniqueness are orthogonal concerns that can be implemented as add-ons to our system, following a modular approach, like in OCaml [32]. Our specific contributions are summarized as follows.

(1) Sound Transitive Immutability: We introduce novel viewpoint adaptation rules and a new `@Lost` qualifier to achieve sound transitive immutability with readonly references.
(2) State-Preservation Guarantees: We formally define and prove abstract-, concrete-, readonly-, and transitive-state preservation by selecting corresponding viewpoint adaptation rules.
(3) Class Mutability Polymorphism: Our design ensures safe instantiation and subclassing of mutability-polymorphic classes, without duplicating the class hierarchy.
(4) Language Formalization: We formalize our system and prove its soundness and four state-preservation guarantees in the Rocq proof assistant [47].
(5) System Evaluation: We implement a type checker and conduct a case study using the JCF and other benchmarks, covering approximately 26,000 lines of non-comment code.

The remainder of this paper is organized as follows. Section 2 introduces the design of our system with code examples. Section 3 presents the core calculus, including syntax, static viewpoint adaptation, well-formedness, and type rules. Section 4 introduces the runtime model, the operational semantics, and the proofs of soundness and four state-preservation guarantees. Section 5 describes the implementation and case study on JCF and benchmarks, and also discusses our system's design trade-offs. Section 6 compares our system with related works, and Section 7 concludes.

2 Language Design

This section presents PICO's design informally through a series of examples. We use a Java-like syntax because our formalization is based on Featherweight Java [25] and our type checker targets Java, though the design applies similarly to other OO languages. The examples use Java 8 or later syntax. In code examples, `...` denotes an omitted expression or argument list and is not literal Java syntax. For readability, we use abbreviated annotation names.

[1]Kotlin's standard library distinguishes read-only and mutable collection interfaces; immutable persistent collections are provided separately by `kotlinx.collections.immutable` [26].

2.1 Qualifiers and Shallow Immutability

In the following code snippet, we create a mutable (abbreviated `@Mut`) object and an immutable (abbreviated `@Imm`) object, and then use a `@Mut` reference `mutRef` and an `@Imm` reference `immRef` to refer to them, respectively. We also use a readonly reference (abbreviated `@RO`) `roRef` to refer to both. PICO's mutability qualifiers are Java type-use annotations [27]. Immutable and mutable references are not subtypes of each other, and thus cannot be assigned to each other.

```
@Mut Object mutRef = new @Mut Object(); // create a mutable object
@Imm Object immRef = new @Imm Object(); // create an immutable object
@RO Object roRef = mutRef; // ok, @RO can refer to a mutable object
roRef = immRef; // ok, and to an immutable object
immRef = mutRef; // error: cannot assign, @Mut is not a subtype of @Imm
```

Assignability modifiers apply only to field declarations. A field slot may either be fixed after initialization or remain reassignable. PICO writes these alternatives as final (`@Final`, our notation for Java's `final` modifier) and assignable (abbreviated `@Assign`), respectively.

```
class Box {
  @Final @Imm int i1 = 1;
  @Assign @Imm int i2 = 2; }
@Mut Box mutRef = new @Mut Box(); // create a mutable box
mutRef.i1 = 3; // error: cannot assign to a final field of a mutable object
mutRef.i2 = 4; // ok
```

In the `Box` example, `i1` is `@Final` so it cannot be reassigned even through a mutable reference, while `i2` is `@Assign` and can be freely reassigned. Together, these two modifiers specify whether a field slot may be reassigned, independently of the receiver's mutability.

2.2 Transitive Immutability and Context Sensitivity

Transitive immutability means that a reference should not allow writes anywhere along its reachable object graph. Context sensitivity means that the mutability and assignability of fields depend on the receiver's mutability. To capture both behaviours, we introduce receiver-dependent forms of mutability and assignability. Receiver-dependent mutability (abbreviated `@RDM`) determines the effective mutability of a field's type from the receiver's mutability, while receiver-dependent assignability (abbreviated `@RDA`) determines whether the field slot is effectively assignable or final. To support transitive immutability, fields default to `@RDA`. Programmers use `final` or `@Assign` explicitly when assignability should not depend on the receiver.

```
class Box { @RDA @Imm int i3 = 3; }
class OuterBox { @RDA @RDM Box box = ...; }
@Mut OuterBox mutOuterRef = new @Mut OuterBox();
mutOuterRef.box = new @Mut Box(); // ok, box is effectively assignable
mutOuterRef.box.i3 = 5; // ok, i3 is effectively assignable
@Imm OuterBox immOuterRef = new @Imm OuterBox();
immOuterRef.box = new @Imm Box(); // error, box is effectively final
immOuterRef.box.i3 = 5; // error, i3 is effectively final
```

In the example above, `@RDM` in `OuterBox` makes the type of the reference stored in `box` effectively `@Mut` or `@Imm` based on the receiver. In `Box`, `@RDA` makes the `i3` slot effectively assignable through a mutable receiver and final through an immutable receiver. The `box` field uses both receiver-dependent annotations, so both properties depend on its receiver. For this running example, we assume both `Box` and `OuterBox` can be instantiated as either `@Mut` or `@Imm`. We discuss class mutability polymorphism in detail in Section 2.5. This mechanism is called *viewpoint adaptation* [13,

15]: the receiver's qualifier determines the effective type of fields declared with an `@RDM` type and the effective assignability of fields declared `@RDA`. For *assignability*, any declared modifier other than `@RDA` is left unchanged; `@RDA` adapts to `@Assign` if the receiver is `@Mut`, and to `@Final` otherwise. For *mutability*, any declared qualifier other than `@RDM` is left unchanged; `@RDM` adapts to the receiver's qualifier (with one exception for `@RO` receivers, discussed in Section 2.3). The formal rules are presented in Section 3.2. Different rule variants accommodate different state-preservation guarantees, which will be discussed in Section 2.4. Together, the example shows transitive immutability and context sensitivity along the path `OuterBox.box.i3`.

Method Receiver Typing. The preceding examples show how field typing enforces transitive immutability and expresses context sensitivity for direct field writes. But what if the class has a method that mutates object state? We extend `OuterBox` with an instance method `setBox` that assigns a new value to the field `box`. To prevent accidental mutation, we specify the mutability qualifier of the method receiver parameter using Java 8's explicit receiver parameter syntax [22]. In `setBox`, the declaration `@Mut OuterBox this` means that the method can only be invoked on a mutable receiver argument. As a consequence, `immOuterRef` cannot invoke `setBox`. If the method is declared with an `@Imm` receiver parameter, the method body cannot assign to `box` because the receiver parameter is immutable.

```
class OuterBox {
  void setBox(@Mut OuterBox this, @Mut Box box) {
    this.box = box; // ok, the receiver is mutable and thus box is assignable
} }
immOuterRef.setBox(...); // error, receiver argument is immutable
mutOuterRef.setBox(...); // ok
```

Context sensitivity is also useful for method receiver parameters, where it avoids code duplication. Consider extending `OuterBox` with a `getBox` method. Can we get a mutable object from a mutable receiver, an immutable object from an immutable receiver, and a readonly object from a readonly receiver? One solution is to have multiple method overloads, each with different mutability qualifiers. However, this approach can lead to code duplication. Languages like Java do not allow method overloading based solely on differences in annotations, so we would need to use different method names for each overload, which can be confusing and error-prone. In our system, we can simply declare the method with a `@RDM` receiver and return type. The single `@RDM` declaration in the last line below replaces the three boilerplate methods above it:

```
@RO  Box getBoxRO (@RO  OuterBox this) { return this.box; }
@Mut Box getBoxMut(@Mut OuterBox this) { return this.box; }
@Imm Box getBoxImm(@Imm OuterBox this) { return this.box; }
// all three replaced by
@RDM Box getBox   (@RDM OuterBox this) { return this.box; }
```

For a readonly receiver, viewpoint adaptation gives `this.box` a type with the helper qualifier `@Lost`, which records unknown underlying mutability. Because `@Lost` <: `@RO`, the result can be used as readonly but not as mutable or immutable. In summary, the method receiver parameter allows programmers to specify the receiver's mutability, thereby controlling the mutability of the method body and preventing accidental mutation. Context-sensitive receiver parameters can reduce code duplication.

2.3 Sound Transitive Immutability

Sound transitive immutability needs additional design consideration when interacting with read-only references and assignability. In systems like IGJ [51] and OIGJ [52], accessing an `@RDM` field

through a `@RO` reference results in a `@RO` type, and in systems like L42 [20], SIFO [41], and an experimental extension for C# [21], accessing a `@Mut` field through a `@RO` reference results in a `@RO` type. If these systems were combined with assignability modifiers, they would permit unsound assignments, resulting in cross-type aliasing [13, 46]. For example, in the code below we introduce a field `assignableBox` and a `swapField` method in the `OuterBox` class. The `swapField` method assigns the `assignableBox` field of `box1` to the `assignableBox` field of `box2`. Calling `swapField(immOuterRef, mutOuterRef)` creates a cross-type alias: an immutable reference and a mutable reference refer to the same object. Capturing a mutable object in an immutable object's `@RDM` field is unsound because the state can be mutated through the mutable reference while it is expected to be immutable.

In our system, when accessing a `@RDM` field from a `@RO` reference, the mutability qualifier is `@Lost`, similar to the one introduced by GUT [13]. The intuition is that we do not know whether the object referred to by the `@RO` reference is mutable or immutable, and the mutability information of the field is lost. Our system disallows `@Lost`-to-`@Lost` assignments, and thus prevents the unsoundness. One simpler alternative is to ban all writes through `@RO` receivers, but this would reject benign updates to fields outside the abstract state, such as memoization caches or access-order bookkeeping, which we discuss in Section 2.4.

```
class OuterBox {
  @Assign @RDM Box assignableBox = ...;
  void swapField(@RO OuterBox box1, @RO OuterBox box2) {
    // error: @Lost-to-@Lost assignment is forbidden
    box1.assignableBox = box2.assignableBox;
} }
// without the error above, this call would alias the two fields
swapField(immOuterRef, mutOuterRef);
```

2.4 Four State-Preservation Guarantees

Transitive immutability is a desirable property for immutability systems. However, real-world code often uses caching or lazy initialization to optimize performance, and such updates should not invalidate an object's immutability. We therefore make precise which parts of an object graph must remain unchanged: PICO defines four *states*, each a set of field slots.

Each state is fixed by two independent choices: which objects it ranges over, and which of their field slots it contains. For the abstract state and the concrete state, the objects are those reachable from a *root object* through field slots whose assignability modifier is `@RDA` or `@Final` and whose mutability qualifier is `@RDM` or `@Imm`. We call these the *abstract-state-reachable* objects. For the readonly state and the transitive state, the roots are the receiver and the arguments of a method call. The transitive state ranges over every object transitively reachable from them, whereas the readonly state ranges only over the objects reachable through non-`@Assign` field slots. Reachability is reflexive in all cases, so the roots are always included. On the slot dimension, the abstract state and the readonly state contain only the non-`@Assign` field slots of those objects, whereas the concrete state and the transitive state contain every field slot. For brevity, we say that an object is *in* a state when the state ranges over it. A state is a set of heap slots, so its roots are objects, and the static qualifier of a reference affects viewpoint adaptation rather than the state itself. AS and CS constrain direct field writes, including through readonly references, and preserve the abstract state and the concrete state of an immutable object. RS and TS additionally constrain method invocations, bounding what the callee may change in the graph reachable from the receiver and the arguments.

Programmers use `@Assign` to permit reassignment of a field slot and `@Mut` to permit mutation of the referenced object. The `ImmContainer` class below uses each for a common optimization pattern.

First, the `@Assign index` field caches the last look-up index for an underlying collection, so a value can be retrieved quickly if the next query targets the same position. Second, the `@Mut table` field acts as a memoization cache, storing the results of expensive computations so that the value only needs to be evaluated once when a specific query first occurs.

```
class ImmContainer {
  @Assign @Imm int index; // the field slot can be reassigned
  @Final @Mut HashTable table; // points to a mutable object
}
```

Method Scopes. If we allow selected fields to be mutated, what guarantees do we have for immutability? PICO provides four guarantees by independently choosing mutability and assignability adaptation: abstract-state preservation (AS) is the baseline, concrete-state preservation (CS) strengthens assignability, readonly-state preservation (RS) strengthens mutability, and transitive-state preservation (TS) strengthens both. The corresponding method scope annotations are `@AS`, `@CS`, `@RS`, and `@TS`. The mechanism can be generalized to select rules for classes, packages, and modules.

Moving downward in Figure 1 gives stronger guarantees: AS is weakest and TS is strongest. This hierarchy also governs calls: a method may call only methods with equal or stronger guarantees. RS and CS are incomparable because they strengthen different adaptation dimensions.

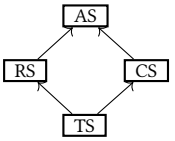


Fig. 1. Method scopes.

We illustrate the difference between AS and CS by showing the abstract state within the full object graph reachable from an immutable object in Figure 2. `Child 1` is referenced by the abstract state field `f_abs` of the immutable object `Root`, so it is immutable. `Child 2` is referenced by the non-abstract state field `f_aux`, and under AS that field slot can be reassigned to another object. For immutable objects, the AS viewpoint adaptation rules ensure that objects in the abstract state are immutable and that there is no mutable aliasing to them. For example, the field `f_alias1` from the `Root` object and the field `f_alias2` from `Child 2` cannot refer to `Child 1` because `@Imm` is not a subtype of `@Mut`. Because of the viewpoint adaptation rules, mutations to the abstract state fields of `Child 1` and `Root` are not possible, so the abstract state is preserved.

CS selects stricter assignability adaptation: a field selected through a non-`@Mut` receiver is effectively `@Final`, so `f_aux` is fixed under CS but reassignable under AS. The RS and TS rules presented in the figure are for reference only and are explained in the next paragraph.

Readonly-State and Transitive-State Preservation. To characterize the side effects of a method call, we ask which parts of the pre-call object graph remain unchanged, taking the receiver and each argument as a root object. RS preserves their readonly state, whereas TS preserves their transitive state. These two guarantees are harder to establish than AS and CS, because the object referenced by a readonly reference may be mutable, and mutable aliases may reach its abstract state. In Figure 3, the readonly reference `xro` refers to a mutable `Root` object. `Child 1` is referenced both by the abstract state field `f_abs` and by `f_alias1` and `f_alias2`, whose declarations do not satisfy the abstract state constraints. Consequently, `Child 1` belongs to the abstract state but is also reachable through mutable references.

RS strengthens the AS mutability adaptation: a declared `@Mut` remains `@Mut` from a `@Mut` viewpoint and otherwise becomes `@Lost`. The other rules are unchanged. The intuition is that a field declared `@Mut` could expose a mutable reference to an object in the abstract state. Thus references to `Child 1` obtained within the RS scope are non-mutable. Conservatively, so are references to non-abstract

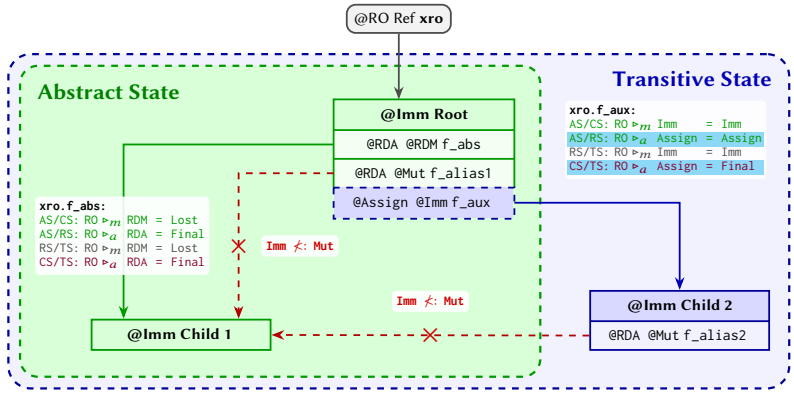


Fig. 2. Abstract State, Concrete State, and Transitive State for `@Imm Root`. Solid-line boxes denote objects and their fields, which have a mutability qualifier, an assignability modifier, and a name. `xro` is a `@RO` reference to `Root`, and each rule table shows how a field access through it adapts. The green dashed-line box highlights the objects in the abstract state of `Root`, and the blue dashed-line box highlights all objects transitively reachable from `Root`. Each object takes the colour of the innermost dashed-line box containing it. The slot `f_aux` has a dashed border because it is outside the abstract state, which is also why `Child 2` is excluded. The concrete state has no box of its own and differs only in the cyan assignability rows, where CS fixes the `f_aux` slot as `@Final` while AS leaves it `@Assign`. The green solid arrow is drawn from an abstract state field and is immutable, the blue solid arrow points to an object outside the abstract state that can be updated, and the two red dashed arrows are invalid references. Each field edge is labelled with viewpoint adaptation rules and results, where the m and a subscripts denote mutability and assignability.

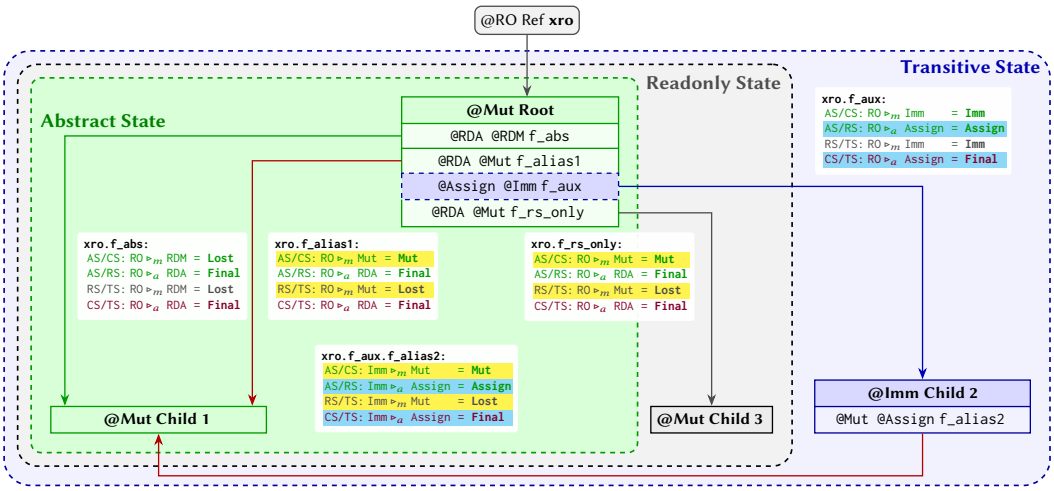


Fig. 3. Abstract State, Concrete State, Readonly State, and Transitive State for mutable `Root`. The notation is as in Figure 2, and the rule tables compare AS, CS, RS, and TS. `Child 2` is blue because the `@Assign` slot `f_aux` places it in the transitive state only. As in Figure 2, the concrete state has no box of its own. In addition, mutable `Root` and `Child 2` can have a valid mutable reference to `Child 1`, which we highlight with red solid arrows. The grey solid arrow leads to `Child 3`, which is in the readonly state but outside the abstract state. We highlight rows that differ in mutability in yellow and rows that differ in assignability in cyan.

state objects such as `Child 3`, which is in the readonly state because `Root` reaches it through the non-`@Assign` field `f_rs_only`. `Child 2`, in contrast, is reachable only through the `@Assign` slot `f_aux`, so it is outside the readonly state altogether and RS does not constrain it. TS combines the RS mutability restriction with the CS assignability restriction. TS therefore preserves every field slot of every object transitively reachable from the receiver or arguments before the call. To enforce the RS and TS guarantees soundly, a method of either scope cannot have an `@Mut` receiver or formal parameter, either of which could otherwise mutate the protected object graph. All four guarantees constrain only the objects that already exist before the statement or call, so a method in any scope may still allocate new objects and mutate them.

2.5 Class Mutability Polymorphism

Class mutability polymorphism allows a single class definition to be instantiated as either mutable or immutable, enabling code reuse without duplicating logic, like the `Box` and `OuterBox` classes we used in Section 2.2. This is valuable in new codebases: rather than writing separate immutable and mutable classes with identical structure, a single mutability-polymorphic class can be used to instantiate both immutable and mutable instances. Beyond new code, this design also supports backward compatibility: existing classes can be retrofitted with mutability qualifiers without duplicating the classes. In this subsection, we discuss the class mutability polymorphism design.

Class Declaration and Object Creation. Our system allows classes to be annotated with mutability qualifiers. Only three qualifiers are permitted at the class level: `@Imm`, `@Mut`, and `@RDM`. There are only two kinds of objects at runtime: immutable and mutable. If a class is annotated with `@Imm` or `@Mut`, all instances of the class are guaranteed to be immutable or mutable, respectively. If a class is annotated with `@RDM`, its instances can be either mutable or immutable depending on the object creation expression and the constructor's return type.

Polymorphic Object Creation. If a constructor is declared to return `@RDM`, it can be instantiated with `@Imm`, `@Mut`, or `@RDM`. This allows polymorphic classes to define a single constructor reused for both immutable and mutable object instantiations. The code below illustrates this design by extending `OuterBox` with a polymorphic constructor. Constructor parameters are viewpoint-adapted from the object creation expression. For example, an `@Imm` creation adapts `@RDM` parameters to `@Imm`. Creating an `@RDM` object is allowed statically, and allocation then uses the runtime mutability of `this`. Thus, `new @RDM OuterBox` allocates a mutable object when `this` is mutable and an immutable object when `this` is immutable. This design allows polymorphic object creation in a polymorphic method. The preservation and safety are defined and proved in Section 4.4. Our system forbids creating objects with the `@RO` qualifier, as objects must be either immutable or mutable. Like methods, constructors may also carry one of the four method scopes, although our formalization checks constructor bodies under `@AS`.

```
@RDM class OuterBox {
  @RDA @RDM Box box;
  @RDM OuterBox(@RDM Box box) { this.box = box; }
  void setBox(@Mut OuterBox this, @Mut Box box) { this.box = box; }
  @RDM Box getBox(@RDM OuterBox this) { return this.box; }
}
@Mut OuterBox mutOuterRef = new @Mut OuterBox(new @Mut Box()); // ok
@Imm OuterBox immOuterRef = new @Imm OuterBox(new @Imm Box()); // ok
@RDM OuterBox rdmOuterRef = new @RDM OuterBox(new @RDM Box()); // ok
// error: @Mut creation adapts the @RDM parameter to @Mut
@Mut OuterBox badRef = new @Mut OuterBox(new @Imm Box());
```

Subclassing. Subclassing is straightforward if the subclass and superclass share the same class mutability declaration. To make our system compatible with the `Object`-rooted class hierarchy and

libraries like JCF, we introduce flexible subclassing rules. A class annotated with `@Imm` or `@Mut` may extend a class, or implement an interface, whose class mutability declaration is either the same qualifier or `@RDM`. In contrast, an `@RDM` class or interface may only extend or implement other `@RDM` classes and interfaces. Since all classes extend `Object`, our system annotates the root class `Object` with `@RDM`: it is the only qualifier that permits both `@Imm` and `@Mut` subclasses without restricting the entire hierarchy to a single mutability. The code below illustrates flexible subclassing by two subclasses of the polymorphic `OuterBox`: `@Imm ImmOuterBox` and `@Mut MutOuterBox`. All three of their methods override methods of `OuterBox`, and we omit `@Override`.

```
@Imm class ImmOuterBox extends OuterBox {
  // no setBox here: it would need a @Mut receiver
  @Imm Box getBox(@Imm ImmOuterBox this) { return this.box; }
}
@Mut class MutOuterBox extends OuterBox {
  void setBox(@Mut MutOuterBox this, @Mut Box box) { this.box = box; }
  @Mut Box getBox(@Mut MutOuterBox this) { return this.box; }
}
```

Method Overriding. Using an `@RDM`-annotated, mutability-polymorphic class such as `OuterBox` as a superclass poses the risk that mutation methods may become accessible to subclasses. Our system ensures safe method overriding using viewpoint adaptation. A method can only be invoked if the receiver argument is a subtype of the method's receiver parameter type after adaptation or the method has a `@RDM` receiver parameter and the receiver argument is `@RO`. When an `@RDM` class is instantiated as `@Imm`, viewpoint adaptation turns its receiver-dependent types into `@Imm` types. Thus, a reference to such an immutable object cannot call a method requiring a `@Mut` receiver parameter. In addition, the `ImmOuterBox` class cannot override the `setBox` method with mutation in its body as it requires a `@Mut` receiver parameter. It still inherits `setBox`, but receiver typing prevents any instance of `ImmOuterBox` from invoking it. The parameter types are contravariant, and the return type is covariant after viewpoint adaptation. The method overriding design ensures immutable subclasses can neither declare mutation methods nor invoke the ones they inherit.

3 Programming Language and Static Model

In this section, we present our formalization of the core PICO by extending Featherweight Java [25] with mutable references, statements, and qualifiers. We introduce the syntax (Section 3.1), static viewpoint adaptation rules (Section 3.2), subclassing, subqualifier, and subtyping rules (Section 3.3), static well-formedness rules (Section 3.4), and type rules (Section 3.5). The static lookup functions are standard in Featherweight Java, and we leave them in Appendix A.1. The calculus abstracts from general constructor bodies, method and constructor overloading, and generics; the implementation supports them. The runtime model and safety properties are presented in Section 4, and the formalization is mechanized in Rocq.

3.1 Syntax

The syntax is presented in Figure 4. We assume all identifiers are globally unique except method identifiers, which may be reused across classes for overriding and are unique within each class.

A program declaration P is a sequence of class declarations and a main statement. A main statement is the starting point of program execution.

A class declaration cd contains class C extending class D with a set of fields $\overline{fd}$, one constructor kd, and methods $\overline{md}$. The class has a class declaration mutability qualifier q_c.

A field declaration fd is composed of an assignability modifier a, a qualified type T, and a field identifier f, where the assignability modifier can be Assign, Final, or RDA.

$$
\begin{array}{rll}
P & ::= & \overline{cd}\ s & \text{Programs} \\
cd & ::= & q_c \text{ class } C \text{ extends } D\ \{\overline{fd}\ kd\ \overline{md}\} & \text{Class definitions} \\
fd & ::= & a\ T\ f & \text{Field declarations} \\
kd & ::= & q_c\ C\ (\overline{T\ x},\ \overline{T'\ y})\ \{\text{super}(\overline{x});\ \overline{\text{this}.f = y}\} & \text{Constructor definitions} \\
md & ::= & p\ T'\ m(T\ \text{this},\ \overline{T''\ x})\{s;\ \text{return } x\} & \text{Method definitions} \\
p & ::= & \text{AS} \mid \text{CS} \mid \text{RS} \mid \text{TS} & \text{Method scope annotations} \\
e & ::= & \text{null} \mid x \mid x.f & \text{Expressions} \\
s & ::= & T\ x \mid x := e \mid x.f := y \mid x := \text{new } q_c\ C(\overline{y}) & \text{Statements} \\
 & & \mid x := y.m(\overline{z}) \mid s; s & \\
T & ::= & q_u\ C & \text{Qualified types} \\
q_c & ::= & \text{Mut} \mid \text{RDM} \mid \text{Imm} & \text{Class mutability qualifiers} \\
q_u & ::= & q_c \mid \text{RO} & \text{User mutability qualifiers} \\
q_h & ::= & \text{Lost} \mid \text{Bottom} & \text{Helper mutability qualifiers} \\
q & ::= & q_u \mid q_h & \text{All mutability qualifiers} \\
a & ::= & \text{Assign} \mid \text{Final} \mid \text{RDA} & \text{Assignability modifiers} \\
{}^{s}\Gamma & ::= & \overline{x \mapsto T} & \text{Static environments} \\
\multicolumn{4}{l}{this, x, y, z \text{ (variables)},\ f, m \text{ (fields, methods)},\ C, D \text{ (classes)}} & \text{Identifiers}
\end{array}
$$

Fig. 4. The syntax of PICO.

A constructor declaration kd has two sets of parameters, $\overline{x}$ and $\overline{y}$; one is used as arguments to the super constructor, and the other initializes all fields of C. We use q_c before the constructor name to indicate the mutability of the resulting object.

A method declaration md has a method scope annotation p, a return type T', a receiver parameter this of type T, and a sequence of formal parameters $\overline{x}$ of type $\overline{T''}$. A method scope annotation is one of abstract state (AS), concrete state (CS), readonly state (RS), and transitive state (TS). These four scopes independently select one of two mutability adaptations and one of two assignability adaptations. The method body is a statement s followed by a return statement return x.

An expression can be a variable, a field access, or a null.

A statement can be (1) a local variable declaration, (2) a variable assignment, (3) a field update, (4) an object creation, (5) a method invocation, or (6) a sequence of statements.

A type T is the composition of a mutability qualifier q_u and a base class C. We use q_c and q_u to distinguish qualifiers that can appear only on class declarations (q_c) from qualifiers that can appear in general type use (q_u), where q_u includes all class declaration qualifiers and additionally RO. q_h is a set of two helper mutability qualifiers, which programmers should not write, and we list them here for the result of viewpoint adaptation and make the lattice bounded.

3.2 Static Viewpoint Adaptation

Our system has static viewpoint adaptation rules for both mutability qualifiers and assignability modifiers within different method scopes p. We write $q_1 \triangleright_{m,p} q_2$ for mutability adaptation and $q \triangleright_{a,p} a$ for assignability adaptation. In both operators, the left operand is the viewpoint qualifier and the right operand is the declared mutability qualifier or assignablity modifier.

For mutability qualifiers, the viewpoint adaptation rules are defined as: In every method scope, a declared RDM adapts to Lost under a RO viewpoint and otherwise to the viewpoint qualifier. AS and CS leave every other declared qualifier unchanged. RS and TS additionally adapt a declared Mut to Mut under a Mut viewpoint and to Lost otherwise.

AS and CS thereby preserve precise mutability information. RS and TS conservatively hide mutable references reached through non-Mut viewpoints because they may expose objects in the abstract state.

$$q_1 \triangleright_{m,\mathrm{AS/CS}} q_2 = \begin{cases} \text{if } q_1 = \text{RO then Lost else } q_1, & q_2 = \text{RDM}, \\ q_2, & \text{otherwise}, \end{cases}$$

$$q_1 \triangleright_{m,\mathrm{RS/TS}} q_2 = \begin{cases} \text{if } q_1 = \text{RO then Lost else } q_1, & q_2 = \text{RDM}, \\ \text{if } q_1 = \text{Mut then Mut else Lost}, & q_2 = \text{Mut}, \\ q_2, & \text{otherwise}. \end{cases}$$

For adapting assignability modifiers, the viewpoint adaptation rules are defined as: In an AS or RS method scope, the resulting modifier is Assign when the declared modifier is Assign, or when the viewpoint qualifier is Mut and the declared modifier is RDA; otherwise, it is Final. In a CS or TS method scope, the resulting modifier is Assign only when the viewpoint qualifier is Mut and the declared modifier is RDA or Assign; otherwise, it is Final.

$$q \triangleright_{a,\mathrm{AS/RS}} a = \begin{cases} \text{Assign} & \text{if } a = \text{Assign} \vee (q = \text{Mut} \wedge a = \text{RDA}), \\ \text{Final} & \text{otherwise}, \end{cases}$$

$$q \triangleright_{a,\mathrm{CS/TS}} a = \begin{cases} \text{Assign} & \text{if } q = \text{Mut} \wedge a \in \{\text{RDA}, \text{Assign}\}, \\ \text{Final} & \text{otherwise}. \end{cases}$$

The stricter assignability adaptation shared by CS and TS makes a field effectively Final under any non-Mut viewpoint, providing concrete-state preservation in CS. Combined with RS's stricter mutability adaptation, it gives TS transitive-state preservation of the pre-call object graph while still allowing updates to newly allocated mutable objects.

3.3 Subclassing, Subqualifier, and Subtyping

We use subclass $\sqsubseteq$ to express the relationship between two classes connected by "extends" and subqualifier $<:_q$ to express the relationship between two qualifiers. Subclassing is the transitive and reflexive closure of "extends" relations. We assume that the superclass is defined before the subclass and that subclassing is acyclic. The qualifier hierarchy is RO as the top and Bottom as the bottom qualifier. Other qualifiers are not subqualifiers of each other, and Lost is not a subqualifier of itself. Based on the definitions of subclass and subqualifier, we define subtype $<:$ between two qualified types. A qualified type is a subtype of another qualified type if the qualifiers follow the subqualifier relationship and the base types follow the subclassing relationship. The formal definition of subclass is presented in Appendix A.2.

3.4 Static Well-Formedness

In this subsection, we present the static well-formedness rules of PICO in Figure 5. Declaration well-formedness uses AS viewpoint adaptation because it preserves mutability information; CS shares this mutability adaptation, while RS and TS apply their stricter adaptation when type-checking method bodies. We present selected rules in the main text and the complete rules in Appendix A.3.

WF-TypeUse states that a type use $q\ C$ is well-formed when the class declaration qualifier, adapted from viewpoint q, is a subqualifier of q. The function bound(C) yields the class declaration qualifier of class C.

Example 1 (Type use). If bound(C) = RDM, then Mut C, Imm C, RDM C, and RO C are well-formed type uses: for example, RO $\triangleright_{m,\mathrm{AS}}$ RDM = Lost and Lost $<:_q$ RO. If bound(C) = Mut, then Mut C and RO C are well-formed, but Imm C is rejected because Imm $\triangleright_{m,\mathrm{AS}}$ Mut = Mut, and Mut is not a subqualifier of Imm.

WF-Field states that a field declaration is well-formed if its qualified type is a valid type use according to WF-TypeUse. A field in a Mut or Imm class may still be declared RDM or RDA. This is harmless because the class declaration qualifier already fixes the object's mutability, so the field's adapted mutability and assignability are fixed as well.

WF-Cons states that a constructor declaration is well-formed in class C if its parameter types are valid type uses according to WF-TypeUse. Since we use constructor parameters to initialize fields, we require that the constructor parameter types be subtypes of the fields' types.

We require the constructor's return type to match the class declaration because our language does not support constructor overloading. In our checker implementation, a class with a RDM class declaration can have mutable, immutable, and RDM return type constructors. Because every constructor will call its super constructor, we require that the super constructor call is valid as defined by checkSuperCall. The rule states that, from the viewpoint of this constructor, the super constructor's return type should match this constructor's return type, and its arguments should be subtypes of the parameter types, adapted from the viewpoint of this constructor's return type. The rules ensure correct object initialization when the superclass and this class have different constructor return types.

WF-Meth expresses that the method declaration is well-formed in class C if the method receiver type, formal parameter types, and return types are well-formed as defined by WF-TypeUse. After type-checking its body, x should be defined in the resulting static environment, and it should be a subtype of the return type. The type rules are defined in Section 3.5, where statement typing has both an input and an output environment to allow local variable declaration. For method overriding, the receiver and formal-parameter types must be supertypes of their adapted counterparts in the parent class, preserving contravariance, while the return type must be a subtype of its adapted counterpart, preserving covariance. This rule ensures that an immutable subclass of an RDM parent class cannot override the mutation methods it inherits. For a class C and method m, NoMut(C, m) means that the declared receiver and formal parameters of m in C have non-Mut types; WF-Meth requires it for every method whose scope is RS or TS, which the readonly-state and transitive-state guarantees in Section 4.5 rely on.

Example 2 (Method overriding). Suppose an RDM class declares a method with an RDM receiver and RDM return type. An Imm subclass may override it with an Imm receiver and Imm return type, because Imm $\triangleright_{m,\mathrm{AS}}$ RDM = Imm for both the adapted receiver and return. However, if the superclass receiver is Mut, the subclass cannot override the method with an Imm receiver: Imm $\triangleright_{m,\mathrm{AS}}$ Mut = Mut, and Mut is not a subqualifier of Imm. The subclass still inherits the method, but it is not callable on an Imm or RO receiver.

WF-Class expresses that a class is well-formed if the class contains well-formed fields, a constructor, and methods. A class must not be its own superclass, and the class declaration should be the same after it adapts its parent class's declaration. This design ensures immutable classes can extend RDM or immutable classes, mutable classes can only extend RDM or mutable classes, and RDM classes can only extend RDM classes. Otherwise, the mutability property can be violated, for example, if an immutable class can extend a mutable class, it would be allowed to pass immutable references to a method expecting mutable references, leading to runtime errors.

Example 3 (Class subclassing). If bound(D) = RDM and bound(C) = Mut, then C may extend D, because Mut $\triangleright_{m,\text{AS}}$ RDM = Mut. If bound(D) = RDM and bound(C) = Imm, then C may also extend D, because Imm $\triangleright_{m,\text{AS}}$ RDM = Imm. In contrast, an Imm class cannot extend a Mut class: the adapted parent qualifier is Mut, not Imm.

$$\frac{(q \triangleright_{m,\text{AS}} \text{bound}(C)) <: q}{q\ C\ \text{OK}}\ (\text{WF-TypeUse}) \qquad \frac{q_f\ C_f\ \text{OK}}{a\ q_f\ C_f\ f\ \text{OK}}\ (\text{WF-Field})$$

$$\frac{\begin{array}{c}\text{parentClass}(C) = D \qquad \text{conSig}(D) = q'_c\ D(\overline{q_s\ D'\ x}) \\ q_c = \text{bound}(C) \qquad q_c \triangleright_{m,\text{AS}} q'_c = q_c \qquad \overline{q_x\ C_x <: (q_c \triangleright_{m,\text{AS}} q_s)\ D'}\end{array}}{\text{checkSuperCall}(C, \overline{q_x\ C_x\ x})\ \text{OK}}\ (\text{checkSuperCall})$$

$$\frac{\begin{array}{c}\text{checkSuperCall}(C, \overline{q'\ C'\ x})\ \text{OK} \qquad \overline{q'\ C'}\ \text{OK},\ \overline{q''\ C''}\ \text{OK} \qquad q = \text{bound(C)} \\ {}^{s}\text{fType}(C, f_i) = \_\ q_{fi}\ C_{fi} \qquad \overline{(q \triangleright_{m,\text{AS}} q'')\ C'' <: (q \triangleright_{m,\text{AS}} q_{fi})\ C_{fi}}\end{array}}{q\ C\ (\overline{q'\ C'\ x},\ \overline{q''\ C''\ y})\ \{\text{super}(\overline{x});\ \overline{\text{this}.f = y}\}\ \text{OK in C}}\ (\text{WF-Cons})$$

$$\frac{\begin{array}{c}q\ C\ \text{OK},\ q'\ C'\ \text{OK},\ \overline{q''\ C''}\ \text{OK} \qquad {}^{s}\Gamma, p \vdash s \dashv {}^{s}\Gamma' \\ {}^{s}\Gamma' \vdash x : q_x\ C_x \qquad q_x\ C_x <: q'\ C' \qquad \text{parentClass}(C) = D \\ q_c = \text{bound}(C) \qquad (p \in \{\text{RS}, \text{TS}\} \Rightarrow \text{NoMut}(C, m)) \\ \text{mSig}(D, m) = p_s\ q'_s\ C'\ m(q_s\ D\ \text{this},\ \overline{q''_s\ C''\ x}) \Rightarrow \big(p = p_s \wedge \\ q' <:_q\ q_c \triangleright_{m,\text{AS}} q'_s \wedge q_c \triangleright_{m,\text{AS}} q_s <:_q\ q \wedge q_c \triangleright_{m,\text{AS}} q''_s <:_q\ q''\big)\end{array}}{p\ q'\ C'\ m(q\ C\ \text{this},\ \overline{q''\ C''\ x})\{s;\ \text{return}\ x\}\ \text{OK in C}}\ (\text{WF-Meth})$$

$$\frac{\overline{fd}\ \text{OK},\ kd\ \text{OK in C},\ \overline{md}\ \text{OK in C} \qquad C \neq D \qquad q_c \triangleright_{m,\text{AS}} \text{bound}(D) = q_c}{q_c\ \text{class}\ C\ \text{extends}\ D\ \{\overline{fd}\ kd\ \overline{md}\}\ \text{OK}}\ (\text{WF-Class})$$

Fig. 5. Selected static well-formedness rules of PICO.

3.5 Type Rules

In this subsection, we present the selected type rules for PICO in Figure 6. The type rules are defined under the assumption that all types used in the rule and in the static environment are well-formed. The viewpoint adaptations are parametrized by the method scope p, which is passed from the enclosing method declaration. The complete type rules are presented in Appendix A.4. WF-Meth and ST-Call are the two places where method scopes interact with viewpoint adaptation: WF-Meth checks method bodies and overriding after adapting superclass signatures from the subclass's class declaration viewpoint bound(C), while ST-Call adapts receiver-dependent return and parameter types from the actual receiver. Together, these rules prevent subclasses and call sites from weakening the guarantees promised by receiver qualifiers and method scopes.

Expression Typing. Typing an expression takes a static environment ${}^{s}\Gamma$, which maps variables to qualified types, a method scope p, and an expression e, yielding a qualified type. ET-FldRead expresses that a field access $x.f$ is well-typed if the field f is defined in the class of x and the resulting field type is adapted from the viewpoint of x.

Statement Typing. Typing a statement takes a static environment ${}^{s}\Gamma$, a method scope p, and a statement s, yielding a new static environment ${}^{s}\Gamma'$.

ST-FldWrite expresses that a field assignment $x.f := y$ is well-typed if the field f is defined in the class of x and the type of y is a subtype of the adapted field type.

Example 4 (Field write). Assume field f is declared with assignability RDA. An Imm C receiver adapts it to Final, so $x.f := y$ is rejected. A Mut C receiver adapts it to Assign, so the write may type-check.

ST-New expresses that a constructor call $x := \text{new } q_c\ C(\overline{y})$ is well-typed if the arguments are subtypes of the adapted parameter types, and the qualifier of the constructor invocation is an adapted subtype of the qualifier of the constructor return type. This rule allows polymorphic object creation. When creating immutable objects, the RDM fields are initialized with Imm arguments; when creating mutable objects, the RDM fields are initialized with Mut arguments.

Example 5 (Object creation). For example, if the constructor of C has an RDM D parameter, then $x := \text{new Imm } C(y)$ requires y to have an Imm D type, while $x := \text{new Mut } C(y)$ requires y to have a Mut D type.

ST-Call expresses that a method call $x := y.m(\overline{z})$ is well-typed if the receiver y is well-typed, the method is defined in the class of y, the method signature is adapted from the viewpoint of y, and the arguments are subtypes of the adapted parameter types. One special case we allow is that a RO receiver argument may call a method whose receiver parameter is RDM, which provides the context sensitivity described in Section 2.2.

Example 6 (Method call). Assume m has an RDM receiver parameter and returns an RDM D. If y has type Mut C, then the receiver premise succeeds and the return type adapts to Mut D, so the result may be assigned to a Mut D variable. If y has type RO C, then the special receiver premise permits the call, but the return type adapts to Lost D; therefore the result may be assigned to a RO D variable, but not to a Mut D variable. A method with a Mut receiver parameter is rejected on the same RO receiver argument because RO $\triangleright_{m,p}$ Mut is Mut in AS/CS scope and Lost in RS/TS scope, and RO is a subqualifier of neither.

Method scope annotations impose an ordering relationship, which we call submethod $<:_m$, as established in Figure 1. A method checked under scope p may call a method with scope p' when $p' <:_m p$, meaning that p' selects equally or more conservative viewpoint adaptations. Thus, a method checked under a stronger method scope may call helper methods that make at least the same guarantees, but not methods whose scope permits forbidden mutations.

In summary, we present the syntax, static viewpoint adaptation and well-formedness rules, and type rules. The type safety is formalized and proved in Section 4.4.

4 Runtime Model

In this section, we present the runtime model, runtime viewpoint adaptation and typability rules (Section 4.1), runtime well-formedness rules (Section 4.2), operational semantics (Section 4.3), preservation property (Section 4.4), state-preservation guarantees (Section 4.5), and proof mechanization experience (Section 4.6).

The runtime model is defined in Figure 7. We define the heap h, mapping addresses to objects. A value v is either an address ι or a special address null_a. An object o is a tuple of a runtime type ${}^rT$ and field mappings $\overline{fv}$. A runtime type ${}^rT$ is composed of a runtime mutability qualifier q_r and a class identifier C. A runtime mutability qualifier q_r is Imm_r or Mut_r as an object can only be an immutable object or a mutable object. A field mapping fv maps from a field identifier f to a value v. An evaluation result res is either OK or MutEXC, which records an illegal field write.

$$\frac{{}^s\Gamma, p \vdash x : q_x\ C_x \qquad {}^s\mathrm{fType}(C_x, f) = \_\ q_f\ C_f \qquad q_x \triangleright_{m,p} q_f = q'}{{}^s\Gamma, p \vdash x.f :\ q'\ C_f} \quad \text{(ET-FldRead)}$$

$$\frac{\begin{array}{c}{}^s\Gamma, p \vdash x : q_x\ C_x \qquad {}^s\Gamma, p \vdash y : q_y\ C_y \qquad {}^s\mathrm{fType}(C_x, f) = a_f\ q_f\ C_f \\ q_y\ C_y <:\ (q_x \triangleright_{m,p} q_f)\ C_f \qquad q_x \triangleright_{a,p} a_f = \mathrm{Assign}\end{array}}{{}^s\Gamma, p \vdash x.f := y \dashv {}^s\Gamma} \quad \text{(ST-FldWrite)}$$

$$\frac{\begin{array}{c}{}^s\Gamma, p \vdash x : q_x\ C_x \qquad {}^s\Gamma, p \vdash \overline{y : q_y\ C_y} \qquad \mathrm{conSig}(C) = q_c'\ C(\overline{q_s\ C_s\ x}) \\ \overline{q_y\ C_y <: (q_c \triangleright_{m,\mathrm{AS}} q_s)\ C_s} \qquad q_c \triangleright_{m,\mathrm{AS}} q_c' = q_c \qquad (q_c \triangleright_{m,\mathrm{AS}} q_c')\ C <: q_x\ C_x\end{array}}{{}^s\Gamma, p \vdash x := \mathrm{new}\ q_c\ C(\overline{y}) \dashv {}^s\Gamma} \quad \text{(ST-New)}$$

$$\frac{\begin{array}{c}{}^s\Gamma, p \vdash x : q_x\ C_x \qquad {}^s\Gamma, p \vdash y : q_y\ C_y \qquad {}^s\Gamma, p \vdash \overline{z : q_z\ C_z} \\ \mathrm{mSig}(m, C_y) = p'\ q'\ C'\ \mathrm{m}(q\ C\ \mathrm{this},\ \overline{q''\ C''x}) \qquad p' <:_m\ p \\ (q_y \triangleright_{m,p} q')\ C' <: q_x\ C_x \qquad \overline{q_z\ C_z <:\ (q_y \triangleright_{m,p} q'')\ C''} \\ (q_y\ C_y <: (q_y \triangleright_{m,p} q)\ C) \vee (q_y = \mathrm{RO} \wedge q = \mathrm{RDM} \wedge C_y \sqsubseteq C)\end{array}}{{}^s\Gamma, p \vdash x := y.m(\overline{z}) \dashv {}^s\Gamma} \quad \text{(ST-Call)}$$

Fig. 6. Selected type rules of PICO.

4.1 Runtime Viewpoint Adaptation and Typability

If the static type is Imm or Mut, the runtime mutability qualifier can only be Imm_r or Mut_r, respectively. But if the static type is RDM, RO, or Lost, what runtime mutability qualifier can it refer to? To connect the runtime object with static references, we introduce runtime viewpoint adaptation rules, denoted by $\triangleright_r$. Similar to static viewpoint adaptation rules, we define runtime viewpoint adaptation rules as follows. The rules guarantee the static mutability qualifier RDM will be resolved to a concrete mutability qualifier. Note that we only have one version of runtime viewpoint adaptation rules for mutability qualifiers, because the mutability is precisely tracked at runtime, and different state-preservation guarantees are provided by static viewpoint adaptation rules.

h	::=	$\overline{\iota \mapsto o}$	Heaps
ι	::=	Address	Addresses
v	::=	$\iota \mid \mathrm{null}_a$	Values
o	::=	$({}^rT, \overline{fv})$	Runtime objects
${}^rT$	::=	$q_r\ C$	Runtime types
q_r	::=	$\mathrm{Imm}_r \mid \mathrm{Mut}_r$	Runtime mutabilities
fv	::=	$f \mapsto v$	Field mappings
${}^r\Gamma$	::=	$\overline{x \mapsto v}$	Runtime environments
res	::=	OK | MutEXC	Results

Fig. 7. The runtime model of PICO.

$$\frac{}{\mathrm{Imm}_r \triangleright_r \mathrm{RDM} = \mathrm{Imm}}\ \text{(R-M-VPA-1)} \qquad \frac{}{\mathrm{Mut}_r \triangleright_r \mathrm{RDM} = \mathrm{Mut}}\ \text{(R-M-VPA-2)}$$

$$\frac{q' \neq \mathrm{RDM}}{q_r \triangleright_r q' = q'}\ \text{(R-M-VPA-3)}$$

Similarly, we define runtime viewpoint adaptation rules for assignability. These rules correspond to the AS/RS static rules, except that they take a runtime mutability qualifier as input. We will use them to define where the operational semantics can go wrong and to show that our type system prevents such runtime errors. The formalization is presented in Appendix A.6.

We define the typability $<:_r$ between the runtime type and the static qualified type. Mut-Typable states that Mut_r is typable if the corresponding static mutability qualifier is Mut, Lost, or RO. Imm-Typable is similar except that the first static qualifier is Imm. The typability relation allows Lost and RO to refer to both immutable and mutable objects at runtime. RDM qualifier will be resolved to either Mut_r or Imm_r by runtime viewpoint adaptation rules, so there is no rule directly for RDM typability. Consequently, QT-Typable requires that the runtime and static mutability qualifier are typable and the runtime base type and static base type are subclasses.

$$\frac{q \in \{\text{Mut}, \text{Lost}, \text{RO}\}}{\text{Mut}_r <:_r \ q} \quad (\textsc{Mut-Typable}) \qquad \frac{q \in \{\text{Imm}, \text{Lost}, \text{RO}\}}{\text{Imm}_r <:_r \ q} \quad (\textsc{Imm-Typable})$$

$$\frac{q_r <:_r \ q \qquad C \sqsubseteq D}{q_r \ C <:_r \ q \ D} \quad (\text{QT-}\textsc{Typable})$$

We present one helper function, `canAssign`, to define an error case and provide dynamic safety in the operational semantics. Other helper functions are defined in Appendix A.5. The function $\text{canAssign}(h, \iota, f)$ returns a boolean indicating whether field f of the object at address ι is assignable in heap h. It is true exactly when the receiver's runtime mutability, as declared by the field's assignability modifier, yields Assign, and false otherwise.

$$\frac{h(\iota)_{\downarrow_1} = q_r \ C \quad {}^{\text{s}}\text{fType}(C, f) = a \ q_f \ C_f \quad q_r \rhd_r a = a'}{\text{canAssign}(h, \iota, f) = (a' = \text{Assign})}$$

4.2 Runtime Well-Formedness

In this subsection, we present the runtime well-formedness rules for PICO in Figure 8.

WF-RType defines the well-formedness of runtime type $q_r \ C$. A runtime type is well-formed if it is a subtype of the class declaration bound after adapting it with the runtime mutability qualifier.

WF-Tyability defines the well-formedness of assigning a static qualified type $q \ C$ to a runtime value ι. This assignment is well-formed if the runtime type of the object at address ι is typable by the static type after adapting another runtime mutability qualifier q_r. Here ι_r is any address, and the actual receiver address will be used for getting q_r in WF-Config. This is a key rule that bridges the static mutability qualifier with the runtime mutability qualifier. As a result, static RO and Lost types can be used to refer to any runtime object, and static RDM type can be used to refer to a runtime object when q_ι is typable with $q_r \rhd_r q$.

WF-Object defines the well-formedness of runtime object at address ι. A runtime object is well-formed if its runtime type is well-formed, and all fields' runtime types are typable by the declared field type adapted by this object's runtime mutability qualifier. A heap is well-formed if all its objects are well-formed (WF-Heap).

WF-REnv defines the well-formedness of runtime environment ${}^{\text{r}}\Gamma$. A runtime environment is well-formed if all values in the environment are defined in the heap.

WF-Config defines the well-formedness of configuration ${}^{\text{r}}\Gamma, h, {}^{\text{s}}\Gamma$. This rule states that a configuration is well-formed if the heap, runtime environment, and static environment are well-formed. In addition, the types in the static environment can be assigned to the corresponding runtime values.

4.3 Operational Semantics

We use big-step operational semantics. Given a well-formed runtime environment ${}^{\text{r}}\Gamma$ and heap h, the judgment ${}^{\text{r}}\Gamma, h, s \rightsquigarrow {}^{\text{r}}\Gamma', h', res$ means that statement s evaluates to runtime environment ${}^{\text{r}}\Gamma'$, heap h', and result res. The result is OK or MutEXC, which records an illegal field write. Figure 9 presents selected statement rules; the complete semantics appears in Appendix A.7.

The OS-S-FldWrite rule expresses that a field assignment will update the heap by assigning the value to the field of the object. If the field is effectively assignable, the field is updated in the heap,

$$\frac{q_r <:_r \ (q_r \rhd_r \text{bound}(C))}{q_r \ C \ \text{OK}} \quad \text{(WF-RType)}$$

$$\frac{h(\iota_r)_{\downarrow_1} = q_r \ \_ \qquad h(\iota)_{\downarrow_1} = q_\iota \ C_\iota \qquad q_\iota \ C_\iota <:_r \ (q_r \rhd_r q) \ C}{h, \iota_r \vdash \iota : q \ C} \quad \text{(WF-Tyability)}$$

$$\frac{h(\iota)_{\downarrow_1} = q_r \ C \qquad q_r \ C \ \text{OK} \qquad \forall f \in \text{fields(C)}.\ {}^s\text{fType}(C, f) = q_f \ C_f \wedge h, \iota \vdash h(\iota.f) : (q_r \rhd_r q_f) \ C_f}{h \vdash \iota \ \text{OK}} \quad \text{(WF-Object)}$$

$$\frac{\forall \iota \in \text{dom}(h).\ h \vdash \iota \ \text{OK}}{h \ \text{OK}} \quad \text{(WF-Heap)}$$

$$\frac{\forall x \in \text{dom}({}^r\Gamma).\ {}^r\Gamma(x) = \text{null}_a \vee {}^r\Gamma(x) \in \text{dom}(h)}{h \vdash {}^r\Gamma \ \text{OK}} \quad \text{(WF-REnv)}$$

$$\frac{{}^r\Gamma = \{\text{this} \mapsto \iota, \overline{x \mapsto v}\} \qquad {}^s\Gamma = \{\text{this} \mapsto q \ C, \overline{x \mapsto T}\} \qquad h \ \text{OK} \qquad {}^s\Gamma \ \text{OK} \qquad h \vdash {}^r\Gamma \ \text{OK} \qquad h, \iota \vdash \iota : q \ C \qquad h, \iota \vdash \overline{v} : \overline{T}}{{}^r\Gamma, h : {}^s\Gamma \ \text{OK}} \quad \text{(WF-Config)}$$

Fig. 8. The runtime well-formedness rules of PICO.

$$\frac{{}^r\Gamma(x) = \iota_x, \ {}^r\Gamma(y) = v_y \qquad \text{canAssign}(h, \iota_x, f) \qquad h' = h[\iota_x.f = v_y]}{{}^r\Gamma, h, x.f := y \rightsquigarrow {}^r\Gamma, h', \text{OK}} \quad \text{(OS-S-FldWrite)}$$

$$\frac{{}^r\Gamma(x) = \iota_x, \ {}^r\Gamma(y) = v_y \qquad \neg\text{canAssign}(h, \iota_x, f)}{{}^r\Gamma, h, x.f := y \rightsquigarrow {}^r\Gamma, h, \text{MutEXC}} \quad \text{(OS-S-FieldWriteEXC)}$$

$$\frac{{}^r\Gamma(\text{this}) = \iota_{\text{this}} \qquad {}^r\Gamma(\overline{y}) = \overline{v_y} \qquad \text{fields(C)} = \overline{f} \qquad q_a = \begin{cases} \text{Imm}_r \ if \ {}^r\text{MType}(h, \iota_{\text{this}}) \rhd_r q_c \ = \ \text{Imm} \\ \text{Mut}_r \ if \ {}^r\text{MType}(h, \iota_{\text{this}}) \rhd_r q_c \ = \ \text{Mut} \end{cases} \qquad o = (q_a \ C, \overline{f \mapsto v_y}) \qquad h + o = (h', \iota) \qquad {}^r\Gamma' = {}^r\Gamma[x \mapsto \iota]}{{}^r\Gamma, h, x := \text{new} \ q_c \ C(\overline{y}) \rightsquigarrow {}^r\Gamma', h', \text{OK}} \quad \text{(OS-S-New)}$$

Fig. 9. Selected big-step operational semantics of PICO.

and the result is OK. If the field is not assignable, a mutation exception will be raised as expressed by OS-S-FldWrite-EXC rule.

The OS-S-New rule expresses that object creation will first create a new object. The fields are initialized by the constructor arguments. This case is novel because we allow the RDM object to be created statically. At runtime, object creation will be adapted from the viewpoint runtime mutability qualifier of `this`, which can only be Imm_r or Mut_r. The runtime viewpoint adaptation rules ensure that the created object's runtime mutability qualifier is either Imm_r or Mut_r.

4.4 Metatheory

In this subsection, we present the type-safety theorem for PICO. Type safety guarantees the absence of the runtime mutability error raised by OS-S-FldWrite-EXC. Accordingly, we formulate safety

as preservation of well-formed configurations together with the absence of mutation exceptions. Both expression and statement evaluation may get stuck for reasons unrelated to mutability, such as lookup failures (including missing bindings and unknown fields) and null dereferences. Our safety result specifically rules out mutability violations: a well-typed statement cannot produce a mutation exception. Establishing the absence of lookup- or nullness-related stuck states would require additional progress or adequacy results and is outside the scope of our system.

Execution Abbreviation. We write $\mathrm{Exec}_p(s, h, h', res)$ when ${}^r\Gamma, h : {}^s\Gamma$ OK, ${}^s\Gamma, p \vdash s \dashv {}^s\Gamma'$, and ${}^r\Gamma, h, s \rightsquigarrow {}^r\Gamma', h', res$; the initial and final environments ${}^r\Gamma, {}^s\Gamma, {}^r\Gamma', {}^s\Gamma'$ are supplied by the surrounding statement.

Preservation and Safety. This theorem states that for all well-formed configurations and well-typed statements, the resulting runtime configuration remains well-formed and the result is OK.

Theorem 1 (Preservation and Safety). $\forall {}^r\Gamma, h, {}^s\Gamma, p, s, {}^s\Gamma', {}^r\Gamma', h', res.$ *If* $\mathrm{Exec}_p(s, h, h', res)$, *then* ${}^r\Gamma', h' : {}^s\Gamma'$ OK *and* $res = \mathrm{OK}$.

The proof is done by induction on the evaluation rules. In particular, the object creation statement $x := \text{new } q_c\ C(\overline{y})$ is interesting. The proof for it is done by case analysis of the runtime viewpoint adaptation rules. The created object's runtime mutability qualifier is either Imm_r or Mut_r.

4.5 State-Preservation Guarantees: From Shallow to Deep, and From Abstract to Transitive

Shallow Abstract Immutability. Our system allows mutation through fields marked with the Assign modifier, and the immutability property is restricted to a lenient version called shallow abstract immutability. It is shallow because it constrains only the immutable object's own fields, and abstract because Assign slots are exempt. The theorem quantifies over p, so it holds in all four scopes and is the building block for the deeper guarantees below.

Theorem 2 (Shallow Abstract Immutability).
$\forall {}^r\Gamma, h, {}^s\Gamma, p, s, {}^s\Gamma', {}^r\Gamma', h', \iota, C, f.$ *If* $\mathrm{Exec}_p(s, h, h', \mathrm{OK})$, $h(\iota)_{\downarrow_1} = \mathrm{Imm}_r\ C$, ${}^s\mathrm{fType}(C, f) \neq \text{Assign } \_\ \_$, *and* $f \in h(\iota)_{\downarrow_2}$, *then* $h(\iota.f) = h'(\iota.f)$.

The proof is done by induction on the evaluation rules. The key idea is that the field write operation will need an assignable field after adaptation. We show that the static viewpoint adaptation rules guarantee that the field is not assignable after adaptation, except when the field is explicitly marked as Assign.

Reachability Relations. We define two reachability relations in the runtime heap. $\iota \longrightarrow \iota'$ denotes reachability: ι' can be reached from ι by following a (possibly empty) sequence of field references. $\iota \rightarrowtail \iota'$ denotes reachable-abstract-state (RAS) reachability: ι' can be reached from ι only through fields that belong to ι's abstract state (i.e., fields whose static declarations satisfy the abstract-state qualifier constraints). Hence, $\rightarrowtail$ is a restricted form of $\longrightarrow$ that follows only abstract-state fields and is defined by the following rules.

$$\frac{\iota \in \mathrm{dom}(h)}{\iota \rightarrowtail \iota} \quad \text{(RAS-1)} \qquad \frac{\iota \rightarrowtail \iota' \qquad \iota' \rightarrowtail \iota''}{\iota \rightarrowtail \iota''} \quad \text{(RAS-2)}$$

$$\frac{\begin{array}{c}\iota \in \mathrm{dom}(h) \qquad h(\iota) = (q_r\ C, \overline{fv}) \qquad fv(f) = \iota' \\ \iota' \in \mathrm{dom}(h) \qquad {}^s\mathrm{fType}(C, f) = \text{Final/RDA RDM/Imm } \_\end{array}}{\iota \rightarrowtail \iota'} \quad \text{(RAS-3)}$$

RAS reachability preserves runtime immutability:

Lemma 1. $\forall {}^{r}\Gamma, h, {}^{s}\Gamma, \iota, \iota', C.\ {}^{r}\Gamma, h : {}^{s}\Gamma\ \mathrm{OK} \wedge \iota \hookrightarrow \iota' \wedge h(\iota)_{\downarrow_1} = \mathrm{Imm}_r\ C \implies \exists C'.\ h(\iota')_{\downarrow_1} = \mathrm{Imm}_r\ C'.$

The proof is by induction on the RAS derivation; RAS-3 and runtime viewpoint adaptation establish that each object reached through an abstract-state field remains immutable.

Abstract-State Preservation. Under AS, every non-Assign field of every RAS-reachable object is preserved (Theorem 3). Starting from an immutable root, Lemma 1 makes every object reached through the abstract state immutable, so Theorem 2 lifts the shallow guarantee to all of them.

Concrete-State Preservation. CS uses the same RAS-reachable objects as AS but combines AS mutability adaptation with stricter assignability adaptation. Under a non-Mut receiver, even a declared Assign field becomes effectively Final, so CS preserves the concrete state of those objects while still permitting mutable access to objects outside the abstract state.

Theorem 3 (AS and CS state-preservation guarantees).

$$
\begin{aligned}
&\textit{AS:}\quad \forall p, {}^{r}\Gamma, h, {}^{s}\Gamma, s, {}^{s}\Gamma', {}^{r}\Gamma', h', \iota, \iota', C, C', f.\\
&\qquad \mathrm{Exec}_p(s, h, h', \mathrm{OK}) \wedge h(\iota)_{\downarrow_1} = \mathrm{Imm}_r\ C \wedge \iota \hookrightarrow \iota' \wedge h(\iota')_{\downarrow_1} = \_\ C'\\
&\qquad \wedge\ f \in h(\iota')_{\downarrow_2} \wedge {}^{s}\mathrm{fType}(C', f) \neq \mathrm{Assign}\ \_\ \_ \implies h(\iota'.f) = h'(\iota'.f),\\
&\textit{CS:}\quad \forall {}^{r}\Gamma, h, {}^{s}\Gamma, s, {}^{s}\Gamma', {}^{r}\Gamma', h', \iota, \iota', C, f.\\
&\qquad \mathrm{Exec}_{\mathrm{CS}}(s, h, h', \mathrm{OK}) \wedge h(\iota)_{\downarrow_1} = \mathrm{Imm}_r\ C \wedge \iota \hookrightarrow \iota' \wedge f \in h(\iota')_{\downarrow_2} \implies h(\iota'.f) = h'(\iota'.f).
\end{aligned}
$$

For the RS and TS method-call guarantees, we define:

$$
\begin{aligned}
\mathrm{CallExec}_p(y, \overline{z}; \iota, \overline{v_i}; h, h') \triangleq\ &\exists {}^{r}\Gamma, {}^{s}\Gamma, {}^{r}\Gamma', {}^{s}\Gamma', x, m, q_y, C_y.\\
&{}^{r}\Gamma, h : {}^{s}\Gamma\ \mathrm{OK} \wedge {}^{s}\Gamma(y) = q_y\ C_y \wedge {}^{s}\Gamma, p \vdash x := y.m(\overline{z}) \dashv {}^{s}\Gamma'\\
&\wedge\ {}^{r}\Gamma, h, x := y.m(\overline{z}) \rightsquigarrow {}^{r}\Gamma', h', \mathrm{OK}\\
&\wedge\ {}^{r}\Gamma(y) = \iota \wedge \overline{{}^{r}\Gamma(z_i) = v_i}.
\end{aligned}
$$

The premise ${}^{s}\Gamma(y) = q_y\ C_y$ ties this signature to the static class of the call receiver. The call abbreviation leaves the initial receiver address and argument values explicit; only arguments whose value is an address contribute reachability roots.

Readonly-State Preservation. A readonly reference gives a weaker guarantee than an immutable reference because a mutable alias to the same object may exist. Our system guarantees that a call type-checked in the RS scope, whose target method has a non-Mut declared receiver and non-Mut formal parameters, does not modify objects reachable from the receiver or arguments, except through Assign fields. The readonly state is the closure defined in Section 2.4, but Theorem 4 is stronger, preserving the non-Assign fields of every transitively reachable object. This is because mutability adaptation yields no mutable reference to any object that existed at the call, so the guarantee reaches past the Assign slots at which the closure stops.

Transitive-State Preservation. Further, our system guarantees that invoking a TS method whose declared receiver and formal parameters are non-Mut does not modify objects reachable from the receiver or arguments. From the same call roots, the transitive state follows every field reference and contains every field slot of every reached object. Recall from Section 3.4 that WF-Meth requires $\mathrm{NoMut}(C, m)$ for every method whose scope is RS or TS, so well-formedness discharges this condition and the state-preservation theorems below carry no explicit non-mutability premise.

Theorem 4 (RS and TS state-preservation guarantees).

RS: $\forall h, h', y, \overline{z}, \iota, \overline{v_i}, \iota', C_{rch}, f.$
$\text{CallExec}_{\text{RS}}(y, \overline{z}; \iota, \overline{v_i}; h, h') \wedge (\iota \to \iota' \vee \exists i, \iota_i. v_i = \iota_i \wedge \iota_i \to \iota') \wedge h(\iota')_{\downarrow_1} =\_ C_{rch}$
$\wedge\ {}^{\text{s}}\text{fType}(C_{rch}, f) \neq \text{Assign}\ \_\ \_ \wedge f \in h(\iota')_{\downarrow_2} \implies h(\iota'.f) = h'(\iota'.f),$

TS: $\forall h, h', y, \overline{z}, \iota, \overline{v_i}, \iota', f.$
$\text{CallExec}_{\text{TS}}(y, \overline{z}; \iota, \overline{v_i}; h, h') \wedge (\iota \to \iota' \vee \exists i, \iota_i. v_i = \iota_i \wedge \iota_i \to \iota') \wedge f \in h(\iota')_{\downarrow_2} \implies h(\iota'.f) = h'(\iota'.f).$

Comparing the Four Guarantees. AS/CS share the objects reachable through an immutable root's abstract state, whereas the RS/TS theorem clauses quantify over objects transitively reachable from the receiver and arguments of a call whose declared receiver and formal parameter types are non-Mut. Within each object set, AS and RS protect non-Assign fields, whereas CS and TS remove that premise and protect every field.

Proof Challenges. The RS and TS proofs must distinguish protected objects already present at call entry from mutable objects allocated during execution, a distinction that reachability in the final heap alone cannot recover. We therefore fix the set of addresses reachable from the receiver and arguments in the pre-call heap and preserve invariants for this set through statement sequencing and nested method calls.

4.6 Proof Mechanization in Rocq

We have formalized our system and proved the above properties using the Rocq proof assistant [47]. In the mechanization, a null dereference raises an explicit exception rather than getting stuck. The mechanization covers all presented features except super calls in constructor bodies, which it models as direct field assignments. Rocq helped us analyze thousands of qualifier combinations and their consequences, thereby increasing the confidence in our design. The Rocq development comprises 39 files and about 27,000 lines of specification and proof. To our knowledge, PICO is the first mechanized system combining these features: transitive abstract immutability, readonly references, assignability, and class mutability polymorphism in one calculus.

5 Evaluation

We describe the implementation of PICO in Section 5.1 and case studies on several benchmarks in Section 5.2. We discuss PICO's limitations and potential solutions in Section 5.3.

5.1 Implementation

Our implementation is based on the Checker Framework [12, 37] and integrates with its Initialization Checker, which is based on the Freedom Before Commitment system [46]. The implementation has approximately 2000 lines of code. In the Checker Framework, *defaulting* means that unannotated types are automatically assigned a qualifier, reducing the annotation burden. We default unannotated types with qualifier `@Mut`, which aligns with Java's assumption that objects are mutable by default; programmers only need to add explicit annotations, such as `@Imm` or `@RO`, when a stricter mutability guarantee is desired. In addition, classes declared as `@Imm` (e.g., String and other immutable classes) are defaulted to immutable, which further reduces annotation effort. The Checker Framework also supports qualifier polymorphism, which enables the additional `@PolyMut` qualifier. The current Java checker uses the `@AS` scope for all methods, and we performed our evaluation under this scope to allow flexible use of `@RO` references. We plan to infer stronger method scope annotations when a method signature does not have a `@Mut` receiver or `@Mut` formal parameters. Beyond the core system, the implementation also supports generics, behavioral subtyping, annotation files, warning suppression, and PICO-specific diagnostic messages.

5.2 Case Studies

We conducted several case studies to evaluate the practicality of our system. In all evaluations, we focused on mutability with the Initialization Checker disabled.

5.2.1 Case Study 1 – JCF Benchmarks. We evaluated our system on the JCF in JDK 17 and retrofitted the JCF with mutability qualifiers. The JCF already provides unmodifiable and immutable collection implementations. Our system statically checks the intended mutability properties of the JCF, but it encountered limitations in both the type-system design and the Checker Framework implementation, discussed in Section 5.3. We type-checked 22,992 lines of non-comment JCF code and manually added 349 `@Imm`, 736 `@Mut`, 1919 `@RO`, 241 `@RDM`, 650 `@PolyMut`, and 66 `@Assign` annotations. The high count of `@Imm` annotations is due to annotating map key type parameters to avoid mutability-based map key errors. The `@RO` and `@Mut` annotations are on explicit method receiver parameters and on type parameter bounds. Because the JCF classes we annotate are declared `@RDM`, uses of their types default to `@RDM` rather than to `@Mut`, so both receiver kinds must be written out: a `@Mut` receiver for methods restricted to mutable objects, such as `Collection.removeIf`, and a `@RO` receiver for methods callable on both mutable and immutable objects, such as `size()` in the `Collection` interface. We need other annotations because we are retrofitting the JCF with mutability properties without separate mutable and immutable interfaces. We believe that complex library code, such as the JCF, requires more annotations to maximize flexibility and that this is a one-time cost. Consequently, client code will be easier to annotate because it can rely on the library's strong mutability guarantees. Constraint-based type inference [14] is a promising direction for future work to reduce the annotation burden. The detailed results are listed in Appendix B.1.

5.2.2 Case Study 2 – Constrictor Benchmarks. To evaluate our system's expressiveness for abstract immutability, we ran it on the Constrictor [28] inheritance and non-inheritance benchmarks. We translated their Python benchmark into Java and type-checked it using our system. For their `viewmethod` and `immutable` method annotation, we annotated the receiver of the method as `@RO`, and for fields that are not part of the abstract state defined by `viewmethod`, we annotated them as `@Assign`; for their `immutable` class annotation, we annotated the class and its subclasses as `@Imm`. We type-checked the 1,184 lines of non-comment code after adding 12 `@Assign`, 16 `@Imm`, and 85 `@RO` annotations. Our system always terminates and reports errors that Constrictor misses because our type checker does not perform loop unrolling. However, our system could not detect view-immutability design violations that leak information from outside the abstract state. Supporting such method contracts remains future work. The detailed results are listed in Appendix B.2.

5.2.3 Case Study 3 – PICO Test Suite. We stress-tested our system's features and error reporting using its test suites. These are handwritten test cases for the checker rather than realistic programs: each is written to exercise a particular combination of qualifiers, so the annotation density is higher than in the case studies above. We type-checked 1,738 lines of non-comment code with 665 annotations. The details are listed in Appendix B.3.

5.3 Limitations

While our system is expressive, we found a few limitations during the case study. In this subsection, we outline representative limitations of our type-system design to motivate future work.

Immutability from Freeze. Our system does not support the freeze pattern, in which an object is built mutably and then converted to immutable. When constructing an immutable collection, its elements are added gradually rather than passed to the constructor. This limitation is well known

in object immutability type systems [31], and we plan to add a uniqueness system to support this feature.

Lazy Initialization. Our system does not directly support lazy initialization of fields. Currently, we annotate lazily initialized fields such as `String.hash` as `@Assign` to permit initialization. However, using `@Assign` implies that the field is not part of the abstract state, which is semantically imprecise because its value should remain unchanged after initialization. We plan to extend PICO to enforce this stabilization property directly.

Iterator and Outer Receiver Dependence. Our system adds the `@RDM` qualifier to support class mutability polymorphism and express receiver-dependent types. However, during the case study, we found that a field's mutability could depend on the outer receiver's mutability. For example, an iterator can hold an entry field that comes from the outer collection class. To support this feature, we can extend `@RDM` to indicate that a field's mutability depends on the outer receiver's mutability.

Calling a Mutation Method from a Polymorphic Constructor. Our system uses `@RDM` to indicate that a class can be used to create both mutable and immutable objects. It would be natural to expect that mutation methods can be invoked from the constructor because the underlying object is still being initialized. However, `@RDM` is not a subtype of `@Mut`, so mutation methods cannot be called from the constructor. Supporting such safe mutation method calls during initialization remains future work.

Despite these limitations, our system represents a significant step forward in practical immutability type systems. It is the first system to combine transitive immutability, abstract immutability, and class mutability polymorphism in a unified, sound framework. The case studies demonstrate that our system can type-check real-world Java code, including the JCF, and its integration with the Checker Framework makes it directly usable in practice. The limitations identified above are understood challenges and represent directions for future work rather than fundamental obstacles.

6 Related Work

IGJ and OIGJ. IGJ adds object immutability and readonly references to Java and formalizes them using Java generics by adding an immutability type parameter. OIGJ extends IGJ by adding an extra ownership parameter to enforce the owner-as-dominator and prolong the initialization phase to the object's owner's constructor. IGJ and OIGJ demonstrate that Java generics are an effective vehicle for mutability polymorphism. PICO targets a complementary design point: instead of threading a mutability type parameter through each class declaration, PICO treats mutability polymorphism as a class-level qualifier, so an `@RDM` superclass may have `@Mut` or `@Imm` subclasses and may be instantiated at either mutability. This helps retrofit existing hierarchies because mutable and immutable variants can share one hierarchy without rewriting every class as generic over mutability. PICO also makes subclassing and method overriding part of the core design by formalizing how polymorphic and concrete class qualifiers interact across the class hierarchy.

PICO also differs from IGJ and OIGJ in its treatment of assignability and abstract state. Javari, IGJ, and OIGJ support assignable fields as a programming mechanism, but their formal models do not give a separate abstract-state theorem for the benign mutations enabled by such fields; PICO defines the abstract state from mutability and assignability annotations and proves abstract-, concrete-, readonly-, and transitive-state preservation in one system. The `@Lost` qualifier addresses cross-type aliasing when assignable receiver-dependent fields are accessed through readonly references: a blanket ban on all readonly writes would also reject benign updates permitted by assignability, whereas PICO rejects only the cases whose receiver-dependent mutability has been hidden.

Jimuva. Another line of work in object immutability is Jimuva [23]. Their motivation is to support object immutability in an open-world setting, which restricts the language to a more conservative setting. Jimuva uses ownership types to specify an object's state and observational properties, whereas PICO directly uses mutability and assignability to determine the abstract state. We formalize the more permissive notion of abstract immutability, whereas Jimuva establishes the stricter notion it calls concrete immutability. Jimuva's extension uses a lexically scoped region to prolong the initialization phase and enable conversion between mutable and immutable objects. This approach avoids a complex alias-tracking type system. Its lightweight uniqueness tracking could be added to PICO.

Glacier. Coblenz et al. [10] propose a language extension that supports class immutability and presents a user study to show the effectiveness of immutability in programming and the usability of type annotations. It is simple and easy to use, but it does not support mutation through selected fields or class mutability polymorphism, which makes it less practical. Glacier-style class immutability can be expressed directly in PICO by declaring the class as `@Imm`. In that case, the type use of that class is immutable by default, so the resulting behaviour closely matches Glacier while still allowing PICO's additional features when needed.

Constrictor. Kinsbruner et al. [28] propose view object immutability, which distinguishes the abstract state from the transitive state and is verified through model checking. PICO instead uses lighter-weight type checking and provides clearer error messages. Constrictor's `@viewmethod` permits changes outside the abstract state to be ignored by the view-immutability property; supporting an analogous guarantee in PICO remains future work.

CiFi. Roth et al. propose CiFi [40], an analysis of fine-grained assignability and immutability. Their immutability and assignability lattices are finer-grained than PICO's to increase the precision of the analysis. Their system supports transitive immutability, lazy initialization, and more. They do not support abstract immutability, class mutability polymorphism, or readonly references, although they suggest abstract immutability would be easy to add. CiFi trades soundness for precision and therefore provides no formal soundness proof.

Readonly References. Javari [48] is the first language extension to add a readonly reference to Java. Additional analysis is required to determine the object's immutability property. It also supports mutation through selected fields and has a more implicit viewpoint adaptation rule, designating fields as *this-mutable* or *this-assignable*. ReIm [24] is another language extension adding a readonly reference to Java. It is simpler and focuses on inferring method purity. ReIm supports viewpoint adaptation and context-sensitivity, which PICO also supports and mechanizes. In practice, the systems agree on many common call patterns, but their viewpoint adaptation uses different contexts: ReIm adapts from the assignment target's qualifier [24, 35], while PICO adapts method results from the receiver qualifier. ReIm does not support object immutability.

roDOT [16] encodes readonly references in Dependent Object Types (DOT) using path-dependent, intersection, and union types. It does not enforce object immutability or distinguish abstract state. $F_{<:M}$ [30] uses seal values to provide dynamic immutability safety, under which illegal writes get stuck, and uses readonly types to prove static safety after removing runtime seals. We formalize PICO by introducing a mutation runtime exception for illegal writes as a form of dynamic safety, and we prove that there is no mutation exception for well-typed programs. PICO has no runtime seal value, but an immutable object provides analogous protection: an illegal write under a TS scope produces a mutation exception. $F_{<:M}$ does not support object immutability or abstract immutability.

GUT [13] is a type system that structures heap topology and enforces the owner-as-modifier encapsulation property. The write access to the object is determined by whether the object is

accessed by its owner. Otherwise, the user can only obtain read access to the object. Boyland [5] argued that readonly references should not be added into Java unless transitivity is made clear, context-sensitivity is supported, and observational exposure is prevented. PICO addresses the former two issues by introducing `@RDM`. Observational exposure cannot be solved by an immutability system alone. PICO could incorporate safe methods [8], Joe$_3$'s type system [36], or Universe Encapsulation Types [38] to prevent observational exposure.

Reference Capabilities. Another line of work is reference capabilities, which is a more general system to enforce various properties, including immutability, uniqueness, and isolation. It is useful in enforcing safe parallelism [21], concurrency [7, 9, 44, 45], and memory management [3]. They do not support abstract immutability because their intended properties require transitive-state preservation rather than PICO's more permissive abstract-state preservation.

C++. C++ has a built-in const qualifier to indicate the immutability property. Eyolfson and Lam [18] conduct an empirical study demonstrating that the const keyword is not expressive enough for real-world use. They propose a static analysis algorithm [17] to verify that exposed fields are not modified in const methods. However, their system only supports abstract immutability related to caching, not move-to-front optimizations due to its design. In PICO, we support both abstract immutability patterns.

Spec#. Leino and Müller [31] use Spec#'s dynamic ownership to encode object immutability. An object is frozen if the ownership is transferred to a special owner called a freezer. Verification is performed by generating verification conditions and delegating them to an SMT solver. It is more expressive than PICO because it can freeze a mutable object. Supporting such mutable-to-immutable conversion in PICO is an interesting direction for future work; we plan to explore this using uniqueness types.

Scala and Kotlin. Scala [42] provides immutable collections in the standard library, while Kotlin's standard library separates read-only and mutable collection interfaces and provides immutable persistent collections through `kotlinx.collections.immutable` [26]. PICO's class mutability polymorphism instead retrofits JCF with a mutability property while maintaining a single collection hierarchy.

Rust. Rust [33] prevents unwanted mutation by using ownership and borrowing in its built-in type system. If a reference is not mutably borrowed, it cannot be mutated through the reference. Rust's immutability is also transitive, and programmers can use `RefCell` fields to permit interior mutation, which corresponds to abstract immutability in PICO. The principal difference between PICO's and Rust's immutability invariants is that PICO permits a mutable and a readonly reference to alias the same mutable object. In Rust, if a reference is a mutable borrow, other references to the same object are not allowed.

7 Conclusion

We present PICO, a sound object immutability type system whose viewpoint adaptation rules support transitive immutability and prevent unsound cross-type aliasing. By independently strengthening assignability and mutability adaptation, PICO provides abstract-, concrete-, readonly-, and transitive-state preservation, together with class mutability polymorphism and safe inheritance and overriding. Our Rocq mechanization proves soundness and four state-preservation guarantees, while our checker retrofits mutability properties in JCF and type-checks additional benchmarks, making PICO a practical step toward sound immutability for real-world OO codebases.

Data availability

All materials required to reproduce the evaluation—including the PICO type checker implementation, annotated JDK, benchmarks, Rocq proofs, and instructions—are available in the archival artifact [50]. The type checker is built on the EISOP Checker Framework, available at https://github.com/eisop/checker-framework, and was developed at https://github.com/opprop/immutability. The annotated JDK and the benchmarks are included in the artifact.

Acknowledgments

We thank the reviewers for their valuable feedback on this paper. We are grateful to Alex Cook, John Boyland, and Marat Akhin for their comments on this manuscript. We thank Arie Gurfinkel for bringing Rust's interior mutability to our attention. We also thank Jenny Xiang and Sean McLaughlin for their discussions and support. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants program, RGPIN-2020-05502 and RGPIN-2026-06207, an Early Researcher Award from the Government of Ontario, and a Fall 2023 Amazon Research Award. This work was also supported by AFRL contract FA8750-15-C-0010. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL, DARPA, NSERC, or the Governments of Ontario, Canada, or the United States.

A Formalization

A.1 Static Lookup Functions

In this subsection, we present lookup functions that are useful for defining static well-formedness and type rules.

Bound Lookup. The function "bound(C)" yields the class declaration on class C. For Object, the class declaration is RDM.

$$\frac{}{\text{bound}(\text{Object}) = \text{RDM}}$$

$$\frac{q_c \text{ class } C \text{ extends } D \ \{\overline{fd}\ kd\ \overline{md}\}}{\text{bound}(C) = q_c}$$

Parent Lookup. The function "parentClass(C)" yields the parent class of class C. For Object, it is the root class and there is no parent class for Object.

$$\frac{}{\text{parentClass}(\text{Object}) = \emptyset}$$

$$\frac{q_c \text{ class } C \text{ extends } D \ \{\overline{fd}\ kd\ \overline{md}\}}{\text{parentClass}(C) = D}$$

Field Lookup. The function "fields(C)" yields the sequence of fields of class C, which contains fields from its parent class. For Object, the result is empty because there is no field in Object class.

$$\frac{}{\text{fields}(\text{Object}) = \emptyset}$$

$$\frac{q_c \text{ class } C \text{ extends } D \ \{\overline{fd}\ kd\ \overline{md}\} \qquad \text{fields}(D) = \overline{fd'}}{\text{fields}(C) = \overline{fd'}, \overline{fd}}$$

Field Type Lookup. The function "$^{s}\text{fType}(C, f)$" yields the field type and qualifier of field f in class C, including assignability and mutability qualifier.

$$\frac{\text{fields}(\text{C}) = \overline{a\ q_f\ C_f f}}{^{s}\text{fType}(C, f_i) = a_i\ q_{fi}\ C_{fi}}$$

Method Signature Lookup. The function "mSig(C, m)" yields the method signature of method m in class C, including the return type, receiver type, and parameter types.

$$\frac{q_c \text{ class } C \text{ extends } D \ \{\overline{fd}\ kd\ \overline{md}\} \qquad md_i = p\ T'\ m(T \text{ this}, \ \overline{T''\ x})\{s;\ \text{return } x\}}{\text{mSig}(C, m) = p\ T'\ m(T \text{ this}, \ \overline{T''\ x})}$$

Constructor Signature Lookup. The function "conSig(C)" yields the constructor signature of class C, including the receiver type and parameter types.

$$\frac{q_c \text{ class } C \text{ extends } D \ \{\overline{fd}\ kd\ \overline{md}\} \qquad kd = q_c\ C\ (\overline{T\ x}, \ \overline{T'\ y})\ \{\text{super}(\overline{x});\ \overline{\text{this}.f = y}\}}{\text{conSig}(C) = q_c\ C\ (\overline{T\ x}, \ \overline{T'\ y})}$$

A.2 Subclassing, Subqualifier, and Subtyping

In this subsection, we present the subclassing, subqualifier, and subtyping rules of PICO in Figure 10, Figure 11, and Figure 12.

$$\frac{}{q <:_q \text{ RO}} \text{ (Q-Top)} \qquad \frac{}{\text{Bottom} <:_q q} \text{ (Q-Bottom)}$$

$$\frac{q \neq \text{Lost}}{q <:_q q} \text{ (Q-Reflexivity)} \qquad \frac{q <:_q q' \qquad q' <:_q q''}{q <:_q q''} \text{ (Q-Transitivity)}$$

Fig. 10. The qualifier ordering rules.

$$\frac{q_c \text{ class } C \text{ extends } D \ \{\overline{fd}\ kd\ \overline{md}\}}{C \sqsubseteq D} \text{ (SC-Extends)}$$

$$\frac{C \sqsubseteq C' \qquad C' \sqsubseteq D}{C \sqsubseteq D} \text{ (SC-Transitivity)} \qquad \frac{}{C \sqsubseteq C} \text{ (SC-Reflexivty)}$$

Fig. 11. The subclassing rules.

$$\frac{q <:_q q' \qquad C \sqsubseteq D}{q\ C <:\ q'\ D} \text{ (ST-SubType)}$$

Fig. 12. The subtyping rules.

A.3 Static Well-Formedness

In this subsection, we present the static well-formedness rules of PICO in Figure 13. Declaration well-formedness uses AS viewpoint adaptation because it preserves mutability information; CS shares this mutability adaptation, while RS and TS use their stricter adaptation when type-checking method bodies.

WF-TypeUse-appendix expresses a type use composed of a mutability qualifier q with a class identifier C is well-formed if the qualifier q is a subtype of q after adapting the class declaration. The function "bound(C)" yields the class declaration qualifier of class C.

WF-Field-appendix expresses that the field declaration is well-formed if the field declaration is a valid type use as defined by WF-TypeUse-appendix.

WF-Cons-appendix expresses that the constructor declaration is well-formed in class C if the constructor parameter types are valid type use as defined by WF-TypeUse-appendix. Since we use constructor parameters to initialize fields, we require that the constructor parameter types be subtypes of the fields' types.

We require the constructor's return type to match the class declaration because our language does not support constructor overloading. In our checker implementation, a class with a RDM class declaration can have mutable, immutable and RDM return type constructors. Because every constructor will call its super constructor, we require that the super constructor call is valid as defined by checkSuperCall-appendix. The rule states that, from the viewpoint of this constructor, the super constructor's return type should match this constructor's return type, and its arguments should be subtypes of the parameter types, adapted from the viewpoint of this constructor's return type. The rules ensure correct object initialization when the superclass and this class have different constructor return types.

WF-Meth-appendix expresses that the method declaration is well-formed in class C if the method receiver type, formal parameter types and return types are well-formed as defined by WF-TypeUse-appendix. After type-checking its body, x should be defined in the resulting static environment, and it should be a subtype of the return type. The type rules are defined in Section A.4, where

statement typing has both an input and an output environment to allow local variable declaration. For method overriding, the method receiver type and formal parameter types are the supertype of the method in the parent class after adaptation, which preserves contravariant typing, while the method return type is a subtype of the method in the parent class after adaptation, which preserves covariant typing. This rule ensures method overriding does not inherit mutation methods when the parent class is an RDM class, and the child class is an immutable class.

WF-Class-appendix expresses that a class is well-formed if the class contains well-formed fields, a constructor, and methods. The class should not inherit itself, and the class declaration should be the same after it adapts its parent class's declaration. This design ensures immutable classes can extend RDM or immutable classes, mutable classes can only extend RDM or mutable classes, and RDM classes can only extend RDM classes. Otherwise, the mutability property can be violated, for example, if an immutable class can extend a mutable class, it is allowed to pass immutable references to a method expecting mutable references, leading to runtime errors.

WF-Prog-appendix express that a program is well-formed if all class declarations are well-formed and the main statement is well-formed.

WF-Env-appendix expresses that the static environment is well-formed if it contains a mapping from receiver to type and the type is well-formed. If there are other mappings from identifiers to types, the types need to be well-formed as well.

A.4 Type Rules

In this subsection, we present the type rules for PICO in Figure 14. The type rules are defined with the assumptions that all types used in the rule and the type environment are well-formed. The viewpoint adaptations are parametrized by the method scope p, which is passed from the enclosing method declaration.

Expression Typing. Typing an expression takes a static environment ${}^{s}\Gamma$, a method scope p, and an expression e, yielding a qualified type.

ET-Null-appendix expresses that a null value is well-typed with any qualified type.

ET-Var-appendix expresses that a variable x is well-typed if the variable is defined in the static environment ${}^{s}\Gamma$.

ET-FldRead-appendix expresses that a field access $x.f$ is well-typed if the field f is defined in the class of x and the resulting field type is adapted from the viewpoint of x.

Statement Typing. Typing a statement takes a static environment ${}^{s}\Gamma$, a method scope p, and a statement s, yielding a new static environment ${}^{s}\Gamma'$.

ST-LocalVar-appendix expresses that a local variable declaration $T\ x$ is well-typed if the variable does not exist in the old static environment.

ST-VarAss-appendix expresses that a variable assignment $x := e$ is well-typed if the expression e is well-typed and the expression type is a subtype of the variable type.

ST-FldWrite-appendix expresses that a field assignment $x.f := y$ is well-typed if the field f is defined in the class of x and the type of y should be a subtype of the adapted field type.

ST-New-appendix expresses that a constructor call $x :=$ new $q_c\ C(\overline{y})$ is well-typed if the arguments are subtypes of the adapted parameter types, and the qualifier of the constructor invocation is an adapted subtype of the qualifier of the constructor return type. This rule allows polymorphic object creation. When creating immutable objects, the RDM fields are initialized with Imm arguments; when creating mutable objects, the RDM fields are initialized with Mut arguments.

ST-Call-appendix expresses that a method call $x := y.m(\overline{z})$ is well-typed if the receiver y is well-typed, the method is defined in the class of y, the method signature is adapted from the viewpoint of y, and the arguments are subtypes of the adapted parameter types. One special case we allow

$$\frac{(q \triangleright_{m,\text{AS}} \text{bound}(C)) <: q}{q\ C\ \text{OK}}\ \text{(WF-TypeUse)} \qquad \frac{q_f\ C_f\ \text{OK}}{a\ q_f\ C_f\ f\ \text{OK}}\ \text{(WF-Field)}$$

$$\frac{\text{parentClass}(C) = D \qquad \text{conSig}(D) = q_c'\ D(\overline{q_s\ D'\ x}) \qquad q_c = \text{bound}(C) \qquad q_c \triangleright_{m,\text{AS}} q_c' = q_c \qquad \overline{q_x\ C_x <: (q_c \triangleright_{m,\text{AS}} q_s)\ D'}}{\text{checkSuperCall}(C, \overline{q_x\ C_x\ x})\ \text{OK}}\ \text{(checkSuperCall)}$$

$$\frac{\text{checkSuperCall}(C, \overline{q'\ C'\ x})\ \text{OK} \qquad \overline{q'\ C'}\ \text{OK},\ \overline{q''\ C''}\ \text{OK} \qquad q = \text{bound(C)} \qquad {}^s\text{fType}(C, f_i) = \_\ q_{fi}\ C_{fi} \qquad \overline{(q \triangleright_{m,\text{AS}} q'')\ C'' <: (q \triangleright_{m,\text{AS}} q_{fi})\ C_{fi}}}{q\ C\ (\overline{q'\ C'\ x},\ \overline{q''\ C''\ y})\ \{\text{super}(\overline{x});\ \overline{\text{this}.f = y}\}\ \text{OK in C}}\ \text{(WF-Cons)}$$

$$\frac{\begin{array}{c} q\ C\ \text{OK},\ q'\ C'\ \text{OK},\ \overline{q''\ C''}\ \text{OK} \qquad {}^s\Gamma, p \vdash s \dashv {}^s\Gamma' \\ {}^s\Gamma' \vdash x : q_x\ C_x \qquad q_x\ C_x <:\ q'\ C' \qquad \text{parentClass}(C) = D \\ q_c = \text{bound}(C) \qquad (p \in \{\text{RS}, \text{TS}\} \Rightarrow \text{NoMut}(C, m)) \\ \text{mSig}(D, m) = p_s\ q_s'\ C'\ m(q_s\ D\ \text{this},\ \overline{q_s''\ C''\ x}) \Rightarrow \big(p = p_s \wedge \\ q' <:_q\ q_c \triangleright_{m,\text{AS}} q_s' \wedge q_c \triangleright_{m,\text{AS}} q_s <:_q\ q \wedge \overline{q_c \triangleright_{m,\text{AS}} q_s'' <:_q\ q''}\big) \end{array}}{p\ q'\ C'\ m(q\ C\ \text{this},\ \overline{q''\ C''\ x})\{s;\ \text{return}\ x\}\ \text{OK in C}}\ \text{(WF-Meth)}$$

$$\frac{\overline{fd}\ \text{OK, } kd\ \text{OK in C},\ \overline{md}\ \text{OK in C} \qquad C \neq D \qquad q_c \triangleright_{m,\text{AS}} \text{bound}(D) = q_c}{q_c\ \text{class}\ C\ \text{extends}\ D\ \{\overline{fd}\ kd\ \overline{md}\}\ \text{OK}}\ \text{(WF-Class)}$$

$$\frac{P = \overline{cd}\ s \qquad \overline{cd}\ \text{OK} \qquad s\ \text{OK}}{P\ \text{OK}}\ \text{(WF-Prog)}$$

$$\frac{{}^s\Gamma = \{\text{this} \mapsto q\ C, \overline{x \mapsto q'\ C'}\} \qquad q\ C\ \text{OK} \qquad \overline{q'\ C'}\ \text{OK}}{{}^s\Gamma\ \text{OK}}\ \text{(WF-Env)}$$

Fig. 13. The static well-formedness rules of PICO.

is that a RO receiver can also call a method with a RDM receiver. Method scope annotations are ordered by the submethod relation $p' <:_m p$. A method checked under scope p may call a method with scope p' when $p' <:_m p$, meaning that p' selects equally or more conservative viewpoint adaptations.

ST-Seq-appendix expresses a sequence of statements $s; s$ is well-typed if the first statement is well-typed and the second statement is well-typed in the updated static environment.

In summary, we present the syntax, static viewpoint adaptation and well-formedness rules, and type rules.

A.5 Runtime Helper Functions

In this subsection, we present helper functions for defining runtime well-formedness and operational semantics.

Runtime Mutability Type Lookup. The function "${}^r\text{MType}(h, \iota)$" returns the runtime mutability type of the object at address ι in heap h.

$$\frac{h(\iota)_{\downarrow_1} = q_r\ C}{{}^r\text{MType}(h, \iota) = q_r}$$

$$\frac{}{{}^s\Gamma, p \vdash \text{null} : T} \quad \text{(ET-Null)} \qquad \frac{{}^s\Gamma(x) = q_x\ C_x}{{}^s\Gamma, p \vdash x : q_x\ C_x} \quad \text{(ET-Var)}$$

$$\frac{{}^s\Gamma, p \vdash x : q_x\ C_x \qquad {}^s\text{fType}(C_x, f) = \_\ q_f\ C_f \qquad q_x \rhd_{m,p} q_f = q'}{{}^s\Gamma, p \vdash x.f :\ q'\ C_f} \quad \text{(ET-FldRead)}$$

$$\frac{{}^s\Gamma' = {}^s\Gamma + \{x \mapsto T\}}{{}^s\Gamma, p \vdash T\ x \dashv {}^s\Gamma'} \quad \text{(ST-LocalVar)}$$

$$\frac{{}^s\Gamma, p \vdash e : T \qquad {}^s\Gamma, p \vdash x : T_x \qquad T <:\ T_x}{{}^s\Gamma, p \vdash x := e \dashv {}^s\Gamma} \quad \text{(ST-VarAss)}$$

$$\frac{\begin{array}{c}{}^s\Gamma, p \vdash x : q_x\ C_x \qquad {}^s\Gamma, p \vdash y : q_y\ C_y \qquad {}^s\text{fType}(C_x, f) = a_f\ q_f\ C_f \\ q_y\ C_y <:\ (q_x \rhd_{m,p} q_f)\ C_f \qquad q_x \rhd_{a,p} a_f = \text{Assign}\end{array}}{{}^s\Gamma, p \vdash x.f := y \dashv {}^s\Gamma} \quad \text{(ST-FldWrite)}$$

$$\frac{\begin{array}{c}{}^s\Gamma, p \vdash x : q_x\ C_x \qquad {}^s\Gamma, p \vdash \overline{y : q_y\ C_y} \qquad \text{conSig}(C) = q_c'\ C(\overline{q_p\ C_p\ x}) \\ \overline{q_y\ C_y <: (q_c \rhd_{m,AS} q_p)\ C_p} \qquad q_c \rhd_{m,AS} q_c' = q_c \qquad (q_c \rhd_{m,AS} q_c')\ C <: q_x\ C_x\end{array}}{{}^s\Gamma, p \vdash x := \text{new}\ q_c\ C(\overline{y}) \dashv {}^s\Gamma} \quad \text{(ST-New)}$$

$$\frac{\begin{array}{c}{}^s\Gamma, p \vdash x : q_x\ C_x \qquad {}^s\Gamma, p \vdash y : q_y\ C_y \qquad {}^s\Gamma, p \vdash \overline{z : q_z\ C_z} \\ \text{mSig}(m, C_y) = p'\ q'\ C'\ \text{m}(q\ C\ \text{this},\ \overline{q''\ C''x}) \qquad p' <:_m\ p \\ (q_y \rhd_{m,p} q')\ C' <: q_x\ C_x \qquad \overline{q_z\ C_z <:\ (q_y \rhd_{m,p} q'')\ C''} \\ q_y\ C_y <: (q_y \rhd_{m,p} q)\ C \vee (q_y = \text{RO} \wedge q = \text{RDM} \wedge C_y \sqsubseteq C)\end{array}}{{}^s\Gamma, p \vdash x := y.m(\overline{z}) \dashv {}^s\Gamma} \quad \text{(ST-Call)}$$

$$\frac{{}^s\Gamma, p \vdash s_1 \dashv {}^s\Gamma' \qquad {}^s\Gamma', p \vdash s_2 \dashv {}^s\Gamma''}{{}^s\Gamma, p \vdash s_1; s_2 \dashv {}^s\Gamma''} \quad \text{(ST-Seq)}$$

Fig. 14. The type rules of PICO.

Runtime Field Lookup. The function "$h(\iota.f)$" returns the value stored in field f of the object at address ι in heap h. We use shorthand notation $\iota.f$ for field access.

$$\frac{h(\iota)_{\downarrow_2} = \overline{fv} \qquad fv(f) = v}{h(\iota.f) = v}$$

Object Addition. The function "$h + o = (h', \iota)$" adds object o to heap h, producing heap h' and fresh address ι.

$$\frac{\iota \notin \text{dom}(h) \qquad h' = h + \{\iota \mapsto o\}}{h + o = (h', \iota)}$$

Field Update. The function "$h[\iota.f = v] = h'$" returns the heap h' after updating the field f of object at address ι with value v.

$$\frac{\begin{array}{c}v = \text{null}_a \vee (v = \iota' \wedge \iota' \in \text{dom}(h)) \qquad h(\iota) = ({}^rT, \overline{fv}) \\ f \in \text{dom}(\overline{fv}) \qquad \overline{fv}' = \overline{fv}[f \mapsto v] \qquad h' = h + \{\iota \mapsto ({}^rT, \overline{fv}')\}\end{array}}{h[\iota.f = v] = h'}$$

A.6 Runtime Viewpoint Adaptation

$$\frac{}{\mathrm{Mut}_r \triangleright_r \mathrm{RDA} = \mathrm{Assign}} \quad \text{(R-A-VPA-1)} \qquad \frac{}{q_r \triangleright_r \mathrm{Assign} = \mathrm{Assign}} \quad \text{(R-A-VPA-2)}$$

$$\frac{a \neq \mathrm{Assign} \wedge (a \neq \mathrm{RDA} \vee q_r \neq \mathrm{Mut}_r)}{q_r \triangleright_r a = \mathrm{Final}} \quad \text{(R-A-VPA-3)}$$

A.7 Operational Semantics

The evaluation rules are defined in big-step operational semantics in Figure 15. We assume that the runtime environment $^r\Gamma$ is well-formed and the heap h is well-formed for every evaluation rule. We provide a mutation exception as a form of dynamic safety: an illegal write raises a mutation exception. Both expression and statement evaluation may get stuck for other reasons, such as missing bindings, lookup failures, or null dereferences. Our safety result specifically rules out mutability and assignability violations in well-typed programs. Nullness-related stuck states (such as dereferencing $null_a$) are outside the scope of PICO.

Expression evaluation. Evaluation of an expression yields a value. It does not change the heap or the runtime environment.

Statement evaluation. The transition relation $^r\Gamma, h, s \rightsquigarrow h', {}^r\Gamma'$ expresses that evaluating a statement with initial runtime environment $^r\Gamma$ and heap h will result in a new runtime environment $^r\Gamma'$ and a new heap h'.

The OS-S-LocalVar-appendix and OS-S-VarAss-appendix rules are straightforward, as they update the runtime environment with a null and the new value, respectively.

The OS-S-FldWrite-appendix rule expresses that a field assignment will update the heap by assigning the value to the field of the object. If the field is effectively assignable, the field is updated in the heap. If the field is not assignable, a mutation exception will be raised as expressed by OS-S-FldWrite-EXC-appendix rule.

The OS-S-New-appendix rule expresses that object creation will first create a new object. It is interesting because we allow creation of RDM object statically. At runtime, object creation will be adapted from the viewpoint runtime mutability qualifier of `this`, which can only be Imm_r or Mut_r. The runtime viewpoint adaptation rules ensure that the created object's runtime mutability qualifier is well-formed, i.e., it is either Imm_r or Mut_r.

The OS-S-Call-appendix rule states that an object invocation first creates a new runtime environment for the method body and updates the heap by evaluating it. After executing the method body, the old runtime environment is updated by the variable assignment.

The OS-P-Prog-appendix rule states that program evaluation begins with the main statement, with an empty runtime environment and an empty heap.

$$\frac{}{{}^{r}\Gamma, h, \text{null} \rightsquigarrow \text{null}_a, \text{OK}} \quad \text{(OS-E-NULL)} \qquad \frac{{}^{r}\Gamma(x) = v}{{}^{r}\Gamma, h, x \rightsquigarrow v, \text{OK}} \quad \text{(OS-E-VAR)}$$

$$\frac{{}^{r}\Gamma(x) = \iota \qquad h(\iota.f) = v_f}{{}^{r}\Gamma, h, x.f \rightsquigarrow v_f, \text{OK}} \quad \text{(OS-E-FLDREAD)}$$

$$\frac{{}^{r}\Gamma' = {}^{r}\Gamma + \{x \mapsto \text{null}_a\}}{{}^{r}\Gamma, h, T\ x \rightsquigarrow {}^{r}\Gamma', h, \text{OK}} \quad \text{(OS-S-LOCALVAR)}$$

$$\frac{{}^{r}\Gamma, h, e \rightsquigarrow v \qquad {}^{r}\Gamma' = {}^{r}\Gamma[x \mapsto v]}{{}^{r}\Gamma, h, x := e \rightsquigarrow {}^{r}\Gamma', h, \text{OK}} \quad \text{(OS-S-VARASS)}$$

$$\frac{{}^{r}\Gamma(x) = \iota_x,\ {}^{r}\Gamma(y) = v_y \qquad \text{canAssign}(h, \iota_x, f) \qquad h' = h[\iota_x.f = v_y]}{{}^{r}\Gamma, h, x.f := y \rightsquigarrow {}^{r}\Gamma, h', \text{OK}} \quad \text{(OS-S-FLDWRITE)}$$

$$\frac{{}^{r}\Gamma(x) = \iota_x,\ {}^{r}\Gamma(y) = v_y \qquad \neg\text{canAssign}(h, \iota_x, f)}{{}^{r}\Gamma, h, x.f := y \rightsquigarrow {}^{r}\Gamma, h, \text{MutEXC}} \quad \text{(OS-S-FIELDWRITEEXC)}$$

$$\frac{\begin{array}{c}{}^{r}\Gamma(\text{this}) = \iota_{\text{this}} \qquad {}^{r}\Gamma(\overline{y}) = \overline{v_y} \qquad \text{fields(C)} = \overline{f} \\ q_a = \begin{cases} \text{Imm}_r \ if\ {}^{r}\text{MType}(h, \iota_{\text{this}}) \rhd_r q_c\ =\ \text{Imm} \\ \text{Mut}_r \ if\ {}^{r}\text{MType}(h, \iota_{\text{this}}) \rhd_r q_c\ =\ \text{Mut} \end{cases} \\ o = (q_a\ C, \overline{f \mapsto v_y}) \qquad h + o = (h', \iota) \qquad {}^{r}\Gamma' = {}^{r}\Gamma[x \mapsto \iota]\end{array}}{{}^{r}\Gamma, h, x := \text{new}\ q_c\ C(\overline{y}) \rightsquigarrow {}^{r}\Gamma', h', \text{OK}} \quad \text{(OS-S-NEW)}$$

$$\frac{\begin{array}{c}{}^{r}\Gamma = \{x \mapsto v_x,\ y \mapsto \iota_y,\ \overline{z \mapsto z_w}, \ldots\} \qquad h(\iota_y)_{\downarrow_1} = \_\ C_y \\ \text{mBody}(m, C_y) = \{s; \text{return}\ x;\} \qquad {}^{r}\Gamma' = \{\text{this} \mapsto \iota_y,\ \overline{z \mapsto v_z}\} \\ {}^{r}\Gamma', h, s \rightsquigarrow {}^{r}\Gamma'', h', \text{OK} \qquad {}^{r}\Gamma''(x) = v_x \qquad {}^{r}\Gamma''' = {}^{r}\Gamma[x \mapsto v_x]\end{array}}{{}^{r}\Gamma, h, x := y.m(\overline{z}) \rightsquigarrow {}^{r}\Gamma''', h', \text{OK}} \quad \text{(OS-S-CALL)}$$

$$\frac{\begin{array}{c}{}^{r}\Gamma, h, s_1 \rightsquigarrow {}^{r}\Gamma', h', \text{OK} \\ {}^{r}\Gamma', h', s_2 \rightsquigarrow {}^{r}\Gamma'', h'', \text{OK}\end{array}}{{}^{r}\Gamma, h, s_1; s_2 \rightsquigarrow {}^{r}\Gamma'', h'', \text{OK}} \quad \text{(OS-S-SEQ)}$$

$$\frac{\emptyset, s \rightsquigarrow h}{\overline{cd}\ s \rightsquigarrow h, \text{OK}} \quad \text{(OS-P-PROG)}$$

Fig. 15. The big step operational semantics of PICO.

B Evaluation details

B.1 Case study 1 - JDK library

We performed our first case study by annotating the Java Collections Framework (JCF) in OpenJDK 17 and evaluating PICO's practicality. Tables 1–4 list the files checked and the annotations added. The benchmark includes representative JCF classes, including Collection, List, Map, Set, Queue, Deque, and their implementations.

Table 1. Java Collection Framework Part 1

File	LOC	Assignable	Imm	RO	RDM	Mut	PolyMut
AbstractCollection.java	184	0	0	16	1	6	5
AbstractList.java	480	4	0	25	9	22	10
AbstractMap.java	376	2	4	24	1	5	9
AbstractQueue.java	52	0	0	2	1	4	0
AbstractSequentialList.java	70	0	0	4	1	4	0
AbstractSet.java	60	0	0	6	1	1	0
ArrayDeque.java	644	0	0	54	3	31	3
ArrayList.java	1063	0	0	104	7	49	15
ArrayPrefixHelpers.java	602	0	0	0	0	0	0
Arrays.java	2383	0	1	301	3	4	9
BitSet.java	645	1	0	27	3	20	0
Collection.java	88	0	0	24	1	7	2
Collections.java	3445	11	145	330	60	104	230
Comparator.java	102	0	2	13	1	0	0
Comparators.java	57	0	0	5	0	0	0
Deque.java	57	0	0	15	1	18	0
EmptyStackException.java	11	0	0	0	1	0	0
EnumMap.java	526	4	0	60	16	24	13
EnumSet.java	165	0	0	1	2	3	5
Total	11010	22	152	1011	112	302	301

Table 2. Java Collection Framework Part 2

File	LOC	Assignable	Imm	RO	RDM	Mut	PolyMut
Enumeration.java	27	0	0	0	1	0	0
FormatterClosedException.java	10	0	0	0	1	0	0
HashMap.java	1750	6	13	85	15	46	24
HashSet.java	143	0	3	19	6	6	0
Hashtable.java	893	9	4	84	10	40	10
IdentityHashMap.java	989	7	8	107	14	40	11
IllformedLocaleException.java	23	0	0	0	1	0	0
ImmutableCollections.java	1151	1	51	61	1	3	4
InputMismatchException.java	15	0	0	0	1	0	0
Iterator.java	32	0	0	1	1	3	0
JumboEnumSet.java	212	0	0	0	0	0	0
Total	5245	23	79	357	51	138	49

Table 3. Java Collection Framework Part 3

File	LOC	Assignable	Imm	RO	RDM	Mut	PolyMut
KeyValueHolder.java	49	0	2	7	0	0	0
LinkedHashMap.java	425	3	2	36	10	28	11
LinkedHashSet.java	32	0	1	0	1	0	0
LinkedList.java	663	3	0	42	8	55	6
List.java	160	0	15	41	1	12	10
ListIterator.java	29	0	0	5	1	4	0
Map.java	325	0	38	43	2	15	7
NavigableMap.java	48	0	1	4	1	2	30
NavigableSet.java	38	0	0	6	1	2	14
NoSuchElementException.java	21	0	0	0	3	0	0
Objects.java	127	0	0	10	0	0	0
Queue.java	28	0	0	3	1	4	0
Total	1945	6	59	197	29	122	78

Table 4. Java Collection Framework Part 4

File	LOC	Assignable	Imm	RO	RDM	Mut	PolyMut
RegularEnumSet.java	160	0	0	15	3	11	0
Set.java	121	0	18	33	1	6	2
SortedMap.java	36	0	1	3	1	0	13
SortedSet.java	39	0	0	3	1	1	6
Spliterator.java	160	0	0	6	5	20	0
Spliterators.java	1041	0	0	30	0	0	4
Stack.java	49	0	0	4	1	2	0
TreeMap.java	2310	14	39	171	28	86	177
TreeSet.java	191	0	1	22	4	9	18
Vector.java	685	1	0	67	5	39	2
Total	4792	15	59	354	49	174	222

B.2 Case study 2 details - Constrictor inheritance and non-inheritance benchmarks

Tables 5 and 6 list the Constrictor inheritance and non-inheritance benchmarks, with the lines checked and the annotations we added to each.

B.3 Case study 3 details - PICO test suite

Table 7 lists the files of our test suite, each written to exercise a particular combination of qualifiers.

Table 5. Inheritance benchmarks

File	LOC	Assignable	Immut	RO
ListImpl.java	14	1	2	0
MutableList.java	11	0	2	0
AlrightPoint.java	15	1	1	0
EvilPoint.java	12	0	1	0
GoodPoint.java	11	0	1	0
InauspiciousPoint.java	14	0	1	0
MaliciousPoint.java	15	0	1	0
MutablePoint.java	21	0	0	2
Point.java	16	0	1	2
WrongfullyAnnotatedMutablePoint.java	21	0	0	3
EvilHashSet.java	53	0	0	1
GenericSet.java	6	0	0	1
HashSet.java	37	0	0	1
MoveToFrontListSet.java	33	0	0	3
WrongImplMoveFrontListSet.java	39	0	0	2
ColoredShape.java	16	0	0	1
EvilMemoizedRectangle.java	11	0	1	1
EvilSquare.java	14	0	1	0
LeakyMemoizedRectangle.java	11	0	1	1
MemoizedRectangle.java	26	1	1	2
Rectangle.java	25	0	0	2
SimpleWrongImplRectangle.java	17	0	0	2
SizedShape.java	7	0	1	1
WrongfullyImplementedRectangle.java	27	0	0	3
Total	472	3	15	28

Table 6. Non-Inheritance benchmarks

File	LOC	Assignable	Immut	RO
BicounterFirst.java	22	1	0	2
BicounterSecond.java	22	1	0	2
BinarySearchTree.java	81	1	0	2
CachedList.java	21	2	0	1
Collatz.java	25	1	0	2
CounterWithAccessCount.java	20	1	0	2
DefaultDict.java	17	0	0	3
EvilBinarySearchTree.java	81	0	0	2
EvilUnionFind.java	35	0	0	1
FaithfulClass.java	14	0	0	1
FlaggedValue.java	14	0	0	1
Graph.java	50	0	0	1
ImmutablePerson.java	20	0	1	3
ImmutableRgb.java	24	0	0	3
ListWithAccessCount.java	29	1	0	1
LongLoopMutator.java	12	0	0	2
MultiplyingDictionary.java	11	0	0	1
MutablePerson.java	22	0	0	4
NumberShuffler.java	31	0	0	6
StringShuffler.java	17	0	0	2
UnionFind.java	36	0	0	1
UnreachablyMutating.java	16	0	0	2
VariableTypesMatter.java	21	0	0	4
ViewMutatingButFaithful.java	12	0	0	2
ViewNonMutatingButUnfaithful.java	16	1	0	2
WrongfullyAnnotatedCachedList.java	20	0	0	2
WrongfullyImplementedCollatz.java	23	0	0	2
Total	712	9	1	57

Table 7. PICO test suite

File	LOC	Assignable	Imm	RO	RDM	Mut	PolyMut
Arrays.java	72	0	12	5	9	8	0
AssignabilityAnnotation.java	21	6	8	0	0	0	0
Builder.java	197	0	3	0	0	1	0
CastTest.java	18	0	1	0	0	1	0
ChainOfCallLost.java	12	0	0	1	1	0	2
ClassAnnotation.java	116	0	29	10	20	18	10
CloneTest.java	37	0	0	0	0	0	0
EnumTest.java	32	0	4	1	1	4	0
FactoryPattern.java	22	0	3	1	2	3	5
Generics.java	51	0	14	4	0	11	0
HashCodeSafetyExample.java	33	0	1	1	0	1	0
ImmutableClass.java	51	1	23	3	2	7	0
ImmutableList.java	38	0	10	1	0	2	0
ImplicitMutable.java	30	0	2	0	1	4	0
LocalVariableRefinement.java	28	0	2	4	0	2	0
ObjectMethods.java	62	0	2	6	4	2	0
Operators.java	43	0	1	2	0	0	0
OverrideEquals.java	41	0	0	5	0	1	0
PICOCircularInit.java	16	0	0	0	0	0	0
PICOException.java	8	0	0	0	0	0	0
PICOFlowTest.java	24	0	2	1	2	0	5
PICOInitialization.java	97	20	33	10	22	15	0
PICOMethodInit.java	21	0	1	0	0	0	0
PICOTypeUseLocation.java	19	0	4	4	1	1	3
PolyMutableTest.java	88	0	15	1	6	13	13
RDMClass.java	25	0	5	0	2	5	0
RDMClassConstructor.java	41	0	8	3	11	6	0
RDMClassUse.java	55	0	4	2	6	5	0
RGB.java	56	0	2	3	1	4	0
RawTypeTest.java	51	0	0	4	0	2	0
ReimStudy.java	35	0	7	2	5	3	1
StaticTest.java	23	0	1	1	9	1	0
SuperClassTest.java	103	0	14	1	24	17	0
SupportedBuilderPattern.java	44	0	6	0	2	3	2
Transitive.java	30	0	0	1	0	3	0
ViewpointAdaptationRules.java	98	2	7	7	6	7	0
Total	1738	29	224	84	137	150	41

References

[1] Marat Akhin et al. 2026. Better Immutability in Kotlin aka "Value Classes 2.0": Motivation and Design Space. https://github.com/Kotlin/KEEP/blob/main/proposals/KEEP-0453-better-immutability-value-classes-motivation.md. Accessed: March 6, 2026.

[2] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. 2006. A Framework for Implementing Pluggable Type Systems. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. Association for Computing Machinery, New York, NY, USA, 57–74. doi:10.1145/1167515.1167479

[3] Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J Parkinson, and Tobias Wrigstad. 2023. Reference capabilities for flexible memory management. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 1363–1393. doi:10.1145/3622846

[4] Joshua Bloch. 2017. *Effective Java* (3 ed.). Addison-Wesley Professional, Boston, MA, USA.

[5] John Boyland. 2006. Why We Should Not Add readonly to Java (Yet). *J. Object Technol.* 5, 5 (2006), 5–29. doi:10.5381/jot.2006.5.5.a1

[6] Gilad Bracha. 2004. Pluggable Type Systems. OOPSLA 2004 Workshop on the Revival of Dynamic Languages.

[7] Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:26. doi:10.4230/LIPIcs.ECOOP.2016.5

[8] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. 2008. Minimal Ownership for Active Objects. In *Asian Symposium on Programming Languages and Systems*. Springer, Berlin, Heidelberg, 139–154. doi:10.1007/978-3-540-89330-1_11

[9] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/2824815.2824816

[10] Michael Coblenz, Whitney Nelson, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 2017. Glacier: Transitive Class Immutability for Java. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Buenos Aires, Argentina, 496–506. doi:10.1109/ICSE.2017.52

[11] Cristina David and Wei-Ngan Chin. 2011. Immutable Specifications for More Concise and Precise Verification. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications*. Association for Computing Machinery, New York, NY, USA, 359–374. doi:10.1145/2048066.2048096

[12] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muşlu, and Todd Schiller. 2011. Building and using pluggable type-checkers. In *ICSE 2011, Proceedings of the 33rd International Conference on Software Engineering*. Association for Computing Machinery, Waikiki, Hawaii, USA, 681–690. doi:10.1145/1985793.1985889

[13] Werner Dietl, Sophia Drossopoulou, and Peter Müller. 2007. Generic Universe Types. In *European Conference on Object-Oriented Programming (ECOOP'07)*. Springer-Verlag, Berlin, Heidelberg, 28–53. doi:10.1007/978-3-540-73589-2_3

[14] Werner Dietl, Michael D. Ernst, and Peter Müller. 2011. Tunable Static Inference for Generic Universe Types. In *European Conference on Object-Oriented Programming*. Springer, Berlin, Heidelberg, 333–357. doi:10.1007/978-3-642-22655-7_16

[15] Werner Dietl and Peter Müller. 2005. Universes: Lightweight Ownership for JML. *J. Object Technol.* 4, 8 (2005), 5–32. doi:10.5381/jot.2005.4.8.a1

[16] Vlastimil Dort and Ondřej Lhoták. 2020. Reference Mutability for DOT. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 18:1–18:28. doi:10.4230/LIPIcs.ECOOP.2020.18

[17] Jonathan Eyolfson. 2018. *Enforcing Abstract Immutability*. Ph. D. Dissertation. University of Waterloo. https://hdl.handle.net/10012/13507

[18] Jon Eyolfson and Patrick Lam. 2016. C++ const and Immutability: An Empirical Study of Writes-Through-const. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 8:1–8:25. doi:10.4230/LIPIcs.ECOOP.2016.8

[19] Jeffrey S Foster, Manuel Fähndrich, and Alexander Aiken. 1999. A Theory of Type Qualifiers. *ACM Sigplan Notices* 34, 5 (1999), 192–203. doi:10.1145/301631.301665

[20] Paola Giannini, Marco Servetto, Elena Zucca, and James Cone. 2019. Flexible Recovery of Uniqueness and Immutability. *Theoretical Computer Science* 764 (2019), 145–172. doi:10.1016/j.tcs.2018.09.001

[21] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and Reference Immutability for Safe Parallelism. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 21–40. doi:10.1145/2384616.2384619

[22] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Gavin Bierman. 2015. The Java Language Specification, Java SE 8 Edition. https://docs.oracle.com/javase/specs/jls/se8/html/index.html. Section 8.4.1, Formal Parameters, accessed 2026-03-17.

[23] Christian Haack, Erik Poll, Jan Schäfer, and Aleksy Schubert. 2007. Immutable Objects for a Java-like Language. In *European Symposium on Programming (ESOP)*. Springer, Berlin, Heidelberg, 347–362. doi:10.1007/978-3-540-71316-6_24

[24] Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. 2012. ReIm & ReImInfer: Checking and Inference of Reference Immutability and Method Purity. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 879–896. doi:10.1145/2384616.2384680

[25] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (may 2001), 396–450. doi:10.1145/503502.503505

[26] JetBrains Team. 2025. kotlinx.collections.immutable: Kotlin Immutable Collections Multiplatform Library. https://github.com/Kotlin/kotlinx.collections.immutable. Version 0.3.8. Apache License 2.0.

[27] JSR 308 Expert Group. 2008. Type Annotations Specification (JSR 308). https://jcp.org/en/jsr/detail?id=308.

[28] Elad Kinsbruner, Shachar Itzhaky, and Hila Peleg. 2024. Constrictor: Immutability as a Design Concept. In *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 22:1–22:29. doi:10.4230/LIPIcs.ECOOP.2024.22

[29] Florian Lanzinger, Alexander Weigl, Mattias Ulbrich, and Werner Dietl. 2021. Scalability and Precision by Combining Expressive Type Systems and Deductive Verification. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–29. doi:10.1145/3485520

[30] Edward Lee and Ondřej Lhoták. 2023. Simple Reference Immutability for System F. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 857–881. doi:10.1145/3622828

[31] K Rustan M Leino, Peter Müller, and Angela Wallenburg. 2008. Flexible Immutability with Frozen Objects. In *Verified Software: Theories, Tools, and Experiments (VSTTE)*. Springer, Berlin, Heidelberg, 192–208. doi:10.1007/978-3-540-87873-5_17

[32] Anton Lorenzen, Leo White, Stephen Dolan, Richard A Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. *Proc. ACM Program. Lang.* 8, ICFP (2024), 485–514. doi:10.1145/3674642

[33] Nicholas D Matsakis and Felix S Klock. 2014. The Rust Language. *ACM SIGAda Ada Letters* 34, 3 (2014), 103–104. doi:10.1145/2663171.2663188

[34] Gary McGraw and Edward W Felten. 1999. *Securing Java: Getting Down to Business with Mobile Code* (2 ed.). John Wiley & Sons, Inc., New York, NY, USA.

[35] Ana Milanova. 2020. FlowCFL: generalized type-based reachability analysis: graph reduction and equivalence of CFL-based and type-based reachability. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 178 (Nov. 2020), 29 pages. doi:10.1145/3428246

[36] Johan Östlund, Tobias Wrigstad, Dave Clarke, and Beatrice Åkerblom. 2008. Ownership, Uniqueness, and Immutability. In *International Conference on Objects, Components, Models and Patterns*. Springer, Berlin, Heidelberg, 178–197. doi:10.1007/978-3-540-69824-1_11

[37] Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D. Ernst. 2008. Practical Pluggable Types for Java. In *International Symposium on Software Testing and Analysis (ISSTA)*. Association for Computing Machinery, New York, NY, USA, 201–212. doi:10.1145/1390630.1390656

[38] Wolfram Pfeifer, Werner Dietl, and Mattias Ulbrich. 2026. A Framework for the Interoperable Specification and Verification of Encapsulated Data Structures. In *Formal Methods*. Springer Nature Switzerland, Cham, Switzerland, 90–110. doi:10.1007/978-3-032-26220-2_5

[39] Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D. Ernst. 2013. Immutability. In *Aliasing in Object-Oriented Programming: Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, Berlin, Heidelberg, 233–269. doi:10.1007/978-3-642-36946-9_9

[40] Tobias Roth, Dominik Helm, Michael Reif, and Mira Mezini. 2021. CiFi: Versatile Analysis of Class and Field Immutability. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE '21)*. IEEE Press, Melbourne, Australia, 979–990. doi:10.1109/ASE51524.2021.9678903

[41] Tobias Runge, Marco Servetto, Alex Potanin, and Ina Schaefer. 2023. Immutability and Encapsulation for Sound OO Information Flow Control. *ACM Trans. Program. Lang. Syst.* 45, 1, Article 3 (mar 2023), 35 pages. doi:10.1145/3573270

[42] Scala Documentation Team. 2025. Scala Collections Overview (2.13). https://docs.scala-lang.org/overviews/collections-2.13/overview.html. Accessed: 2025-04-22.

[43] Dan Smith. 2020. JEP 401: Value Classes and Objects (Preview). https://openjdk.org/jeps/401. Accessed: March 6, 2026.

[44] Fridtjof Stoldt, Sylvan Clebsch, Matthew A. Johnson, Matthew J. Parkinson, and Tobias Wrigstad. 2026. Dynamically Checked Deep Immutability in Python. *Proc. ACM Program. Lang.* 10, PLDI, Article 274 (June 2026), 24 pages. doi:10.1145/3808352

[45] Fridtjof Peer Stoldt, Brandt Bucher, Sylvan Clebsch, Matthew A Johnson, Matthew J Parkinson, Guido Van Rossum, Eric Snow, and Tobias Wrigstad. 2025. Dynamic Region Ownership for Concurrency Safety. *Proc. ACM Program. Lang.* 9, PLDI (2025), 1565–1590. doi:10.1145/3729313

[46] Alexander J. Summers and Peter Müller. 2011. Freedom before Commitment: A Lightweight Type System for Object Initialisation. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 1013–1032. doi:10.1145/2048066.2048142

[47] The Rocq Development Team. 2026. The Rocq Reference Manual – Release 9.1.0. https://rocq-prover.org/doc/V9.1.0/refman/index.html. Accessed: 2026-02-01.

[48] Matthew S. Tschantz and Michael D. Ernst. 2005. Javari: Adding Reference Immutability to Java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 211–230. doi:10.1145/1094811.1094828

[49] Sam Weber, Michael Coblenz, Brad Myers, Jonathan Aldrich, and Joshua Sunshine. 2017. Empirical Studies on the Security and Usability Impact of Immutability. In *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, Cambridge, MA, USA, 50–53. doi:10.1109/SecDev.2017.21

[50] Aosen Xiong, Yudi Bai, Haifeng Shi, Lian Sun, Mier Ta, and Werner Dietl. 2026. Artifact for Transitive, Abstract, and Class Polymorphic Immutability. Zenodo. doi:10.5281/zenodo.21519743

[51] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kieżun, and Michael D. Ernst. 2007. Object and Reference Immutability Using Java Generics. In *Foundations of Software Engineering (FSE)*. Association for Computing Machinery, New York, NY, USA, 75–84. doi:10.1145/1287624.1287637

[52] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. 2010. Ownership and Immutability in Generic Java. *ACM Sigplan Notices* 45, 10 (2010), 598–617. doi:10.1145/1932682.1869509

Received 2026-03-17; revised 2026-07-21; accepted 2026-08-06