Transitive, Abstract, and Class Polymorphic Immutability

Aosen Xiong Yudi Bai Haifeng Shi Lian Sun Mier Ta Werner Dietl

University of Waterloo

OOPSLA 2026







This Talk in One Slide

Immutability An object's state never changes.



This Talk in One Slide

Immutability An object's state never changes.

Transitive Immutability extends to everything an object reaches.



This Talk in One Slide

Immutability	An object's state never changes.
Transitive	Immutability extends to everything an object reaches.
Abstract	Protection by default. Some state, such as caches, may change.



This Talk in One Slide

Immutability	An object's state never changes.
Transitive	Immutability extends to everything an object reaches.
Abstract	Protection by default. Some state, such as caches, may change.
Class Polymorphic	One class can have both mutable and immutable instances.



This Talk in One Slide

Immutability	An object's state never changes.
Transitive	Immutability extends to everything an object reaches.
Abstract	Protection by default. Some state, such as caches, may change.
Class Polymorphic	One class can have both mutable and immutable instances.

Beyond the title

- Precise specification of immutability
- Type system mechanized in Rocq
- Implemented in the EISOP Checker Framework for Java
- Evaluated on the Java Collections Framework and other benchmarks



Effective Java's Advice

Effective Java advises a **final class with final fields**.

By convention

```
final class Person {
  final int id;
  final int born;
}
```



J. Bloch. *Effective Java*, 3rd edition. Addison-Wesley, 2017.

Effective Java's Advice

Effective Java advises a **final class with final fields**.

By convention

```
final class Person {
  final int id;
  final int born;
}
```

By a type system

```
@Imm class Person {
  int id;
  int born;
}
```



J. Bloch. *Effective Java*, 3rd edition. Addison-Wesley, 2017.

Effective Java's Advice

Effective Java advises a **final class with final fields**.

By convention

```
final class Person {
  final int id;
  final int born;
}
```

By a type system

```
@Imm class Person {
  int id;
  int born;
}
```

`final` constrains assignment and subclassing. `@Imm` states the property directly.



J. Bloch. *Effective Java*, 3rd edition. Addison-Wesley, 2017.

Effective Java's Advice

Effective Java advises a **final class with final fields**.

By convention

```
final class Person {
  final int id;
  final int born;
}
```

By a type system

```
@Imm class Person {
  int id;
  int born;
}
```

`final` constrains assignment and subclassing. `@Imm` states the property directly.

What are assignability and mutability?



J. Bloch. *Effective Java*, 3rd edition. Addison-Wesley, 2017.

Assignability

In our type system, every **field** carries an **assignability modifier**.

The assignability modifiers

`@Final`: assigned once, during initialization

`@Assignable`: can be reassigned at any time



Assignability

In our type system, every **field** carries an **assignability modifier**.

The assignability modifiers

`@Final`: assigned once, during initialization

`@Assignable`: can be reassigned at any time

But `final` says nothing about whether the **reachable object** changes.



Mutable and Immutable Objects

At runtime, objects are always immutable or mutable.

Runtime objects

```
new @Imm Person(...)
new @Mut Buffer(...)
```

Two kinds of objects.



Mutable and Immutable Objects

At runtime, objects are always immutable or mutable.

Runtime objects

```
new @Imm Person(...)
new @Mut Buffer(...)
```

Two kinds of objects.

Types

```
@Imm Person sherlock;
@Mut Buffer scratch;
```

Two type qualifiers.



Mutable and Immutable Objects

At runtime, objects are always immutable or mutable.

Runtime objects

```
new @Imm Person(...)
new @Mut Buffer(...)
```

Two kinds of objects.

Types

```
@Imm Person sherlock;
@Mut Buffer scratch;
```

Two type qualifiers.

Class declarations

```
@Imm class Person {...}
@Mut class Buffer {...}
```

All instances are of that kind.



Mutable and Immutable Objects

At runtime, objects are always immutable or mutable.

Runtime objects

```
new @Imm Person(...)
new @Mut Buffer(...)
```

Two kinds of objects.

Types

```
@Imm Person sherlock;
@Mut Buffer scratch;
```

Two type qualifiers.

Class declarations

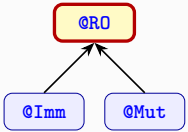
```
@Imm class Person {...}
@Mut class Buffer {...}
```

All instances are of that kind.

From here on, every assignability modifier and mutability qualifier is written out explicitly.



Readonly, a Common Supertype of `@Imm` and `@Mut`



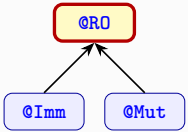
`@RO`: no mutation through this reference

```
void log(@RO Object o) { ... }

log(sherlock);  // allowed  @Imm Person
log(scratch);   // allowed  @Mut Buffer
```



Readonly, a Common Supertype of @Imm and @Mut



@RO: no mutation through this reference

```
void log(@RO Object o) { ... }

log(sherlock);  // allowed  @Imm Person
log(scratch);   // allowed  @Mut Buffer
```

```
@Mut class Buffer {
  void append(@Mut Buffer this, char c) {...}
}
@RO Buffer view = scratch;
view.append('x');  // rejected
// @RO is not a subtype of @Mut
```



Abstract Immutability

```
@Imm class Person {
  ...
  @Assignable @Imm int hash;  // lazy init
  @Final @Mut Counter hits;   // access counter

  @Imm int hashCode(@RO Person this) {
    hits.increment();         // allowed
    if (hash == 0) {
      hash = compute();       // allowed
    }
    return hash;
  }
}
```



Abstract Immutability

```
@Imm class Person {
  ...
  @Assignable @Imm int hash;  // lazy init
  @Final @Mut Counter hits;   // access counter

  @Imm int hashCode(@RO Person this) {
    hits.increment();         // allowed
    if (hash == 0) {
      hash = compute();       // allowed
    }
    return hash;
  }
}
```

Two ways to opt out

`@Assignable`: the **slot** can change

`@Mut`: the **object** is mutable



How can one class have both mutable and immutable instances?

```
@RDM class Outer {
  ...
}
@Imm Outer immOuter = new @Imm Outer(...);  // ✓
@Mut Outer mutOuter = new @Mut Outer(...);  // ✓
```

@RDM, receiver-dependent mutable: a class whose instances have to be either immutable or mutable.



How can one class have both mutable and immutable instances?

```
@RDM class Outer {
  void setInner(@Mut Outer this, ...) {...}
}
@Imm Outer immOuter = new @Imm Outer(...);  // ✓
@Mut Outer mutOuter = new @Mut Outer(...);  // ✓

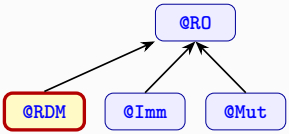
immOuter.setInner(...);  // ✗ needs a @Mut receiver
mutOuter.setInner(...);  // ✓
```

@RDM, receiver-dependent mutable: a class whose instances have to be either immutable or mutable.



Receiver-Dependent Mutability

Immutability should be transitive: the type of a field depends on the type of the receiver.



`@RDM` on a type: it follows the receiver

```
@RDM class Outer {
  @Final @RDM Inner inner;
}

@Imm Outer immOuter = ...;
@Mut Outer mutOuter = ...;

immOuter.inner  // @Imm Inner
mutOuter.inner  // @Mut Inner
```

W. Dietl, P. Müller. Universes: Lightweight Ownership for JML. *JOT*, 2005.
W. Dietl, S. Drossopoulou, P. Müller. Generic Universe Types. *ECOOP*, 2007.



Receiver-Dependent Assignability

The assignability modifier of a field can depend on the receiver too.

The assignability modifier

`@Final`: assigned once, during initialization

`@Assignable`: can be reassigned at any time

`@RDA`: assignable through a `@Mut` receiver



Receiver-Dependent Assignability

The assignability modifier of a field can depend on the receiver too.

The assignability modifier

`@Final`: assigned once, during initialization

`@Assignable`: can be reassigned at any time

`@RDA`: assignable through a `@Mut` receiver

```
@RDM class Outer {
  @RDA @RDM Inner inner;
}

@Imm Outer immOuter = ...;
@Mut Outer mutOuter = ...;
immOuter.inner = new @Imm Inner();  // rejected, effectively final
mutOuter.inner = new @Mut Inner();  // allowed, effectively assignable
```



How can readonly references stay sound?

```
@RDM class Outer { @Assignable @RDM Inner inner; }
void swapField(@RO Outer o1, @RO Outer o2) {
  o1.inner = o2.inner;
}
```

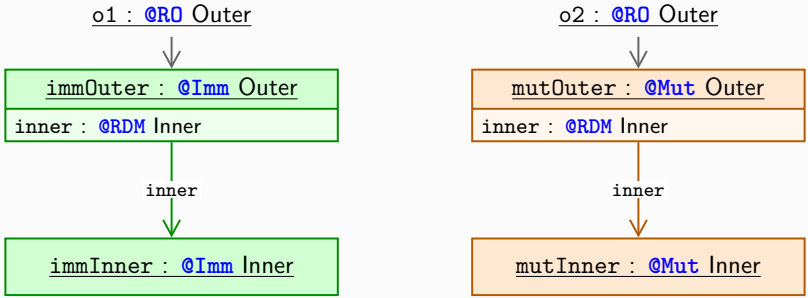
A `@RO` reference may point to either kind of object.



How can readonly references stay sound?

```
@RDM class Outer { @Assignable @RDM Inner inner; }
void swapField(@RO Outer o1, @RO Outer o2) {
  o1.inner = o2.inner;
}
```

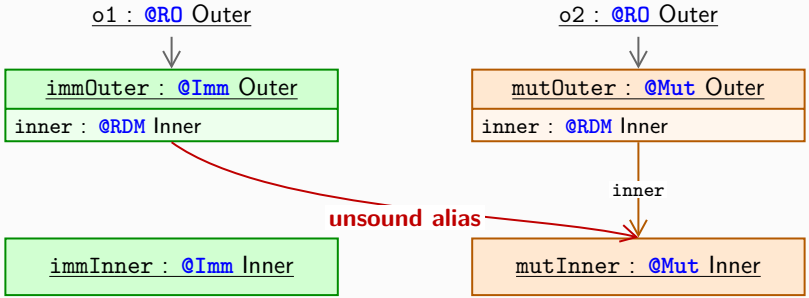
A @RO reference may point to either kind of object.



How can readonly references stay sound?

```
@RDM class Outer { @Assignable @RDM Inner inner; }
void swapField(@RO Outer o1, @RO Outer o2) {
  o1.inner = o2.inner;
}
```

A @RO reference may point to either kind of object.

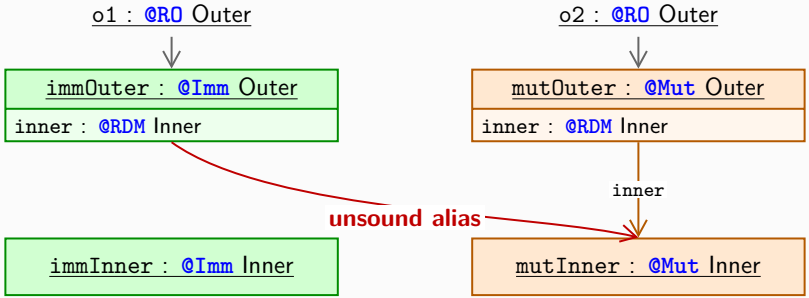


An unsound system would let `immOuter` hold `mutInner`, which `mutOuter` could still change.

How can readonly references stay sound?

```
@RDM class Outer { @Assignable @RDM Inner inner; }
void swapField(@RO Outer o1, @RO Outer o2) {
  o1.inner = o2.inner;
}
```

A `@RO` reference may point to either kind of object.



An unsound system would let `immOuter` hold `mutInner`, which `mutOuter` could still change.

Our rules forbid this: `o1.inner = o2.inner` is rejected.

Precise Immutability Specification

We identify four state-preservation properties, ordered by strength:

weakest

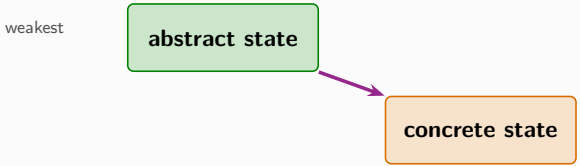
abstract state

G. T. Leavens, A. L. Baker, C. Ruby. Preliminary Design of JML. *SIGSOFT SEN* 31(3), 2006.



Precise Immutability Specification

We identify four state-preservation properties, ordered by strength:



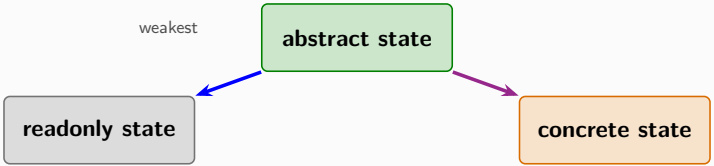
stronger assignability → stronger mutability



G. T. Leavens, A. L. Baker, C. Ruby. Preliminary Design of JML. *SIGSOFT SEN* 31(3), 2006.

Precise Immutability Specification

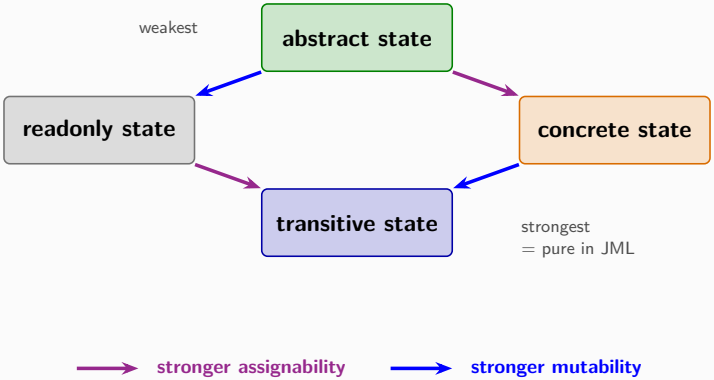
We identify four state-preservation properties, ordered by strength:



G. T. Leavens, A. L. Baker, C. Ruby. Preliminary Design of JML. *SIGSOFT SEN* 31(3), 2006.

Precise Immutability Specification

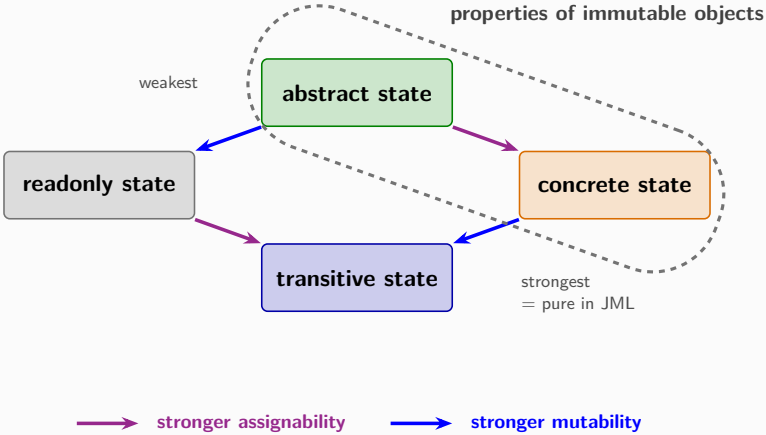
We identify four state-preservation properties, ordered by strength:



G. T. Leavens, A. L. Baker, C. Ruby. Preliminary Design of JML. *SIGSOFT SEN* 31(3), 2006.

Precise Immutability Specification

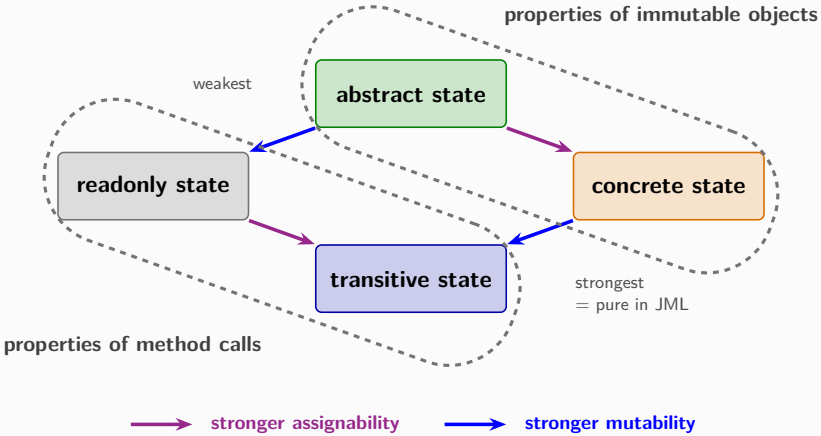
We identify four state-preservation properties, ordered by strength:



G. T. Leavens, A. L. Baker, C. Ruby. Preliminary Design of JML. *SIGSOFT SEN* 31(3), 2006.

Precise Immutability Specification

We identify four state-preservation properties, ordered by strength:



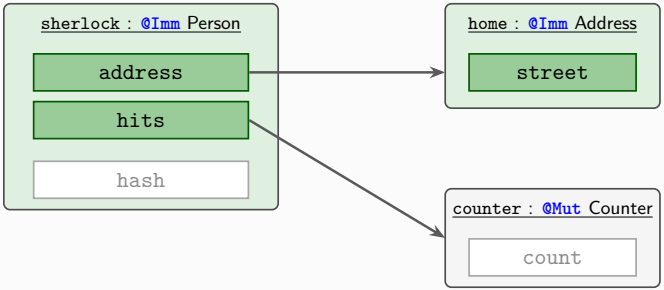
G. T. Leavens, A. L. Baker, C. Ruby. Preliminary Design of JML. *SIGSOFT SEN* 31(3), 2006.

Protection Boundaries: abstract state

```
@Imm class Person {
  @Final @Imm Address address;
  @Final @Mut Counter hits;
  @Assignable @Imm int hash;

  @Imm int hashCode(@RO Person this) {
    hits.increment(); allowed
    if (hash == 0) {
      hash = compute(); allowed
    }
    return hash;
  }
}
```

The object graph and protected state of an immutable Person.



protected state

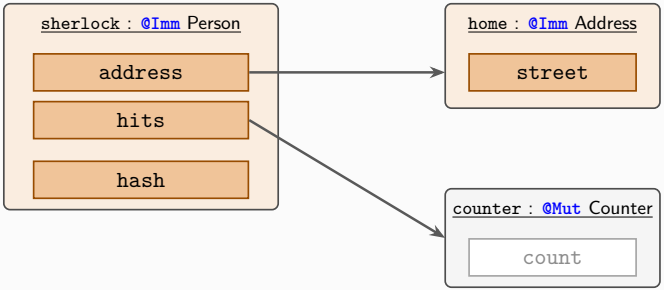
may change

Protection Boundaries: concrete state

```
@Imm class Person {
  @Final @Imm Address address;
  @Final @Mut Counter hits;
  @Assignable @Imm int hash;

  @Imm int hashCode(@RO Person this) {
    hits.increment(); allowed
    if (hash == 0) {
      hash = compute(); rejected
    }
    return hash;
  }
}
```

The object graph and protected state of an immutable Person.



protected state

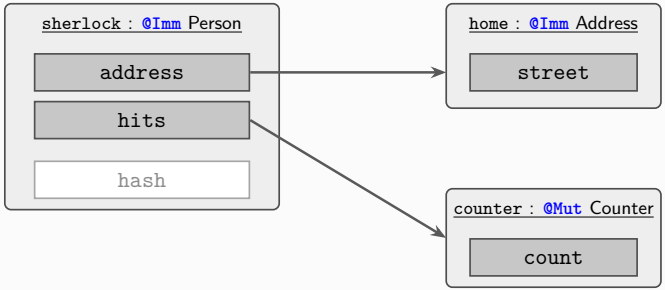
may change

Protection Boundaries: readonly state

```
@Imm class Person {
  @Final @Imm Address address;
  @Final @Mut Counter hits;
  @Assignable @Imm int hash;

  @Imm int hashCode(@RO Person this) {
    hits.increment(); rejected
    if (hash == 0) {
      hash = compute(); allowed
    }
    return hash;
  }
}
```

The object graph and protected state of an immutable Person.



protected state

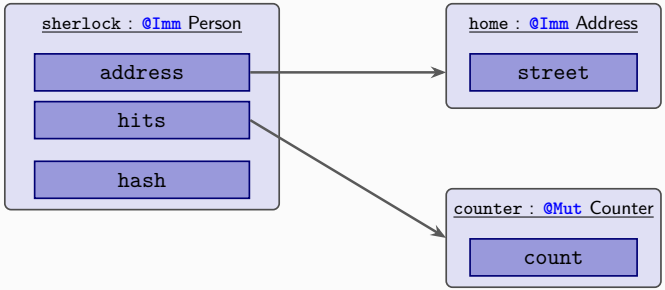
may change

Protection Boundaries: transitive state

```
@Imm class Person {
  @Final @Imm Address address;
  @Final @Mut Counter hits;
  @Assignable @Imm int hash;

  @Imm int hashCode(@RO Person this) {
    hits.increment(); rejected
    if (hash == 0) {
      hash = compute(); rejected
    }
    return hash;
  }
}
```

The object graph and protected state of an immutable Person.



protected state

may change

Default Types for Code Without Annotations

- Types default to `@Mut`, since Java objects are mutable.

M. Coblenz, W. Nelson, J. Aldrich, B. Myers, J. Sunshine. Glacier: Transitive Class Immutability for Java. *ICSE*, 2017.



Default Types for Code Without Annotations

- Types default to `@Mut`, since Java objects are mutable.
- Uses of an `@Imm` or `@RDM` class, such as `String` or `Address`, default to its qualifier.



M. Coblenz, W. Nelson, J. Aldrich, B. Myers, J. Sunshine. Glacier: Transitive Class Immutability for Java. *ICSE*, 2017.

Default Types for Code Without Annotations

- Types default to `@Mut`, since Java objects are mutable.
- Uses of an `@Imm` or `@RDM` class, such as `String` or `Address`, default to its qualifier.
- Fields default to `@RDA`, so transitivity needs no extra annotation.



M. Coblenz, W. Nelson, J. Aldrich, B. Myers, J. Sunshine. Glacier: Transitive Class Immutability for Java. *ICSE*, 2017.

Default Types for Code Without Annotations

- Types default to `@Mut`, since Java objects are mutable.
- Uses of an `@Imm` or `@RDM` class, such as `String` or `Address`, default to its qualifier.
- Fields default to `@RDA`, so transitivity needs no extra annotation.

```
@Imm class Person {
  @RDA @Imm String name;
  @RDA @RDM Address home;
}

@Imm Person sherlock = new @Imm Person(...);
void greet(@Imm Person p) {...}
```

The grey annotations are the defaults.



M. Coblenz, W. Nelson, J. Aldrich, B. Myers, J. Sunshine. Glacier: Transitive Class Immutability for Java. *ICSE*, 2017.

Default Types for Code Without Annotations

- Types default to `@Mut`, since Java objects are mutable.
- Uses of an `@Imm` or `@RDM` class, such as `String` or `Address`, default to its qualifier.
- Fields default to `@RDA`, so transitivity needs no extra annotation.

```
@Imm class Person {
  @RDA @Imm String name;
  @RDA @RDM Address home;
}

@Imm Person sherlock = new @Imm Person(...);
void greet(@Imm Person p) {...}
```

The grey annotations are the defaults.
Glacier's immutable classes are a special case of an `@Imm` class.



M. Coblenz, W. Nelson, J. Aldrich, B. Myers, J. Sunshine. Glacier: Transitive Class Immutability for Java. *ICSE*, 2017.

Results and Future Work

Mechanized in Rocq

- Featherweight Java with mutable references
- Preservation and safety
- All four state-preservation guarantees

[1] E. Kinsbruner, S. Itzhaky, H. Peleg. Constrictor: Immutability as a Design Concept. *ECOOP*, 2024.



Results and Future Work

Mechanized in Rocq

- Featherweight Java with mutable references
- Preservation and safety
- All four state-preservation guarantees

Checker Framework implementation

Case study	Lines	Annotations
JCF (OpenJDK)	22,992	3,961
Constrictor benchmarks[1]	1,184	113
Checker test suite	1,738	665

Retrofits the JCF, and finds errors Constrictor misses.

[1] E. Kinsbruner, S. Itzhaky, H. Peleg. Constrictor: Immutability as a Design Concept. *ECOOP*, 2024.



Results and Future Work

Mechanized in Rocq

- Featherweight Java with mutable references
- Preservation and safety
- All four state-preservation guarantees

Checker Framework implementation

Case study	Lines	Annotations
JCF (OpenJDK)	22,992	3,961
Constrictor benchmarks[1]	1,184	113
Checker test suite	1,738	665

Retrofits the JCF, and finds errors Constrictor misses.

Future work

- Annotation inference
- Generics formalization
- Observational immutability



[1] E. Kinsbruner, S. Itzhaky, H. Peleg. Constrictor: Immutability as a Design Concept. *ECOOP*, 2024.

Results and Future Work

Mechanized in Rocq

- Featherweight Java with mutable references
- Preservation and safety
- All four state-preservation guarantees

Checker Framework implementation

Case study	Lines	Annotations
JCF (OpenJDK)	22,992	3,961
Constrictor benchmarks[1]	1,184	113
Checker test suite	1,738	665

Retrofits the JCF, and finds errors Constrictor misses.

Future work

- Annotation inference
- Generics formalization
- Observational immutability

Paper

Artifact



Thank you!



[1] E. Kinsbruner, S. Itzhaky, H. Peleg. Constrictor: Immutability as a Design Concept. *ECOOP*, 2024.

Backup – Codename PICO



Precise **I**mmutability for **C**lasses and **O**bjects

Checker Framework implementation

- About 2,000 lines of checker code
- Generics, behavioral subtyping, annotation files, warning suppression, and diagnostics
- Annotated JCF: 3,961 annotations over 22,992 lines

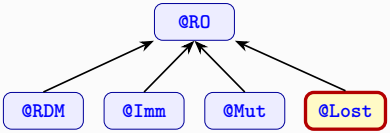
Rocq mechanization

- About 26,000 lines in 38 files
- 549 lemmas and theorems
- No admitted proofs, and no axioms behind the public results



Photo: jeffreyw, *Mmm... Pico de Gallo*, CC BY 2.0, via Wikimedia Commons.

Backup – The @Lost Qualifier



@Lost: statically unknown mutability

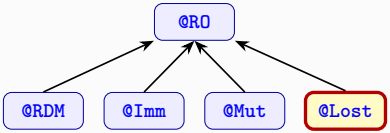
Two rules

1. @RO ▷ @RDM = @Lost
   "we have lost track of the mutability"

W. Dietl, S. Drossopoulou, P. Müller. Generic Universe Types. *ECOOP*, 2007.



Backup – The @Lost Qualifier



`@Lost`: statically unknown mutability

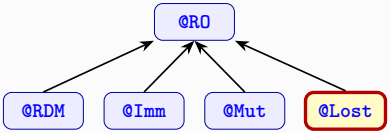
Two rules

1. `@RO` ▷ `@RDM` = `@Lost`
   "we have lost track of the mutability"
2. `@Lost`-to-`@Lost` assignment is forbidden

W. Dietl, S. Drossopoulou, P. Müller. Generic Universe Types. *ECOOP*, 2007.



Backup – The @Lost Qualifier



@Lost: statically unknown mutability

Two rules

1. **@RO** ▷ **@RDM** = **@Lost**
   "we have lost track of the mutability"
2. **@Lost**-to-**@Lost** assignment is forbidden

Effect on swapField

```
o1.inner = o2.inner;
```

rejected: cross-type aliasing cannot arise.

W. Dietl, S. Drossopoulou, P. Müller. Generic Universe Types. *ECOOP*, 2007.



Backup – Related Work

- **Javari** (Tschantz, Ernst, 2005): readonly references and assignable fields, but no object immutability.



Backup – Related Work

- **Javari** (Tschantz, Ernst, 2005): readonly references and assignable fields, but no object immutability.
- **IGJ, OIGJ** (Zibin et al., 2007, 2010): object immutability, with mutability polymorphism through generics. Assignable fields, but no abstract-state theorem.



Backup – Related Work

- **Javari** (Tschantz, Ernst, 2005): readonly references and assignable fields, but no object immutability.
- **IGJ, OIGJ** (Zibin et al., 2007, 2010): object immutability, with mutability polymorphism through generics. Assignable fields, but no abstract-state theorem.
- **ReIm** (Huang et al., 2012): readonly references and purity inference, with viewpoint adaptation, but no object immutability.



Backup – Related Work

- **Javari** (Tschantz, Ernst, 2005): readonly references and assignable fields, but no object immutability.
- **IGJ, OIGJ** (Zibin et al., 2007, 2010): object immutability, with mutability polymorphism through generics. Assignable fields, but no abstract-state theorem.
- **ReIm** (Huang et al., 2012): readonly references and purity inference, with viewpoint adaptation, but no object immutability.
- **Glacier** (Coblenz et al., 2017): class immutability for Java, with a user study, but no mutation through selected fields and no class mutability polymorphism. It is expressible here by declaring the class `@Imm`.



Backup – Related Work

- **Javari** (Tschantz, Ernst, 2005): readonly references and assignable fields, but no object immutability.
- **IGJ, OIGJ** (Zibin et al., 2007, 2010): object immutability, with mutability polymorphism through generics. Assignable fields, but no abstract-state theorem.
- **ReIm** (Huang et al., 2012): readonly references and purity inference, with viewpoint adaptation, but no object immutability.
- **Glacier** (Coblenz et al., 2017): class immutability for Java, with a user study, but no mutation through selected fields and no class mutability polymorphism. It is expressible here by declaring the class `@Imm`.
- **JML** (Leavens et al., 2006): pure methods, as a specification.



Backup – Related Work

- **Javari** (Tschantz, Ernst, 2005): readonly references and assignable fields, but no object immutability.
- **IGJ, OIGJ** (Zibin et al., 2007, 2010): object immutability, with mutability polymorphism through generics. Assignable fields, but no abstract-state theorem.
- **ReIm** (Huang et al., 2012): readonly references and purity inference, with viewpoint adaptation, but no object immutability.
- **Glacier** (Coblenz et al., 2017): class immutability for Java, with a user study, but no mutation through selected fields and no class mutability polymorphism. It is expressible here by declaring the class `@Imm`.
- **JML** (Leavens et al., 2006): pure methods, as a specification.
- **This work**: object immutability with sound viewpoint adaptation, four state-preservation properties, and class-level mutability polymorphism, mechanized in Rocq.



Backup – Viewpoint Adaptation Rules

Back to `Person`: an `@Assignable` `hash`, and a `@Final` `@Mut` `hits` counter.

	Assignability `@Assignable` slots may change	**Assignability** `@Assignable` slots are fixed

Mutability
objects behind `@Mut` fields may change
	`@AbstractState`
	`@Imm int hashCode(@RO Person this) {`
	`  hash = compute();` **allowed**
	`  hits.increment();` **allowed**
	`}`
		`@ConcreteState`
		`@Imm int hashCode(@RO Person this) {`
		`  hash = compute();` **rejected**
		`  hits.increment();` **allowed**
		`}`

Mutability
everything reachable is protected
	`@ReadonlyState`
	`@Imm int hashCode(@RO Person this) {`
	`  hash = compute();` **allowed**
	`  hits.increment();` **rejected**
	`}`
		`@TransitiveState`
		`@Imm int hashCode(@RO Person this) {`
		`  hash = compute();` **rejected**
		`  hits.increment();` **rejected**
		`}`



Backup – Soundness

Preservation and Safety

If a well-typed statement runs from a well-formed configuration, in any method scope, then the resulting configuration is well-formed and the result is `OK`: no mutation exception occurs.

- Proved by induction on the evaluation rules.
- The interesting case is object creation, by cases on the runtime viewpoint adaptation.
- Mechanized in Rocq: `preservation_pico` and `well_typed_no_mutation_exp`.
- The state-preservation guarantees build on this theorem.



Backup – Related Designs

System	Field read through `@RO`	If assignable fields are added
IGJ, OIGJ	`@RO` ▷ `@RDM` = `@RO`	cross-type aliasing
L42, SIFO, C# extension	`@RO` ▷ `@Mut` = `@RO`	cross-type aliasing
This work	`@RO` ▷ `@RDM` = `@Lost`	rejected

Each design is sound as published. The hole appears only when it is combined with assignable fields.



Backup – Ownership and Rust

Orthogonal, and composable

Our type system leaves aliasing unrestricted. Ownership and uniqueness are orthogonal and can be layered on top, in the modular style of OCaml's modes.

Adoption

Rust's ownership and borrowing give stronger properties, but they require a different programming style that is hard to retrofit into existing object-oriented code.



Backup – Limitations

- **Freeze pattern**: no mutable-then-frozen construction. A uniqueness extension is future work.
- **Lazy initialization**: `String.hash` is marked `@Assignable`, which is imprecise. A stabilization property is future work.
- **Outer receiver dependence**: an iterator's entry field depends on the enclosing collection.
- **View immutability**: the type system does not detect violations that leak information from outside the abstract state. Constrictor's `@viewmethod` covers this case, and an analogous guarantee is future work.
- **Polymorphic constructors**: `@RDM` is not a subtype of `@Mut`, so a constructor cannot call mutation methods.



Backup – Annotation Counts

	`@Imm`	`@Mut`	`@RO`	`@RDM`	`@PolyMut`	`@Assignable`
JCF	349	736	1,919	241	650	66
Constrictor	16	–	85	–	–	12

The checker test suite has 665 annotations over 1,738 lines.
Many JCF annotations are explicit receivers: `@Mut` for methods such as `removeIf`, and `@RO` for methods such as `size()`.

