Object and Reference Immutability using Java Generics

Yoav Zibin, Alex Potanin(*), Mahmood Ali, Shay Artzi, Adam Kiezun, and Michael D. Ernst

MIT Computer Science and Artificial Intelligence Lab, USA
* Victoria University of Wellington, New Zealand
Immutability – What for?

- Program comprehension
- Verification
- Compile- & run-time optimizations
- Invariant detection

- Refactoring
- Test input generation
- Regression oracle creation
- Specification mining
- Modelling
Immutability varieties

- **Class immutability**
  - No instance of an immutable class can be mutated after creation (e.g., String, Integer)

- **Object immutability**
  - The same class may have both mutable and immutable instances

- **Reference immutability**
  - A particular reference cannot be used to mutate its referent (but other aliases might cause mutations)
Previous work

- **Access rights**
  - Java with Access-Control (JAC)
    - `readnothing < readimmutable < readonly < writeable`
  - Capabilities for sharing
    - Lower-level rights that can be enforced at compile- or run- time

- **Reference immutability:**
  - Universes (ownership + reference immutability)
  - C++’s `const`
  - Javari
IGJ - *Immutability Generic Java*

- **Class immutability**
  - All instances are immutable objects

- **Object immutability:**
  - An object: mutable or immutable

- **Reference immutability:**
  - A reference: mutable, immutable, or readonly
IGJ syntax

1: // An immutable reference to an immutable date;  
   // Mutating the referent is prohibited, via this or any other reference.
   Date<Immutable> immutD = new Date<Immutable>();

2: // A mutable reference to a mutable date;  
   // Mutating the referent is permitted, via this or any other reference.
   Date<Mutable> mutD = new Date<Mutable>();

3: // A readonly reference to any date;  
   // Mutating the referent is prohibited via this reference.
   Date<ReadOnly> roD = ... ? immutD : mutD;

Java syntax is not modified:

- One new generic parameter was added
- Some method annotations were added (shown later)
IGJ design principles

- **Transitivity**
  - Transitive (deep) immutability protects the entire abstract state from mutation
  - Mutable fields are excluded from the abstract state
- **Static**
  - No runtime representation for immutability
- **Polymorphism**
  - Abstracting over immutability without code duplication
- **Simplicity**
  - No change to Java’s syntax; a small set of typing rules
Hierarchies in IGJ

Immutability parameters hierarchy

The **subclass** hierarchy for Object and Date

The **subtype** hierarchy for Object and Date
Covariance problem and immutability

void foo(ArrayList<Object> a) { … }
foo(new ArrayList<Object>()); // OK
foo(new ArrayList<String>()); // Compilation error!

void foo(Object[] a) { a[0] = new Integer(1); }
foo(new Object[42]); // OK, stores an Integer in an Object array
foo(new String[42]); // Causes ArrayStoreException at runtime

- IGJ’s Solution:
  - readOnly, Immutable — allow covariance
  - Mutable — disallow covariance

List<ReadOnly, String> is a subtype of List<ReadOnly, Object>
List<Mutable, String> is NOT a subtype of List<Mutable, Object>
IGJ typing rules

- There are several typing rules (next slides)
  - Field assignment
  - Immutability of `this`
  - Method invocation
- Let $I(x)$ denote the immutability of $x$
  - Example:
    ```
    Date<Mutable> d;
    I(d) is Mutable
    ```
Field assignment rule

- Field assignment rule:
  
  \[ o.\text{someField} = \ldots; \]
  
  is legal iff \( I(o) = \text{Mutable} \)

Example:

- \( \text{Employee<ReadOnly>} \ roE = \ldots; \)
- \( roE.\text{address} = \ldots; \ // \text{Compilation error!} \)
Immutability of this

- **this** immutability is indicated by a method annotation
  - @ReadOnly, @Mutable, @Immutable

- We write `I(m.this)` to show the context of **this**

- Example:
  - @Mutable void m() { ... this ... }
  - `I(m.this) = Mutable`
Method invocation rule

\[ o.m(...) \]
\text{is legal iff } I(o) \text{ is a subtype of } I(m.\text{this})

1: \text{Employee<Mutable> } mutE = \ldots;
2: \text{mutE.setAddress(...);} \quad // \text{OK}
3: \text{mutE.getAddress();} \quad // \text{OK}
4: \text{Employee<ReadOnly> } roE = mutE;
5: \text{roE.setAddress(...);} \quad // \text{Compilation error!}
Reference immutability (ReadOnly)

```java
1: class Edge<I extends ReadOnly> {
2:     long id;
3:     @Mutable Edge(long id) { this.setId(id); }
4:     @Mutable void setId(long id) { this.id = id; }
5:     @ReadOnly long getId() { return this.id; }
6:     @ReadOnly Edge<I> copy() { return new Edge<I>(this.id); }
7:     static void print(Edge<ReadOnly> e) {... }
8: }

10: class Graph<I extends ReadOnly> {
11:     List<I,Edge<I>> edges;
12:     @Mutable Graph(List<I,Edge<I>> edges) { this.edges = edges; }
13:     @Mutable void addEdge(Edge<Mutable> e) { this.edges.add(e); }
14:     static <X extends ReadOnly>
15:         Edge<X> findEdge(Graph<X> g, long id) { ... }
16: }
```
Object immutability: Motivation

- Compile- & run-time optimizations
- Program comprehension
- Verification
- Invariant detection
- Test input generation
- ...
- Example: Immutable objects need no synchronization

```java
@ReadOnly @synchronized long getId() { return id; }
@Immutable long getIdImmutable() { return id; }
```
Object immutability: Challenge

1: class Edge<I extends ReadOnly> { 
2:     private long id; 
3:     @??????????????? Edge(long id) { this.setId(id); } 
4:     @Mutable void setId(long id) { this.id = id; } 

Challenge: How should the constructor be annotated? 

- @Mutable ? 
  - A mutable alias for this might escape 
- @Immutable or @ReadOnly ? 
  - Cannot assign to any field, nor call this.setId
Object immutability: Solution

```java
1: class Edge<I extends ReadOnly> { 
2:    private long id;
3:    @AssignsFields Edge(long id) { this.setId(id); } 
4:    @AssignsFields void setId(long id) { this.id = id; } 
5:    Edge<I> e;
6:    @Mutable void foo(long id) { this.e.id = id; }
```

- `@AssignsFields`
  - Can only assign to the fields of this, i.e., it is not transitive
  - Private: **cannot** write `Date<AssignsFields>`
  - Conclusion: **this** can only escape as `ReadOnly`

```mermaid
graph TD
    IMutable --> IAssignsFields
    IAssignsFields --> IReadOnly
    IMutable --> IImmutable
    IImmutable --> IMutable
```

18/23
Case studies

- IGJ compiler
  - Small and simple extension of javac
  - Using the visitor pattern for the AST
  - Modified `isSubType` according to IGJ’s covariant subtyping

- Case studies:
  - Jolden benchmark, htmlparser, svn client
  - 328 classes (106 KLOC)
  - 113 JDK classes and interfaces
Case studies conclusions

- **Representation exposure errors**
  - In `htmlparser`: constructor takes an array and assigns it to a field, without copying; an accessor method also returns that array

- **Conceptual problems**
  - In Jolden: an immutable object is mutated only once immediately after its creation. We refactored the code, inserting the mutation to the constructor

- **Found both immutable classes and objects**
  - `Date`, `SVNURL`, `lists`
See the paper for ...

- CoVariant and NoVariant type parameters
- Method overriding
- Mutable and assignable fields
- Inner classes
- **Circular immutable data-structures**
- Formal proof (Featherweight IGJ)
Conclusions

- Immutability Generic Java (IGJ)
  - Both reference, object, and class immutability
  - Simple, intuitive, small, no syntax changes
  - Static – no runtime penalties (like generics)
  - Backward compatible, no JVM changes
  - High degree of polymorphism using generics and safe covariant subtyping

- Case study proving usefulness
- Formal proof of soundness
Future work

- Add default immutability

```java
class Graph<I extends ReadOnly default Mutable>
```

- An alternative syntax
  (in JSR 308 for Java 7)

```java
new @mutable ArrayList:@immutable Edge>(...)
```

- Runtime support (e.g. down-cast)