Object and Reference Immutability using Java Generics

Yoav Zibin, Alex Potanin†, Mahmood Ali, Shay Artzi, Adam Kieżun, Michael D. Ernst
MIT Computer Science & Artificial Intelligence Lab — † Victoria University of Wellington
{zyoav|mali|artzi|akiezun|mernst}@csail.mit.edu — alex@mcs.vuw.ac.nz

Abstract

A compiler-checked immutability guarantee provides useful documentation, facilitates reasoning, and enables optimizations. This paper presents *Immutability Generic Java* (IGJ), a novel language extension that expresses immutability without changing Java's syntax by building upon Java's generics and annotation mechanisms. In IGJ, each class has one additional type parameter that is `Mutable`, `Immutable`, or `ReadOnly`. IGJ guarantees both *reference immutability* (only mutable references can mutate an object) and *object immutability* (an immutable reference points to an immutable object). IGJ is the first proposal for enforcing object immutability within Java's syntax and type system, and its reference immutability is more expressive than previous work. IGJ also permits covariant changes of type parameters in a type-safe manner, e.g., a readonly list of integers is a subtype of a readonly list of numbers. IGJ extends Java's type system with a few simple rules. We formalize this type system and prove it sound. Our IGJ compiler works by type-erasure and generates byte-code that can be executed on any JVM without runtime penalty.

Categories and Subject Descriptors: D.1.5 [Programming Techniques]: Object-oriented Programming; D.2.1 [Software Engineering]: Requirements / Specifications[Languages]; D.3.3 [Programming Languages]Language Constructs and Features

General Terms: Design, Languages, Theory

Keywords: const, Generic, IGJ, Immutability, Java, Readonly

1. Introduction

Immutability information is useful in many software engineering tasks, such as modeling [7], verification [30], compile- and runtime optimizations [9, 25, 28], refactoring [17], test input generation [1], regression oracle creation [24,32], invariant detection [15], specification mining [10], and program comprehension [13]. Three varieties of immutability guarantee are:

Class immutability No instance of an *immutable class* may be changed; examples in Java include `String` and most subclasses of `Number` such as `Integer` and `BigDecimal`.

Object immutability An *immutable object* can not be modified, even if other instances of the same class can be. For example, some instances of `List` in a given program may be immutable, whereas others can be modified. Object immutability can be used for pointer analysis and optimizations, such as sharing between threads without synchronization, and to help prevent hard-to-detect bugs, e.g., the documentation of the `Map` interface in Java states that "Great care must be exercised if *mutable objects* are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map."

Reference immutability A *readonly reference* [2, 5, 12, 22, 25, 29, 31] (or a *const* pointer in C++) cannot be used to modify its referent. However, the referent might be modified using an aliasing mutable reference. Reference immutability is required to specify interfaces, such as that a procedure may not modify its arguments (even if the caller retains the right to do so) or a client may not modify values returned from a module. Past work does not guarantee object immutability, unless reference immutability is combined with an alias/escape analysis to guarantee that no mutable aliases to an object exist [2, 3].

This paper presents *Immutability Generic Java* (IGJ), a language that supports class, object, and reference immutability. Each object is either mutable or immutable, and each reference is `Immutable`, `Mutable`, or `ReadOnly`. Inspired by work that combines ownership and generics [27], the distinctions are expressed without changing Java's syntax by adding one new type parameter (at the beginning of the list of type parameters):

```
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 mutable reference.
   Date<Mutable> mutD = new Date<Mutable>();
3: // A readonly reference to any date; mutating the referent is
   // prohibited via this reference, but the referent may be changed
   // via an aliasing mutable reference.
   Date<ReadOnly> roD = ... ? immutD : mutD;
```

Statement 1 shows object immutability in IGJ, and statement 3 shows reference immutability. Fig. 4 shows a larger IGJ example.

Java's type arguments are no-variant, to avoid a type loophole [6, 20], so statement 3 is illegal in Java. Statement 3 is legal in IGJ, because IGJ allows covariant changes in the immutability parameter. IGJ even allows covariant changes in other type parameters if mutation is disallowed, e.g., `List<ReadOnly,Integer>` is a subtype of `List<ReadOnly,Number>`.

IGJ satisfies the following design principles:

Transitivity IGJ provides transitive (deep) immutability that protects the entire abstract state of an object. For example, an immutable graph contains an immutable set of immutable edges. C++ does not support such transitivity because its `const`-guarantee does not traverse pointers, i.e., a pointer in a `const` object can mutate its referent.

IGJ also permits excluding a field from the abstract state. For example, fields used for caching can be mutated even in an immutable object.

Static IGJ has no runtime representation for immutability, such as an "immutability bit" that is checked before assignments or method calls. Testing at runtime whether an object is immutable [29] hampers program understanding.

ESEC/FSE'07, September 3–7, 2007, Cavat near Dubrovnik, Croatia.

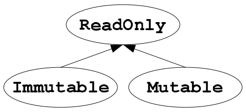

Figure 1: The hierarchy of immutability parameters, which have special meaning when used as the first type argument, as in `List<Mutable,T>`. (See also Fig. 6 in Sec. 2.4.)

The IGJ compiler works by type-erasure, without any run-time representation of reference or object immutability, which enables executing the resulting code on any JVM without runtime penalty. A similar approach was taken by Generic Java (GJ) [6] that extended Java 1.4. As with GJ, libraries must either be retrofitted with IGJ types, or fully converted to IGJ, before clients can be compiled. IGJ is backward compatible: every legal Java program is a legal IGJ program.

Polymorphism IGJ abstracts over immutability without code duplication by using generics and a flexible subtype relation. For instance, all the collection classes in C++'s STL have two overloaded versions of `iterator`, `operator[]`, etc. The underlying problem is the inability to return a reference whose immutability depends on the immutability of `this`:

```
const Foo& getFieldFoo() const;
      Foo& getFieldFoo();
```

Simplicity IGJ does not change Java's syntax. A small number of additional typing rules make IGJ more restrictive than Java. On the other hand, IGJ's subtyping rules are more relaxed, allowing covariant changes in a type-safe manner.

The contributions of this paper are: (i) a novel and simple design that naturally fits into Java's generics framework, (ii) an implementation of an IGJ compiler, proving feasibility of the design, and (iii) a formalization of IGJ with a proof of type soundness. Our ideas, though demonstrated using Java, are applicable to any statically typed language with generics, such as C++, C#, and Eiffel.

Outline. The remainder of this paper is organized as follows. Sec. 2 describes the IGJ language, which is compared to previous work in Sec. 3. Sec. 4 discusses case studies of our IGJ implementation, and Sec. 5 formalizes IGJ and gives a proof of soundness. Sec. 6 concludes.

2. IGJ language

The first type parameter of a class/interface in IGJ is called the *immutability parameter*. The first type argument of a type in IGJ is called the *immutability argument*, and it can be `Mutable`, `Immutable`, or `ReadOnly`.

2.1 Type hierarchy

Fig. 1 depicts the type hierarchy of immutability parameters. The subtyping relation is denoted by $\preceq$, e.g., $\texttt{Mutable} \preceq \texttt{ReadOnly}$. The classes `Mutable`, `Immutable`, and `ReadOnly` may not be extended, they have no subtype relation with any other types, and they can be used only as the first type argument or as a type bound.

The root of the IGJ type hierarchy (excluding `ReadOnly` and its descendants) is `Object<ReadOnly>`. Fig. 2 depicts the subtype relation for the classes `Object` and `Date`.

In Java, all type parameters are no-variant[1] (except when using wildcards, see Sec. 3.2). The subtyping rules for IGJ are more relaxed. IGJ permits covariant changes in the immutability parameter, e.g., $\texttt{Date<Mutable>} \preceq \texttt{Date<ReadOnly>}$. This satisfies the polymorphism design principle, because a programmer can write a single method that accepts a reference of any immutability, for example: `void print(Date<ReadOnly> d)`.

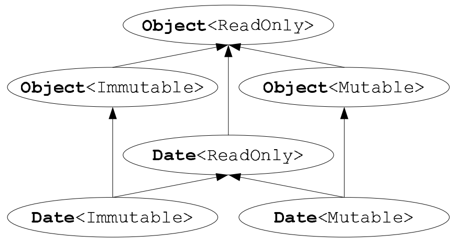

Figure 2: The subtype hierarchy for `Object` and `Date`. The classes (in bold) still have an underlying tree structure.

IGJ also permits covariant changes in other type parameters if the immutability argument is `ReadOnly` or `Immutable`, e.g.,

$$\texttt{List<ReadOnly,Integer>} \preceq \texttt{List<ReadOnly,Number>}$$

Covariant changes are safe only when the object is readonly or immutable because it cannot be mutated in a way which is not type-safe. Therefore, the following pair is not in the subtype relation:

$$\texttt{List<Mutable,Integer>} \not\preceq \texttt{List<Mutable,Number>}$$

To illustrate why covariant changes are prohibited in Java, consider a method that accepts `List<Mutable,Number>` as one of its arguments. If `List<Mutable,Integer>` is allowed to be passed into the method, then inside the method an element in the list could be changed to some `Number` that is not an `Integer`. This will break the guarantee that a list of the type `List<Mutable,Integer>` only contains instances of `Integer`.

A type parameter `X` in class `C` can be annotated[2] with `@NoVariant` to prevent covariant changes, in which case we say that `X` is no-variant and write *NoVariant*(X, C). Otherwise we say that `X` is covariant and write *CoVariant*(X, C). Sections 2.3 and 2.5 discuss when a type parameter should be no-variant, e.g., the type parameter `X` in the interface `Comparable<I,X>` is no-variant:

```
interface Comparable<I extends ReadOnly, @NoVariant X>
    { @ReadOnly int compareTo(X o); }
```

For instance, `Comparable<ReadOnly,Integer>` is not a subtype of `Comparable<ReadOnly,Number>`.

IGJ's subtype definition for two types of the same class is given in Def. 2.1. The full subtype definition (formally given in Fig. 12 of Sec. 5) includes all of Java's subtyping rules, therefore IGJ's subtype relation is a superset of Java's subtype relation.

DEFINITION 2.1. *Let* $\texttt{C<I}, \texttt{X}_1, \ldots, \texttt{X}_n\texttt{>}$ *be a class. Then, type* $\texttt{S} = \texttt{C<J}, \texttt{S}_1, \ldots, \texttt{S}_n\texttt{>}$ *is a subtype of* $\texttt{T} = \texttt{C<J}', \texttt{T}_1, \ldots, \texttt{T}_n\texttt{>}$*, written as* $\texttt{S} \preceq \texttt{T}$*, iff* $\texttt{J} \preceq \texttt{J}'$ *and for* $i = 1, \ldots, n$*, either* $\texttt{S}_i = \texttt{T}_i$ *or* ($\texttt{Immutable} \preceq \texttt{J}'$ *and* $\texttt{S}_i \preceq \texttt{T}_i$ *and CoVariant*($\texttt{X}_i, \texttt{C}$)).

Example. Fig. 3 presents the subtype hierarchy for `List<Object>`. The types `L<M,O<M>>`, `L<M,O<IM>>`, and `L<M,O<R>>` have a common *mutable* supertype `L<M,? extends O<? extends R>>`, but the only value that can be inserted in such a list is `null`. (See Sec. 3.2 for a discussion of Java's wildcards.)

2.2 Reference immutability

This section gives the three key type rules of IGJ that enforce reference immutability: that is, only a `Mutable` reference can modify its referent. To support reference immutability it is sufficient to use `ReadOnly` and `Mutable` references; Sec. 2.4 adds object immutability by using `Immutable` references as well.

[1] We use "no-variant" rather than "invariant" to avoid confusion with other meanings of the latter term.

[2] Annotating type parameters is planned for Java 7 [14]. In Java 5, a class or interface annotation would have to specify which positions are no-variant.

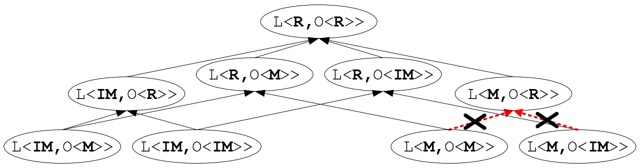

Figure 3: The subtype hierarchy for `List<Object>`, abbreviated as `L<O>`. The types `ReadOnly`, `Mutable`, and `Immutable` are abbreviated as `R`, `M`, and `IM`, respectively. The crossed-out arrows emphasize pairs that are *not* subtypes.

We use $I(\ldots)$ to denote a function that takes a type or a reference, and returns its immutability argument. For example, given a reference `mutD` of type `Date<Mutable>`, $I(\texttt{mutD}) = \texttt{Mutable}$, and we say that the *immutability of* `mutD` is `Mutable`.

A field can be assigned only by a mutable reference:

FIELD-ASSIGNMENT RULE:
`o.someField = ...` is legal iff $I(\texttt{o}) = \texttt{Mutable}$.

Thus, you cannot assign to fields of a readonly reference, e.g.,

```
Employee<ReadOnly> roE = ...;
roE.address = ...; // Illegal!
```

The immutability of `this` depends on the context, i.e., the method in which `this` appears:

THIS RULE: $I(\texttt{this}) = I(\texttt{m})$, in a method `m`.

Since there is no obvious syntactic way to denote the immutability of `this`, IGJ uses method annotations: `@ReadOnly`, `@Mutable`, etc.[3]

For example, below we have $I(\texttt{this}) = I(\texttt{m}) = \texttt{Mutable}$:

```
@Mutable void m() { ... this ... }
```

The default method annotation in IGJ is `@Mutable`, but for clarity of presentation, this paper explicitly annotates all methods.

The third type rule of IGJ states when a method call is legal:

METHOD-INVOCATION RULE: `o.m(...)` is legal iff $I(\texttt{o}) \preceq I(\texttt{m})$.

IGJ requires that $I(\texttt{o}) \preceq I(\texttt{m})$, and not simply $I(\texttt{o}) = I(\texttt{m})$, to allow a mutable reference to call readonly methods, e.g.,

```
Employee<Mutable> o = ...;
o.setAddress(...); // OK: I(o) ⪯ I(setAddress) = Mutable
o.getAddress();    // OK: I(o) ⪯ I(getAddress) = ReadOnly
((Employee<ReadOnly>) o).setAddress(...); // Illegal!
```

Example. Fig. 4 presents two IGJ classes: `Edge` and `Graph`. The immutability parameter `I` is declared in lines 1 and 10; by convention we always denote it by `I`. If the `extends` clause is missing from a class declaration, then we assume it extends `Object<I>`. We can use any subtype of `ReadOnly` in place of `I`, e.g., `ReadOnly` (on line 9), `Mutable` (on line 14), or another parameter such as `I` (on line 11) or `X` (on line 17).

We will now demonstrate the type-checking rules by example. The assignment `this.id = id` on line 5 is legal because according to THIS RULE we have that $I(\texttt{this}) = I(\texttt{setId}) = \texttt{Mutable}$, and according to FIELD-ASSIGNMENT RULE a mutable reference can assign to a field. That assignment would be illegal if it was moved to line 6, because `this` is readonly in the context of method `getId`. The method invocation `this.setId(...)` on line 3 is legal according to METHOD-INVOCATION RULE because $I(\texttt{this}) \preceq I(\texttt{setId})$. That method invocation would be illegal on line 6.

Observe on line 9 that the static method `print` does not have an annotation because it does not have an associated `this` object. According to Def. 2.1 of the subtype relation, an edge of any immutability can be passed to `print`.

[3] The paper uses the annotation `@ReadOnly` whereas the IGJ compiler uses `@ReadOnlyThis`, because an annotation and a class cannot have the same qualified name. The same applies for the other annotations.

```
1: class Edge<I extends ReadOnly> {
2:   private long id;
3:   @AssignsFields Edge(long id) { this.setId(id); }
4:   @AssignsFields synchronized void setId(long id) {
5:       this.id = id; }
6:   @ReadOnly synchronized long getId() { return id; }
7:   @Immutable long getIdImmutable() { return id; }
8:   @ReadOnly Edge<I> copy() { return new Edge<I>(id); }
9:   static void print(Edge<ReadOnly> e) { ... } }
10: class Graph<I extends ReadOnly> {
11:   List<I,Edge<I>> edges;
12:   @AssignsFields Graph(List<I,Edge<I>> edges) {
13:       this.edges = edges; }
14:   @Mutable void addEdge(Edge<Mutable> e) {
15:       this.edges.add(e); }
16:   static <X extends ReadOnly> Edge<X>
17:       findEdge(Graph<X> g, long id) { ... } }
```

Figure 4: IGJ classes `Edge<I>` and `Graph<I>`, with the immutability parameters and annotations underlined. Erasing the immutability parameters and annotations yields a legal Java program with the same semantics. The annotations `@Immutable` and `@AssignsFields` are explained in Sec. 2.4; for now assume that `@Immutable` is the same as `@ReadOnly`, and `@AssignsFields` is the same as `@Mutable`.

Recall that the **Transitivity** design principle states that the design must support transitive (deep) immutability. In our example, in a mutable `Graph` the field `edges` on line 11 will contain a mutable list of mutable edges. We call such a field `this`-*mutable* [31] because its immutability depends on the immutability of `this`: in a mutable object this field is mutable and in a readonly object it is readonly. C++ has similar behavior for fields without the keywords `const` or `mutable`. The advantage of IGJ syntax is that the concept of `this`-*mutable* is made explicit in the syntax: a class can reuse its immutability parameter in its fields, and the underlying generic type system propagates the immutability information without the need for special type rules. Using generics simplifies both the design and the implementation.

Moreover, C++ has no `this`-mutable local variables, return types, method parameters, or type arguments, whereas IGJ treats `I` as a regular type parameter. For example, the following are `this`-mutable: the return type on line 8, the type argument `Edge<I>` on line 11, and the method parameter `edges` on line 12.

2.3 Method overriding

IGJ respects the Java class hierarchy. An overriding method cannot weaken the specification of the overridden method:

METHOD-OVERRIDING RULE:
If method `m'` overrides `m`, then $I(\texttt{m}) \preceq I(\texttt{m'})$.

For example, overriding can change a mutable method to a readonly method, but not vice versa.

The *erased signature* of a method is obtained by replacing type parameters with their bounds. When the erased signature of an overriding method changes, the normal `javac` inserts a *bridge method* to cast the arguments to the correct type [6]. IGJ requires that the *erased signature* of an overriding method remains the same if that method is either readonly or immutable:

ERASED-SIGNATURE RULE: If method `m'` overrides method `m` and `Immutable` $\preceq I(\texttt{m})$, then the erased signatures of `m'` and `m`, excluding no-variant type parameters, must be identical.

Fig. 5 demonstrates why the ERASED-SIGNATURE RULE prohibits method overriding if the erased signature changes. As another example, if `X` was annotated as `@NoVariant` in line 1, then the over-

```
1: class MyVector<I extends ReadOnly, X> { ...
2:   @ReadOnly void isIn(X o) {...} }  // The erased signature is isIn(Object)
3: class MyIntVector<I extends ReadOnly> extends MyVector<I,Integer> { ...
4:    // Overriding isIn is illegal due to ERASED-SIGNATURE RULE: the erased signature isIn(Integer) is different from isIn(Object)
5:   @ReadOnly void isIn(Integer o) {...} }  // Would be legal if X was annotated with @NoVariant
6: MyVector<ReadOnly,Object> v = new MyIntVector<ReadOnly>(); // Would be illegal if X was annotated with @NoVariant
7: v.isIn( new Object() ); // If overriding were legal, the bridge method of isIn(Integer) would cast an Object to an Integer
```

Figure 5: An example of illegal method-overriding due to the ERASED-SIGNATURE RULE.

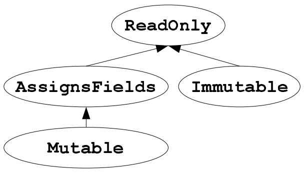

Figure 6: The full type hierarchy of immutability parameters.

riding in line 5 would be legal, and covariantly changing `X` (line 6) would be illegal.

Out of 82,262 methods in Java SDK 1.5, 30,169 methods override other methods, out of which only 51 have a different erased signature, and only the method `compareTo(X)` is readonly (the rest are mutable: `add`, `put`, `offer`, `create`, and `setValue`). Because `X` is no-variant in the `Comparable` interface, we conclude that ERASED-SIGNATURE RULE imposes no restrictions on the Java SDK.

2.4 Object immutability

One advantage of *object* immutability is enabling safe sharing between different threads without the cost of synchronization. Consider lines 6–7 in Fig. 4. A `long` read/write is not atomic in Java; synchronization is necessary. Only an immutable `Edge` can use `getIdImmutable()` to avoid the cost of synchronization.

The referent of a readonly reference (Sec. 2.2) is not immutable: it could be changed via another pointer. A separate analysis can indicate some cases when such changes are impossible [31], but it is preferable for the type system to guarantee that the referent of immutable references cannot change.

The IGJ type system makes such a guarantee:

> A mutable reference points to a mutable object, and
> an immutable reference points to an immutable object. (1)

In order to enforce this property, no immutable reference may be aliased by a mutable one; equivalently, no mutable reference may point to an immutable object.

2.4.1 Constructors and the annotation `@AssignsFields`

The rules given so far are sufficient to guarantee object immutability for IGJ with the exception of constructors. A constructor that is making an immutable object must be able to set the fields of the object (which becomes immutable as soon as the constructor returns). It is not acceptable to mark the constructor of an immutable object as `@Mutable`, which would permit arbitrary side effects, possibly including making mutable aliases to `this`.

IGJ uses a fourth kind of reference immutability, `AssignsFields`, to permit constructors to perform limited side effects without permitting modification of immutable objects. Whereas a `@Mutable` method can assign *and mutate* an object's fields, an `@AssignsFields` method can only assign (*not mutate*) the fields of <u>`this`</u>. A programmer can write the `@AssignsFields` method annotation but may *not* write the `AssignsFields` type in any other way, such as `Edge <AssignsFields>`. Therefore, in an `@AssignsFields` constructor, `this` can only escape as `ReadOnly`. Fig. 6 shows the full hierarchy of immutability parameters.

`AssignsFields` is not transitive, i.e., you can assign to fields of `this` but not to fields of fields of `this`. Specifically, we relax FIELD-ASSIGNMENT RULE by allowing a field of `this` to be assigned in an `@AssignsFields` method:

> **<u>FIELD-ASSIGNMENT RULE</u> revised**:
> `o.someField = ...` is legal iff
> $I(\texttt{o}) = \texttt{Mutable}$ or ($I(\texttt{o}) = \texttt{AssignsFields}$ and $\texttt{o} = \underline{\texttt{this}}$).

Next, we restrict the METHOD-INVOCATION RULE:

> **<u>METHOD-INVOCATION RULE</u> revised**:
> `o.m(...)` is legal iff
> $I(\texttt{o}) \preceq I(\texttt{m})$ and ($I(\texttt{m}) = \texttt{AssignsFields}$ implies $\texttt{o} = \underline{\texttt{this}}$).

(Adding $\texttt{o} = \underline{\texttt{this}}$ ensures that `AssignsFields` is not transitive.)

Creating a mutable object is legal only when using a `@Mutable`, `@AssignsFields`, or `@ReadOnly` constructor, i.e., it is illegal to create a mutable object using an `@Immutable` constructor because an immutable alias might escape the constructor. Similarly, it is illegal to create an immutable object using a `@Mutable` constructor. It is not always known at compile time whether a `new` operation creates a mutable or an immutable object, e.g., see line 8 of Fig. 4. In such cases, IGJ prohibits using either a `@Mutable` or an `@Immutable` constructor.

> **<u>OBJECT-CREATION RULE</u>**: `new SomeClass<X,...>(...)` is *illegal* iff the annotation `Y` of the constructor satisfies:
> $\texttt{Y} = \texttt{@Mutable}$ and $\texttt{X} \neq \texttt{Mutable}$, or
> $\texttt{Y} = \texttt{@Immutable}$ and $\texttt{X} \neq \texttt{Immutable}$.

Example. The assignment `this.id = id`, on line 5 of Fig. 4, is legal according to the revised FIELD-ASSIGNMENT RULE because method `setId` is annotated with `@AssignsFields` and thus the immutability of `this` is $I(\texttt{this}) = \texttt{AssignsFields}$. The method call `this.setId(...)` on line 3 is legal because

$$I(\texttt{this}) = \texttt{AssignsFields} \preceq I(\texttt{setId}) = \texttt{AssignsFields}.$$

The METHOD-INVOCATION RULE was revised to avoid transitivity of `AssignsFields`. E.g., adding `this.edges.get(0).setId(42)` to line 13 is legal in the old METHOD-INVOCATION RULE, but not in the revised one. Note that this addition must be illegal because it could mutate an immutable edge in the list `edges`.

IGJ can express *immutable cyclic* data-structures, as the following example of a bi-directional list shows:

```
class BiNode<I extends ReadOnly> {
  BiNode<I> prev, next;
  @AssignsFields BiNode(int len, BiNode<I> p) {
    next = len==0 ? null : new BiNode<I>(len-1, this);
    prev = p; } }
BiNode<Immutable> l = new BiNode<Immutable>(42,null);
```

A field of an immutable object can be assigned multiple times in the constructor or even in other `@AssignsFields` methods. This is harmless, and the programmer can mark a field as `final` to ensure that it is assigned in the constructor once and no more than once.

Field initializers. Field initializers are expressions that are used to initialize the object's fields. The immutability of `this` in such expressions is the maximal immutability among all constructors. For example, if all constructors are mutable, then `this` is mutable; if there exists a readonly constructor, then `this` is readonly.

```
1: class AccessOrderedSet<I extends ReadOnly,
2:     @NoVariant X> {
3:   private List<Mutable,X> l;
4:   public @ReadOnly boolean contains(X x) { ...
5:     // We can mutate this.l even though this is ReadOnly
6:     this.l.addFirst(x); } }
```

Figure 7: Class `AccessOrderedSet` with a mutable field `l`. Variable `X` must be no-variant because it is used in a mutable field.

2.5 Mutable and assignable fields

A type system should guarantee facts about the abstract state of an object, not merely its concrete representation. Therefore, a transitive guarantee of immutability for *all* fields of an object may be too strong. For example, fields used for caching are not part of the abstract state. This section discusses how to permit a given field to be assigned or mutated even in an immutable object, and than discusses special restrictions involving such fields.

Assignable fields. An assignable field is in essence the reverse of a final field: a final field cannot be re-assigned whereas an assignable field can always be assigned (even using a readonly reference). An assignable field is denoted by `@Assignable`. We revise FIELD-ASSIGNMENT RULE to always allow assigning to an assignable field:

FIELD-ASSIGNMENT RULE revised again:
`o.someField = ...` is legal iff
$I(\texttt{o}) = \texttt{Mutable}$ or ($I(\texttt{o}) = \texttt{AssignsFields}$ and $\texttt{o} = \texttt{this}$) or `someField` is annotated as `@Assignable`.

For example, consider this code snippet:

```
private @Assignable int memoizedHashCode = 0;
public @ReadOnly int hashCode() {
  if (memoizedHashCode == 0)
    memoizedHashCode = ...;
  return memoizedHashCode; }
```

The assignment `memoizedHashCode=...` is legal even though `hashCode` is readonly, due to the `@Assignable` annotation.

Mutable fields. A *mutable field* can always be mutated, even using a readonly reference. No new linguistic mechanism is required to express a mutable field: its immutability argument is `Mutable`.

For instance, `AccessOrderedSet` in Fig. 7 implements a set using a list `l` (line 3). As an optimization, `l` is maintained in access-order, even during calls to readonly methods such as `contains` (line 4). Because `l` is mutated (line 6) in a readonly method, `l` is declared as a mutable field.

Covariant and no-variant type parameters. The type parameter `X` in Fig. 7 must be annotated as `@NoVariant` (line 2) due to its use in the mutable field `l`. If `X` could change covariantly, we would have:

`AccessOrderedSet<ReadOnly, Integer>` $\preceq$
`AccessOrderedSet<ReadOnly, Number>`

We could than add a `Number` to an `Integer` list using the `contains` method in line 4 of Fig. 7. To avoid such type loopholes, IGJ requires a `@NoVariant` annotation on a type parameter which is used in a mutable field. An assignable field or a mutable superclass cause the same restriction as a mutable field:

NOVARIANT RULE: A type parameter must be no-variant if it is used in a mutable field, an assignable field, a mutable superclass, or in the position of another no-variant type parameter. (See a formal definition in Fig. 11 of Sec. 5.)

The immutability parameter must be allowed to change covariantly, or else a mutable reference could not call a readonly method:

COVARIANT RULE: *CoVariant*$(\texttt{I}, \texttt{C})$ must hold for any class `C`.

For example, declaring the following field is prohibited:

`@Assignable Edge<I> f;` *// Illegal!* `I` *must be covariant.*

2.6 Exceptions, immutable classes, reflection, and arrays

In IGJ's syntax, the immutability is an integral part of the type. In Javari [31] (see Sec. 3) it is syntactically possible but semantically illegal to write this code:

`class Cell<X> { readonly X x; ...}`

It is semantically illegal because the immutability of `X` is determined in the client code, e.g., `Cell<readonly Date>`. In comparison, IGJ's syntax does not even enable such a declaration: it is syntactically and semantically illegal.

Throwable. Generics and immutability naturally combine in another aspect: their *usage limitations*. For example, it is forbidden to throw a readonly reference because the catcher can mutate that reference. Similarly, Java prohibits adding type parameters to any subclass of `Throwable` because the compiler cannot statically connect the throwing and catching positions. IGJ implicitly inherits this usage limitation from the underlying generics mechanism. In contrast, Javari explicitly prohibits throwing readonly exceptions.

Manifest classes and immutable classes. IGJ supports *manifest classes* [27], which are classes without an immutability parameter. Manifest classes can be used to express *class immutability*, e.g.,

`class String extends Object<Immutable> ...`

IGJ treats all methods of `String` as if they were annotated with `@Immutable`, and issues errors if mutable methods exist.

Reflection. It is discouraged in IGJ to use reflection or to remove the immutability parameter by casting to a raw type. The IGJ compiler issues a *warning* in both cases because they can create holes in the type system. (IGJ does not consider these *errors* because they might be necessary to call legacy code.)

Arrays. Java does not permit arrays to have type parameters. IGJ supplies a wrapper class `Array<I extends ReadOnly,T>` that enables the creation of immutable arrays. IGJ treats an array type `T[]` as `Array<Mutable,T>`, i.e., arrays are mutable by default.

2.7 Inner classes

Nested classes that are *static* can be treated the same as normal classes. An *inner class* is a nested class that is not explicitly or implicitly declared static (see JLS 8.1.3 [18]). Inner classes have an additional `this` reference: `OuterClass.this`. According to THIS RULE, the immutability of `this` depends on the immutability of the method. Because methods in IGJ have a single method annotation, the immutability of `this` and `OuterClass.this` should be the same. Therefore, in IGJ an inner class cannot have its own immutability parameter:

INNER-CLASS RULE: An inner class inherits the immutability parameter of the outer class.

Lines 1–5 in Fig. 8 show the declaration of the `Iterator` interface, in which the only mutable method is `remove`. The immutability of an *iterator* is inherited from its *container*. Even though method `next` (line 3) changes the state of the *iterator*, it does not change the state of the *container*, and is thus readonly. In contrast, method `remove` (line 4) changes the container, and is thus mutable.

Now consider the class `ArrayList` and its inner class `ArrItr`. The inner class `ArrItr` lacks an explicit immutability parameter (line 8), because it is implicitly inherited from `ArrayList`. On line 13 both `this` references are mutable because `remove()` is mutable. Finally, consider the creation of a new iterator on line 7. We handle this `new` operation using METHOD-INVOCATION RULE: this method call

```
interface Iterator<I extends ReadOnly,E> {
  @ReadOnly boolean hasNext();
  @ReadOnly E next();       // Although next changes the iterator, it is readonly because it does not change the container.
  @Mutable void remove();  // remove is mutable because it changes the container.
}
class ArrayList<I extends ReadOnly, E> ...   { ...
  public @ReadOnly Iterator<I,E> iterator() { return this.new ArrItr(); }  // OK: I(this) ⪯ I(ArrItr) = ReadOnly
  class ArrItr implements Iterator<I,E> { // ArrItr has no explicit immutability parameter: it is inherited from the outer class
    private @Assignable int currPos;
    public @ReadOnly ArrItr() { this.currPos=0; }  // OK: currPos is @Assignable
    public @ReadOnly boolean hasNext() { return this.currPos < ArrayList.this.size(); }
    public @ReadOnly E next() { return ArrayList.this.get( this.currPos++ ); }     // OK: I(this) ⪯ I(get) = ReadOnly
    public @Mutable void remove() { ArrayList.this.remove( this.currPos - 1 ); }  // OK: I(this) ⪯ I(remove) = Mutable
  } }
```

Figure 8: Declaration of the interface `Iterator` and the class `ArrayList` with an inner class `ArrItr`.

is legal because `this` is readonly and the constructor of `ArrItr` is readonly. We do not use OBJECT-CREATION RULE because the inner object inherits the immutability of the outer object.

Anonymous inner classes have no name and no constructor. IGJ assumes that the immutability of the missing constructor is the same as the immutability of the method declaring the anonymous inner class. For instance, the code in Fig. 8 can be converted to use an anonymous inner class:

```
public @ReadOnly Iterator<I,E> iterator() {
  return new Iterator<I,E>() { ... }; }
```

In the example above the immutability of the constructor is readonly because `iterator()` is readonly.

3. Previous Work

We are not aware of any previous work that proposed a static object-oriented typing-system for *object immutability* and not just reference immutability. Pechtchanski and Sarkar [25] describe annotations for immutability assertions, such as `@immutableField`, `@immutableParam`, etc., and show that such assertions enable optimizations that can speed up some benchmarks by 5–10%. They do not present any typing rules to enforce such assertions.

Functional languages such as ML default all fields to being immutable, with mutable (“`ref`”) fields being the exception. Such languages do not support `this`-mutable fields nor allow partially initialized objects to escape from the constructor.

Java already includes various classes whose instances are immutable, and it supports non-transitive immutability using `final`, which prohibits field assignments after the constructor finishes.

C++'s `const` mechanism has similar semantics to IGJ: a field can be declared as `const` (similar to readonly in IGJ), `mutable`, or by default as `this`-mutable. In contrast to IGJ, C++ has no `this`-mutable parameters, return types, local variables, or type arguments. Other disadvantages are: (i) `const` can be cast away at any time, making it more a suggestion than a binding contract, (ii) `const` protects only the state of the enclosing object and not objects it points to, e.g., you cannot mutate an element inside a `const` node in a list, but the `next` node is mutable, and (iii) using `const` results in code duplication such as two versions of `operator[]` in every collection class in the STL.

Most of the IGJ terminology was borrowed from Javari [31] such as assignable, readonly, mutable, and `this`-mutable. In Javari, `this`-mutable fields are mutable as lvalue and readonly as rvalue. Javari does not support object immutability, and its reference immutability is more limited than that of IGJ because Javari has no `this`-mutable parameters, return types, or local variables. Javari's keyword `romaybe` is in essence a template over immutability. IGJ uses generics directly to achieve the same goal, as demonstrated by the static method `findEdge` on line 16 of Fig. 4. The same method in Javari would be written as

```
romaybe Edge findEdge(romaybe Graph g, long id)
```

Finally, Javari uses `? readonly` which is similar to Java's wildcards. Consider, for instance, the class `Foo` written in Javari's syntax:

```
class Foo { mutable List<Object> list; }
```

Then in a `readonly Foo` the type of `list` is

```
mutable List<? readonly Object>
```

which is syntactic sugar for

```
mutable List<? extends readonly Object
            super mutable Object>
```

Thus, it is possible to insert only mutable elements to `list`, and retrieve only readonly elements.

Skoglund and Wrigstad [29] propose a system with read and write references with similar semantics to C++'s `const`. They also introduce a `caseModeOf` construct which permits run-time checking of reference writeability.

Several papers proposed a mechanism of *access rights*. JAC [22] is a compile-time access-right system with this access-right order: `readnothing < readimmutable < readonly < writeable`. Right `readnothing` cannot access fields of `this` (only the identity for equality), and `readimmutable` can only access immutable state of `this`. JAC uses additional keywords (such as `nontransferable`) that address other concerns than immutability. Capabilities for sharing [5] are intended to generalize various other proposals for access rights, ownership and immutability, by giving a lower level semantics that can be enforced at compile- or run-time. A reference can possess any combination of these 7 access rights: read, write, identity (permitting address comparisons), exclusive read, exclusive write, exclusive identity, and ownership (giving the capability to assert rights). Immutability, for example, is represented by the lack of the write right and possession of the exclusive write right.

Boyland [4] concludes that readonly does not address observational exposure, i.e., modifications on one side of an abstraction boundary that are observable on the other side. IGJ's immutable objects address such exposure because their state cannot change. Boyland's second criticism was that the **transitivity** principle (see Sec. 1) should be selectively applied by the designer, because, “the elements in the container are not notionally part of the container” [4]. In IGJ, a programmer can solve this problem by using a different immutability for the container and its elements.

Non-null types [16] has a similar challenge that IGJ has in constructing immutable objects: a partially-initialized object may escape its constructor. IGJ uses `@AssignsFields` to mark a constructor of immutable objects, and a partially initialized object can escape only as `ReadOnly`. Non-null types uses a `Raw` annotation *on* *references* that might point to a partially-initialized object, and *on methods* to denote that the receiver can be `Raw`. A non-null field of a `Raw` object has different lvalue and rvalue: it is possible to assign only non-null values to such field, whereas reading from such field may return `null`. Similarly to IGJ, non-null types cannot express the staged initialization paradigm in which the construction of an object continues after its constructor finishes.

Huang et al. [19] propose an extension of Java (called cJ) that allows methods to be provided only under some static subtyping condition. For instance, a cJ generic class, `Date<I>`, can define

```
<I extends Mutable>? void setDate(...)
```

which will be provided only when the type provided for parameter `I` is a subtype of `Mutable`. Designing IGJ on top of cJ would make METHOD-INVOCATION RULE redundant, at the cost of replacing IGJ's method annotations with cJ's conditional method syntax.

Finally, IGJ uses the type system to check immutability statically. Controlling immutability at runtime (for example using assertions or Eiffel-like contractual obligations) falls outside the scope of this paper.

3.1 Ownership types and readonly references

Ownership types [3, 23] impose a structure on the references between objects in a program's memory. Ownership-enabled languages such as Ownership Generic Java [27] prevent aliasing to the internal state of an object. While preventing exposure of owned objects, ownership does not address exposing immutable parts of an object that cannot break encapsulation.

One possible application of ownership types is the ability to reason about read and write effects [8] which has complimentary goals to object immutability. Universes [12] is a Java language extension combining *ownership and reference immutability*. Most ownership systems enforce that all reference chains to an owned object pass through the owner. Universes relaxes this demand by enforcing this rule only for mutable references, i.e., readonly references can be shared without restriction.

3.2 Covariant subtyping

Covariant subtyping allows type arguments to covariantly change in a type-safe manner. Variant parametric types [21] attach a variance annotation to a type argument, e.g., `Vector<+Number>` (for covariant typing) or `Vector<-Number>` (for contravariant typing). Its subtype relation contains this chain:

```
Vector<Integer> ⪯ Vector<+Integer> ⪯
 ⪯ Vector<+Number> ⪯ Vector<+Object>
```

The type checker prohibits calling `someMethod(X)` when the receiver is of type `Foo<+X>`. For instance, suppose there is a method `isIn(X)` in class `Vector<X>`. Then, it is prohibited to call `isIn(Number)` on a reference of type `Vector<+Number>`.

Java's wildcards have a similar chain in the subtype relation:

```
Vector<Integer> ⪯ Vector<? extends Integer> ⪯
 ⪯ Vector<? extends Number> ⪯ Vector<? extends Object>
```

Java's wildcards and variant parametric types are different in the legality of invoking `isIn(? extends Number)` on a reference of type `Vector<? extends Number>`. A variant parametric type system prohibits such an invocation. Java permits such an invocation, but the only value of type `? extends Number` is `null`.

IGJ also contains a similar chain:

```
Vector<Mutable,Integer> ⪯ Vector<ReadOnly,Integer> ⪯
 ⪯ Vector<ReadOnly,Number> ⪯ Vector<ReadOnly,Object>
```

The restriction on method calls in IGJ is based on user-chosen *semantics* (whether the method is readonly or not) rather than on *method signature* as in wildcards and variant parametric types. For example, IGJ allows calling `isIn(Number)` on a reference of type `Vector<ReadOnly,Number>` iff `isIn` is readonly.

3.3 Typestates for objects

In a typestate system, each object is in a certain state, and the set of applicable methods depends on the current state. Verifying typestates statically is challenging due to the existence of aliases, i.e., a state-change in a particular object must affect all its aliases. Typestates for objects [11] uses linear types to manage aliasing.

Object immutability can be partially expressed using typestates: by using two states (mutable and immutable) and declaring that mutating methods are applicable only in the mutable state. An additional method should mark the transition from a mutable state to an immutable state, and it should be called after the initialization of the object has finished. It remains to be seen if systems such as [11] can handle arbitrary aliases that occur in real programs, e.g., `this` references that escape the constructor.

4. Case studies

Our preliminary experience suggests that IGJ is useful in expressing and checking important immutability properties.

We created two Eclipse plug-ins for converting Java code into IGJ. The first plug-in converts a class to IGJ by adding to it an immutability parameter, and setting the immutability argument to `Mutable` in all clients of that class. The second plug-in generates IGJ skeletons of libraries' public signatures, permitting signature annotation without the need to modify the library code.

The IGJ compiler is a relatively small and simple extension to Sun's javac. The IGJ compiler uses a visitor pattern to visit every element in the Abstract Syntax Tree (AST) before the Java attribution phase, checking for appropriate use of the immutability parameter. After the Java attribution phase, it uses another AST visitor to detect any violation of the typing rules. Finally, it modifies javac's `isSubType` according to Def. 2.1.

We performed case studies on the jolden benchmark programs[4], the htmlparser library[5], and the SVNKit Subversion client[6]. In all, we converted 328 classes (106 KLOC) of code to type-correct IGJ, refactoring the code only in minor ways noted below. We also annotated the signatures of 113 JDK classes and interfaces.

Conversion to IGJ revealed representation exposure errors. For example, in the `htmlparser` library, the "and" filter constructor takes an array of predicates and assigns it to a private field without copying; an accessor method also returns that private field without copying. Clients of either method can mutate both the array's length and its contents.

Conversion to IGJ also allowed us to find and fix a conceptual problem in several immutable classes, where the constructor left the object in an inconsistent state. For example, consider jolden's `perimeter` program, which computes the perimeter of a region in a binary image represented by a quad-tree. All instances of `Quadrant` and `QuadTreeNode` are immutable, so we made these classes and their subclasses immutable. Factory method `createTree` (Fig. 9) creates a new `GreyNode` `QuadTreeNode` with no children (line 7), then later calls `setChildren` (line 10). Such a call is illegal because `QuadTreeNode` is an immutable class. Solving such problems was easy: we added parameters to the constructor/factory to give it access to the complete state of the new object, or moved all of the logic of object construction into a single method rather than dispersing it. (In the case of `QuadTreeNode`, we could have used the `AssignsFields` immutability for `setChildren`.)

[4] http://www-ali.cs.umass.edu/DaCapo/benchmarks.html
[5] http://htmlparser.sourceforge.net/
[6] http://svnkit.com/

$\mathtt{T} ::= \mathtt{X} \mid \mathtt{N}$	Type.
$\mathtt{N} ::= \mathtt{C}\texttt{<}\mathtt{J}, \overline{\mathtt{T}}\texttt{>}$	Non-variable type.
$\mathtt{J} ::= \mathtt{ReadOnly} \mid \mathtt{Mutable} \mid \mathtt{Immutable} \mid \mathtt{I}$	Immutability arguments.
$\mathtt{L} ::= \mathtt{class}\ \mathtt{C}\texttt{<}\mathtt{I} \lhd \mathtt{ReadOnly}, \overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\texttt{>} \lhd \mathtt{C}'\texttt{<}\mathtt{I}, \overline{\mathtt{T}'}\texttt{>}\{\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\}$	Class declaration.
$\mathtt{M} ::= \texttt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\texttt{>}\ \mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return}\ \mathtt{e};\ \}$	Method declaration.
$\mathtt{e} ::= \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}\texttt{<}\overline{\mathtt{T}}\texttt{>}(\overline{\mathtt{e}}) \mid \mathtt{new}\ \mathtt{N}(\overline{\mathtt{e}}) \mid (\mathtt{N})\ \mathtt{e} \mid \mathtt{e.f} = \mathtt{e} \mid \mathtt{l}$	Expressions.

Figure 10: FIGJ Syntax.

$$\frac{\mathtt{C}'\texttt{<Mutable},\overline{\mathtt{T}}\texttt{>} \in \mathit{subterm}(\mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>}) \quad \mathtt{X}_i \in \mathit{TP}(\mathtt{T}_j)}{\mathit{NoVariant}(\mathtt{X}_i, \mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>})}\ (\mathrm{MC1}) \qquad \frac{\mathtt{C}'\texttt{<}\overline{\mathtt{T}}\texttt{>} \in \mathit{subterm}(\mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>}) \quad \mathit{NoVariant}(\mathtt{Y}_j, \mathtt{C}'\texttt{<}\overline{\mathtt{Y}}\texttt{>}) \quad \mathtt{X}_i \in \mathit{TP}(\mathtt{T}_j)}{\mathit{NoVariant}(\mathtt{X}_i, \mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>})}\ (\mathrm{MC2})$$

Figure 11: Definition of $\mathit{NoVariant}(\mathtt{X}_i, \mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>})$.

```
 1: public static
 2: QuadTreeNode createTree(QuadTreeNode parent,...)  {
 3:    QuadTreeNode node;
 4:    if (...)  { node = new BlackNode(...); }
 5:    else if (...)  { node = new WhiteNode(...); }
 6:    else {
 7:      node = new GreyNode(...);
 8:      sw = createTree(node, ...);
 9:      se=...; nw=...; ne=...;
10:      node.setChildren(sw,se,nw,ne); }
11:    return node;
12: }
```

Figure 9: The `QuadTreeNode.createTree` method of the `perimeter` program. Class `QuadTreeNode` should be immutable, so the call to `setChildren` on line 10 fails to type-check.

We were able to use both immutable classes and immutable objects. SVNKit used the latter for `Date` objects that represent the beginning and expiration of file locks, the URL to the repository (IGJ could simplify the current design, which uses an immutable `SVNURL` class with setter methods that return new instances), and many `List`s and `Array`s of metadata. IGJ could also permit use of immutable objects in some places where immutable classes are currently used, increasing flexibility.

Our biggest problem in the case study was the fact that Java does not permit generic arguments to be attached to arrays. The jolden benchmarks had been transliterated from C and used many arrays; we expect such uses to be much rarer in good object-oriented code.

Most fields used the containing class's immutability parameter. We used few mutable fields; one of the rare exceptions was a collection (in `SVNErrorCode`) that contains all `SVNErrorCode`s ever created. We never used the *NoVariant* rule for mutable fields. We used `@Assignable` fields only 5 times — to allow the receiver of a tree rebalancing operation, or the receiver of a method that resizes a buffer without mutating the contents, to be marked as `ReadOnly`.

Annotating existing code is an important test of IGJ, but IGJ is likely to be even more effective on code that is designed with immutability in mind. We saw many places that a different — and better! — design would have been encouraged by IGJ.

5. Proof of Type Soundness

Proving soundness is essential in the face of complexities such as the new subtype definition (Def. 2.1) and mutable fields (Sec. 2.5). This section gives the type rules of a simplified version of IGJ and proves property (1) from Sec. 2.4. We are not aware of any previous work that proved a reference immutability theorem such as "readonly references cannot be converted to mutable". Property (1) implies such theorem, or else it would be possible to convert immutable to readonly, and then to mutable.

Our type system, called Featherweight IGJ (FIGJ), is based on Featherweight Generic Java (FGJ) [20]. FIGJ models the essence of IGJ: the fact that only mutable references can assign to fields, and the new subtype definition. All the features of IGJ that are not in FIGJ do not introduce any new difficulties — merely more tedious but conceptually similar cases — in the proof. Similar to the way FGJ removed many features from Java (such as null values, assignment, overloading, private, etc.), we removed from IGJ all method annotations. All methods in FIGJ are readonly, thus assignment must be done from the "outside", i.e., instead of calling a setter method we must set the field from the outside (all fields are considered public in FGJ). We removed the `AssignsFields` immutability; FIGJ has a single constructor that assigns its arguments to the object's fields. Finally, we restrict each class in FIGJ to have a *single* immutability parameter which extends `ReadOnly`, i.e., FIGJ cannot express manifest classes such as `String`.

Sec. 5.1 describes the syntax of FIGJ. Sec. 5.2 presents the FIGJ subtype relation. Sec. 5.3 modifies FGJ typing and reduction rules. Sec. 5.4 proves preservation and progress.

5.1 Featherweight IGJ Syntax

FIGJ adds imperative extensions to FGJ such as assignment to fields, object locations, and a store [26]. Fig. 10 presents the syntax of FIGJ. It defines types ($\mathtt{T}$), non-variable types ($\mathtt{N}$), immutability arguments ($\mathtt{J}$), class declarations ($\mathtt{L}$), method declarations ($\mathtt{M}$), and expressions ($\mathtt{e}$). Expressions in FIGJ include the five expressions in FGJ (method parameter, field access, method invocation, new instance creation, and cast), as well as the imperative extensions (field update and locations). Note that an immutability argument ($\mathtt{J}$) only appears in the syntax as the first type argument. Thus, the syntax does not allow defining a field of type `ReadOnly` nor defining a class with two type parameters extending `ReadOnly`. The root of the class hierarchy is $\mathtt{Object}\texttt{<}\mathtt{X} \lhd \mathtt{ReadOnly}\texttt{>}$.

The store $\mathtt{S} = \{\mathtt{l} \mapsto \mathtt{N}(\overline{\mathtt{l}})\}$ maps locations to objects. Note that we do not need a store typing [26] because the store already contains the type of each location. For a simpler notation, we use a single symbol Δ to denote an environment that maps (i) method parameters to their types, and (ii) type parameters to their bounds (which are non-variable types): $\Delta = \{\mathtt{x} : \mathtt{T}\} \cup \{\mathtt{X} \preceq \mathtt{N}\}$.

The field, method type, and method body lookup functions, are based on their counterparts in FGJ, and thus omitted from this paper. We define an additional auxiliary function that returns the immutability argument $I_\Delta(\mathtt{C}\texttt{<}\mathtt{J}, \overline{\mathtt{T}}\texttt{>}) = \mathtt{J}$, and for a type parameter returns the immutability argument of its bound $I_\Delta(\mathtt{X}) = I_\Delta(\Delta(\mathtt{X}))$. We remove the subscript Δ when it is clear from the context.

We make the same assumptions as in FGJ about the correctness of the class declarations (e.g., that there are no circles in the subclass relation, that we have no method overloading, etc). We also use the same judgements as in FGJ, such as type, store, expressions, method and class wellformedness, with minor differences (e.g., instead of `Object`, we use `Object<I>`).

Fig. 11 defines which type parameters are no-variant. We use two auxiliary functions: (i) $\mathit{TP}(\mathtt{T})$ is the set of type parameters in $\mathtt{T}$, (ii) $\mathit{subterm}(\mathtt{N})$ is the set of all subterms of types in $\mathit{fields}(\mathtt{N})$. (We do not need to consider the superclass as in the NoVariant Rule because *fields* includes all the fields of the superclass as well.)

After *NoVariant* reaches a fixed point, we define *CoVariant* as the negation of *NoVariant*. In order for the class declarations to be well-formed, the immutability parameter must always be covariant (see CoVariant Rule): $\mathit{CoVariant}(\mathtt{X}_1, \mathtt{C<\overline{X}>})$ for any class $\mathtt{C}$.

FIGJ is more strict than FGJ regarding method overriding when the signature contains covariant type parameters, i.e., FIGJ requires that the *erased signature* of an overriding method (excluding no-variant type parameters) does not change (see Erased-Signature Rule).

5.2 Subtyping

Fig. 12 shows FIGJ subtyping rules. The first four rules are the same as FGJ rules. Additionally, two special classes — `Mutable` and `Immutable`— are considered a separate class hierarchy extending `ReadOnly`. The rule S1 is a formalization of Def. 2.1. We write $\mathtt{T} \preceq \mathtt{T}'$ as a shorthand for $\Delta \vdash \mathtt{T} \preceq \mathtt{T}'$. Observe that the subtype relation is reflexive and transitive. Note that from rule S1 and the CoVariant Rule we have that

if $\mathtt{I} \preceq \mathtt{I}'$, then $\mathtt{C<I,\overline{T}>} \preceq \mathtt{C<I',\overline{T}>}$.

Observe in rule S1 that the requirement $\mathtt{Immutable} \preceq \mathtt{T}'_1$ is equivalent to $\mathtt{T}'_1 \neq \mathtt{Mutable}$ and $\mathtt{T}'_1 \neq \mathtt{I}$, because an immutability argument can have only four values according to the syntax rule for $\mathtt{J}$ in Fig. 10.

In FGJ, if $\mathtt{A} \preceq \mathtt{B}$ and $\mathtt{f}$ is a field of $\mathtt{B}$, then the type of field $\mathtt{f}$ in $\mathtt{A}$ and $\mathtt{B}$ is exactly the same, i.e., $\mathtt{A.f} = \mathtt{B.f}$. In FIGJ, we prove instead that $\mathtt{A.f} \preceq \mathtt{B.f}$. For instance, consider the field `edges` on line 11 of Fig. 4, and the following two references:

```
Graph<Mutable> mutG;
Graph<ReadOnly> roG;
// mutG.edges has the type List<Mutable,Edge<Mutable>>
// roG.edges has the type List<ReadOnly,Edge<ReadOnly>>
```

And indeed the first is a subtype of the latter.

Lemma 5.1. *Let* $\mathtt{T} \preceq \mathtt{T}'$, $\mathtt{F}'\ \mathtt{f} \in \mathit{fields}(\mathtt{T}')$. *Then* $\mathtt{F}\ \mathtt{f} \in \mathit{fields}(\mathtt{T})$ *and* $\mathtt{F} \preceq \mathtt{F}'$.

Proof. It is trivial to prove that field $\mathtt{f}$ exists in $\mathit{fields}(\mathtt{T})$. We will prove that $\mathtt{F} \preceq \mathtt{F}'$ by induction on the derivation of $\mathtt{T} \preceq \mathtt{T}'$. Consider the last rule in the derivation sequence. The proof for the first six rules is immediate from the definition of *fields* and the fact that subtyping is transitive.

Now consider rule S1, where $\mathtt{T} = \mathtt{C<\overline{U}>}$, and $\mathtt{T}' = \mathtt{C<\overline{U'}>}$:

$$\begin{array}{l} \mathtt{U}_i = \mathtt{U}'_i \text{ or} \\ \left(\mathtt{Immutable} \preceq \mathtt{U}'_1 \text{ and } \mathtt{U}_i \preceq \mathtt{U}'_i \text{ and } \mathit{CoVariant}(\mathtt{X}_i, \mathtt{C<\overline{X}>})\right) \end{array} \tag{2}$$

Let $\mathtt{V}$ denote the type of field $\mathtt{f}$ in $\mathtt{C<\overline{X}>}$. Then, $\mathtt{F} = [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{V}$ and $\mathtt{F}' = [\overline{\mathtt{U}'}/\overline{\mathtt{X}}]\mathtt{V}$. We wish to prove that $\mathtt{F} \preceq \mathtt{F}'$.

We will prove by induction (on the subterm size) that for every subterm $\mathtt{A<\overline{S}>}$ in $\mathtt{V}$,

$$[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{A<\overline{S}>} \preceq [\overline{\mathtt{U}'}/\overline{\mathtt{X}}]\mathtt{A<\overline{S}>}. \tag{3}$$

From (3), we will be able to conclude that $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{V} \preceq [\overline{\mathtt{U}'}/\overline{\mathtt{X}}]\mathtt{V}$, i.e., $\mathtt{F} \preceq \mathtt{F}'$. In order to apply rule S1 on (3), we need to prove that for all j:

$$\begin{array}{l} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}_j = [\overline{\mathtt{U}'}/\overline{\mathtt{X}}]\mathtt{S}_j \text{ or } \left(\mathtt{Immutable} \preceq [\overline{\mathtt{U}'}/\overline{\mathtt{X}}]\mathtt{S}_1 \text{ and} \right. \\ \left. [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}_j \preceq [\overline{\mathtt{U}'}/\overline{\mathtt{X}}]\mathtt{S}_j \text{ and } \mathit{CoVariant}(\mathtt{Y}_j, \mathtt{A<\overline{Y}>})\right) \end{array} \tag{4}$$

Let j be fixed. If for all i, $\mathtt{X}_i \notin \mathit{TP}(\mathtt{S}_j)$ or $\mathtt{U}_i = \mathtt{U}'_i$, then $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}_j = [\overline{\mathtt{U}'}/\overline{\mathtt{X}}]\mathtt{S}_j$, and we proved (4) for this j. Thus, exists i for which hold

$$\frac{\ldots \qquad \Delta, \mathtt{S} \vdash \mathtt{e} : \mathtt{T} \qquad I(\mathtt{T}) = \mathtt{Mutable}}{\Delta, \mathtt{S} \vdash \mathtt{e.f}_i = \mathtt{e}' : \mathtt{T}'} \quad \text{(T-Field-Set)}$$

Figure 13: Typing Rules (only T-Field-Set was modified).

$$\frac{\begin{array}{c} \mathtt{S}[\mathtt{l}] = \mathtt{N}(\overline{\mathtt{l}}) \qquad I(\mathbf{N}) = \mathbf{Mutable} \\ \mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \mathtt{S}' = \mathtt{S}[\mathtt{l} \mapsto [\mathtt{l}'/\mathtt{l}_i]\mathtt{N}(\overline{\mathtt{l}})] \end{array}}{\mathtt{l.f}_i = \mathtt{l}', \mathtt{S} \rightarrow \mathtt{l}', \mathtt{S}'} \quad \text{(R-Field-Set)}$$

Figure 14: Reduction Rules (only R-Field-Set was modified).

both $\mathtt{U}_i \neq \mathtt{U}'_i$ and

$$\mathtt{X}_i \in \mathit{TP}(\mathtt{S}_j). \tag{5}$$

From (2) and $\mathtt{U}_i \neq \mathtt{U}'_i$, we have that

$$\mathtt{Immutable} \preceq \mathtt{U}'_1 \text{ and } \mathit{CoVariant}(\mathtt{X}_i, \mathtt{C<\overline{X}>}). \tag{6}$$

If $\mathit{NoVariant}(\mathtt{Y}_j, \mathtt{A<\overline{Y}>})$, according to rule MC2 in Fig. 11 and (5), we have $\mathit{NoVariant}(\mathtt{X}_i, \mathtt{C<\overline{X}>})$, which contradicts (6). Thus,

$$\mathit{CoVariant}(\mathtt{Y}_j, \mathtt{A<\overline{Y}>}). \tag{7}$$

We consider all 4 options for the immutability argument $\mathtt{S}_1$ to prove

$$\mathtt{Immutable} \preceq [\overline{\mathtt{U}'}/\overline{\mathtt{X}}]\mathtt{S}_1. \tag{8}$$

If $\mathtt{S}_1 = \mathtt{ReadOnly}$ or $\mathtt{S}_1 = \mathtt{Immutable}$, then (8) holds. If $\mathtt{S}_1 = \mathtt{Mutable}$, then according to rule MC1 in Fig. 11 and (5), we have $\mathit{NoVariant}(\mathtt{X}_i, \mathtt{C<\overline{X}>})$, which contradicts (6). If $\mathtt{S}_1 = \mathtt{I} = \mathtt{X}_1$, then $[\overline{\mathtt{U}'}/\overline{\mathtt{X}}]\mathtt{S}_1 = \mathtt{U}'_1$, and from (6) we proved (8).

If $\mathtt{S}_j$ is some type parameter $\mathtt{X}_k$, then

$$[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}_j \preceq [\overline{\mathtt{U}'}/\overline{\mathtt{X}}]\mathtt{S}_j, \tag{9}$$

because from (2) we have $\mathtt{U}_k \preceq \mathtt{U}'_k$. If $\mathtt{S}_j$ is a non-variable type, then from the induction hypothesis, we also have (9). Using, (7), (8), and (9), we proved (4) for this j. □

In a legal assignment $\mathtt{e}_1\mathtt{.f} = \mathtt{e}_2$, where $\mathtt{e}_1 : \mathtt{T}$, we have that $I(\mathtt{T}) = \mathtt{Mutable}$. Lem. 5.2 proves that in all subtypes $\mathtt{T}' \preceq \mathtt{T}$, the field $\mathtt{f}$ does not change covariantly.

Lemma 5.2. *Let* $\Delta \vdash \mathtt{T} \preceq \mathtt{T}'$, $\mathtt{F}'\ \mathtt{f} \in \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}'))$, $\mathtt{F}\ \mathtt{f} \in \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}))$, *and* $I(\mathtt{T}') = \mathtt{Mutable}$. *Then* $\mathtt{F} = \mathtt{F}'$.

Proof. By induction on the derivation of $\Delta \vdash \mathtt{T} \preceq \mathtt{T}'$, similarly to Lemma A.2.8 in [20]. Because $I(\mathtt{T}') = \mathtt{Mutable}$, whenever rule S1 is applied, it can never be that $\Delta \vdash \mathtt{Immutable} \preceq \mathtt{T}'_1$, thus we need to consider the same set of subtyping rules as in FGJ. □

5.3 FIGJ Typing and Reduction Rules

Fig. 13 and Fig. 14 present FIGJ typing and reduction rules. Rule T-Field-Set checks at "compile-time" that only a mutable expression can set a field, whereas R-Field-Set checks at "run-time" that only fields of a mutable object can be set. Note that only objects have an immutability at run-time, not locations.

5.4 FIGJ Type Soundness

Type preservation states that if an expression reduces to another expression, then the latter is always a subtype of the former.

Theorem 5.3. ***(Type Preservation)*** *If* $\Delta, \mathtt{S} \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e}, \mathtt{S} \rightarrow \mathtt{e}', \mathtt{S}'$, *then* $\exists \mathtt{T}'$ *such that* $\Delta \vdash \mathtt{T}' \preceq \mathtt{T}$ *and* $\Delta, \mathtt{S}' \vdash \mathtt{e}' : \mathtt{T}'$.

Proof. By induction on the derivation of $\mathtt{e}, \mathtt{S} \rightarrow \mathtt{e}', \mathtt{S}'$, similarly to the proof of Theorem 3.4.1 in [20], which uses Lemma A.2.8 on page 436 for the case of a field access. Lemma A.2.8 states that the type of a field does not change in a subclass, but the proof is still valid if the type of a field changes covariantly, as we have proven in Lem. 5.1. Our proof also needs to consider field assignment in R-Field-Set, for which we use Lem. 5.2 which showed that fields that are assigned are no-variant. □

$$\frac{\Delta \vdash \mathtt{T}_1 \preceq \mathtt{T}_2 \quad \Delta \vdash \mathtt{T}_2 \preceq \mathtt{T}_3}{\Delta \vdash \mathtt{T}_1 \preceq \mathtt{T}_3} \qquad \frac{}{\Delta \vdash \mathtt{X} \preceq \Delta(\mathtt{X})} \qquad \frac{\mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\rangle \triangleleft \mathtt{N}\ \{\ \ldots\}}{\Delta \vdash \mathtt{C}\langle\overline{\mathtt{T}}\rangle \preceq [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}}$$

$$\frac{}{\Delta \vdash \mathtt{T} \preceq \mathtt{T}} \qquad \frac{}{\Delta \vdash \mathtt{Mutable} \preceq \mathtt{ReadOnly}} \qquad \frac{}{\Delta \vdash \mathtt{Immutable} \preceq \mathtt{ReadOnly}}$$

$$\frac{\forall \mathtt{T}_i \in \overline{\mathtt{T}} : \quad \mathtt{T}_i = \mathtt{T}'_i \text{ or } (\Delta \vdash \mathtt{Immutable} \preceq \mathtt{T}'_1 \text{ and } \Delta \vdash \mathtt{T}_i \preceq \mathtt{T}'_i \text{ and } \mathit{CoVariant}(\mathtt{X}_i, \mathtt{C}\langle\overline{\mathtt{X}}\rangle))}{\Delta \vdash \mathtt{C}\langle\overline{\mathtt{T}}\rangle \preceq \mathtt{C}\langle\overline{\mathtt{T}'}\rangle} \quad (\text{S1})$$

Figure 12: FIGJ Subtyping Rules.

The progress theorem shows that FIGJ programs don't get "stuck" and any closed well-typed FIGJ expression can be reduced to some location or contains a failed downcast.

THEOREM 5.4. ***(Progress)*** *Suppose* $\mathtt{e}$ *is a closed well-typed expression. Then, either* $\mathtt{e}$ *is a location, or it contains a failed downcast, or there is an applicable reduction rule for* $\mathtt{e}$.

PROOF. Using a case by case analysis of all possible expression types in Fig. 13. The only change from the proof in FGJ is the use of T-FIELD-SET to prove that $I(\mathtt{l}) = \mathtt{Mutable}$ in R-FIELD-SET, and thus we never get stuck due to that rule. $\square$

Thm. 5.5 is a formalization of property (1) from Sec. 2.4.

THEOREM 5.5. *Let* $\Delta, \mathtt{S} \vdash \mathtt{e} : \mathtt{T}$, *and* $\mathtt{e}, \mathtt{S} \rightarrow^* \mathtt{l}, \mathtt{S}'$, *where* $\mathtt{S}'[\mathtt{l}] = \mathtt{N}(\overline{\mathtt{l}})$. *Then* $\Delta \vdash I(\mathtt{N}) \preceq I(\mathtt{T})$.

PROOF. From Thm. 5.3, $\Delta \vdash \mathtt{N} \preceq \mathtt{T}$. Thus $\Delta \vdash I(\mathtt{N}) \preceq I(\mathtt{T})$. $\square$

6. Conclusion

This paper presented *Immutability Generic Java* (IGJ), a design for adding reference and object immutability on top of the existing generic mechanism in Java. IGJ satisfies the design principles in Sec. 1: transitivity, static, polymorphism, and simplicity. IGJ provides transitive immutability to protect the entire abstract state, but a user can exclude fields from the abstract state. IGJ is purely static, backward compatible, and the resulting code can run on any JVM without runtime penalty. IGJ achieves a high degree of polymorphism using generics and safe covariant subtyping. Finally, IGJ does not change Java's syntax, and has a small number of typing rules.

Acknowledgments. This work was funded in part by DARPA contracts FA8750-06-2-0189 and HR0011-07-1-0023.

References

[1] S. Artzi, M. D. Ernst, A. Kieżun, C. Pacheco, and J. H. Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *M-TOOS*, Oct. 2006.

[2] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *OOPSLA*, pages 35–49, Oct. 2004.

[3] C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT Dept. of EECS, Feb. 2004.

[4] J. Boyland. Why we should not add `readonly` to Java (yet). In *FTfJP*, July 2005.

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

[6] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA*, pages 183–200, Oct. 1998.

[7] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, June 2005.

[8] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, pages 292–310, Oct. 2002.

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

[10] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *WODA*, pages 17–24, May 2006.

[11] R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP*, pages 465–490, June 2004.

[12] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *J. Object Tech.*, 4(8):5–32, Oct. 2005.

[13] J. J. Dolado, M. Harman, M. C. Otero, and L. Hu. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE TSE*, 29(7):665–670, July 2003.

[14] M. D. Ernst. Annotations on Java types: JSR 308 working document. `http://pag.csail.mit.edu/jsr308/`, Nov. 12, 2007.

[15] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, Feb. 2001.

[16] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA*, pages 302–312, Nov. 2003.

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

[18] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, Boston, MA, third edition, 2005.

[19] S. S. Huang, D. Zook, and Y. Smaragdakis. cJ: Enhancing Java with safe type conditions. In *AOSD*, pages 185–198, Mar. 2007.

[20] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, May 2001.

[21] A. Igarashi and M. Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM TOPLAS*, 28(5):795–847, 2006.

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

[23] Y. Lu and J. Potter. On ownership and accessibility. In *ECOOP*, pages 99–123, July 2006.

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

[25] I. Pechtchanski and V. Sarkar. Immutability specification and its applications. In *Java Grande*, pages 202–211, Nov. 2002.

[26] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[27] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic Java. In *OOPSLA*, pages 311–324, Oct. 2006.

[28] A. Sălcianu. *Pointer analysis for Java programs: Novel techniques and applications*. PhD thesis, MIT Dept. of EECS, Sep. 2006.

[29] M. Skoglund and T. Wrigstad. A mode system for read-only references in Java. In *FTfJP*, June 2001.

[30] O. Tkachuk and M. B. Dwyer. Adapting side effects analysis for modular program model checking. In *ESEC/FSE*, pages 188–197, Sep. 2003.

[31] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, Oct. 2005.

[32] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *ECOOP*, pages 380–403, July 2006.