Type-based Object Immutability with Flexible Initialization

Christian Haack    Erik Poll

Radboud University, Nijmegen

IPA Herfstdagen on Software Analysis in Nunspeet on November 24-28, 2008

http://mobius.inria.fr
Outline

1. Introduction

2. Explaining the Type System
Kinds of Immutability

Class immutability.

An immutable class is a class whose instances cannot be modified.

Object immutability.

An object is immutable if its state cannot be modified.

Reference immutability.

A reference is immutable if the state of the object it refers to cannot be modified through this reference.
Kinds of Immutability

Class immutability.

An immutable class is a class whose instances cannot be modified.

- **Object initialization:** necessarily tied to constructor.

Object immutability.

An object is immutable if its state cannot be modified.

- **Object initialization:** may extend beyond constructor.

Reference immutability.

A reference is immutable if the state of the object it refers to cannot be modified through this reference.
Kinds of Immutability

Class immutability.

An immutable class is a class whose instances cannot be modified.

- **Object initialization**: necessarily tied to constructor.
- **Closed world**: assumes that clients of immutable classes follow the rules of the pluggable type system.
- **Open world**: assumes that clients only follow Java’s standard typing rules.

Object immutability.

An object is immutable if its state cannot be modified.

- **Object initialization**: may extend beyond constructor.

Reference immutability.

A reference is immutable if the state of the object it refers to cannot be modified through this reference.
Design Goals

- Design goal: support the various kinds of immutability
- Design goal: simplicity and directness
  - Want to make do with the core annotations `@Rd`, `@RdWr`, `@Any`, plus qualifier polymorphism.
- Design goal: compatibility with JSR 308
  - a version with explicit ghost commands that indicate the end of the initialization phase
  - a ghost command inference algorithm (in progress)
Previous Related Work

- **Jimuva** by Haack, Poll, Schäfer, Schubert (ESOP'07)
  - class immutability
  - object immutability without flexible initialization
  - support for immutable objects with mutable rep objects
    - no reference immutability
    - no flexible initialization
    - ownership and effect annotations needed

- **IGJ** by Zibin, Potanin, Ali, Artzi, Kiezun, Ernst (ESEC/FSE’07)
  - object immutability without flexible initialization
  - reference immutability
  - no ownership and effect annotations needed
    - no flexible initialization
    - no support for immutable objects with mutable rep objects
      (String can’t be checked)
Previous Related Work (cont.)

- **Frozen Objects** by Leino, Müller, Wallenburg (VSTTE’08)
  - object immutability with late initialization (freezing)
  - not type-based but based on the Boogie programming methodology
  - can encode class immutability, but assuming closed world

- **SafeJava** by Boyapatti (Ph.D. thesis 2004)

- **Joe3** by Ostlund, Wrigstad, Clarke, Akerblom (IWACO’07),
  - object immutability with late initialization through unique references
  - **Joe3** supports all kinds of immutability, but builds on an expressive ownership type system (violating our design goal of directly supporting immutability without additional attributes)
The flexible initialization technique employs a ghost command for dynamically generating new names that can occur in types. Similar to:

- region creation command in region-based memory management (Tofte/Talpin)
Outline

1. Introduction

2. Explaining the Type System
Type Qualifiers: RdWr and Rd

Qualifiers.

\[
q \ ::= \\
    \text{RdWr} \quad \text{read-write access (default)} \\
    \text{Rd} \quad \text{read-only access}
\]

Types.

\[
T \ ::= \ q \ C
\]

If an object has type \text{Rd} \ C then its fields may only be read.
Type Qualifiers: RdWr and Rd

Qualifiers.

\[ q ::= \]

\[ \text{RdWr} \text{ read-write access (default)} \]

\[ \text{Rd} \text{ read-only access} \]

Types.

\[ T ::= q\ C \]

If an object has type \( \text{Rd}\ C \) then its fields may only be read.

```java
class C { int f; }

static void m(Rd C x) {
    x.f = 42; // TYPE ERROR
}

static void m(RdWr C x) {
    x.f = 42; // OK
}
```

Soundness.

Well-typed programs never write to \( \text{Rd}\)-objects.
Type Qualifiers: Any for Reference Immutability

Qualifiers.

\[ q ::= \cdots \]

\[ \text{Any} \quad \text{“the referred object is either } \text{Rd} \text{ or } \text{RdWr”} \]

Subqualifying.

\[ \text{Rd <: Any} \quad \text{RdWr <: Any} \]

Subtyping.

\[ \frac{p <: q \quad C <: D}{p \ C <: q \ D} \]

Writes through \text{Any}-references are prohibited.
Type Qualifiers: Any for Reference Immutability

Qualifiers.

\[ q ::= \ldots \]

\[ \text{Any} \quad \text{“the referred object is either \text{Rd} or \text{RdWr}”} \]

Subqualifying.

\[ \text{Rd} <: \text{Any} \quad \text{RdWr} <: \text{Any} \]

Subtyping.

\[
\begin{array}{c}
\frac{p <: q \quad C <: D}{p \; C <: q \; D}
\end{array}
\]

Writes through \textit{Any}-references are prohibited.

interface Util {
    void foo(int \text{Any} []; x);
}

static void m(Util util) {
    int[] a = new int \text{RdWr} []; \{42,43,44\};
    util.foo(a);
    assert a[0] == 42;
}

Due to IGJ.
The Access Qualifier is a Class Parameter

- Classes have a special class parameter `MyAccess`.
- `MyAccess` refers to the access qualifier for `this`.

```java
class Point {
    int x;
    int y;
}

class Square {
    MyAccess Point upperleft;
    MyAccess Point lowerright;
}

static void m(Rd Square s) {
    s.upperleft = s.lowerright; // TYPE ERROR
    s.upperleft.x = 42; // TYPE ERROR
}
```
The Access Qualifier is a Class Parameter

- A covariant class parameter? Is that sound?
- Yes it is, because writeable qualifiers are minimal.
- So whenever one upcasts a qualifier, the associated reference becomes immutable.
Flexible Initialization

Initialization Token.

\[ n \in \text{Token} \quad \text{token for initializing a set of related objects} \]

Qualifiers.

\[ q ::= \ldots \]

\[
\text{Fresh}(n) \quad \text{fresh object under initialization}
\]

**Fresh** objects are writeable, even if they later turn immutable.
Flexible Initialization

 Initialization Token.

 \[ n \in \text{Token} \quad \text{token for initializing a set of related objects} \]

 Qualifiers.

 \[ q ::= \ldots \]

 \[ \text{Fresh}(n) \quad \text{fresh object under initialization} \]

 \text{Fresh} objects are writeable, even if they later turn immutable.

 Ghost commands.

 \[ \text{newtoken } n \quad \text{create a new initialization token} \]

 \[ \text{commit Fresh}(n) \text{ as } q \quad \text{globally convert } \text{Fresh}(n) \text{ to } q \]

 These have no runtime effect.
Flexible Initialization

Initialization Token.

\[ n \in \text{Token} \] token for initializing a set of related objects

Qualifiers.

\[ q ::= \ldots \]

\[ \text{Fresh}(n) \] fresh object under initialization

\text{Fresh} objects are writeable, even if they later turn immutable.

Ghost commands.

\begin{align*}
\text{newtoken } n & \quad \text{create a new initialization token} \\
\text{commit } \text{Fresh}(n) \text{ as } q & \quad \text{globally convert } \text{Fresh}(n) \text{ to } q
\end{align*}

These have no runtime effect.

```c
static char Rd [] copy (char RdWr [] w) {
    newtoken n;
    char[] r = new char Fresh(n) [w.length];
    for (int i=0; i++; i < w.length) r[i] = w[i];
    commit Fresh(n) as Rd;
    return r;
}
```
Flexible Initialization: Soundness of Commit

Only the method that creates an initialization token can refer to it.

- Classes cannot have field types with \texttt{Fresh(n)}-qualifiers.

```java
class C {
    Fresh(n) D x; // TYPE ERROR: n out of scope
}
```

- Only the method that generated \texttt{n} can commit \texttt{Fresh(n)}.

```java
static Rd C commit(Fresh(n) C x) { // TYPE ERROR: n out of scope
    commit Fresh(n) as Rd;
    return x;
}
```
Flexible Initialization: the Heap Topology

(picture slightly inaccurate for Generics)

- **RdWr-object**
- **Rd-object**
- **Any-object**

initialized

- references inside regions allowed (not shown)

**Fresh(blue)**

**Fresh(red)**
Flexible Initialization: the Heap Topology

(picture slightly inaccurate for Generics)

- RdWr-object
- Rd-object
- Any-object

- references inside regions allowed (not shown)
- incoming references into initialized region allowed
Flexible Initialization: the Heap Topology

(picture slightly inaccurate for Generics)

- references inside regions allowed (not shown)
- incoming references into initialized region allowed
- incoming references into Fresh-regions disallowed
Flexible Initialization: Committing a Fresh Region

- RdWr-object
- Rd-object
- Any-object

Initialized

Fresh(blue)

Fresh(red)
Flexible Initialization: Committing a Fresh Region

- RdWr-object
- Rd-object
- Any-object

- Fresh(red)
- Fresh(blue)

Initialized

Stack

initialized

red region turns black

colors of references on stack
Flexible Initialization: Committing a Fresh Region

- RdWr-object
- Rd-object
- Any-object

Initialized

Stack

- Top
- Rest

Fresh(blue)

Fresh(red)

- Colors of references on stack must be adjusted.
- Red region turns black.
Flexible Initialization: Committing a Fresh Region

Initialized

RdWr-object
Rd-object
Any-object

commit Fresh(red) as Rd

top
rest

Stack

Fresh(red)

Fresh(blue)

colors of references on stack must be adjusted

red region turns black
Flexible Initialization: Committing a Fresh Region

- RdWr-object
- Rd-object
- Any-object

Initialized

red region turns black

commit Fresh(red) as Rd

red region turns black

Stack

Christian Haack, Erik Poll

Type-based Object Immutability
Flexible Initialization: Committing a Fresh Region

committed Fresh(red) as Rd

red region turns black

Initialized

Stack
Flexible Initialization: Committing a Fresh Region

- **RdWr-object**
- **Rd-object**
- **Any-object**

Initialized

red region turns black

colors of references on stack must be adjusted

Stack

commit Fresh(red) as Rd

Christian Haack, Erik Poll

Type-based Object Immutability
Flexible Initialization: Committing a Fresh Region

Initialized

Commit Fresh(red) as Rd

- Red region turns black
- Colors of references on stack must be adjusted

Stack
Flexible Initialization: Committing a Fresh Region

- RdWr-object
- Rd-object
- Any-object

Initialized

commit Fresh(red) as Rd

red region turns black

colors of references on stack must be adjusted

Stack

Christian Haack, Erik Poll
static void copy(Point src, Point dst) {
    dst.x = src.x; dst.y = src.y;
}

It should be allowed to pass actual $dst$-parameters of types $RdWr\ Point$ and $Fresh(n)\ Point$.

- This method is similar to $arraycopy()$.
- $arraycopy()$ is called in constructors of immutable $Strings$. 
Qualifier Polymorphism for Methods (cont.)

\[
\text{Any} \\
\text{Rd} \quad \text{RdWr} \quad \text{Fresh}(n) \quad \text{Fresh}(m) \quad \cdots
\]
Qualifier Polymorphism for Methods (cont.)

Writeable can only be used as a bound, not as a type.
Writeable can only be used as a bound, not as a type.

Typing rule for field sets (incomplete sketch).

\[
\frac{x : q \ C \quad q \text{ extends } \text{Writeable}}{x.f = v : \text{ok}}
\]

static <a, b extends Writeable> void copy(a Point src, b Point dst) {
    dst.x = src.x; dst.y = src.y;
}
Why can Writeable not be a Type?

This would lead to unsoundness for two reasons:

- fields of type `Writeable` could refer to `Fresh(n)`-objects.  
  ⇒ unsoundness of `commit`

- `Writeable` would be a non-minimal qualifier that allows writes.  
  ⇒ unsoundness of covariance for `MyAccess` class parameter
More on Qualifier-Polymorphic Methods

- **static `<a>` void foo(int `a` [] x)**
  - does not write to object `x` through reference `x`
  - does not write object `x` to the heap

- **static void faa(int `Any` [] x)**
  - does not write to object `x` through reference `x`
  - may write object `x` to the heap (into `Any`-fields)

- **static `<a extends Writeable>` void fee(int `a` [] x)**
  - may write to object `x` through reference `x`
  - does not write object `x` to the heap
Receiver Qualifiers (supported by JSR 308)

- JSR 308 provides a slot for annotations on the receiver type.
- **Inspectors** can be called on any receivers.
- **Mutators** can only be called on *Writable* receivers.

```java
class Hashtable<K,V> {
    <a> V get(K key) a { ... }
    <a extends Writeable> put(K key, V value) a { ... }
}
```

```java
newtoken n;
Hashtable<String,String> t = <Fresh(n)>Hashtable<String,String>();
t.put("Wojtek", "Nijmegen");
t.put("Vladimir", "Koblenz");
commit Fresh(n) as Rd;
t.get("Wojtek"); // OK
t.put("Daniel", "Madrid"); // TYPE ERROR
```

Christian Haack, Erik Poll
Type-based Object Immutability 21
Constructors

Constructors are special and must have one of the following forms:

1. \(<\tilde{a} \text{ extends } \tilde{B}> q C(\tilde{T} \tilde{x}) p \{ \text{body} \} \)

   Typically:
   
   \(<a \text{ extends } \text{Writeable}> a C(\tilde{T} \tilde{x}) a \{ \text{body} \} \)
   
   - Caller commits.
   - Useful for immutability with late initialization.

2. \(<\tilde{a} \text{ extends } \tilde{B}> q C(\tilde{T} \tilde{x}) \{ \text{newtoken} n; \text{body} \} \)

   - Constructor commits.
   - Useful for immutability at the end (or in the middle) of constructor.

It is disallowed to call constructors of the second form using \textit{super()}. The second form is therefore only recommended in \textit{final} classes.
Class Immutability (Open World)

A class annotation: Immutable

```java
Immutable final class String {
    private final char MyAccess [] arr; ...
}
```

Rules.

- immutable classes must be final and direct subclasses of `Object`
- methods and constructors may only call static or final methods (transitively)
- all fields must be private
- constructors must have the following form
  ```java
  Rd C(\bar{T} \bar{x}) \{ newtoken n; ...; commit Fresh(n) as Rd; \}
  ```
  where `MyAccess` does not occur in \( \bar{T} \).
- types of public methods must have the following form:
  ```java
  <a> U m(\bar{T} \bar{x}) a \{ ... \}
  ```
  where `MyAccess` and `a` do not occur in \( U, \bar{T} \).
We must ensure that thread-shared objects are initialized (i.e., not Fresh).

Why? A thread can commit Fresh(n) without other threads knowing about it.

class Thread {
    void run() RdWr { }
    void start(); // Treated specially. The type system uses run()'s type.
}

- Note that subclasses of Thread may override run() with receiver type RdWr or Any (by contravariance).
- Because the thread object is forced to be initialized when the thread is started, so are all objects reachable from it.
- For generics: all actual class parameters of subclasses of Thread must extend Any Object.
Qualifier-parametrized Classes

Types.

\[ T ::= q \ C \langle \bar{p} \rangle \]

Subtyping.

\[
\begin{align*}
q & <: r & C <: D \\
\hline
q \ C \langle \bar{p} \rangle & <: r \ D \langle \bar{p} \rangle
\end{align*}
\]

Types are invariant in the angle-bracketed class parameters.

Restriction:

- In field types, MyAccess must not occur inside angle brackets.

```java
class C {
    Rd C<MyAccess> x; // FORBIDDEN

    // In full Java with generics:
    Rd List<MyAccess> Object> y; // FORBIDDEN
}
```
Qualifier-parametrized Classes: Example

class ListIterator<collection_access, E> implements Iterator<collection_access, E>
{
    collection_access Node<E> current, prev, pprev;

    RdWr Iterator(collection_access Node<E> head) RdWr {
        this.current = head;
    }

    <a> E next() RdWr<a> { ... writes this, reads current ... }

    <a extends Writeable> E remove() RdWr<a> {
        ... writes this, reads current, writes prev ...
    }
}

An iterator client:

interface Util {
    void foo(RdWr Iterator<Any,E> it);
}

foo(it) does not mutate the underlying collection through it.
Conclusion

Summary.

- Design of a pluggable type system supporting various kinds of immutability.

- The system does not need ownership types, unique references, or effect annotations.

- Qualifier polymorphism is very expressive (both for flexibility and for security/confinement).
Conclusion

Summary.

- Design of a pluggable type system supporting various kinds of immutability.
- The system does not need ownership types, unique references, or effect annotations.
- Qualifier polymorphism is very expressive (both for flexibility and for security/confinement).

Done.

- Formalization of the type system for a small model language.
- Soundness proof: Well-typed programs do not write to $\text{Rd}$-objects.
- Inference algorithm that infers all annotations inside method bodies, including ghost statements. (Proofs in progress.)

To do.

- Implementation on top of the JSR 308 checker framework.