Type-based Object Immutability with Flexible Initialization

Christian Haack[1,2] Erik Poll[1]

[1]Radboud University, Nijmegen, The Netherlands

[2]aicas GmbH, Karlsruhe, Germany

ECOOP 2009 in Genoa, July 5–10

http://mobius.inria.fr

What is this Talk About?

- A pluggable type system ...
- ... for statically checking various immutability properties ...
- ... in Java-like languages.

Kinds of Immutability

Object immutability.

An object is immutable if its state cannot be modified.

Class immutability.

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

- 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.

Read-only references.

A reference is read-only if the state of the object it refers to cannot be modified through this reference.

Object Immutability and Object Initialization

Immutable objects mutate during their initialization phase!

- Tying object initialization to object constructors ...
- ... is neither sufficient ...
  - E.g., immutable arrays,
  - immutable collections implemented in terms of mutable collection APIs,
  - immutable cyclic structures.
- ... nor does it simplify the type analysis.
  - Java imposes no restrictions on constructor bodies.

Design Goals

- support the various kinds of immutability
- support object initialization outside constructors
- self-containedness (we do not want to build on top of other more general systems)
- compatibility with JSR 308 annotations
  - a version with explicit ghost commands that indicate the end of object initialization phases
  - an inference algorithm that infers the end of object initialization phases

Type Qualifiers: RdWr and Rd

Type Qualifiers.

q ::=

`RdWr`	read-write access (default) (aka. `@Mutable`)
`Rd`	read-only access (aka. `@Immutable`)

Type Qualifiers: RdWr and Rd

Type Qualifiers.

q ::=

`RdWr`	read-write access (default) (aka. `@Mutable`)
`Rd`	read-only access (aka. `@Immutable`)

Types.

$$T ::= q\ C$$

If an object has type `Rd` C then its fields may only be read.

Type Qualifiers: RdWr and Rd

Type Qualifiers.

q ::=

`RdWr`	read-write access (default) (aka. `@Mutable`)
`Rd`	read-only access (aka. `@Immutable`)

Types.

$$T ::= q\ C$$

If an object has type `Rd` C then its fields may only be read.

```
class C { int f; }

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

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

Type Qualifiers: RdWr and Rd

Type Qualifiers.

q ::=

RdWr	read-write access (default) (aka. @Mutable)
Rd	read-only access (aka. @Immutable)

Types.

$$T \ ::= \ q\ C$$

If an object has type Rd C then its fields may only be read.

```
class C { int f; }

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

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

Soundness.

Well-typed programs never write to Rd-objects.

Type Qualifiers: Any (aka @Readonly-Reference)

Type Qualifiers.

$q ::= \cdots$

Any "the referred object is either Rd or RdWr"

Subqualifying.

Rd <: Any   RdWr <: Any

Subtyping.

$$\frac{p <: q \qquad C <: D}{p\ C <: q\ D}$$

Writes through Any-references are prohibited.

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

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

The Access Right is a Class Parameter

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

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

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

static void m(Rd Square s) {
    s.upperleft = new Point();      // TYPE ERROR
    s.upperleft.x = 42;             // TYPE ERROR
}
```

The Initialization Discipline

Initialization Token.

$n \in$ Token token for initializing a set of related objects

Ghost command (has no effect at runtime).

`newtoken` n create a new initialization token

The Initialization Discipline

Initialization Token.

$n \in$ Token   token for initializing a set of related objects

Ghost command (has no effect at runtime).

`newtoken` n create a new initialization token

Type Qualifier.

`Fresh(`n`)`   fresh object under initialization

- Typical use: `newtoken n; new Fresh(n)` C`(); new Fresh(n)` D`(); ...`
- `Fresh` objects are writeable (even if they later turn immutable).

The Initialization Discipline

Initialization Token.

$n \in$ Token — token for initializing a set of related objects

Ghost command (has no effect at runtime).

`newtoken` n — create a new initialization token

Type Qualifier.

`Fresh(`n`)` — fresh object under initialization

- Typical use: `newtoken n; new Fresh(n)` C`(); new Fresh(n)` D`(); ...`
- `Fresh` objects are writeable (even if they later turn immutable).

Ghost command (has no effect at runtime).

`commit Fresh(`n`) as` q — globally convert `Fresh(`n`)` to q

The Initialization Discipline

Initialization Token.

$n \in$ Token token for initializing a set of related objects

Ghost command (has no effect at runtime).

`newtoken` n create a new initialization token

Type Qualifier.

`Fresh(`n`)` fresh object under initialization

- Typical use: `newtoken n; new Fresh(n)` C`(); new Fresh(n)` D`(); ...`
- `Fresh` objects are writeable (even if they later turn immutable).

Ghost command (has no effect at runtime).

`commit Fresh(`n`) as` q globally convert `Fresh(`n`)` to q

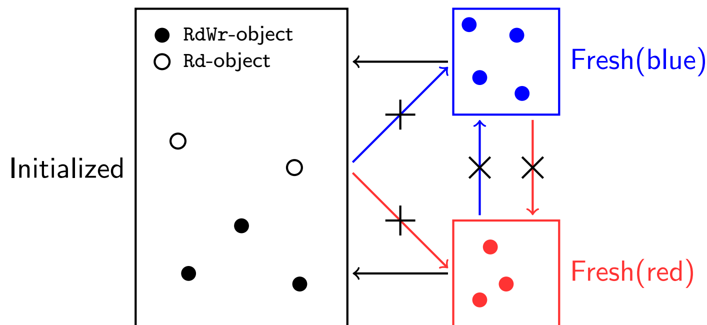
```
static char Rd [] copy (char RdWr [] w) {
    newtoken n;
    char[] r = new char Fresh(n) [w.length];
    for (int i=0; i++; i < w.length) r[i] = w[i];
    commit Fresh(n) as Rd;
    return r;
}
```

Soundness of Commit (Scoping of Init-Tokens)

- Fields cannot have Fresh(n)-qualifiers.

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

Soundness of Commit (The Heap Invariant)



Heap Invariant.

There are no ingoing references into `Fresh`-regions.

N.B.: references inside `Fresh` regions are possible, by `MyAccess` qualifier on fields.

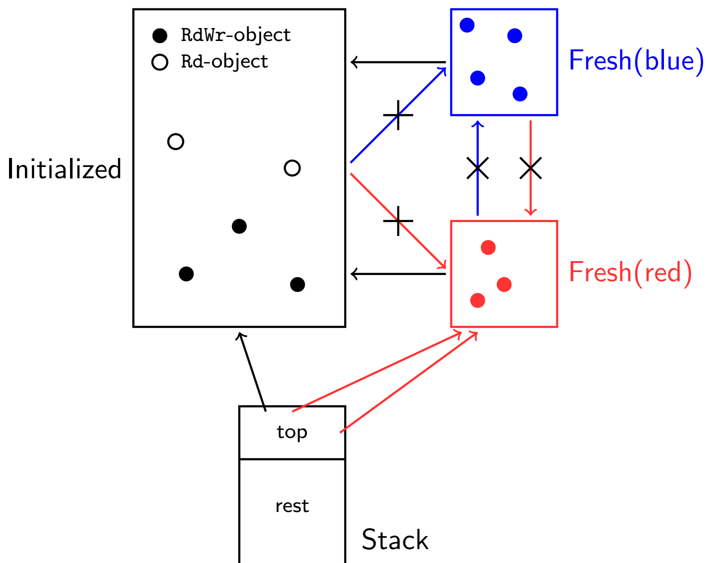
Soundness of Commit (Meth. Confinement of Init-Tokens)

- Each initialization token is to confined to a single method.

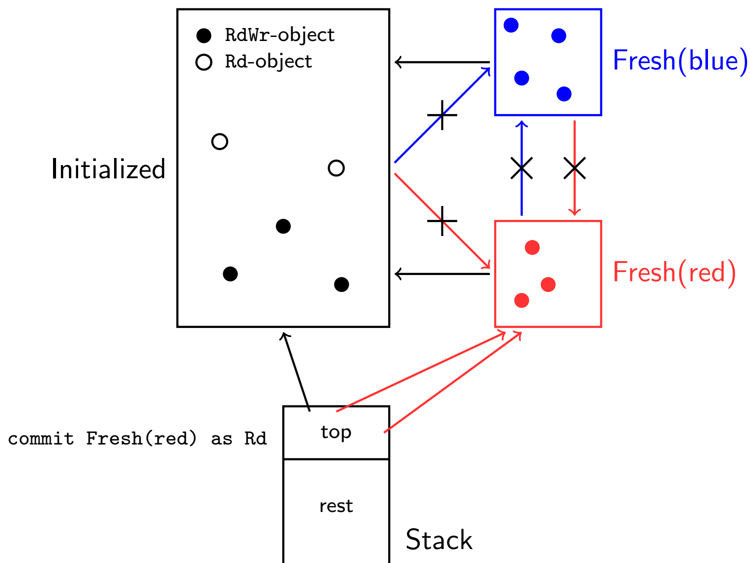
```
Rd C commit(Fresh(n) C x) { // TYPE ERROR: n out of scope
    commit Fresh(n) as Rd;
    return x;
}
```

- So, only the method that generates an initialization token has the right to commit the associated fresh region.

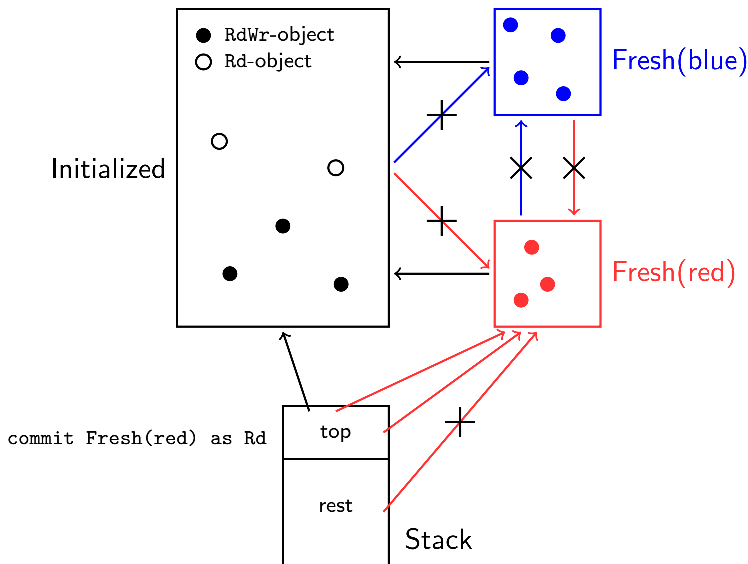
Soundness of Commit (The Big Picture)



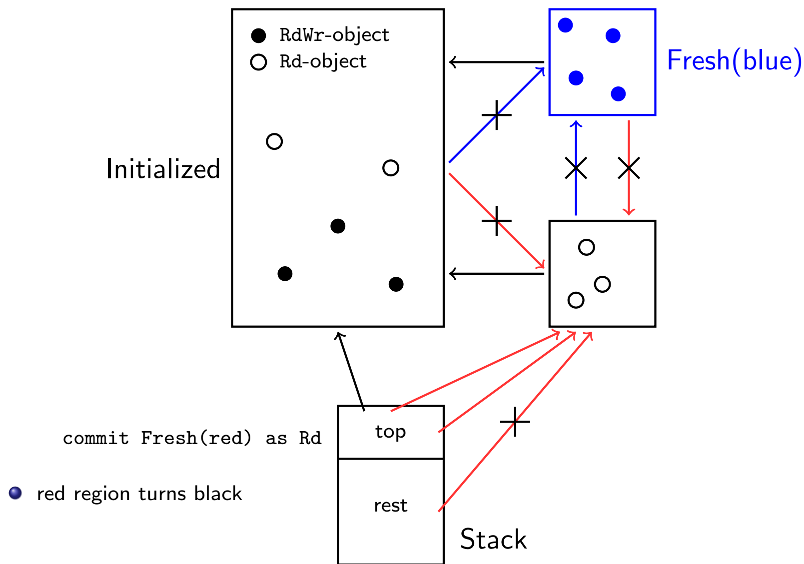
Soundness of Commit (The Big Picture)



Soundness of Commit (The Big Picture)

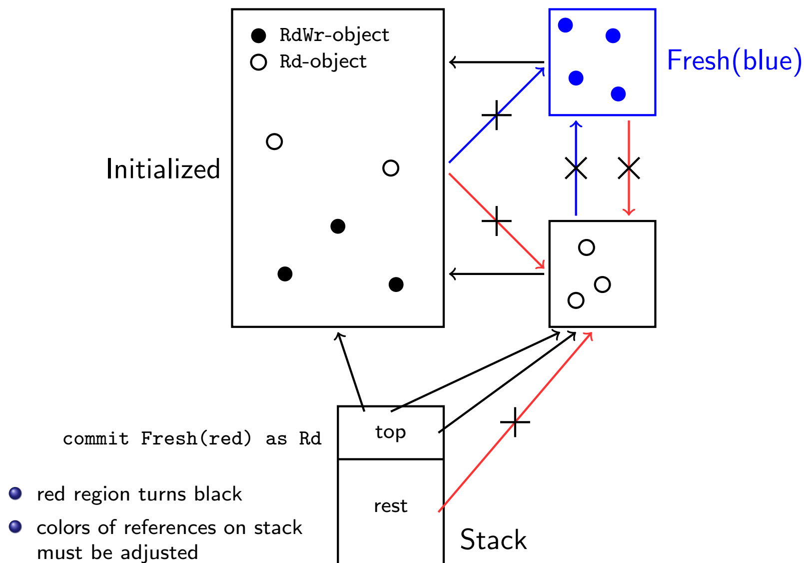


Soundness of Commit (The Big Picture)



- red region turns black

Soundness of Commit (The Big Picture)



- red region turns black
- colors of references on stack must be adjusted

Qualifier Polymorphism for Methods

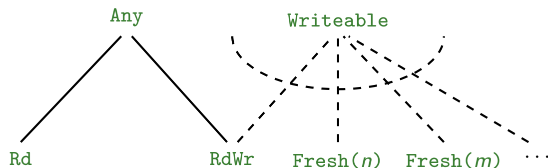
```
static void copy(Point src, Point dst) {
    dst.x = src.x;
    dst.y = src.y;
}
```

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 `String`s.

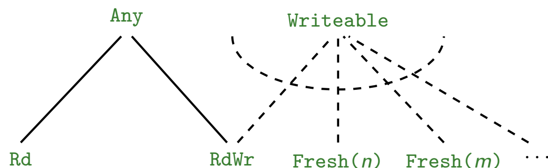
Qualifier Polymorphism for Methods (cont.)

Qualifier Polymorphism for Methods (cont.)



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

Qualifier Polymorphism for Methods (cont.)



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

Typing rule for field sets (incomplete sketch).

$$\frac{x : q\ C \qquad q \text{ extends } \texttt{Writeable}}{x.f = v : \text{ok}}$$

```
static <a, b extends Writeable> void copy(a Point src, b Point dst) {
    dst.x = src.x;
    dst.y = src.y;
}
```

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 `Writeable` receivers.

```
class Hashtable<K,V> {

    <a> V get(K key) a { ... }                                   // INSPECTOR

    <a extends Writeable> V put(K key, V value) a { ... }   // MUTATOR
}
```

```
newtoken n;
Hashtable<String,String> t = new <Fresh(n)> Hashtable<String,String>();
t.put("Alice", "Amsterdam");
t.put("Bob", "Berlin");
commit Fresh(n) as Rd;
t.get("Alice"); // OK
t.put("Charlie", "Chicago"); // TYPE ERROR
```

Constructors

Constructors must have one of the following forms:

1. $<\bar{a}\ \texttt{extends}\ \bar{B}>\ q\ C(\bar{T}\ \bar{x})\ p\ \{\ \mathit{body}\ \}$

   Typically:

   $<a\ \texttt{extends}\ \texttt{Writeable}>\ a\ C(\bar{T}\ \bar{x})\ a\ \{\ \mathit{body}\ \}$

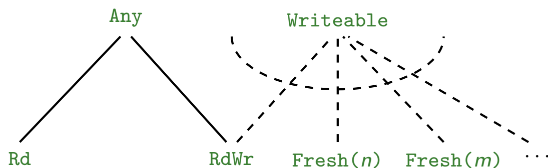
   - Caller commits.
   - Better choice most of the time.

2. $<\bar{a}\ \texttt{extends}\ \bar{B}>\ q\ C(\bar{T}\ \bar{x})\ \{\ \texttt{newtoken n;}\ \mathit{body}\ \}$

   - Constructor commits.
   - Useful for class immutability in an open world.

It is disallowed to call constructors of the second form using `super()`. The second form is therefore most appropriate for `final` classes.

Qualifier-Polymorphic Methods and Confinement



- `static <a> void foo(int a [] x)`
  - does not write to object x through reference x
  - does not create a reference to object x on the heap
- `static void faa(int Any [] x)`
  - does not write to object x through reference x
  - may create a reference to object x on the heap
- `static <a extends Writeable> void fee(int a [] x)`
  - may write to object x through reference x
  - does not create a reference to object x on the heap

Class Immutability (Open World)

```
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 or final
- types of public methods must have the following form:

  `<a>` U `m(`$\bar{T}\ \bar{x}$`) a { ... }`

  where `MyAccess` and `a` do not occur in U.
- constructors must have the following form

  `Rd C(`$\bar{T}\ \bar{x}$`) { newtoken n; ...; commit Fresh(n) as Rd; }`

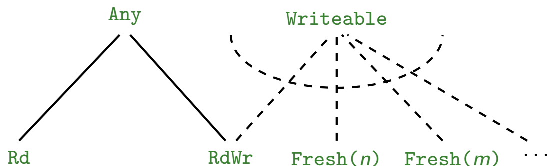
  where `MyAccess` does not occur in $\bar{T}$.

Threads

- We enforce that objects are committed before they get thread-shared.

```
class Thread {

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



Local Annotation Inference

- An algorithm that infers all annotations inside method and constructor bodies.
- In particular: all ghost statements are inferred.
- The algorithm is syntax-directed.
- Basic idea: insert `commit`-statements lazily whenever this is required to satisfy annotations on fields, methods and constructors.

Contributions

A pluggable type system for immutability:

- Object immutability, class immutability and read-only references.
- Simple and direct.
  - Annotation language is small: only `Rd`, `RdWr`, `Any` and qualifier polymorphism. (`Fresh` can always be inferred.)
  - Only annotations on types are needed. (No effect annotations, constraints, pre/postconditions, loop invariants, or the like.)
- Flexible object initialization (not tied to constructors).
  - Without the need for complex aliasing annotations or destructive reads, as in systems that support typestate changes through unique references.