

Immutable Objects for a Java-like Language

Christian Haack Erik Poll Jan Schäfer Aleksy Schubert

Radboud University of Nijmegen

University of Kaiserslautern

University of Warsaw

March 29, 2007



<http://mobius.inria.fr>



What are Immutable Objects?

- An **object is immutable** if its state appears to be constant during the entire program execution.
- A **class is immutable** if all its instances are immutable in every execution of every program.

Immutable Objects — Why are They Useful?

“Immutable objects are simple.”

(Joshua Bloch in *Effective Java*.)

- Maintaining object invariants of immutable objects is unproblematic.
- Sharing immutable objects among concurrent threads is unproblematic. Immutable objects are thread-safe.
- Immutable objects are secure in the presence of untrusted components.
- Methods on immutable objects are state-independent. Useful for program verification.
- More?

What Have We Done?

- Designed a static type system for specifying and verifying object immutability.
- Central keyword: a class modifier `immutable`.
- Our system supports multiple common programming patterns.
- We have shown that the system is sound in an `open world with legal subclassing`.

Open World with Legal Subclassing

Our type system guarantees object immutability even in the presence of untrusted clients ... as long as they subclass legally.

Open World

- Immutable classes and those that they depend on must be checked by our immutability type system ...
- ... but clients of immutable classes need not be. These need only be checked by Java's standard type system.
- Some techniques that don't work in an open world:
 - Reference immutability/read-only references (e.g., Javari).
 - Method signatures that constrain clients.

Legal Subclassing

- Untrusted clients must not subclass **immutable** classes.
- We could drop this constraint if we required that all immutable classes are **final**. But this seems too restrictive.
- It would be nice if Java had a more flexible way of prohibiting untrusted components to subclass critical classes.

An Immutable Class with Shallow Object State

```
immutable class Integer {  
    private final int val;  
    Integer (int val) { this.val = val; }  
    int get () { return val; }      // an inspector method  
}
```

- We say that an object has **shallow state** if its state consists of its fields only.
- An **Integer** is immutable because its state is **shallow** and its fields are **final**.
- Another reason: An **Integer** is immutable because its fields are **private** and its methods are **inspectors**.

Another Immutable Class with Shallow Object State

```
immutable interface List { ... }  
  
immutable class ConsList implements IntList {  
    private final Object head;  
    private final List tail;  
  
    ConsList (Object head, List tail) {  
        this.head = head; this.tail = tail; }  
  
    // ... inspector methods ...  
}
```

- A `ConsList` is immutable, because its state is `shallow` and its fields are `final`.
- A `List` is immutable, because all its implementations are.

Immutable Objects with Shallow States are Simple

- For shallow states, Java's `final field declarations` are enough.
- Alternatively, one can enforce `private fields` and `inspector methods`.
- Our type system uses the latter technique, because it generalizes to immutable classes with deep states.

An Immutable Object with a Mutable Subcomponent

```
immutable class String {  
    private final MutableCharList cs;  
  
    String (MutableCharList cs) { this.cs = cs.deepcopy(); }  
  
    // ... methods do not mutate cs ...  
}
```

- The entire state of the character list `cs` is part of `this`'s state.
- Technical jargon: `cs` is a **rep-object** of `this`.
- Final fields alone do not suffice to enforce immutability of `String`.

Preventing Rep-Exposure

```
immutable class String {  
    private final MutableCharList cs;  
  
    String (MutableCharList cs) {  
        this.cs = cs.deepcopy(); // this is safe  
        // this.cs = cs;         this is unsafe in an open world  
    }  
}
```

- Our type system must enforce encapsulation of rep-objects.

Ownership Types

- Our type system for immutability builds on top of a simple ownership type system.
- Ownership types provide a simple language for specifying the depth of encapsulated object state.
- The ownership type system statically checks that objects properly encapsulate their state (i.e., it checks absence of rep-exposure).

Ownership Types Specify the Depth of Object States

```
class C/*<myowner>*/ {  
    private D<this> x;      //object x is owned by this ("rep")  
    private D<myowner> y;  //obj. y owned by this's owner ("peer")  
    private D<world> z;    //object z is ownerless  
}
```

- `myowner` is an implicit class parameter that refers to the owner of `this`.
- “ p is owned by o ” \triangleq “ p is o 's rep-object”
- `world` is a special constant.

Object Creation

Object Creation

```
... new C<x>() ...
```

- Creates a new object owned by x .
- x can be `this`, `world`, `myowner` or an owner parameter of an owner-polymorphic method.

How do Ownership Type Systems Prevent Rep-Exposure?

- By using a restrictive enough subtyping judgment.

Subtyping for Ownership Types

$$\frac{C <: D}{C\langle x \rangle <: D\langle x \rangle}$$

- So we have: $C\langle x \rangle <: D\langle y \rangle \Leftrightarrow x = y$.
- The subtyping judgment disallows flows of objects owned by o into locations for objects with a different owner o' .

Ownership Types Prevent Rep-Exposure

```
class C {  
    private D<this> d;  
  
    C (D<world> d) {                // D<world> ↯: D<this>  
        this.d = new D<this>(d.x); // This type-checks.  
        // this.d = d;             Type Error!  
    }  
}  
  
class D {    public int x;    D (int x) { this.x = x; } }
```

- **Restriction:** Method and constructor signatures must not mention types of the form `C<this>`.
- Needed for **soundness in an open world**.

But What About the String Example?

```
immutable class String {  
    private final MutableCharList<this> cs;  
  
    String (MutableCharList<world> cs) {  
        this.cs = cs.deepcopy();    // deepcopy()'s return type??  
    }  
    ...  
}
```

- What would be `deepcopy()`'s return type?
- We need `owner-polymorphic methods`.

But What About the String Example?

```
class MutableCharList/*<myowner>*/ {
  <x> MutableCharList<x> deepcopy() {
    Node<myowner> in = this.hd;
    ... while (in != null) { ... new Node<x>(in.val) ... } ...
    return new MutableCharList<x> (...)
  }
}

immutable class String {
  private final MutableCharList<this> cs;

  String (MutableCharList<world> cs) {
    this.cs = cs.deepcopy<this>();
  }
}
```

Owner-polymorphic Methods

But owner-polymorphic methods allow exposing rep-objects:

```
public static <x> void m (D<x> y) { ... }  
immutable class C {  
    private D<this> reobject;  
    ... m<this>(reobject) ...  
}
```

That's dangerous! Yes, that's true. But ...

- ... while our type system **allows constructors** of immutable objects to instantiate the owner parameter `x` by `this` ...
- ... it **prevents methods** of immutable objects from instantiating `x` by `this`.
- So after the constructor of an immutable object has terminated, clients can't see its rep-objects anymore.

Owner-polymorphic Methods

```
immutable class C {  
  C() { ... m<this>(reproject) ... }  
}
```

But m could create a static alias to `reproject`!

- No. Fortunately, this is impossible by scoping.

```
public static <x> void m (D<x> y) {  
  ...  
  No heap location outside  $y$ 's own state has a supertype of  $D<x>$ .  
  ...  
}
```

Read-only Methods

Requirement on immutable classes.

Methods must be *read-only*.

Semantic goal for read-only expressions.

Read-only expressions must not mutate the state of *immutable* receivers.

Static rules.

An *expression* is *read-only* if:

- It contains no field assignments.
- Method calls have the form $e.m\langle\bar{v}\rangle(\bar{e})$ where either
 - m is *read-only*, or
 - e has a type of the form $C\langle\text{world}\rangle$ and all \bar{v} are *world*.
- Constructor calls have the form $\text{new } C\langle\text{world}\rangle(\bar{e})$.

Read-only Methods vs. Pure Methods

- Read-only methods need not be pure (aka side-effect-free).

A read-only method that is not pure.

```
getChars (int scrBegin, int srcEnc, char[]⟨world⟩ dst, int dstBegin)
```

- This non-pure method is implemented by Java's immutable `String` class.

Write-local Constructors

Requirement on immutable classes.

Constructors must be *write-local*.

Semantic goal for *write-local* expressions.

Write-local expressions must not write to the state of immutable objects except to the state of *this*.

Static rules.

An expression is *write-local* if:

- In field assignments $e.f = e'$:
 - $e = \text{this}$ or e has type $C\langle\text{this}\rangle$.
- In method calls $e.m\langle\bar{v}\rangle(\bar{e})$:
 - m is *read-only*
 - or $e = \text{this}$ and m is *write-local*
 - or e has type $C\langle\text{this}\rangle$
 - or e has type $C\langle\text{world}\rangle$.

Anonymous Constructors

Requirement on immutable classes.

Constructors must be *anonymous*.

Semantic goal for anonymous expressions.

Anonymous expressions must not leak *this*.

Static rules.

An expression is *anonymous* if:

- It is not *this*.
- It does not assign *this* to fields.
- It does not pass *this* to methods.
- In method calls $e.m\langle\bar{v}\rangle(\bar{e})$ either m or e is *anonymous*.

These rules are taken from Vitek and Bokowski's confinement type systems.

Formalization

- We have formalized our type system for a core language in the style of [Featherweight Java \(FJ\)](#).
- Our extensions to [FJ](#):
 - A mutable heap.
 - A more faithful model of Java's object constructors.
 - Protected fields.

Formalization: Visible States

Visible States

We call the following kinds of states **visible states** for object o :

- Pre-states for method calls with receiver o and caller distinct from o .
- Pre-states for reads of o 's fields with reader distinct from o .
- The **anonymity** requirement for constructors ensures that an **immutable** object does not become visible until its constructor has terminated.

Formalization: Semantics of Immutability

Immutability in a Fixed Program

Class C is immutable in program P whenever the following holds:

If $P \rightarrow^* (h_1, s_1) \rightarrow^* (h_2, s_2)$,
and (h_1, s_1) and (h_2, s_2) are visible states for o ,
and $\text{class}(o) <: C$, then $\text{state}(h_1)(o) = \text{state}(h_2)(o)$.

Immutability in an Open World

$C \in \bar{C}$ is immutable in \bar{C} whenever it is immutable in all Java-programs $P = (\bar{C} \cup \bar{D}; \text{main})$ that legally subclass \bar{C} .

- \bar{D} and main are *unchecked components*. They follow Java's standard typing rules, but need not follow the rules of our immutability type system.

Formalization: Type Soundness

Theorem (Type Soundness)

In a well-typed class table, every class that is declared immutable is semantically immutable.

- Type soundness is a corollary of a type preservation theorem.

Sharing Mutable Representation Objects

- Sometimes immutable objects share mutable representation objects.
- For instance, Java's immutable `BigInteger` class:
 - A `BigInteger` has a sign field and a field for the absolute value, which is implemented as an encapsulated integer array.
 - A `BigInteger` and its negation share the integer array.
- An extension of our type system supports sharing of mutable rep-objects.
- We have to be careful to prevent an immutable object during its construction phase from writing into a shared rep-object.
- To control this, we introduce `read-only objects`.
- `Read-only objects` may have mutator methods but clients may not call these.

Read-only Objects

Object Access Permissions

ar	$::=$	$rdwr$	read-write access
		rd	read access only (“read-only objects”)
		$myaccess$	implicit class parameter
ty	$::=$	$C\langle ar, x \rangle$	object type

- Only **read-only** methods may be called on **rd**-restricted receivers.
- In context **world**, access constraints are ignored.

$$\frac{}{\Gamma \vdash C\langle ar, world \rangle <: C\langle ar', world \rangle}$$

- Immutable objects may share **rd**-restricted rep-objects.

$$\frac{\Gamma \vdash x, y : D, D' \quad \text{immutable} \in \text{atts}(D) \cap \text{atts}(D')}{\Gamma \vdash C\langle rd, x \rangle <: C\langle rd, y \rangle}$$

Conclusion

Summary.

- We have put together a type system for statically checking object immutability
 - using ownership types to ensure encapsulation.
 - controlling write-effects on top of the ownership types.
- We have formalized this system for a core language and proven that it is sound in an open world with legal subclassing.

Future Work.

- Implement a prototype immutability checker for Java.
- Automatic inference of auxiliary specs:
 - read-only, write-local, anonymous.
 - Ownership annotations on local variables.
 - Type inference for owner-polymorphic methods?
- It is often important that immutable objects do not read outside their own state.
 - Combine with ideas of systems for controlling read-effects (e.g. Clarke and Drossopolous)

Acknowledgements

Mobius: Mobility, Ubiquity and Security

Mobius is funded from 2005–2009 as project IST-015905 under the Global Computing II proactive initiative of the Future and Emerging Technologies objective in the Information Society Technologies priority of the European Commission's 6th Framework programme.

Disclaimer: This document reflects only the author's views and the European Community is not liable for any use that may be made of the information contained therein.