

Immutable Objects for A Java-like Language

Christian Haack (Nijmegen)

Erik Poll (Nijmegen)

Jan Schäfer (Kaiserslautern)

Aleksy Schubert (Nijmegen/Warsaw)

Object immutability

What's that?

- 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.

Object immutability

What's that?

- 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.

What have we done?

- Designed a *static type system* for specifying and verifying object immutability.
- Sound in an *open world with legal subclassing*.

Object immutability

What's that?

- 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.

What have we done?

- Designed a *static type system* for specifying and verifying object immutability.
- Sound in an *open world with legal subclassing*.

Central keyword:

- A class modifier `immutable`.

Why is immutability useful?

“Immutable objects are simple.”

(Joshua Bloch in *Effective Java*.)

- *Sharing* immutable objects is *unproblematic*.
- Immutable objects are *secure* in the presence of *untrusted components*.
- Immutable objects are *thread safe*. No synchronization needed.
- Methods on immutable objects are *state-independent*. Useful for program verification.
- More?

Semantics

C is immutable in program *P* if 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 $\text{state}(h_1)(o) = \text{state}(h_2)(o)$.

Semantics

C is *immutable in program P* if 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 $\text{state}(h_1)(o) = \text{state}(h_2)(o)$.

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

Semantics

C is *immutable in program P* if 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 $\text{state}(h_1)(o) = \text{state}(h_2)(o)$.

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

- \bar{D} and main are unchecked (“untrusted”) components, i.e., they do not follow the rules of our immutability type system. The only constraint is that they do not extend or override immutability-annotated classes or methods

Semantics

C is *immutable in program P* if 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 $\text{state}(h_1)(o) = \text{state}(h_2)(o)$.

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

- \bar{D} and *main* are unchecked (“untrusted”) components, i.e., they do not follow the rules of our immutability type system. The only constraint is that they do not extend or override immutability-annotated classes or methods
- For this reason, we call this *Immutability in an Open World with Legal Subclassing*.

Semantics

C is *immutable in program P* if 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 $\text{state}(h_1)(o) = \text{state}(h_2)(o)$.

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

- \bar{D} and *main* are unchecked (“untrusted”) components, i.e., they do not follow the rules of our immutability type system. The only constraint is that they do not extend or override immutability-annotated classes or methods
- For this reason, we call this *Immutability in an Open World with Legal Subclassing*.

Soundness of Type System:

In a well-typed class table, every class that is declared `immutable` is in fact immutable.

The simple case: shallow state

- The object state consists of the fields only.

```
immutable class Int {  
    private final int value;  
    Int(int i) { value = i; }  
    int get() { return value; }  
}
```

- This class is immutable because:
 - it has no mutator methods
- Its fields have primitive types

The simple case: shallow state

- One can build up complex data structures just from immutable objects with shallow state.

```
immutable class Node {  
    private final int value;  
    private final Node next;  
    Node(int i, Node n) { value=i; next=n; }  
    getVal() { return value; }  
    getNext() { return next; }  
    Node cons(int i) { return new Node(i,this); }  
}
```

- This class is immutable because:
 - it has no mutator methods
- Its fields have primitive *or immutable* types

The complex case: deep state

- Some immutable objects have deep states.

```
immutable class String {  
    private final rep char[] a;  
    ...  
}
```

- The object `this.a` is part of a `String`'s local state.
- `String` is immutable because:
 - it has no mutator methods
 - *subobject* `this.a` is *encapsulated*

Specifying deep object states

```
class C {  
    D<this> x;        // x owned by this  
    D<myowner> y;    // y owned by this's owner  
    D<world> z;      // z ownerless    }  
class D {    Object<myowner> x;    }
```

- `myowner` is a special variable (like `this`)
- `world` is a constant

Specifying deep object states

```
class C {  
    D<this> x;        // x owned by this  
    D<myowner> y;    // y owned by this's owner  
    D<world> z;      // z ownerless    }  
class D {    Object<myowner> x;    }
```

- `myowner` is a special variable (like `this`)
- `world` is a constant
- The object state of `o` includes objects owned by `o`.

Specifying deep object states

```
class C {  
    D<this> x;        // x owned by this  
    D<myowner> y;    // y owned by this's owner  
    D<world> z;      // z ownerless    }  
class D {    Object<myowner> x;    }
```

- `myowner` is a special variable (like `this`)
- `world` is a constant
- The object state of `o` includes objects owned by `o`.

Example: Suppose `o` has type `C<world>`.

- Objects `o.x` and `o.x.x` are part of `o`'s state.
- Objects `o.y` and `o.z` are *not* part of `o`'s state.

Object Creation

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

- creates a new object owned by `x`
- `x` instantiates occurrences of `myowner` in `C`
- `x` can be `this`, `world` or `myowner`

Rep and peer as syntax sugar

$$\begin{aligned} \text{rep } C &\triangleq C\langle\text{this}\rangle \\ \text{peer } C &\triangleq C\langle\text{myowner}\rangle \\ C &\triangleq C\langle\text{world}\rangle \end{aligned}$$

... and also:

$$\text{myowner variable} \approx \text{JML's owner ghost field}$$

Ownership types prevent rep exposure

Subtyping: $C\langle x \rangle <: D\langle y \rangle$ iff $C <: D$ and $x = y$

Ownership types prevent rep exposure

Subtyping: $C\langle x \rangle <: D\langle y \rangle$ iff $C <: D$ and $x = y$

```
class C {  
    Mtb<this> val;  
    C(Mtb<world> x) {  
        val = x; //type error, Mtb<world>  $\not<: Mtb<this>$  }  
}
```

Ownership types prevent rep exposure

Subtyping: $C\langle x \rangle <: D\langle y \rangle$ iff $C <: D$ and $x = y$

```
class C {  
    Mtb<this> val;  
    C(Mtb<world> x) {  
        val = x; //type error, Mtb<world>  $\not<: Mtb<this>$  }  
    }  
}
```

But that's ok:

```
class C {  
    Mtb<this> val;  
    C(Mtb<world> x) {  
        val = new Mtb<this>(x.get()); }  
    }  
}
```

Ownership types prevent rep exposure

Subtyping: $C\langle x \rangle <: D\langle y \rangle$ iff $C <: D$ and $x = y$

```
class C {  
    Mtb<this> val;  
    C(Mtb<world> x) {  
        val = x; //type error, Mtb<world>  $\not<: Mtb<this>$  }  
    }  
}
```

But that's ok:

```
class C {  
    Mtb<this> val;  
    C(Mtb<world> x) {  
        val = new Mtb<this>(x.get()); }  
    }  
}
```

Rule: Method type signatures must not mention `this`.

Read-only Methods

Methods of `immutable` classes must be `readonly`.

Intent: must not mutate own state

Read-only Methods

Methods of `immutable` classes must be `readonly`.

Intent: must not mutate own state

Static rules: An expression is `readonly`, if:

- it contains no field assignments
- all its method calls have the form $e.m(\bar{e})$ where either
 - m is `readonly`, or
 - e has a type of the form $C\langle\text{world}\rangle$
- `new`-calls have the form `new C<world>.(\bar{e})`

Read-only Methods

Methods of `immutable` classes must be `readonly`.

Intent: must not mutate own state

Static rules: An expression is `readonly`, if:

- it contains no field assignments
- all its method calls have the form $e.m(\bar{e})$ where either
 - m is `readonly`, or
 - e has a type of the form $C\langle\text{world}\rangle$
- `new`-calls have the form `new C<world>.(\bar{e})`

Example from `String`:

```
getChars(int srcBegin, int srcEnd,  
         char[]<world> dst, int dstBegin)
```

is `readonly` but not `pure`.

Write-local Constructors

Constructors of `immutable` objects must be `wrlocal`.

Intent: constructors must not write to other immutable objects of the same class

Write-local Constructors

Constructors of `immutable` objects must be `wrlocal`.

Intent: constructors must not write to other immutable objects of the same class

Static rules: An expression is `wrlocal`, if:

- field assignments $e.f=e'$:
 - $e = \text{this}$ or e has type $C<\text{this}>$
- method calls $e.m(\bar{e})$:
 - either m `rdonly`, or
 - m `wrlocal` and $e = \text{this}$, or
 - m `wrlocal` and e has type $C<\text{this}>$, or
 - e has type $C<\text{world}>$

Anonymous Constructors

Constructors of `immutable` objects must be `anon`.

Intent: constructors must not leak `this`

Anonymous Constructors

Constructors of `immutable` objects must be `anon`.

Intent: constructors must not leak `this`

Static rules: An expression is `anon`, if:

- it is not `this`
- it does not assign `this` to fields
- it does not pass `this` to methods
- it only calls a methods `m` on `this` if `m` is `anon`

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

An Example with Annotations

```
class MutableInt {  
    private int value;  
    MutableInt(int i) { value=i; }  
    readonly int get() { return value; }  
    void set(int i) { value=i; }  
}
```

```
immutable class EncapsulatedMutable {  
    private final MutableInt<this> m;  
    anon wrlocal EncapsulatedMutable(MutableInt m) {  
        this.m = new MutableInt<this>(m.get()); }  
    readonly int get(){ return m.get(); }  
}
```

But what about this?

```
class Util {  
    static void copy(MutableInt from, MutableInt to) {  
        to.set(from.get()); }  
}
```

```
immutable class EncapsulatedMutable {  
    private final MutableInt<this> m;  
    anon wrlocal EncapsulatedMutable(Mutable m) {  
        this.m = new MutableInt<this>(null);  
        Util.copy(m, this.m); // type error }  
    rdonly int get(){ return m.get(); }  
}
```

- this is just like `String(char[] a)`
- which uses `System.arraycopy()`

Our solution: owner-polymorphic methods

```
...  
<x,y> void copy(MtblInt<x> from, MtblInt<y> to)  
...  
Util.copy<world,this>(m,this.m) // this type-checks!  
...
```

Our solution: owner-polymorphic methods

```
...  
<x,y> void copy(MtblInt<x> from, MtblInt<y> to)  
...  
Util.copy<world,this>(m,this.m) // this type-checks!  
...
```

But why is it safe??

- The polymorphic type of `copy` guarantees that its implementation does not create a static alias to `to` from outside `to`'s state. (A *Theorem for Free!*)

Our solution: owner-polymorphic methods

```
...  
<x,y> void copy(MtblInt<x> from, MtblInt<y> to)  
...  
Util.copy<world,this>(m,this.m) // this type-checks!  
...
```

But why is it safe??

- The polymorphic type of `copy` guarantees that its implementation does not create a static alias to `to` from outside `to`'s state. (A *Theorem for Free!*)
- We need a caller side restriction to tame dynamic aliasing:
 - `rdonly`-expressions may only instantiate method's owner parameters by `world`.

And how about this?

```
immutable class BigInteger {
    private int signum;
    private int[]<this> mag;
    ...
    private anon wrlocal
    BigInteger(int[]<this> mag, int signum) {
        this.signum = -signum; this.mag = mag;    }

    rdonly BigInteger negate() {
        return new BigInteger(mag, signum); // type error
    }
}
```

- **mag** has type `int[]<this>`
- ... the constructor requires `int[]<o>`, where `o` is new

A partial solution: readonly references

<i>ar</i>	::=	<i>rdwr</i>	read-write access
		<i>rd</i>	read access only
		<i>myaccess</i>	special variable (like <i>this</i> , <i>myowner</i>)
<i>ty</i>	::=	<i>C</i> <i><ar, x></i>	object type

A partial solution: readonly references

ar ::= $rdwr$ read-write access
| rd read access only
| $myaccess$ special variable (like $this$, $myowner$)
 ty ::= $C\langle ar, x \rangle$ object type

Subtyping:

- In context $world$, access constraints are ignored.

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

A partial solution: readonly references

ar ::= $rdwr$ read-write access
| rd read access only
| $myaccess$ special variable (like `this`, `myowner`)
 ty ::= $C\langle ar, x \rangle$ object type

Subtyping:

- 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.

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

$$C\langle rdwr, x \rangle <: C\langle rd, y \rangle \quad \text{this is unsound}$$

Sharing representation objects

```
immutable class BigInteger {
  private int signum;
  private int[]<rd,this> mag;
  ...
  private anon wrlocal
  BigInteger(int[]<rd,this> mag, int signum) {
    this.signum = -signum; this.mag = mag;
  }

  rdonly BigInteger negate() {
    return new BigInteger(mag,signum); // ok now
  }
}
```

- **mag** has type `int[]<rd,this>`
- ... constructor requires `int[]<rd,o>`, where `o` is new
- ... and these two types are equivalent.

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

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
- It is subtle to put the pieces of the puzzle together in the right way.

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
- It is subtle to put the pieces of the puzzle together in the right way.
- But it is possible.

Future Work

- *Implement an immutability checker* for Java based on this system.
 - Lots of decisions to be made: static variables, private methods, ...

Future Work

- *Implement an immutability checker* for Java based on this system.
 - Lots of decisions to be made: static variables, private methods, ...
- Automatic *inference of auxiliary specs.*
 - `rdonly`, `wrlocal`, `anon`
 - ownership annotations on local variables
 - type inference for owner-polymorphic methods?

Future Work

- *Implement an immutability checker* for Java based on this system.
 - Lots of decisions to be made: static variables, private methods, ...
- Automatic *inference of auxiliary specs.*
 - `rdonly`, `wrlocal`, `anon`
 - 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 a system for controlling *read-effects* (e.g. Clarke and Drossopolous)