A New Viewpoint on Mutating Methods that Preserves Reference Immutability

Jaime Quinonez
MIT Program Analysis Group
December 5, 2006

Reference Immutability

- `@ReadOnly` on a type specifies a reference that cannot be used to modify an object
- `@ReadOnly` can annotate any use of a type
- For a type `T`, `@ReadOnly T` is a supertype of `T`
  - `T` can be used anywhere `@ReadOnly T` is expected
  - `@ReadOnly T` cannot be used where `T` is expected
- Unannotated `T` is `@Mutable` by default

Example mutable class

- `setTime()` mutates object
- `getTime()` does not mutate object

```
public class Date {
        private long time;

        public Date(long t) {
                this.time = time;
        }
        public long getTime() {
                return time;
        }
        public void setTime(long time) {
                this.time = time;
        }
}
```


@ReadOnly receiver annotates method

- `getTime()` does not mutate object
- receiver of `getTime()` is `@ReadOnly`

```
public class Date {
        private long time;

        public Date(long t) {
                this.time = time;
        }
        public long getTime() @ReadOnly {
                return time;
        }
        public void setTime(long time) {
                this.time = time;
        }
}
```


@Mutable receiver can also annotate method

- `setTime()` does mutate object
- receiver of `setTime()` is `@Mutable`

```
public class Date {
        private long time;

        public Date(long t) {
                this.time = time;
        }
        public long getTime() @ReadOnly {
                return time;
        }
        public void setTime(long time) @Mutable {
                this.time = time;
        }
}
```


@ReadOnly on generic types

`ArrayList<@ReadOnly Date> list;`

- `list` is a mutable `ArrayList`
  - Contains immutable references to `Dates`
- The list can be mutated
  - `list.get(0)` - legal
  - `list.clear()` - legal
- Elements in the list cannot be mutated
  - Return type of `list.get()` is `@ReadOnly Date`
  - `list.get(0).getTime()` - legal
  - `list.get(0).setTime(2)` - **illegal**
  - `@ReadOnly Date d = list.get(0)` - legal
  - `@Mutable Date d = list.get(0)` - **illegal**

@ReadOnly doesn't propagate to generics

```
@ReadOnly ArrayList<Date> list;
```

- `list` is an immutable `ArrayList`
  - Contains mutable references to `Dates`
- The list cannot be mutated
  - `list.get(0)` - legal
  - `list.clear()` - **illegal**
- Elements in the list can be mutated
  - Return type of `list.get()` is `@Mutable Date`
  - `list.get(0).getTime()` - legal
  - `list.get(0).setTime(2)` - legal
  - `@ReadOnly Date d = list.get(0)` - legal
  - `@Mutable Date d = list.get(0)` - legal

Fields are @ThisMutable by default

- Mutability of a field is the same as the mutability of receiver `this`

```
public class Cell {
        private @ThisMutable Date d;

        public void read() @ReadOnly {
                // type of this.d is @ReadOnly Date
        }
        public void write() @Mutable {
                // type of this.d is @Mutable Date
        }
}
```


Problem: returning fields requires overloading

- `Date d` is the abstract state of a `Cell`

```
public class Cell {
        private @ThisMutable Date d;

        // protects abstract state from modification
        public @ReadOnly Date getDate() @ReadOnly {
                return d;
        }

        // exposes abstract state to be modified
        public @Mutable Date getDate() @Mutable {
                return d;
        }
}
```


Problem: mutability not present runtime

- Java is a statically-typed language
- Mutability annotations checked at compile time, then discarded

```
public class Cell {
        private @ThisMutable Date d;

        // protects abstract state from modification
        public Date getDate() {
                return d;
        }

        // exposes abstract state to be modified
        public Date getDate() {
                return d;
        }
}
```

- **Error**: `getDate()` methods have identical signatures

C++ allows overloading

- Mutability of return type needs to match mutability of receiver
- C++ approach
  - Use keyword `const` to create overloaded methods
  - `const Date& getDate() const;`
  - `Date& getDate();`
  - Exactly what previous overloading example tried to do
- This approach cannot be done in Java due to type-system representation
  - Similar to inability to template over generics:
    - `public void foo(List<Number> list);`
    - `Public void foo(List<String> list);`
- Entire Standard Template Library is filled with overloaded functions
  - Unnecessary code duplication
  - Increases size of files, but no change in runtime
  - Error-prone since programmers often forget to duplicate updates

IGJ uses generics to specify mutability

- Mutability of return type needs to match mutability of receiver
- IGJ approach
  - Every type is generic, last parameter can be ReadOnly or Mutable

```
public class Cell<T> {
   private Date<T> d;
   public @ReadOnly Date<T> getDate() { return d; }
}

Cell<ReadOnly> cell;
cell.getDate(); // returns a Date<ReadOnly>
Cell<Mutable> cell;
cell.getDate(); // returns a Date<Mutable>
```

- Breaks type system to give mutability meaning to generics
  - Casts between Cell<Mutable> and Cell<ReadOnly> are identical
  - List<T> and List<Q> not related in Java

Javari allows templating over mutability

- Mutability of return type needs to match mutability of receiver
- Javari approach
  - Use annotation `@RoMaybe` to template over mutability
  - `public @RoMaybe Date getDate() @RoMaybe { … };`
    - Simultaneously represents both possibilities:
    - `public @ReadOnly Date getDate() @ReadOnly { … };`
    - `public @Mutable Date getDate() @Mutable { … };`
- Templating eliminates code duplication
  - Only one implementation needs to be maintained
  - Not allowed to have different code in methods

Javari's interpretation of mutating methods

- Javari requires a method to be annotated `@Mutable` if calling the method might somehow lead to a modification
  - The method might modify a field
  - The method might pass some of its fields to mutable contexts
  - The method might return a mutable reference to internal data
- The type system is sound and complete
- Other interpretations can increase program understanding and expressive power of mutability annotations

Proposal: Extend this-mutability to other types

- A method is `@Mutable` if it leads to a modification the caller can't control
- `public @Mutable Cell getDate() @Mutable {return d; }`
  - Method doesn't actually modify anything
  - Given a reference `@Mutable Cell c`, no possible call to `c.getDate()` in any context will modify `c`
- Restriction is that `getDate()` returns something that can be used to modify state
  - Given `@Mutable Cell c`, it is legal for `c.getDate()` to return a `@Mutable Date`
  - Given `@ReadOnly Cell c`, it is illegal for `c.getDate()` to return a `@Mutable Date`
- All the type mutability rules can be expressed as

```
public @ThisMutable Date getDate() @ReadOnly {
   return d; }
```


Proposal: Extend this-mutability to other types

```
public @ThisMutable Date getDate() @ReadOnly {
   return d; }
```

- All necessary information to determine mutability of "this" is present at compile time

```
@ReadOnly Cell rcell;
// return type of rcell.getDate() is @ReadOnly Date
@ReadOnly Date d1 = rcell.getDate(); // legal
@Mutable  Date d2 = rcell.getDate(); // illegal

@Mutable Cell mcell;
// return type of mcell.getDate() is @Mutable Date
@ReadOnly Date d3 = mcell.getDate(); // legal
@Mutable  Date d4 = mcell.getDate(); // legal
```


Modifications to type rules

- Previous viewpoint meant type of reciever always known in containing class

```
public class Cell {
   public @ThisMutable Date d;

   public @ReadOnly Date getDate() @ReadOnly {
        // know that you are in @ReadOnly context,
        // type of this.d is @ReadOnly Date
        return d;
   }
}
```


Modifications to type rules

- If receiver annotation no longer completely defines context, code correctness can't be guaranteed solely by examining each method
- However, code can still be checked against `@ReadOnly` rules
- Code that uses `getDate()` needs to be checked for correctness

```
public class Cell {
   public @ThisMutable Date d;

   public @ThisMutable Date getDate() @ReadOnly {
        // Don't know complete context, this.d
        // might be @Mutable or @ReadOnly
        return d;
   }
}
```

Modifications to type rules

- Code that uses `Cell.getDate()` has complete `this` information and can be checked against type rules
- All rules can still be checked at compile-time
- Task of checking that return type is used properly is shifted from class defining the method to class using the method

```
@ReadOnly Cell rcell;
// return type of rcell.getDate() is @ReadOnly Date
@ReadOnly Date d1 = rcell.getDate(); // legal
@Mutable  Date d2 = rcell.getDate(); // illegal

@Mutable Cell mcell;
// return type of mcell.getDate() is @Mutable Date
@ReadOnly Date d3 = mcell.getDate(); // legal
@Mutable  Date d4 = mcell.getDate(); // legal
```

19

Individual contribution to project

- Modify immutability type inference tool
- Modify core calculus (Featherweight Generic Java) to prove soundness
  - Find all valid uses of `@ThisMutable` and ensure type rules are sound
- Perform case studies to evaluate usefulness
  - Backwards compatibility ensures existing code still technically valid
  - Software design patterns have typical implementations that might be affected
  - Existing Javari code can be rewritten
  - The JDK can be rewritten
  - Large open source projects can be rewritten

20