Writing Immutable Objects with Elegance

Erwann Wernli
Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch/

Oscar Nierstrasz
Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch/

ABSTRACT

The object paradigm is rooted in the concept that objects are *mutable*. Writing *immutable* objects is of course possible, but inelegant. We propose to reconcile object-orientation and immutability by extending the language with a special prefix that alters the usual semantics of assignments and message sends in the following way: first, the left-hand side value of the operation is cloned; second, the operation is applied on the clone; third, the alias used in the left-hand side is rebound to the clone. If the left-hand side is the pseudo-variable `self`, `self` is rebound to the clone in the control flow. While at first surprising, we argue that this approach is conceptually simple and make the code effectively more elegant.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: [Language Constructs and Features]

Keywords

immutability; object-orientation; syntax

1. INTRODUCTION

Many language constructs aim at the elimination of syntactic noise (and other forms of code bloat) that is a known impediment to software maintenance and evolution. One particular trend in object-oriented programming that gains increasing traction is the use of *immutable* objects, since they make reasoning about concurrency and aliasing easier. Unfortunately, the object paradigm is deeply rooted in the concept of *mutable* objects. As a consequence, writing immutable objects is easy but particularly inelegant—this inevitably results in high syntactic noise. We propose to reconciling object-orientation and immutability from the perspective of code elegance.

The key idea is to provide a modified assignment operator `$=` that first copies the value on the left-hand side (thus effectively mutating a clone of the object) and rebinds the alias on the left-hand side to the clone. Let us consider that `temp` is local variable. After the evaluation of `temp.f $= v`, the alias `temp` points to an updated clone of the previous object pointed by `temp`.

In a language were fields are encapsulated, the left-hand side of all assignments is `self`. Consequently, the evaluation of `self.f $= v` results in the update of a clone of the current object and the rebinding of `self` to the clone in the control flow. An imperative sequence of assignments `self.f $= v; self.g $= w; self.h $= x` will then update successive clones!

We can generalize the same idea to arbitrary message sends. Prefixing a message send with `$` means: copy the receiver first, dispatch the message, and update the alias on the left-hand side with the returned value. If fields `f,g,h` have regular setters `f=(),g=(),h=()`, the imperative sequence of message sends `self.$f=(v); self.$g=(w);self.$h=(x)` will update successive clones.

The fact that the value of `self` within a lexical scope can change might seems disturbing at first. We argue that this approach is however more natural to write immutable objects once this crucial point has been understood.

2. EXAMPLES

We now compare how the code of different classes of objects would look like with and without our extension.

2.1 Points

Let us consider a typical class `Point` written in a hypothetical dynamically-typed language. We assume that the language prohibits direct access to fields of objects and supports implicit getters `f()` and setters `f=()` for all fields. For clarity we always mention `self` explicitly.

```
class Point {
 x, y;
 // constructor omitted
 move(offsetX, offsetY) {
   return new Point (
       self.x + offsetX, self.y + offsetY
   );
 }
}
```

Listing 1: A typical `Point` class

This code is inelegant: the need to explicitly instantiate a new object obscures the intent of the method. This is syntactic noise. Also, subclasses of `Point` would fail to instantiate proper classes. This problem could however be alleviated

WXYZ '05 date, City.

with first-class classes or self types, though. It must be noted that the code would be even less elegant when expressed with explicit cloning.

```
class Point {
 x, y;
 // constructor omitted
 move(offsetX, offsetY) {
   | temp |
   temp = self.clone().
   temp.x=( self.x + offsetX ).
   temp.y=( self.y + offsetX ).
   ↑ temp.
 }
}
```

Listing 2: A `Point` class using cloning

Here is how the class `Point` is rewritten with our language extension. We assume that methods return implicitly `self` at the end.

```
class Point {
 x, y;
 // constructor omitted
 move(offsetX, offsetY) {
   self.x $= self.x + offsetX.
   self.y $= self.y + offsetY.
 }
}
```

Listing 3: The `Point` class, revisited

The code expresses its intention without any syntactic noise. The code can alternatively be written using setters `x=()` and `y=()`:

```
class Point {
 x, y;
 // constructor omitted
 move(offsetX, offsetY) {
   self.$x=( self.x + offsetX ).
   self.$y=( self.y + offsetY ).
 }
}
```

Listing 4: The `Point` class, re-revisited

2.2 Circles

Let's now consider circles. A circle has a point as its center, and a radius.

```
class Circle {
 center, radius;
 // constructor omitted
 move(offsetX, offsetY) {
   return new Circle (
       self.center.move(offsetX,offsetY), radius
   );
 }
}
```

Listing 5: A typical `Circle` class

This code is even less elegant than the previous code in Listing 1. Figuring out which object is copied is obscured due to syntactic noise. Here is how it is rewritten with our extension:

```
class Circle {
 center, radius;
 // constructor omitted
 move(offsetX, offsetY) {
   self.center $= self.center.move(offsetX, offsetY).
 }
}
```

Listing 6: The `Circle` class, revisited

The intention of the code is clear. Alternatively, it can be rewritten with setters `center=()` and `radius=()`.

```
class Circle {
 center, radius;
 // constructor omitted
 move(offsetX, offsetY) {
   self.$center=( self.center.move(offsetX, offsetY) ).
 }
}
```

Listing 7: The `Circle` class, re-revisited

We have so far invoked `$`-prefixed methods only on `self` or temporary variables `temp`. What is the effect of invoking such a method on a path `self.f` or `temp.f`? We can define it as follows:

`var.f.$method()` ≡ `var.$f( var.f.method() )`

This way, the code can be made even more compact:

```
class Circle {
 center, radius;
 // constructor omitted
 move(offsetX, offsetY) {
   self.center.$move(offsetX, offsetY).
 }
}
```

Listing 8: The `Circle` class, re-re-revisited

We believe the code above is elegant. Note that since our language prohibits accesses to fields for objects other than `self`, there is no need to generalize the definition for path of arbitrary length, *e.g.*,`var.f.g.h` . We must forbid method chaining and `$`-prefix. For instance, `var.p().q().$r()` is invalid since there is no way to update `var.p().q()` with the new value.

3. FUTURE WORK

The next step would be to implement the language feature. The Smalltalk environment seems a good candidate since the stack if reified with first-class activation records. This would enable us to rebind `self` according to our semantics. Other environments might not offer such flexibility to prototype our proposition.

Also, this preliminary work has presented the semantics of the feature informally. Another next step would be to formalize it and make sure it does not interfere in nasty ways with other language constructs. This proposition seems however reasonable at first. The only complicated part is the rebinding of `self` that must follow precise rules. The type of the new value used to rebind `self` must in particular match the previous type of `self`.

4. RELATED WORK

Controlling and reasoning about mutations and side-effects is a major challenge of object-orientation: unwanted mutations due to uncontrolled aliasing might break invariants [2]; sharing of mutable objects puts thread-safety at risk [6];

comparing mutable objects is confusing [3]. All these reasons make immutable objects, also called value objects [7], very appealing. Some argue that immutability should be the default, and mutability only used when really needed [5, 1]. The increasing use of immutable collections is a clear witness of this trend, *e.g.*, in Google's Guava or Scala. There is an increasing support to verify immutability of objects and groups of objects [4]. The problem of converting mutable objects to immutable objects has also been addressed [5]. We believe we are however the first to address the problem of syntactic elegance when immutability and object-orientation are combined.

5. CONCLUSION

We argue that the combination of immutably and object-orientation results in code with high syntactic noise. To remove this noise, we proposed a language extension that alters the semantics of assignments and message sends. Our proposition has the surprising effect that `self` in a syntactic scope might actually change at run time. Based on examples with the canonical classes `Point` and `Circle`, we argue that this surprising effect is not disturbing and that our proposition effectively improves readability and helps regain code elegance. This is a preliminary work and the next steps are to implement the actual feature and formalize its semantics to ensure it does not interfere in unexpected way with other features.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project Np. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015).

6. REFERENCES

[1] J. Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

[2] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. *SIGPLAN Not.*, 33(10):48–64, Oct. 1998.

[3] P. Grogono and M. Sakkinen. Copying and comparing: Problems and solutions. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, ECOOP '00, pages 226–250, London, UK, UK, 2000. Springer-Verlag.

[4] C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for a java-like language. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 347–362, Berlin, Heidelberg, 2007. Springer-Verlag.

[5] F. Kjolstad, D. Dig, G. Acevedo, and M. Snir. Transformation for class immutability. In *Proceeding of the 33rd International Conference on Software Engineering*, ICSE '11, pages 61–70, New York, NY, USA, 2011. ACM.

[6] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[7] D. Riehle. Value object. In *Proceedings of the 2006 conference on Pattern languages of programs*, PLoP '06, pages 30:1–30:6, New York, NY, USA, 2006. ACM.