Towards Unifying Immutability and Ownership

(Position Paper)

Paley Li, Alex Potanin
Victoria University of Wellington
{lipale|alex}@mcs.vuw.ac.nz

James Noble, Lindsay Groves
Victoria University of Wellington
{kjx|lindsay}@mcs.vuw.ac.nz

Abstract

Immutability is a highly requested feature missing from the mainstream object-oriented languages. Boyland [2] argues that immutability cannot be introduced into a language without additional aliasing-prone mechanisms such as ownership (or fractional permissions). Rather than providing support for both of these important features independently [8, 6, 1], we explore the possibility of unifying them into one. We propose three variations starting with a straight combination of Generic Ownership [12] and Generic Immutability [13] and finishing with a possibility of a Capabilities-like system [3]. We hope to explore a pathway to more general, unified concept.

1. Generic Ownership

Generic Ownership [12, 10] uses generic types supported since Java 5 to support deep ownership [5, 4] guarantees with no change to the syntax of the language. Here is an example of a list declared in an Ownership Generic Java (OGJ) language [9]:

```
class List<E extends Object,
           O extends World> {...}
```

One can see how an additional (last) type parameter is an *owner parameter* declaring the owner variable for the current list. The owner of an element is also present implicitly but does not have to be explicitly declared.

The owners form a small hierarchy with `World` at the top, extended by `Package`, `Class`, and `This` used to specify package-only, class-only, and current instance only ownership.

2. Generic Immutability

Generic Immutability [13] again uses generic types to support both reference- and object-immutability. Here is an example of a list declared in Immutability Generic Java (IGJ) language [11]:

```
class List<I extends ReadOnly,
           E extends Object> {...}
```

Here *immutability parameter* comes first and allows list to be `ReadOnly` (read-only reference), `Mutable` (default reference in Java), and `Immutable` (reference to an immutable object).

The immutabilities again form a small hierarchy with `ReadOnly` at the top, extended by `Mutable` and `Immutable`.

This allows a neat way of expressing a method parameter that only accepts read-only argument without resorting to two method declarations for the same method (as in C++ when using `const`):

```
void foo(Bar<? extends ReadOnly> b) {...}
```

Here `foo` will safely accept any type instantiation of `Bar`, be it `Bar<ReadOnly>`, `Bar<Mutable>`, or `Bar<Immutable>`.

IGJ also improves on Java by allowing limited variance depending on the immutability parameter. It is considered unsafe in Java to allow a list to be variant in its element type parameter. The covariance would allow one to have a list of strings to be treated as a list of objects (potentially trying to add an object instead of string) and contravariance would allow one to have a list of objects treated as a list of strings (potentially trying to take out an object as if though it was a string). Both are bad and Java 5 chose to prohibit variance in the mean time. However, covariant changes to the other type parameters are perfectly safe if the immutability parameter is `ReadOnly` or `Immutable`.

3. Why bother combining immutability and ownership?

Flexible Alias Protection [7] highlighted the problem with "argument dependence" where a read-only reference can be mutated by an object elsewhere. Boyland [2] highlights "observational exposure" as a reason enough not to rush into introducing immutability into the mainstream languages without additional mechanisms such as ownership or fractional permissions.

Some languages provide support for both in a semi-independent manner. For example, Joe$_3$ [8] supports ownership annotations on class declarations together with additional immutability information added using $+$ (writable), $-$ (read only), and $*$ (immutable) symbols. Universes [6] support read only references as a way to refer to objects that one does not have ownership of. Shared objects in AliasJava [1] are also similar in spirit to read only.

We hope that combining ownership and immutability will be a safer way of introducing immutability into the mainstream using a single new concept, rather than two unrelated ones. While at this stage we cannot provide the most elegant way to unify these concepts, we hope that our work will bring us closer to one.

4. Variation 1: Straight Combination of Generic Immutability and Generic Ownership

First variation of combining ownership and immutability would be to create a language that combines OGJ and IGJ by using the first type parameter as immutability parameter and the last type parameter as owner parameter, while the intermediate type parameters carry the two implicitly. Here is an example using such language:

```
class List<I extends ReadOnly,
  E extends Object, O extends World> {...}
```

No care needs to be taken of combining ownership and immutability parameters, but all care needs to be taken to make sure the type systems behind OGJ and IGJ do not contain conflicts when merged together. We have performed the exercise of merging the two type

systems this to see the feasibility of having the two features together. While this is similar in spirit to semi-orthogonal combinations, it does have the advantages of carrying both ownership and immutability implicitly. For the list example we require only three parameters instead of five (the ownership and immutability of the element are implicit inside `E` just like in Generic Ownership or Generic Immutability.

The disadvantage of this approach would be that it may not be expressive enough for the day to day programming tasks. However, given that both OGJ and IGJ were used to implement Java Collections, we hope that it is not going to be a problem.

5. Variation 2: Generic Immutability and Ownership

Second variation would be to combine immutability and ownership parameters into a single hierarchy that can be expressed as immutability parameters parametrised by owners. Here is an example in such a language:

```
class List<E extends Object,
           IO extends ReadOnly<?>> {...}
```

We are currently working on formulating the precise structure of the immutability-and-ownership parameter hierarchy.

The disadvantage of this approach would be that it is not as easily understandable by the implementer unless unification is absolutely clearly defined.

6. Variation 3: Generic Access Rights for Immutability and Ownership

Third and final variation we are proposing for discussion is to use the ideas in Capabilities for Sharing [3] to outline a small number of access rights (e.g. `Rd` for read, `Wr` for write, `Id` for a right to reference, `Im` for object immutability, `XU` for external uniqueness). There can be more basic rights including exclusive versions of the above or common combinations such as `RW` for reading and writing. Here is an example in such a language:

```
class List<E extends Object,
  O extends Rd<Wr<? extends Package>>> {...}
```

It is much more tricky to define a relationship hierarchy for these basic access rights but we are working towards one. Finally, a set of policies must be defined in terms of such rights to represent deep or shallow ownership, external and classical uniqueness, object or reference immutability, and more.

The issues here include whether to wrap the rights around owners as in the example above, or around class instantiations. For example, the list above is fixed to have all of its instances readable and writable and presumably, none of them will be able to be read only. An alternative that would declare list with the most general right and allow the types to be instantiated appropriately can look as follows:

```
class List<E extends Object,
  O extends Any> {...}
```

with the instantiation being: `List<String<Any>,Rd<This>>` for a list of any strings that is (only) readable by the current instance only.

An alternative would be to use annotations such as JSR308 or JML comments while formalising the type system using polymorphic types. This may avoid the unnecessary ordering constraints imposed by type parametrisation.

The disadvantage of this approach can be its excessive complexity, however this may be the most expressive way providing the most promising start on the path to unification.

7. Towards Unifying Immutability and Ownership

The ultimate goal of this work is to come up with a new concept unifying both ownership and immutability to be able to sell immutability for the introduction to a language with generic types as a single and coherent feature. Just as the Capabilities for Sharing work attempted originally, this may coincidentally allow a way to compare the existing varieties of ownership and immutability and other programming language improvements approaches out there right now.

We see external uniqueness as essential to our success and we hope for our resulting language to support it fully. If we allow *exclusive* rights where an object gains not just a right to *write* but to also not having anyone else *writing* at the same time, we are delving into the land of concurrency where we need to make a decision on how to grant the exclusive right to the requester and in which order. Thus, this platform may allow bringing together a lot of aspects of various fields to the ownership-centric programming language design.

We proposed three variations for unifying immutability and ownership. There can be more variations that can be discussed at the workshop. Having both immutability and ownership in the modern languages would have wide ranging benefits especially in the presence of concurrency in this age of multi-core programming.

References

[1] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. In *OOPSLA*, pages 311–330, Seattle, WA, USA, November 2002. ACM Press.

[2] John Boyland. Why we should not add readonly to Java (yet). In *(FTfJP)*, Glasgow, Scotland, July 2005.

[3] John Boyland, James Noble, and William Retert. Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. In *ECOOP*. Springer-Verlag, June 2001.

[4] Dave Clarke. *Object Ownership and Containment*. PhD thesis, School of CSE, UNSW, Australia, 2002.

[5] David Clarke, John Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA*, pages 48–64, Vancouver, Canada, October 1998. ACM Press.

[6] P. Müller and A. Poetzsh-Heffter. *Programming Languages and Fundamentals of Programming*, chapter Universes: a Type System for Controlling Representation Exposure. Fernuniversität Hagen, 1999. Poetzsh-Heffter, A. and Meyer, J. (editors).

[7] James Noble, Jan Vitek, and John Potter. Flexible Alias Protection. In Eric Jul, editor, *ECOOP*, volume 1445 of *(LNCS)*, pages 158–185. Springer-Verlag, July 1998.

[8] Johan Östlund, Tobias Wrigstad, Dave Clarke, and Beatrice Åkerblom. Ownership, uniqueness and immutability. In *International Workshop on Aliasing, Confinement, and Ownership (IWACO'07)*, 2007. Workshop at ECOOP'07.

[9] Alex Potanin. Ownership Generic Java Download. `http://www.mcs.vuw.ac.nz/~alex/ogj/`, 2005.

[10] Alex Potanin. *Generic Ownership*. PhD thesis, SMSCS, Victoria University of Wellington, 2007.

[11] Alex Potanin. Immutability Generic Java Download. `http://www.mcs.vuw.ac.nz/~alex/igj/`, 2007.

[12] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership. In *OOPSLA*, 2006.

[13] Yoav Zibin, Alex Potanin, Shay Artzi, Adam Kiezun, and Michael D. Ernst. Object and reference immutability using Java generics. In *Foundations of Software Engineering*, 2007.