Comparing Transitive to Non-transitive Object Immutability

Michael Coblenz, Joshua Sunshine, Brad Myers
School of Computer Science
Carnegie Mellon University
{mcoblenz, joshua.sunshine, bam}@cs.cmu.edu

Sam Weber, Forrest Shull
Software Engineering Institute
samweber@cert.org, fjshull@sei.cmu.edu

Abstract

Many programming languages provide features that express restrictions on which data structures can be changed. For example, C++ includes `const` and Java includes `final`. Languages that are in widespread use typically provide *non-transitive* immutability: when a reference is specified to be immutable or read-only, the object referenced can still reference mutable structures. However, some languages, particularly research languages, provide *transitive* immutability, in which immutable objects can only reference other immutable objects (with some exceptions). We are designing a lab study of programmers to elucidate the differences in programmer effectiveness between these two approaches.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.2.9 [*Software Engineering*]: Management—Productivity

Keywords Programming languages, Immutability, Mutability

1. Introduction

Many designers of APIs and programming languages recommend using immutability in order to prevent bugs and security flaws. For example, Bloch devoted a section of his book, *Effective Java*, to minimizing mutability [1]. He cited these benefits of immutability: simple state management; thread-safety; and safe and efficient sharing. Likewise, Oracle's Secure Coding Guidelines for Java SE [2] recommend minimizing mutability. There are questions, however, about what immutability means and how to express immutability in programming languages. Some languages support programmer-provided specifications of immutability or read-only restriction in order to express and enforce (either statically or dynamically) that either certain data structures cannot be changed, or that clients who obtain references to certain data structures do not have permission to change them. For example, `final` in Java expresses that a particular reference cannot be reassigned to point to a different object, but the referenced object can still change its contents. Though one can write a class with no setters and all private fields, there is no way to express immutability directly. In C++, `const` can be used to express that state cannot change, but `const` data can still refer to non-`const` data, which can then be changed. Furthermore, `const` provides no guarantees regarding other references to the same object. This means that in addition to not providing the expected benefits of immutability to programmers, such as thread safety and simple state management, these annotations also do not provide the guarantees that would be needed for compiler optimizations, which require that the *transitive state* be immutable.

Immutability annotations can express either *transitive* immutability, in which immutable objects can only reference other immutable objects, or *non-transitive* immutability: when a reference is specified to be immutable, the object referenced can still reference mutable structures, but it cannot have its contents changed directly. By providing guarantees of transitive immutability, programmers and compilers can make much stronger assumptions about state. For example, if a method is invoked on an immutable object in a loop, the compiler might be able to optimize the code by lifting the invocation out of the loop, but this optimization is unsafe if the object is not transitively immutable. Likewise, a transitively immutable object can be shared safely among threads without locks, but the same is not necessarily true of a non-transitively immutable object. In a pragmatic sense, a non-transitively immutable object is, in fact, *mutable*. Note the distinction with *read-only restrictions*, which specify that data cannot be changed through a particular reference, but say nothing about whether any other references might exist through which the data could be changed.

Some researchers have proposed language features for transitive immutability. IGJ [3] extends Java's typesystem to include statically-checked immutability annotations. IGJ

PLATEAU'15, October 26, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3907-0/15/10...$15.00
http://dx.doi.org/10.1145/2846680.2846688

supports parametric immutability, in which immutability annotations can be specified to match the containing object's immutability annotation. For example, a programmer can specify that a field in an object should have the same immutability annotation as the object itself. We call this *ad hoc transitive immutability*: ad hoc in the sense that transitivity only occurs when the programmer specifies that it should, and only to the specified extent. Another approach is taken by Pechtchanski and Sarkar's system, which splits the analysis into static and dynamic stages in order to ensure safety of dynamically-loaded code [5]. In contrast, Microsoft's Freezable class defines an interface for objects that can have a state in which they are immutable, and verification happens only at runtime [6]. Enforcement of Freezable is implementation-dependent, since the author of a Freezable class must add calls to specific APIs before and after modifying state of Freezable objects, and whether the immutability is transitive depends on the locations of those calls.

Despite the benefits of transitive immutability, these guarantees may come at increased cost as programmers must annotate and make immutable larger parts of programs. In addition, *using* APIs that are transitively immutable may require users to bear additional cost. For example, Stylos and Clarke found users prefer and are more effective at instantiating *mutable* classes [7]. One would expect that transitive immutability would result in more immutable objects, making the disadvantages more pervasive. Though some authors, such as Zibin et al. [3] evaluated their work by using their annotation systems on large systems, we have not found any work that uses programmers other than the designers of the systems to compare immutability approaches.

Most systems in widespread use today, such as Java and C++, do not support any kind of transitivity. One exception is Rust [8], which makes mutability (as opposed to immutability) explicit and has an ownership system that restricts which mutable references can exist. However, because of the integration of immutability with the ownership system, it is difficult to evaluate transitive immutability as an independent design decision in Rust.

2. Evaluating Transitivity and Non-transitivity

This overall lack of adoption in practice in contrast to the widespread discussion in the literature motivates us to ask: how useful and usable is transitive immutability? Is it so burdensome despite the guarantees it provides that it is infeasible to use, or are there some cases in which its strong guarantees are worth the cost? In what situations do annotations that enforce transitive immutability benefit programmers more than those that enforce non-transitive immutability, and how should transitive immutability annotation systems be designed? Are there mitigations, either in the language or tooling, that can increase the usability of immutability?

We have started investigating the design of transitive immutability annotation systems by prototyping a modification of IGJ that reflects transitive immutability: if an object has a constructor that is annotated `@Immutable`, then the object's class cannot transitively contain any fields not marked `@Immutable`. We have started conducting user studies evaluating this design and comparing it to a subset of IGJ that is restricted to non-transitive immutability: that is, without the immutability parameter that facilitates ad hoc transitive immutability. Our own experimentation, including pilot user studies, suggests that the full set of features required to enable all the possible kinds of immutability (non-transitive, ad hoc transitive, and transitive immutability; read-only references; and class-based immutability) is too complicated for users to understand easily. Instead, we hope to identify which combination of features would offer the best tradeoff for most users. We hypothesize that transitive immutability better matches people's expectations of what immutability should mean, and if used in software, will prevent bugs.

Acknowledgments

We appreciate help from Michael Ernst and Werner Dietl in helping us understand IGJ and feedback from Jonathan Aldrich. This material is supported in part by NSA lablet contract #H98230-14-C-0140, by SEI contract FA8721-05-C-0003 with CMU, and by NSF grant CNS-1423054. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of any of the sponsors.

References

[1] Bloch, J. Effective Java, Second Edition. Mountain View, CA Sun Microsystems, 2008.

[2] Oracle Corp. Secure Coding Guidelines for the Java SE, version 4.0. http://www.oracle.com/technetwork/java/seccodeguide-139067.html#6

[3] Zibin, Y., Potanin, A., Ali, M., Artzi, S., Kie, A. un, and Ernst, M. D. (2007). Object and Reference Immutability Using Java Generics. In Proceedings of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering (pp. 7584). New York, NY, USA: ACM.

[5] Pechtchanski, I., and Sarkar, V. (2002). Immutability specification and its applications. In Proceedings of the 2002 Joint ACM-ISCOPE conference on Java Grande - JGI '02 (pp. 202-211). New York, NY, USA: ACM Press.

[6] Microsoft, Inc. Freezable Objects Overview. https://msdn.microsoft.com/en-us/library/vstudio/ms750509(v=vs.100).aspx.

[7] Stylos, J. and Clarke, S. Usability Implications of Requiring Parameters in Objects' Constructors. In International Conference on Software Engineering (ICSE'2007). Minneapolis, MN: pp. 529–539.

[8] The Rust Programming Language. Mozilla Research. https://www.rust-lang.org.