Flexible Initialization of Immutable Objects

Tyler Etzel
Cornell University, United States
tje44@cornell.edu

Abstract

Immutability is a valuable feature for programmers in object oriented languages: making objects immutable often simplifies reasoning about the correctness of code, particularly when concurrency is present. Java allows programmers to express and enforce immutability by declaring all fields of an object `final`, but this comes at the cost of decreased expressiveness and intuitiveness of initialization. In this work, we propose a minimalistic type-based mechanism that both enforces immutability and relaxes these constraints on initialization. Furthermore, we propose and formalize two different type systems based on this mechanism that form a meaningful trade-off with respect to complexity, expressiveness, and strength of static guarantees. System One is simple, more expressive, and provides object-level immutability; System Two has more complicated annotation, is less expressive, and ensures that immutable objects are fully initialized in addition to enforcing immutability.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Immutability, Initialization

1. The Problem

Immutability is an important language feature for software engineering because it simplifies design and maintenance of code. Joshua Bloch, in his book *Effective Java*, goes so far as to recommend immutability as a default choice when designing a program, only resorting to mutability when a "very good" reason exists [1]. Current solutions to immutability can nevertheless be quite burdensome. In Java, immutability is enforced by labeling the fields of an object with the keyword `final`. Such fields must be assigned exactly once by a static initializer or directly inside the body of a constructor.

This mechanism for immutability constrains the usability and expressiveness of object initialization. In terms of usability, programmers are more productive with, and prefer, APIs that embrace the "create-set-call" approach to object construction, whereby objects are first constructed with a default constructor, and then brought into a specific initialized state with a series of "set" methods [2]. In terms of expressiveness, cyclic object hierarchies, although not entirely prohibited, are quite limited and awkward with `final`-based immutability.

The existing literature on immutability distinguishes between read-only references, which cannot be used to mutate an object's state, and object-level immutability, which prevents an object's state from being mutated by any reference [4]. Object-level immutability is valuable because it eliminates the entire class of bugs that arise from unexpected state changes, at least if the object in question is declared immutable. Our systems provide object-level immutability for this reason.

SPLASH Companion'16, October 30 – November 4, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4437-1/16/10...$15.00
http://dx.doi.org/10.1145/2984043.2998541

2. Our Approach

Our contributions are (1) A minimalistic, type-based mechanism for allowing flexible initialization of immutable objects; (2) A formalization of this mechanism in a simplified Java-like language, referred to as "System One"; (3) An extension to System One that provides additional initialization guarantees, referred to as "System Two"; (4) A dynamic semantics for a Java-like language, and proofs of the relevant static guarantees for System One and Two.

2.1 System One

Our mechanism provides object-level immutability. It works as follows: first, instead of enforcing immutability at the level of fields, the system require programmers to label classes as immutable. To allow for flexible initialization, we devised a type system inspired by "delayed types" [3]. However, instead of tracking the initialization status of specific fields, this system tracks the status of objects at a much coarser granularity: an object is either "liquid" (i.e. still being initialized), or "frozen' (i.e. initialized and immutable). The liquid type-state is indicated by an additional annotation wherever types occur (e.g. "`@Liquid List`"); frozen is the default type-state for an immutable class. Objects of immutable classes are initially liquid and are bound to some scope at creation. We refer to these as "initialization scopes". Changing from the liquid to frozen state occurs at the end of the initialization scope. Our system guarantees object-level immutability for frozen objects. Objects bind to the smallest scope that contains them.

```
/* Forms a cyclic link between objects.
 * "@Initializes" is needed for System Two  */
@Initializes("{n1.other, n2.other}")
void link(@Liquid Node n1, @Liquid Node n2) {
   n1.other = n2;
   n2.other = n1;
}
Node double_link() {
/* this scope is explicit as in the formalism */
   initialize {
      link(new Node(), new Node());
   }
}
```

Listing 1. Illustrates the use of Liquid types

Making scopes explicit simplifies the proof of correctness. Still, it is possible to infer initialization scopes based on types declared in method signatures. By using inference of initialization scopes with sensible default annotations, the programmer would need to be provide additional annotations only if flexible initialization is actually needed.

To allow unrestricted usage of objects would amount to tracking all aliasing information about an object. Our solution to this problem is, broadly speaking, to prevent the *usage* of still-liquid objects and only allow operations serve to *initialize* these objects. "Usage" of an object entails, for example, reading the fields of that object or assigning it to a field of a mutable object. We also disallow `@Liquid` fields. Of course, the type system allows assigning still-liquid objects to fields of other still liquid-objects: this is what allows the construction of complex object hierarchies. Our solution is unique in its simplicity: other approaches (e.g. the one elaborated in [6]) solve this problem by making the type system more complicated.

2.2 System Two

In switching to System One from the `final`-based system that Java uses, the programmer loses the guarantee that objects are fully initialized once constructed. This is checked in Java simply by ensuring that each `final` field is assigned either statically or directly in the constructor. System Two imposes additional annotation burden and reduces expressiveness but guarantees that all frozen objects are initialized.

Unlike System One, our type formalism for System Two tracks individual fields. If the a field is possibly uninitialized at the end of its object's scope, a compiler error is raised. Checking the initialization status of a field accurately across methods requires additional annotation from the programmer. Just as in the delayed types system, our formalism encodes this information as a set of fields in each method signature (e.g. `{x.f, y.g}`), indicating which fields the method initializes [3]. Listing 1 shows an example of this.

2.3 Formalism and Proofs

We have devised a small-step semantics to represent the execution of programs in a Java-like language. Using this formalism, we proved that (1) In System One and Two, the execution of a well-typed program never mutates a frozen object, nor allows frozen objects to become liquid again; and (2) In System Two, the execution of a well-typed program never freezes an object, unless all fields of the object's runtime type have been assigned.

The formalization and proofs can be found here.

3. Related Work

Our mechanism strikes a novel balance in terms of simplicity and expressiveness. Many existing systems, like IGJ, provide object immutability but do not allow flexible initialization [7]. Similar approaches exist that allow flexible initialization in lexical scopes. The system in [6], for example, provides roughly the same static guarantees as System One and allows flexible initialization. This system allows more usage of immutable objects during construction, but at the cost of additional complexity: the type quantifier hierarchy is substantially more complicated. System One is as minimal as possible, while still allowing flexible initialization.

Placeholder types are the only system we know of that matches our system's simplicity [8]. They are nevertheless not as expressive. As one example, programmatically initializing a tree with parent pointers from the bottom-up is not possible because all nodes of the tree would need to be constructed in the same placeholder declaration.

Similar trade-offs exist with System Two. System Two is comparable to the delayed types and masked types systems in [3] and [5]. Our systems restricts use during initialization to a greater degree than these systems. On the other hand, System Two can handle situations that delayed types cannot (bottom-up initialization of a tree with parent nodes, for example); System Two is also much simpler from the programmers point of view than masked types.

References

[1] J. Bloch. *Effective Java, Second Edition*. 2008.

[2] J. Stylos, and S. Clarke. Usability Implications of Requiring Parameters in Objects' Constructors. In *ICSE*, 2007.

[3] M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *OOPSLA*, 2007.

[4] M. Coblenz, J. Sunshine, J. Aldrich, B. Myers, S. Weber, and F. Shull. Exploring Language Support for Immutability. In *ICSE*, 2016.

[5] X. Qi and A. C. Myers. Masked types for sound object initialization. In *POPL*, 2009.

[6] C. Haack and E. Poll. Type-based object immutability with flexible initialization. In *ECOOP*, 2009.

[7] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kieżun, and M. Ernst. Object and reference immutability using Java generics. In *ESCE/FSE*, 2007.

[8] M. Servetto, J. Mackay, A. Potanin, J. Noble. The Billion-Dollar Fix: Safe Modular Circular Initialisation with Placeholders and Placeholder Types. In *ECOOP*, 2013.