Specifying and Verifying Static Initialization in Deductive Program Verifiers

Bachelor's Thesis Description

Patricia Firlejczyk
Supervised by João Pereira, Dr. Marco Eilers
and Prof. Dr. Peter Müller

September 2023

1 Introduction

Static initializers are blocks of code responsible for initializing the static variables of a module, i.e., the variables whose lifetime is the entire program, before they are first accessed. Static initializers are present in multiple mainstream programming languages: in Java, static initializers are used to initialize the static fields of classes, whereas in Go, static initializers can be used to initialize the global variables of a package.
In mainstream languages, static initializers typically have the two following properties, which are guaranteed by the language runtime:

Property 1: Each static initializer runs before any access to the static variables that it initializes

Property 2: Each static initializer runs at most once

Most mainstream imperative and object-oriented programming languages provide support for initialization code. However, how initialization is done strongly depends on the programming language. One difference is how the first property is implemented. In Go, this property is achieved by executing the static initializers before executing the `main` method. Java uses lazy initialization, which means that initialization happens at runtime and a class is only initialized just before the first use of any item declared in this class. So in Java, a static initializer may run after the start of the `main` method as long as the first property holds.
The two properties listed above must be guaranteed by the language implementation and the client does not have to ensure them manually.

Because of the first property, static initializers can be used to establish properties of static fields independently of the clients of the class. An example of static initializers is printing in Java. If one wants to print to the console in Java, one often uses the `System.out.println()` method. The variable `out` is a static variable of the class `System` and is initialized there statically. Because of that, a client which wants to use the `println()` method, does not need to explicitly initialize the `System` class, but can directly call the method `println()` on the field `out` [1].

```java
class A {
    public static int X;
    static {
        X = 1;
    }
}
```

Listing 1: The block on lines 3-5 is a static initializer of class `A`. The static variable `A.X` is initialized to 1 before the first mention of `A` and the initialization happens only once even if there are multiple objects of type `A`.

In Java, we have static blocks for static initialization as shown in listing 1. There are a lot of different problems with initialization in Java and its implementations. One problem is that the JLS underspecifies when static fields in interfaces are initialized. They don't have to be initialized before the initialization of classes that import them, however they can. This means that the same program can lead to different results under different implementations.

Another problem is that Java allows for cyclic dependencies between the initialization order of classes. In Java, a class is initialized by the runtime before the first active use of that class. There are six situations that are considered active usage, such as creating a new instance of a class, calling a static method declared by a class or the use or assignment of a static field declared by a class or interface. Even if there are multiple active uses of a class, from the second property follows that each static initializer is executed at most once. Uses that are not considered active uses are called passive uses and do not trigger the class's initialization [2].
Therefore, if class `A` has an active use of `B` in its static initializer, then `B` must be initialized before `A` can do its static initialization. If `B` also has an active use of `A` in its initialization, then we have a cyclic dependency. This leads to the problem that concurrent initialization in Java can deadlock.
In listing 2, the initialization dependencies form a cycle, since the accesses of `B.b` and `A.a` are considered active uses. There are three possible executions of this program.

Execution 1: a thread enters class `A` before it enters class `B`

```java
class A {
    static char a = 'a';
    static { a = B.b; }
}
```

```java
class B {
    static char b = 'b';
    static { b = A.a; }
}
```

Listing 2: The static initializers of both classes depend on each other, so concurrent execution may deadlock or lead to nondeterministic results.

Execution 2: a thread enters class `B` before it enters class `A`

Execution 3: classes `A` and `B` are initialized by two threads at the same time

In the first execution, the static char `A.a` is initialized to `'a'` first. Then, when class `B` is entered, `B.b` is set to `'a'`, since `A.a` equals `'a'`. In the end, both `A.a` and `B.b` hold the value `'a'`.
In the second execution, the opposite happens. Both `A.a` and `B.b` end up holding the value `'b'` at the end of the initialization.
In the last execution, `A.a` is set to `'a'` and `B.b` is set to `'b'`. However, in both classes the static block cannot be executed, as neither class `A` nor class `B` have finished their initialization. Both threads wait for the other one to finish the initialization and thus there is a deadlock [3].

Go uses init blocks for package initialization. In Go, a static variable is initialized after all variables that it depends on are initialized.
The initialization in Go happens one package at a time. Across multiple packages, the language specificiation defines that a package will be initialized before all packages that import it. By construction it is guaranteed that there are no cyclic initialization dependencies between packages.

The initialization in Go suffers from a known problem called the *Static Initialization Order Fiasco* [4], which was initially coined for C++. Across multiple files, the initialization order of variables is determined by the order in which the files are compiled. As such, the order of the initialization may depend on the command used to compile the project.

```go
package main

var X map[int]int

func init() {
    X = make(map[int]int)
}
```

```go
package main

func init() {
    X[0] = 0
}

func main() { }
```

If you compile and run the programs using the command `go run file2.go file1.go`, you get an error message. The second file is initialized before the

Listing 3: The outcome of this program depends on the order in which the files are passed to the compiler. The file on the left is called `file1.go` and the file on the right is `file2.go`

first file, thus `X[0]` is accessed, before `X` is created in `file1`. If you run them the other way around, there is no problem, since `X = make(map[int]int)` happens before `X[0]=0`. [5]

The dynamic initialization in C++ suffers from the same problem. In C++, the initialization of static variables happens in two stages. Since static variables refer to variables whose lifetime is the entire program, in C++, if the initial value of a variable can be evaluated at compile time, this variable is also initialized at compile time. This makes the runtime of the program faster. The remaining variables are zero-initialized at compile time and later dynamically initialized at runtime [6].

Because of these problems, it would be useful to use tools like program verifiers to reason about static initialization and the values of static fields at different points in the program.

An existing approach providing a methodology for specifying and verifying static class invariants in object-oriented programs is described in a paper written by K. Rustan M. Leino and Peter Müller [7]. This paper considers three major uses of static fields and invariants in the Java library. In the methodology described in this paper, each class and each object have a field which marks if the class/object invariant holds or may not hold. It thus allows an invariant to be temporarily violated. The paper defines a validity ordering between classes and uses it, among other things, as the initialization order. From the validity ordering follows that subclasses must be initialized before a superclass they extend is initialized. However, this does not hold in some languages like in Java or in C#. Another issue is that this paper does not use separation logic [8]. This makes it more difficult to deal with concurrency. The goal of this project is to define a new approach using separation logic based on the ideas in [7].

2 Goals

2.1 Core Goals

1. Collect and analyze the uses of static initializers and the kinds of properties that they are used to establish. The analysis may be inspired by the work by K. Rustan M. Leino and Peter Müller [7] and the work by Simon Fritsche supervised by Malte Schwerhoff and Peter Müller [9]. Based on that, we plan to categorize the uses of static fields and invariants and find interesting example programs for all of them. These code examples can

be later used as test cases.

2. Devise a modular specification language that extends separation logic (language agnostic or not) capable of describing module invariants and properties established by static initializers.

3. Develop a verification methodology to verify the properties that can be expressed in the specification language defined in goal 2. Define a toy language with support for modules, static initializers, and heap-allocated data-structures. The initialization order may be less defined in this language than in Java, and further assumptions may be taken into account later, or it may be more precisely defined from the beginning. The outcome should be an encoding of the toy language using the developed methodology into Viper [10].

4. Implement the test cases obtained in goal 1 using our obtained methodology from goal 3 in Viper and evaluate this approach.

5. Characterize static initialization implementations from different languages, for example Java, Go, C++, based on the features they provide, e.g., lazy vs eager (including whether they ensure that all static initializers run once, or at most once), whether they allow cyclical dependencies or not.

2.2 Extension Goals

1. Develop a program logic for the toy language mentioned in core goal 3. Implement the methodology obtained in goal 3 as a set of formal rules. The logic can be used to verify the correctness of programs containing static initializers.

2. Develop extensions of the verification methodology catering to different language features then identified in core goal 5. The goal is to incorporate more knowledge (for example, the initialization order if it is known) in order to prove more properties set established static initializers.

References

[1] Javainterviewpoint, "Java – how system.out.println() really work?." `https://www.javainterviewpoint.com/java-how-system-out-println-really-work/`. Accessed: 2023-09-17.

[2] B. Venners, "The lifetime of a type." `https://www.artima.com/insidejvm/ed2/lifetype.html`. Accessed: 2023-09-20.

[3] E. Börger and W. Schulte, "Initialization problems for java," *Softw. Concepts Tools*, vol. 19, no. 4, pp. 175–178, 2000.

[4] "Static initialization order fiasco." https://en.cppreference.com/w/cpp/language/siof. Accessed: 2023-09-25.

[5] "The go programming language specification." https://go.dev/ref/spec\#Program_initialization_and_execution. Accessed: 2023-09-17.

[6] P. Arias, "C++ - initialization of static variables." https://pabloariasal.github.io/2020/01/02/static-variable-initialization/. Accessed: 2023-09-17.

[7] K. R. M. Leino and P. Müller, "Modular verification of static class invariants," in *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings* (J. S. Fitzgerald, I. J. Hayes, and A. Tarlecki, eds.), vol. 3582 of *Lecture Notes in Computer Science*, pp. 26–42, Springer, 2005.

[8] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pp. 55–74, IEEE Computer Society, 2002.

[9] S. Fritsche, M. Schwerho, and P. Müller, "Verifying scala's vals and lazy vals," 2014.

[10] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation* (B. Jobstmann and K. R. M. Leino, eds.), (Berlin, Heidelberg), pp. 41–62, Springer Berlin Heidelberg, 2016.