Modular Reasoning about Global Variables and Their Initialization

JOÃO PEREIRA, ETH Zurich, Switzerland
ISAAC VAN BAKEL, ETH Zurich, Switzerland
PATRICIA FIRLEJCZYK, ETH Zurich, Switzerland
MARCO EILERS, ETH Zurich, Switzerland
PETER MÜLLER, ETH Zurich, Switzerland

Many imperative programming languages offer global variables to implement common functionality such as global caches and counters. Global variables are typically initialized by *module initializers* (e.g., static initializers in Java), code blocks that are executed automatically by the runtime system. When or in what order these initializers run is typically not known statically and modularly. For instance in Java, initialization is triggered dynamically upon the first use of a class, while in Go, the order depends on all packages of a program. As a result, reasoning modularly about global variables and their initialization is difficult, especially because module initializers may perform arbitrary side effects and may have cyclic dependencies. Consequently, existing modular verification techniques either do not support global state or impose drastic restrictions that are not satisfied by mainstream languages and programs.

In this paper, we present the first practical verification technique to reason formally and modularly about global state and its initialization. Our technique is based on separation logic and uses *module invariants* to specify ownership and values of global variables. A partial order on modules and methods allows us to reason modularly about when a module invariant may be soundly assumed to hold, irrespective of when exactly the module initializer establishing it runs. Our technique supports both thread-local and shared global state. We formalize it as a program logic in Iris and prove its soundness in Rocq. We make only minimal assumptions about the initialization semantics, making our technique applicable to a wide range of programming languages. We implemented our technique in existing verifiers for Java and Go and demonstrate its effectiveness on typical uses cases of global state as well as a substantial codebase implementing an Internet router.

CCS Concepts: • **General and reference → Verification**; • **Theory of computation → Separation logic**; • **Software and its engineering → Formal software verification**.

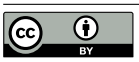
Additional Key Words and Phrases: Static Initialization, Global Variables, Iris, Concurrent Separation Logic

ACM Reference Format:
João Pereira, Isaac van Bakel, Patricia Firlejczyk, Marco Eilers, and Peter Müller. 2025. Modular Reasoning about Global Variables and Their Initialization. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 355 (October 2025), 28 pages. https://doi.org/10.1145/3763133

1 Introduction

Many programming languages support *global state* to implement patterns such as caches, dispatch tables, and global counters. To list a few, C, C++, Go, and Rust support global variables, Java and C#

Authors' Contact Information: João Pereira, joao.pereira@inf.ethz.ch, Department of Computer Science, ETH Zurich, Zurich, Switzerland; Isaac van Bakel, isaac.vanbakel@inf.ethz.ch, Department of Computer Science, ETH Zurich, Zurich, Switzerland; Patricia Firlejczyk, pfirlejczyk@student.ethz.ch, Department of Computer Science, ETH Zurich, Zurich, Switzerland; Marco Eilers, marco.eilers@inf.ethz.ch, Department of Computer Science, ETH Zurich, Zurich, Switzerland; Peter Müller, peter.mueller@inf.ethz.ch, Department of Computer Science, ETH Zurich, Zurich, Switzerland.



ACM 2475-1421/2025/10-ART355
https://doi.org/10.1145/3763133

have static fields, and Scala has singleton objects. Global variables have two defining characteristics: First, their scope corresponds, at least, to the module[1] in which they are declared; often, they are accessible to the entire program. Second, they have a *static lifetime*, and thus, they may be accessed at any point during the execution of the program. In practice, global variables are widely used for storing both shared immutable values (e.g., the OpenJDK implementation of Java's `Byte` class [8] has a static field named `cache` that stores an array of interned instances of `Byte` to be returned by the method `Byte.valueOf`) and mutable state (e.g., Java's `Logger` class [7] stores the hierarchical namespace of loggers in static fields).

Both characteristics pose challenges for modular verification. The global scope complicates local reasoning: global variables are shared among different methods (and different threads in concurrent executions), such that ownership-based verification as in separation logic [49] is not directly applicable. A standard solution to this problem is to maintain invariants over the shared state. We adopt this approach by introducing *module invariants* over the values of global variables. Such invariants need to be established during the initialization of global variables; *from then on*, they may be assumed and need to be preserved by code that accesses global state. However, this simple inductive argument is complicated by the second characteristic of global variables, their static lifetimes. Real programming languages have a complex initialization semantics; for instance, in Java, C#, Go, and Rust[2], global variables are initialized by *module initializers*. Module initializers are guaranteed to *start* executing before the global variables of their modules are accessed for the first time, but are in general *not* guaranteed to *finish*, for instance, when the access occurs *during* the initialization or when module initializers are mutually recursive. Consequently, the module invariant may not have been established by the time of the first access. As a result, this language design poses three challenges for modular verification.

(1) *Validity of module invariants:* A verification technique must determine when module invariants may soundly be assumed to hold. This requires a way to determine when each module initializer has definitely finished (and, thus, established the module invariant) and when a module invariant may be temporarily violated, for instance, during an update of global data through another thread. This challenge is complicated by the fact that the order in which module initializers run is not known modularly (e.g., in Go [10]) or is even determined dynamically (e.g., in C# and Java [6, 9]).
(2) *Side effects of module initializers:* In mainstream languages, module initializers may perform arbitrary side effects on all memory locations reachable from global variables. Consequently, the program behavior depends on when module initializers are executed, and in which order. Since this information is not known modularly (or even statically), a verification technique must take into account all valid timings of the module initializers.
(3) *Cyclic initialization dependencies:* Some languages like Java and C# allow two or more modules to mutually access each other's global variables from their module initializers (possibly indirectly via method calls). Such cyclic dependencies usually indicate latent bugs, such as reading uninitialized global variables and deadlock [18, 39]. Bugs like these plague many concurrent codebases, including the previously mentioned `Logger` class in the Java standard library. They need to be prevented by a verification technique.

[1]While we do not formally define *module*, we note that it corresponds to *classes* in Java, *packages* in Go, and *compilation units* in `C/C++`.
[2]Initializers are not part of the Rust language, but a similar functionality is provided by the `lazy_static!` macro defined in the `lazy-init` crate [4], which precedes all accesses to a global variable by code that runs the initialization code if it hasn't been run yet.

Despite the prevalence of global state, to our knowledge, there are no verification techniques that provide a practical solution to these challenges. Leino and Müller [35] propose a verification technique for invariants over global state. In their technique, modules are initialized eagerly according to a user-defined validity ordering. It is, thus, not applicable to mainstream languages, which have a different initialization semantics. Summers et al. [51] also use invariants to reason about static fields, but require programs to follow a strict, static ownership scheme [22], which is too restrictive for many practical use cases. Both of the above approaches support only sequential programs. Jacobs et al.'s technique [28] targets concurrent programs by requiring *all* accesses to global variables to be synchronized with a lock. They do not support programs that safely access mutable global variables without explicit synchronization. Moreover, like both previous approaches, they impose a restrictive ownership regime. Liu et al. [39] present a static analysis to identify common issues in the initialization of Scala's singleton objects. However, their technique does not allow one to verify functional correctness. Finally, prior to our work, no deductive verifier like Gobra [55], KeY [12], VerCors [17], or VeriFast [27] supports modular reasoning about global state.

This Paper. We present the first modular verification technique for programs with global state that is sufficiently expressive to handle practical implementations and languages. To handle mutable state and concurrency, our technique builds on concurrent separation logic [44, 49]. In particular, we treat global variables as separation logic resources that can be accessed only when the executing methods owns the corresponding permission. This treatment allows us to rule out data races and reason about side effects on global variables.

We specify modules with *module invariants*, separation logic assertions that may contain permissions to the global variables of a module as well as constraints on their values. A proof obligation ensures that the invariant of each module is established by the module's initializer.

Similar to Iris invariants [29], module invariants are represented as separation logic resources called *module tokens* (or *tokens* for short). A method may *open* a module invariant to obtain the contained permissions if it owns the corresponding token. Opening a module invariant is sound only at program points where the initialization of the corresponding module has definitely finished; otherwise, one would potentially assume the module invariant before it has been established. Therefore, opening an invariant is guarded by two proof obligations: (1) An invariant may be opened when the subsequent execution step (or sometimes an earlier operation) ensures that initialization must have started; for instance in Java, initialization of a class is triggered by object creation, static field access, and static method calls and, thus, an invariant may be opened at these operations. This proof obligation guarantees that by the time we attempt to use the invariant, initialization has at least started. (2) We check that the current method is not executed *during* the initialization; the semantics of the programming language then implies that module initialization must also have finished before the current method is executed. To enable this second proof obligation, we introduce a novel specification construct called *initialization levels* that establishes a partial order on the modules and methods of a program. Proof obligations ensure that the initializer of a module with level m may call only methods with a strictly smaller level. Consequently, a method with level l cannot be part of the initialization of any module with a level of at most l, such that their initialization, if started, must also have finished.

The approach described so far already solves the three challenges above. A method may soundly assume the invariant of a module if it holds the corresponding token and the method's level is at least the module's level (Challenge 1). Since module initializers do not own any module tokens, they cannot cause side effects beyond initializing the global variables of the enclosing module (and newly-allocated memory). Consequently, verification does not depend on their exact execution

time (Challenge 2). For the same reason, a module initializer cannot read global variables of other modules, making it impossible to create cyclic initialization dependencies (Challenge 3).

The technique described so far is sound, but overly restrictive for two reasons. First, it is common for global variables to store *immutable* data such as constants or global look-up tables. Once initialized, such data can be safely accessed (also concurrently), such that the verification overhead of passing around module tokens is unnecessary. The same applies to global data that is protected by a synchronization primitive such as a lock. Second, not allowing the initializer of one module to access global variables of another module prevents common idioms, such as printing a debug messages during initialization using Java's `System.out` stream.

To remove these limitations, our technique supports *duplicable* module invariants, in addition to the *non-duplicable* invariants described so far. Duplicable invariants may contain permission to read memory locations as well as value constraints to capture properties that, once established, hold for the rest of the program execution. Therefore, opening a duplicable invariant does *not* require the presence of a module token; every method or module initializer may open the duplicable invariant of every module with at most the same level. Verification that a duplicable invariant has been established is analogous to non-duplicable invariants.

Duplicable invariants provide a convenient way to access immutable data. They can also be used to share a lock among different threads. By including permissions in the lock invariant, threads can get synchronized write access to global state. Duplicable invariants also let the initializer of a module read global variables of other modules, irrespective of their levels, and even modify them if they are protected by a synchronization primitive (we will present a further relaxation for sequential code in Sec. 4.1).

For duplicable invariants, Challenge 1 and 2 are solved analogously to non-duplicable invariants. Challenge 3 is solved by the level requirement. Since a module initializer may only read global variables of modules with a strictly smaller level, cyclic dependencies are not possible.

Our technique is applicable to any programming language that satisfies the following three conditions:

- **C1** We call a point in the execution of a program *definitely initialized for a module M* if it occurs (1) right before an access to a global variable of M or a call to a static method[3] of M, and (2) outside of M's module initializer and methods directly or indirectly called from it. The module initializer of M starts executing at the latest when reaching the first definitely-initialized execution point for M. It also finishes before executing the next operation after the first definitely-initialized execution point for M unless (1) it does not terminate or (2) M is part of a cyclic initialization dependency. In case (1), all threads block at the next definitely-initialized execution point for M.
- **C2** In case the language includes operations that trigger module initialization (e.g., in Java, those mentioned in the previous condition) then only the first occurrence of such an operation in a program execution triggers. No other operations wait for module initialization to finish.
- **C3** The module initializer of a module M runs at most once during the entire execution of the program.[4]

These conditions are satisfied by a wide range of mainstream programming languages with different initialization semantics, including Java, Go, C#, and Scala. In particular, the conditions do not require

[3] In our examples, we treat constructors analogously to static methods.

[4] Conditions **C2** and **C3** may seem redundant at first. However, **C2** ensures that triggering module initialization does not yield an error. Moreover, for languages where initialization may happen only when triggered, **C3** follows from **C2**, but is necessary for our technique to be sound for other languages.

that the initializer of a module runs, unless an access to one of its variables occurs. This allows for both eager and lazy initialization.

In the rest of this paper, we present our technique informally and illustrate it on examples (Sec. 2), and then formalize our techniques as a program logic in the Iris framework [30] and prove its soundness in Rocq (Sec. 3). To demonstrate that our technique is practical, we instantiate it to two mainstream languages with different initialization semantics, Java and Go, and implement it in two state-of-the-art deductive verifiers, VerCors [17] and Gobra [55] (Sec. 4). Our evaluation demonstrates that the technique is expressive and can be automated effectively (Sec. 5).

Contributions. We summarize our main contributions:

- We present the first practical, modular verification technique for programs with global state, written in mainstream programming languages. Our technique is applicable to a wide range of programming languages with different initialization semantics including eager and lazy initialization, such as C++, C#, Go, Java, and Scala.
- We formalize our technique as a separation logic. Our soundness proof is mechanized in the Iris framework and applies to languages whose initialization semantics satisfies conditions **C1-C3**.
- We implement our technique in two state-of-the-art verifiers for languages with different initialization semantics: Gobra for Go and VerCors for Java.
- We demonstrate the effectiveness of our approach on a wide range of challenging examples. Our experiments show that our technique imposes little annotation overhead and can be automated in SMT-based verifiers.

2 Key Ideas

In this section, we present our technique informally and illustrate its key ideas on several examples; the formal proof rules are presented in the next section.

Our examples are written in Java syntax, but we do not assume Java's lazy initialization semantics. As explained in the introduction, our technique is applicable to any language whose module initialization semantics satisfies conditions **C1**–**C3**. Therefore, we assume that a module initializer may start non-deterministically at any moment prior to the first access to a global variable or call to a static method of the module, and its execution may interleave with the threads of the program. This treatment of module initializers over-approximates both lazy and eager initialization semantics, thus providing general reasoning principles that are language agnostic.

2.1 Module Invariants

The method `fib` in Listing 1 recursively computes the `n`-th element of the Fibonacci sequence, and uses the global variable `cache` to memoize all prior results.

Proving that this method computes the `n`-th Fibonacci number when `n` is non-negative requires that the following properties hold for each access to the global variable `cache`:

- **P1** `cache` holds a non-null value, otherwise a `NullPointerException` may be thrown;
- **P2** `cache` is not accessed concurrently by another thread, otherwise data races may occur;
- **P3** `cache` contains the keys `0` and `1`, otherwise the method may not terminate;
- **P4** `cache` maps all of its keys `k` to the `k`-th Fibonacci number, otherwise the method may produce wrong results.

More generally, we must show for each access to a global variable that the access is *safe*, i.e., it does not lead to `null` pointer dereferences or race conditions (properties **P1** and **P2**), and that the values stored in global variables satisfy certain functional properties (**P3** and **P4**).

```
1  class Fib {
2    private final static HashMap<Integer, Integer> cache;
3    static {
4      cache = new HashMap<>();
5      cache.put(0, 0);
6      cache.put(1, 1);
7    }
8    public static int fib(int n) {
9      boolean hasKey = cache.containsKey(n);
10     if (hasKey) {
11       return cache.get(n);
12     }
13     int fibN = fib(n-1) + fib(n-2);
14     cache.put(n, fibN);
15     return fibN;
16   }
17 }
```

Listing 1. Class `Fib` provides a memoized implementation of the Fibonacci sequence. `cache` is a global variable; the code inside the `static`-block is the module initializer.

We use *module invariants* to make both kinds of properties explicit and formal. For our example in Java notation, module invariants can be specified at the level of classes; however, unlike traditional *class invariants* [42], they constrain global variables, not the fields of instances of the class.

We express module invariants in the assertion language of separation logic; thus, they provide information about the ownership of global variables and the values stored in them. A method may access a global variable if it owns it. We use *fractional permissions* [19] to allow concurrent reads, while ensuring exclusive writes. Ownership of a memory location x (which might be a global variable) is denoted with the *points-to* assertion $x \mapsto_p v$, where p is a rational number in the interval $(0, 1]$, and v is the value stored in location x (or _ if the value is irrelevant). A fractional amount p strictly less than 1 provides read-only access, whereas write access requires full ownership ($p = 1$). As is standard, the ownership of a memory location may be split and re-combined:

$$x \mapsto_{p1+p2} v \equiv x \mapsto_{p1} v * x \mapsto_{p2} v$$

In our example, we express properties **P1**–**P4** wit the following module invariant:

$$\begin{aligned} Inv_{\texttt{Fib}} \triangleq \exists m. \exists a.\, &\texttt{cache} \mapsto_1 m * \\ &m \neq null * is\_map(m, a) * \\ &\{0, 1\} \subseteq dom(a) * \\ &(\forall k.\, k \in dom(a) \implies 0 \leq a(k) \wedge a(k) = \mathcal{F}(k)) \end{aligned}$$

The first conjunct provides exclusive access (full ownership) to `cache` and, thus, rules out data races (property **P2**). The second conjunct guarantees property **P1**. The third conjunct uses the separation logic predicate $is\_map(m, a)$ to express full ownership of the hash map (property **P2**) and to abstract the state of the hash map to a partial function from keys to values, a; the concrete definition of *is_map* is irrelevant here. The remaining two conjuncts express property **P3** and **P4**, where $\mathcal{F}$ denotes the mathematical Fibonacci function.

2.1.1 Establishing Module Invariants. The module initializer $Initializer_M$ of a module M must establish M's module invariant, that is, it must satisfy the following triple:

$$\left\{Init_M \ * \ \ast_{g \in vars(M)} g \mapsto_1 \_\right\} \ Initializer_M \ \{Inv_M\}$$

where $Init_M$ indicates that module M has started initializing, as we will discuss in Sec. 2.2, $\ast$ is a syntactic shorthand for the separating conjunction over a set of locations, $vars(M)$ is the set of the identifiers of all global variables declared in module M, and Inv_M is M's module invariant. The precondition allows the module initializer to write to all global variables of the module, but does not directly permit accesses to global variables of other modules. Such accesses are in general not safe because they might introduce data races. We will see in Sec. 2.3 how we can allow such accesses if they are properly synchronized. Note that providing ownership to the module initializer cannot duplicate permissions because, according to condition **C3**, an initializer runs at most once.

Verifying this triple for the module initializer of class `Fib` is straightforward, provided we have appropriate specifications for the constructor and the `put` method of class `HashMap`.

Note that while termination of module initializers is generally desired, it is not required for the soundness of our technique. According to condition **C1**, a non-terminating initializer for a module M causes all thread executions to block at the next definitely-initialized execution points for M, such that any code that might use M's invariant is unreachable.

2.1.2 Using Module Invariants. The module invariant of a module M may be soundly used at any program point where the two following conditions hold:

IC1 M's initializer has started and finished, that is, its module invariant has been established;
IC2 No other thread concurrently relies on or invalidates the module invariant of M.

We explain how to check condition **IC1** in Sec. 2.2.

To establish condition **IC2**, we introduce a separation logic resource $[\text{Token } M]_p$ for every module M, where p denotes fractional ownership. Conceptually, one can think of this resource as an isorecursive predicate [50] that abstracts over the module invariant; like other isorecursive separation logic predicates, it can be unfolded to obtain the module invariant (thereby multiplying ownership amounts with fraction p, as usual) and get access to M's global variables. However, unfolding the predicate is sound only if condition **IC1** holds; otherwise, one might use an invariant that has not (yet) been established. To emphasize this difference to standard separation logic predicates, we use a different terminology. We call $[\text{Token } M]_p$ the *module token* (or *token* for short) of M; exchanging the token for the module invariant is called *opening* the invariant and comes with proof obligations that ensure **IC1** (see Sec. 2.2). *Closing* an invariant is analogous to folding a predicate, without extra proof obligations.

The program entry point (i.e., the `main` method) gets the module token $[\text{Token } M]_1$ for every module M in the program.[5] The `main` method can transfer tokens like any other separation logic resource, for instance, pass them to other methods or threads, or include them in predicates and module invariants. The standard separation logic proof rules ensure that neither a token nor the corresponding module invariant can be duplicated or forged, which guarantees condition **IC2**.

The only way for a method to obtain ownership of a global variable is via a module invariant. Hence, the module invariant of a module M typically contains ownership of all global variables of M. A method may then either obtain M's token (for instance, via its precondition) and open the invariant to obtain ownership of the global variables, or it may require ownership directly in its

[5]The set of all modules is generally not known during modular verification. However, it can be soundly approximated by including all tokens that are mentioned in the specifications of methods called from the main module, or in its module invariant, predicates, lock invariants, etc. Tokens of other modules could not be used during verification of `main` anyway.

precondition, pushing the task of opening the invariant to its caller. Our example uses the latter option, as expressed in the following specification for method `fib`:

$$\{Inv_{\texttt{Fib}} \land 0 \leq \texttt{n}\}\ \texttt{fib(n)}\ \{v.\ Inv_{\texttt{Fib}} \land v = \mathcal{F}(\texttt{n})\}$$

Verification of the implementation from Listing 1 against this specification is straightforward. Any caller of `fib` must hold $Inv_{\texttt{Fib}}$, for instance, by obtaining the token $[\text{Token Fib}]_1$ from `main` and opening the invariant, or by acquiring a lock whose lock invariant provides $Inv_{\texttt{Fib}}$. By building on separation logic, our technique is completely agnostic to how resources are manipulated and how threads are synchronized.

2.2 Definite Initialization

In the previous subsection, we have seen that a module invariant may be opened only at points, where the corresponding module invariant has been established (condition **IC1**). This is the case when the module initializer (1) has started to execute and (2) has terminated.

To check the first condition, we use a dedicated separation logic resource $Init_M$ indicating that M's initialization has started. This resource is obtained when accessing one of M's global variables or calling one of M's static methods. According to condition **C1** from the introduction, module initialization must start there at the latest. This approach soundly models both lazy initialization as in Java and eager initialization as in Go.

Checking the second condition is more involved. We need to make sure that the execution point at which we attempt to open the invariant for module M cannot be reached (directly or transitively) from M's initializer (otherwise, the execution point would be *during* M's initialization). To check this property modularly, we introduce *initialization levels* (or *levels* for short), a novel specification construct, which will later also allow us to rule out cyclic initialization dependencies (Challenge 3 in the introduction).

We require programmers to annotate each module, module initializer, and method with an initialization level, i.e., a value from a pre-selected partial order. We impose the following *level requirements*:

LC1 The level of the module initializer of a module M is *strictly less than* M's level.
LC2 A method or module initializer with level l may call only methods with a level of *at most* l.

Note that requirement **LC2** allows recursive implementations (including mutual recursion if all involved methods have the same level).

The level requirements imply that, in a module M with level l, the initializer can (directly or indirectly) call only methods with a strictly smaller level. Importantly, *all other methods* may safely assume that they are *not* executed while M's module initializer runs. Consequently, according to condition **C1**, if M's initializer has been started, it must also have terminated and, therefore, it is sound to open M's invariant. So in summary, our proof obligations ensure condition **IC1** by checking that (1) the $Init_M$ resource has been obtained and (2) the level of the current method is above M's level.[6] Inside the module initializer of a module N, check (1) is sufficient to establish **IC1**. As we will see in the next section, $Init_M$ cannot be transferred into a module initializer (e.g., via a lock). So the only way for N's initializer to obtain $Init_M$ is to access a field or call a static method of M. Since we prove that cyclic initialization dependencies do not occur (see Sec. 2.4), condition **C1** guarantees that M's initializer has terminated before execution proceeds. When both **IC1** and **IC2** hold, we can soundly open the invariant.

[6]Technically, it would be sufficient to check that the method's level is *not below* M's level. The stronger check here will be necessary to avoid cyclic initialization dependencies, as we will explain in Sec. 2.4.

```
1  class Counter {
2    private final static Lock cl;
3    private static int count;
4
5    static {
6      count = 0;
7      cl = new Lock();
8    }
9
10   public static int getAndInc() {
11     cl.lock();   // acquire lock invariant
12     int result = count++;
13     cl.unlock(); // release lock invariant
14     return result;
15   }
16 }
```

Listing 2. Class `Counter`, which implements a global, thread-safe counter.

To verify the `Fib` module (Listing 1), the assigned initialization levels must satisfy the following constraints: the level of `HashMap`'s constructor and `put` method must be at most that of `Fib`'s initializer; the levels of `HashMap`'s `containsKey`, `get`, and `put` must all be at most `fib`'s. These constraints are needed to allow the various calls (**LC2**).

The need to assign levels may seem laborious at first, but is not difficult in practice. First, many modules do not call any methods from their initializers, because they do not have global variables or their global variables are initialized to constants. All those modules can have an arbitrarily small level. For the other modules, one can choose the smallest level that is greater than the levels of all methods called by their initializer. Second, levels for methods are related to the termination measures used by modular verifiers to prove termination. For instance, Dafny's termination measures [33] are lexicographically-ordered tuples that include the module name as first component, ordered my the import relation. The same approach works for our technique and ensures that it is always possible to call methods in imported modules, such as those of `HashMap` in our example.

2.3 Duplicable Invariants

As explained in Sec. 2.1, module tokens are necessary to ensure condition **IC2**, that is, to prevent different threads from performing unsynchronized accesses to global variables, potentially leading to data races and situations where an invariant is used by one thread while it has been temporarily violated by another. However, requiring the module token to open an invariant is overly restrictive in two common cases: when an invariant constrains only immutable global variables (e.g., global constants) and when access to (mutable) global variables is synchronized by a concurrency primitive such as a lock. In the former case, concurrent read accesses do not cause data races and the invariant cannot be violated after it has been established. In the latter, synchronization prevents data races and confines temporary violations to synchronized critical sections.

The example in Listing 2 illustrates both cases. The global variable `cl` is declared `final`, meaning that it cannot be modified after the module initializer has terminated. The global variable `count` is mutable, but protected by the non-reentrant lock in `cl`.

To handle such cases, we complement the module invariants presented so far with *duplicable module invariants*. A separation logic assertion P is *duplicable* if it satisfies the equivalence $P \equiv P * P$. A duplicable invariant cannot provide full ownership (or even a fixed fraction) of a resource.

Consequently, once the invariant has been established, it cannot be violated by writing to the global variables it mentions, nor can there be data races on these variables. Hence, condition **IC2** holds automatically and does not have to be enforced via module tokens. This allows us to use the following, weaker proof obligations for duplicable invariants: (1) They must be established by the module initializer. (2) To ensure condition **IC1**, the duplicable invariant of a module M may be opened only if the current method or initializer owns $Init_M$ and its level is at least M's level. (3) The level requirements **LC1** and **LC2** from Sec. 2.2 hold.

In our example, we associate the lock `cl` with a lock invariant $lock\_inv \triangleq \texttt{count} \mapsto_1 \_$, which grants full ownership to `count`. As usual in concurrent separation logics, a predicate $is\_lock(v, I)$ expresses that lock v has the lock invariant I. Calls to `lock` require this predicate and provide the lock invariant, whereas calls to `unlock` do the reverse. The predicate $is\_lock(o, P)$ is duplicable, such that all threads may compete for the lock by calling `lock`.

We can then specify the following duplicable module invariant for `Counter`:

$$Inv_{\texttt{Counter}} \triangleq \exists o.\, \exists p.\, 0 < p * \texttt{cl} \mapsto_p o \; * is\_lock(o, lock\_inv)$$

This allows us to discharge the proof obligations mentioned above: (1) The module initializer establishes the duplicable invariant; in particular, `cl`'s lock invariant holds. (2) Method `getAndInc` may open the duplicable module invariant because it obtains $Init_{\texttt{Counter}}$ when accessing `cl`. Moreover, we can assign levels such that `getAndInc`'s is at least `Counter`'s. (3) By assigning levels to all lock operations below those of `Counter`'s, all method and constructor calls are permitted.

Interestingly, these proofs do not require the specification of `getAndInc` to include any permissions to global variables nor the predicates our technique introduces ($Init$, tokens). The handling of `Counter`'s global state is hidden from clients and does not complicate their verification (except assigning suitable levels). In particular, this example demonstrates that duplicable invariants allow the module initializer of a module N to get read and write access to the global variables of another module M by using suitable synchronization. This synchronization is necessary to prevent data races. For immutable global variables of M, N's module initializer may directly open M's duplicable invariant (without synchronizing), as long as the level assignment allows that.

2.4 Cyclic Initialization Dependencies

The technique introduced so far is sound unless the program contains *cylic initialization dependencies*, that is, two or more modules where each module's initializers accesses a global field or static method of the next module on the cycle. The following example illustrates the problem.

In an execution where `A` is initialized first, the call to `B.foo` will start initialization of `B`, such that the assertion in `B`'s initializer is reached *before* `f` has been initialized and `A`'s invariant has been established. Nevertheless, this program verifies using the technique introduced so far! In particular, there is a level assignment that allows the call to `B.foo` and opening `A`'s invariant to prove the assertion in `B`'s initializer, just pick `B.foo` < `A` < `B` in the level ordering. This problem is not surprising, given that our technique so far builds on condition **C1** from the introduction, which fully applies only in programs without cyclic initialization dependencies.

Our level requirements **LC1** and **LC2** ensure that any *method* that executes during the initialization of a module M has a level strictly smaller than M and, thus, we can conclude when a method is *not* part of the initialization and may soundly open M's invariant. However, these requirements do not provide the same guarantee for *module initializers*. In the above example, `B`'s initializer runs during `A`'s initialization, but the level requirements do not prevent it from opening `A`'s invariant.

Intuitively, one can consider accesses to global variables and calls to static methods of a module M as "calling" M's initializer in case the initializer has not been started before. Therefore, we introduce an additional level requirement, which is the analog of **LC2** for these conceptual calls:

```
1  class A {
2    // dup invariant: ∃p. 0 < p ∧ f ↦_p 1
3    static int f;
4    static {
5      B.foo();
6      f = 1;
7    }
8  }
9
10 class B {
11   static {
12     // open A's invariant
13     assert A.f == 1;
14   }
15
16   static void foo() {}
17 }
```

Listing 3. Classes A and B form a cyclic initialization dependency.

```
1  class Main {
2    void main(String args[]) {
3      int x = Fib.fib(3);
4      assert x == 2;
5    }
6  }
```

Listing 4. Client program for the Fib module from Listing 1.

LC3 A method or module initializer with level l may access a global field or call a static method of a module M only if (1) it owns $Init_M$ or (2) M's level is *at most* l.

Together, our three level requirements rule out cyclic initialization dependencies, which makes our technique sound as we will prove later. In particular, the above example now correctly fails to verify because **LC3** introduces an additional constraint for the call to `B.foo`, $\mathsf{B} < \mathsf{A}$, leading to an unsatisfiable set of level constraints.

2.5 Complete Example

Having presented the key concepts of our approach, we can now complete our example by adding a client for the `Fib` module from Listing 1: In order to verify this client program, we need to establish the following proof obligations:

(1) The level requirements **LC1**, **LC2**, and **LC3** hold for the entire program.
(2) The precondition of `fib` holds before the call on line 3.
(3) The conditions **IC1** and **IC2** hold at every program point where the invariant is opened.
(4) The assertion in line 4 holds.

(1) The level requirement **LC1** trivially holds for class `Main` given that it does not have a module initializer. We can satisfy requirement **LC2** by choosing the level L of the method `main` to be higher than that of method `fib`; similarly, we can satisfy requirement **LC3** by choosing L such that it is also higher than that of the `Fib` module.

(2) Recall that the precondition of `fib` is that the invariant of `Fib` holds when the call is performed. We can thus satisfy this precondition by opening the invariant immediately before the call in line 3.

(3) Opening the invariant of `Fib` requires proving that the conditions **IC1** and **IC2** hold right before the call. As explained in Sec. 2.2, the condition **IC1** requires that initialization of the `Fib` module has started and finished. The former can be established from the call to `Fib.fib` itself, which yields the resource $Init_{Fib}$ to indicate that initialization has started. The latter follows from the fact that the `main` method's level is above `Fib`'s. Finally, condition **IC2** follows from the fact that method `main` holds the module token of `Fib` ($[\text{Token } Fib]_1$) at the time of the call, since, as explained in Sec. 2.1.1, the `main` method initially owns the module token $[\text{Token } M]_1$ for every module M in the program.

(4) After the call to `fib`, the assertion in line 4 follows directly from the method's postcondition.

3 Formalization

We formalize our technique as a program logic for an example concurrent language with global state using Iris [30], a separation logic framework. We establish its soundness using our key ideas, proved using the Iris proof mode [31, 32] in the Rocq proof assistant [53]. Our soundness result establishes the correctness of our treatment of invariants, and the absence of initialization cycles.

Our soundness result is shown for an ML-like language with global modules (Section 3.1). It is given a semantics by translation to HeapLang, another ML-like language with existing Iris support but no module system (Section 3.2). We model initializers using concurrent preemptive HeapLang threads which can execute at any time, overapproximating the different initialization semantics of real-world languages. We formally define the assertion language for global state (Section 3.3) and give its embedding into Iris' separation logic (Section 3.4).

Initialization cycle freedom in particular is shown by instrumenting the semantics of our ML-like language with a notion of runtime level, and asserting that all module accesses are strictly monotonic according to this level (Section 3.5). These assertions are then shown to always succeed for verified programs, so they can be elided from the semantics.

3.1 Syntax

In this subsection, we provide the syntax for our core language, which we use as a minimal example for our soundness results. This syntax is presented in Figure 1. Effectively, our language is defined by augmenting Iris' HeapLang with modules and global initialization.

Expr. The three significant additions to the expression language with respect to HeapLang are: loads from module fields $!M.f$; stores to module fields $M.f \leftarrow e$; and calls to module methods `call` $M.m$. Module methods can take arguments by being declared as lambdas.

Module. A module consists of a set of field and method declarations, together with the module initializer expression. Method declarations consist of a module-unique method name, along with an expression body. Field declarations are simply a module-unique field name. Field initialization can be performed in the module initializer.

Program. A program e_ϕ consists of an expression e, together with a set of accompanying named modules ϕ. The modules in the context can refer to each other's fields in their initializers and methods. These references can potentially be cyclic.

3.2 Translational Semantics

The semantics of our language is given by an embedding into HeapLang (shown in Figures 2 to 4), with its usual program state definition. This is done to simplify the formalization overhead:

$$
\begin{aligned}
v \in Val &::= () \mid n \in \mathbb{Z} \mid b \in \{\text{True}, \text{False}\} \mid \ell \in Loc \mid \ldots \\
x \in Var &::= \ldots \\
f \in Field &::= \ldots \\
m \in Method &::= \ldots \\
M \in Id &::= \ldots \\
e \in Expr &::= v \mid x \mid \lambda x.\ e \mid e\ e \mid \texttt{let}\ x := e\ \texttt{in}\ e \\
&\quad \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{while}\ e\ \texttt{do}\ e \\
&\quad \mid \texttt{ref}\ e \mid !e \mid e \leftarrow e \mid !M.f \mid M.f \leftarrow e \\
&\quad \mid \texttt{call}\ M.m \mid \texttt{fork}\ e \mid \ldots
\end{aligned}
$$

$$
\phi \in Modules \triangleq Id \rightharpoonup \left\{ \begin{array}{c} \textsc{Fields} : \overline{Field} \\ \textsc{MethodDefns} : Method \rightharpoonup Expr \\ \textsc{Init} : Expr \end{array} \right\}
$$

$$
e_\phi \in Program \triangleq Expr \times Modules
$$

Fig. 1. The syntax of our source language. The highlighted terms are the significant novelties w.r.t. HeapLang.

Translation of expressions. $\boxed{[\![e]\!]^\omega_\tau}$

$$
\begin{array}{ll}
[\![()]\!]^\omega_\tau \triangleq () & [\![n]\!]^\omega_\tau \triangleq n \\
[\![b]\!]^\omega_\tau \triangleq b & [\![\ell]\!]^\omega_\tau \triangleq \ell \\
[\![x]\!]^\omega_\tau \triangleq x & [\![\lambda x.\ e]\!]^\omega_\tau \triangleq \lambda x.\ [\![e]\!]^\omega_\tau \\
[\![e\ e']\!]^\omega_\tau \triangleq [\![e]\!]^\omega_\tau\ [\![e']\!]^\omega_\tau & [\![\texttt{ref}\ e]\!]^\omega_\tau \triangleq \texttt{ref}\ [\![e]\!]^\omega_\tau \\
[\![!e]\!]^\omega_\tau \triangleq !\,[\![e]\!]^\omega_\tau & [\![e \leftarrow e']\!]^\omega_\tau \triangleq [\![e]\!]^\omega_\tau \leftarrow [\![e']\!]^\omega_\tau \\
\multicolumn{2}{c}{\vdots}
\end{array}
$$

Fig. 2. The translational semantics of basic terms in our language. These terms don't interact with our translation structures ω, τ. We omit the syntax of HeapLang terms for brevity, but the languages are very similar, which is why most terms translate in a direct way [11].

instantiating Iris with an entirely new language requires significant proof and mechanization effort. By instead using a translational semantics, we are able to re-use the existing work for HeapLang.

Since our language is effectively an extension to HeapLang, most expressions are translated directly in the expected style (i.e. variables to variables, lambdas to lambdas, and so on), as shown in Figure 2. The only exceptions are those language features not present in HeapLang - namely, global fields, static methods, and module initializers. Giving these a semantics requires additional book-keeping structures, shown in Figure 3. Global fields are translated to HeapLang variables of heap-allocated values, which can be mutated. The mapping of global field to HeapLang variable is recorded in the FieldVars of the translation context τ. Similarly, static methods also become HeapLang variables of heap-allocated stored expressions (i.e. the method body), which can be loaded and executed. These are also recorded in τ as the MethodVars.

We can overapproximate both eager and lazy initialization approaches by encoding whole programs using HeapLang's support for concurrency. In this encoding, the program expression is

$$\omega \in \mathit{LevelAssignment} \triangleq \mathit{Id} \rightharpoonup L \quad L \text{ is a partially-ordered set}$$

$$\tau \in \mathit{TranslationCtx} \triangleq \left\{ \begin{array}{rl} \textsc{LockVars} : \mathit{Id} & \rightharpoonup \mathit{Var} \\ \textsc{FieldVars} : \mathit{Id} \times \mathit{Field} & \rightharpoonup \mathit{Var} \\ \textsc{MethodVars} : \mathit{Id} \times \mathit{Method} & \rightharpoonup \mathit{Var} \\ \textsc{ThreadIdVar} : \mathit{Var} & \\ \textsc{LevelVar} : \mathit{Var} & \end{array} \right\}$$

Fig. 3. Structures used for the translational semantics. ThreadIdVar stores the identifier of the module the current thread is initializing, if any. LevelVar stores the runtime level of the current thread.

Translation of module accesses. $\boxed{[\![e]\!]^{\omega}_{\tau}}$

$$\begin{array}{l} \textsc{Sync}(M)^{\omega}_{\tau} \triangleq \texttt{if } \tau.\textsc{ThreadIdVar} = M \\ \qquad \texttt{then skip} \\ \qquad \texttt{else if } \omega(M) < \tau.\textsc{LevelVar} \\ \qquad \texttt{then} \\ \qquad\quad \texttt{lock } \tau.\textsc{LockVars}(M); \\ \qquad\quad \texttt{unlock } \tau.\textsc{LockVars}(M) \\ \qquad \texttt{else fail} \end{array}$$

$$\begin{array}{l} [\![!M.f]\!]^{\omega}_{\tau} \triangleq \textsc{Sync}(M)^{\omega}_{\tau}; \\ \qquad\qquad !(\tau.\textsc{FieldVars}(M, f)) \end{array}$$

$$\begin{array}{l} [\![M.f \leftarrow e]\!]^{\omega}_{\tau} \triangleq \textbf{let } tmp = [\![e]\!]_{\tau} \textbf{ in} \\ \qquad\qquad \textsc{Sync}(M)^{\omega}_{\tau}; \\ \qquad\qquad \tau.\textsc{FieldVars}(M, f) \leftarrow tmp \end{array}$$

$$\begin{array}{l} [\![\texttt{call } M.m]\!]^{\omega}_{\tau} \triangleq \textsc{Sync}(M)^{\omega}_{\tau}; \\ \qquad\qquad !(\tau.\textsc{MethodVars}(M, m))\ \tau.\textsc{ThreadIdVar}\ \tau.\textsc{LevelVar} \end{array}$$

Fig. 4. The translational semantics of global state in our language. Sync is a meta-function for code generation that enforces the correct semantics for module accesses - access from the initializer is always allowed, and definitely-initialized execution points both assert monotonicity of the runtime level and block until initialization succeeds.

executed in one thread, and each module initializer is executed in a separate thread forked from the main one. A single thread per initializer suffices because initializers execute at most once (**C3**). All threads share access to the translation variables which model the fields and methods of modules, and HeapLang's concurrent semantics allow each thread to take steps at any point.

Module Access Synchronization. Our model restricts executions to those satisfying **C1** (where initialization starts, and potentially finishes, before the first definitely-initialized execution point). We do this by inserting synchronization code in our translational semantics, leveraging mutual exclusion (mutex) locks already provided in HeapLang. When creating a thread for each module initializer, we allocate and acquire a mutex for that thread until the initializer has completed execution (recording it in LockVars.) Once the initializer thread has completed, we release the mutex.

Since the mutex is taken while the initializer is executing, it then suffices to make a module's field accesses (both reads and writes) and method calls wait on that module's mutex, before performing the corresponding operation on the appropriate HeapLang variable (Figure 4). This prevents field and method use until the initializer has finished execution. By **C2**, this is all we can do - once the mutex is released by the initializer, no further synchronization occurs. We note that the mutex

$$P, Q \in Asst ::= \top \mid \bot \mid \ell \mapsto_p v \mid M.f \mapsto_p v \mid [\text{Token } M]_p \mid Init_M \mid \bigcirc_M P \mid P * Q \mid \ldots$$

Fig. 5. Assertions in our separation logic.

doesn't restrict *simultaneous* field accesses, since the mutex does not guard the access - it only ensures that the thread for the corresponding initializer has finished executing.

The only exception to this mutex-taking is in the initializer thread of a module itself, since **C2** forbids accesses to a module from inside that module's initializer to block. Therefore, our translation has access to a *thread identifier* variable, which marks the thread as either the unique initializer thread for a particular module, or a non-initializer thread. When performing an access, the module name is compared against the thread identifier at runtime. If there is a match, the mutex is ignored, which allows module initializers to access their own module fields and methods without blocking. This logic is implemented in the Sync code generation function shown in Fig. 4.

Acyclic Module Accesses. Finally, for module accesses we also assert that the access is *acyclic*. This is done by assigning threads a *runtime level*, which can be inspected by the semantics. [7] We then make each initializer's runtime level the initialization level of the module it initializes. When a module access occurs, the current thread's runtime level is asserted to be strictly above the initialization level (and thus the runtime level) of the module being blocked on (Figure 4). For non-blocking accesses, no such check is necessary.

3.3 Program Logic

We formally describe our reasoning principles for global state as a Hoare logic over our core language. Our program logic is defined using a separation logic assertion language. We also define how to specify whole programs, including those with module dependencies.

Assertions. Our assertion language is shown in Figure 5. As well as the typical constructor of a (fractional) heap points-to ($x \mapsto_p v$) and logical operators, we also define the specialized assertions:

- $M.f \mapsto_p v$, the (fractional) points-to for the field f of the module M.
- $[\text{Token } M]_p$, the module token of M.
- $\bigcirc_M P$, an assertion under the *initialization-guarded modality* $\bigcirc_M$. The guarded assertion P might not hold in the current heap, but *will* hold once initialization has completed for M. In particular, $\bigcirc_M \bot$ would require that the module initializer for M does not terminate.
- $Init_M$, the fact that M's initialization has started. This assertion holds no resources and is duplicable. It is used to eliminate the initialization-guarded modality $\bigcirc_M$ (see GuardElim).

The modality $\bigcirc_M$ has two roles: it allows us to soundly reason about accessing module invariants which may not yet have been established (because the corresponding module initializer has not run), and it allows us to fearlessly access a module invariant after initialization is known to have completed. It is often useful to be able to "unfold" the module token of a module M into a copy of its module invariant even before $Init_M$ is known. Once $Init_M$ is known, we can then open the invariant of M and soundly eliminate the modality at *any time* (even outside of definitely-initialized execution points).

[7] This comparison must match the partial order for used for initialization levels. We therefore require that partial order to be representable and comparable at runtime. We believe this is not onerous.

Hoare triples of assertions. $\boxed{\{P\}\ e\ |^{l}_{\Phi}\ \{v.\ Q\}}$

$$\textsc{OpenMInv}\ \frac{0 < p \qquad \{P * \bigcirc_M \Phi(M).\textsc{MInv}(p)\}\ e\ |^{l}_{\Phi}\ \{v.\ Q * \Phi(M).\textsc{MInv}(p)\}}{\left\{P * [\text{Token } M]_p\right\}\ e\ |^{l}_{\Phi}\ \left\{v.\ Q * [\text{Token } M]_p\right\}}\ \Phi(M).\textsc{Level} \le l$$

$$\textsc{OpenDupMInv}\ \frac{\{P * \bigcirc_M \Phi(M).\textsc{DupMInv}\}\ e\ |^{l}_{\Phi}\ \{v.\ Q\}}{\{P\}\ e\ |^{l}_{\Phi}\ \{v.\ Q\}}\ \Phi(M).\textsc{Level} \le l \qquad \textsc{GuardElim}\ \frac{\{P\}\ e\ |^{l}_{\Phi}\ \{v.\ Q\}}{\{\bigcirc_M P * \mathit{Init}_M\}\ e\ |^{l}_{\Phi}\ \{v.\ Q\}}$$

$$\textsc{LevelDrop}\ \frac{\{P\}\ e\ |^{l'}_{\Phi}\ \{v.\ Q\}}{\{P\}\ e\ |^{l}_{\Phi}\ \{v.\ Q\}}\ l' \le l \qquad \textsc{GlobFieldRead}\ \frac{0 < p}{\left\{M.f \mapsto_p v * \mathit{Init}_M\right\}\ !M.f\ |^{l}_{\Phi}\ \left\{v'.\ M.f \mapsto_p v * v = v'\right\}}$$

$$\textsc{GlobFieldStore}\ \frac{}{\{M.f \mapsto \_ * \mathit{Init}_M\}\ M.f \leftarrow v\ |^{l}_{\Phi}\ \{v'.\ v' = () * M.f \mapsto v\}}$$

$$\textsc{GlobMethodCall}\ \frac{}{\begin{array}{l}\{\Phi(M).\textsc{Methods}(m).\textsc{Pre}(v) * \mathit{Init}_M\} \\ \quad \texttt{call}\ M.m\ v\ |^{l}_{\Phi} \\ \{v'.\ \Phi(M).\textsc{Methods}(m).\textsc{Post}(v, v')\}\end{array}}\ \Phi(M).\textsc{Methods}(m).\textsc{MLevel} \le l$$

$$\textsc{InitializedI1}\ \frac{\{P * \mathit{Init}_M\}\ !M.f\ |^{l}_{\Phi}\ \{v.\ Q\}}{\{P\}\ !M.f\ |^{l}_{\Phi}\ \{v.\ Q * \mathit{Init}_M\}}\ \Phi(M).\textsc{Level} \le l \qquad \textsc{InitializedI2}\ \frac{\{P * \mathit{Init}_M\}\ M.f \leftarrow v\ |^{l}_{\Phi}\ \{Q\}}{\{P\}\ M.f \leftarrow v\ |^{l}_{\Phi}\ \{Q * \mathit{Init}_M\}}\ \Phi(M).\textsc{Level} \le l$$

$$\textsc{InitializedI3}\ \frac{\{P * \mathit{Init}_M\}\ \texttt{call}\ M.m\ v\ |^{l}_{\Phi}\ \{Q\}}{\{P\}\ \texttt{call}\ M.m\ v\ |^{l}_{\Phi}\ \{Q * \mathit{Init}_M\}}\ \Phi(M).\textsc{Level} \le l$$

Fig. 6. Selected Hoare logic rules for the core language.

Hoare Logic. The triple $\{P\}\ e\ |^{l}_{\Phi}\ \{v.\ Q\}$ for an expression e is parameterized by an initialization level l [8] and set of module specifications Φ. The rules for these triples are given in Figure 6. All triples derivable for a set of module specifications are also derivable for any superset of those specifications. As a consequence, our logic is *modular* in terms of program modules.

A program triple $\{P\}\ e_\phi\ \{v.\ Q\}$ is a specification for the whole program e_ϕ. Its definition is shown in Figure 7. A program is specified if there is a set of module specifications which allows for acyclically specifying the module initializers and methods of all program modules. Each module initializer is required to establish its duplicable and non-duplicable module invariant. Furthermore, all methods are required to establish their specifications, though they may do so at any level.

[8] We implicitly fix the partial order L used for initialization levels.

$$\Phi \in \mathit{ModuleSpecs} \triangleq \mathit{Id} \rightharpoonup \left\{ \begin{array}{l} \textsc{Level} : L \\ \textsc{MInv} : \mathbb{Q} \to \mathit{Asst} \\ \textsc{DupMInv} : \mathit{Asst} \\ \textsc{Methods} : \mathit{Method} \rightharpoonup \left\{ \begin{array}{l} \textsc{MLevel} : L \\ \textsc{Pre} : \mathit{Val} \to \mathit{Asst} \\ \textsc{Post} : \mathit{Val} \times \mathit{Val} \to \mathit{Asst} \end{array} \right\} \end{array} \right\}$$

$$\{P\}\ e_\phi\ \{v.\ Q\} \triangleq \left\{ \begin{array}{l} \textsc{MSpecs} : \mathit{ModuleSpecs} \\ \textsc{MainSpec} : \exists l.\ \{P\}\ e\ |^{l}_{\textsc{MSpecs}}\ \{v.\ Q\} \\ \textsc{InitSpecs} : \forall M \in \mathsf{dom}(\phi).\ \exists l' < \textsc{MSpecs}(M).\textsc{Level}. \\ \qquad \left\{ \ast_{f \in \phi(M).\textsc{Fields}} M.f \mapsto () * \mathit{Init}_M \right\} \\ \qquad \phi(M).\textsc{Init}\ |^{l'}_{\textsc{MSpecs}} \\ \qquad \{\textsc{MSpecs}(M).\textsc{MInv}(1) * \textsc{MSpecs}(M).\textsc{DupMInv}\} \\ \textsc{MethodSpecs} : \forall M \in \mathsf{dom}(\phi).\ \forall m \in \mathsf{dom}(\phi(M).\textsc{MethodDefns}). \\ \qquad \forall v \in \mathit{Val}. \\ \qquad \{\textsc{MSpecs}(M).\textsc{Methods}(m).\textsc{Pre}(v) * \mathit{Init}_M\} \\ \qquad \phi(M).\textsc{MethodDefns}(m)\ v\ |^{\textsc{MSpecs}(M).\textsc{Methods}(m).\textsc{MLevel}}_{\textsc{MSpecs}} \\ \qquad \{v'.\ \textsc{MSpecs}(M).\textsc{Methods}(m).\textsc{Post}(v, v')\} \end{array} \right\}$$

Fig. 7. Program specifications in our core language.

Level Requirements. Perhaps surprisingly, accessing fields of a module M for reading or writing does not directly impose any restrictions on the initialization level. Instead, the user is required to prove that initialization has already started (or the current thread is the initializer) using Init_M. The initializer of M starts with access to this fact, and can thus justify all of its own field accesses. In order to prove that initialization has started, and thus obtain Init_M, outside the initializer of M, a level check *is* required (as in InitializedI1). This can be proved immediately before a definitely-initialized execution point for M. In the case of InitializedI2, this requires that we can only learn Init_M once the right-hand side is a value, since initialization is triggered only at this point.

Methods are allowed to have levels below that of their module, to allow for methods which are called from the module initializer. However, calling these methods from outside the module initializer *would still* trigger initialization, so they can only be safely called at the module level or higher. To call a method at a level below the module level, one must know that initialization will not trigger, which is implied by the Init_M requirement in GlobMethodCall. Similarly to fields, in order to use method calls to justify that initialization will be triggered, the initialization level must be at the module level or higher (InitializedI3).

3.4 Soundness

We show our program logic to be sound in only allowing access to established invariants, and preventing non-strictly-monotonic initializer blocking. Both of these would result in runtime errors in HeapLang under our translational semantics. Therefore, it suffices to show the absence of errors in HeapLang terms generated from source expressions specified by our program logic.

We do this by using the HeapLang weakest-precondition definition generated by Iris for HeapLang terms; these weakest preconditions are automatically sound by construction. Program triples in

our program logic are encoded to weakest-precondition statements on the translated output term, which are then proven. As a result, our program logic ensures source terms are safe to execute under the translational semantics. The full details of this are explained in the extended version of this paper [46].

3.5 Initialization Cycle Freedom

We assert that blocking module accesses are strictly monotonic in terms of the runtime levels involved - so that each initializer can block only on a thread of a strictly lower level. From the soundness of our logic, we know these assertions always succeed for verified programs. By **C2**, we also know that blocking due to initialization only occurs after these assertions. Therefore, at any point during execution, we can strictly order all runtime threads according to which other thread they are blocked on, if any. It is a well-known result that this implies that the directed graph of runtime threads (where edges indicate blocking due to module accesses) is always acyclic [41]. Therefore, our program logic prevents initialization cycles. We do not mechanize this proof beyond showing that accesses are monotonic.

Moreover, since we prove the absence of failed monotonicity assertions in any verified program, we know that our soundness result does not rely on the behavior of the model in the case of an initialization cycle. As such, our result applies to languages with *any* initialization cycle behavior, even if it does not result in a program error. The runtime levels themselves can even be elided from the semantics, since their only purpose is to check the assertions, which always succeed.

4 Implementation

Our technique is applicable to a large set of real languages that support global state, as we show in this section on two complex, real-world languages with very different initialization systems, Java and Go. We also demonstrate that our technique can be integrated into existing (automated) deductive verification tools, VerCors [17], an automated verifier for Java, and Gobra [55] for Go.

Applying our technique to a real-world language requires soundly modeling all constructs in the language that are relevant for initialization and not present in our model. Additionally, if a language provides stronger guarantees (e.g., about the time when initializers execute) than we assume in our logic, we may incorporate these additional guarantees to make our technique more expressiveness and easier to apply.

4.1 Application to Java

First, we will show how our technique can be applied to a language like Java, which performs initialization of static fields lazily on a class level. With very minor differences in some details, everything here also applies to initialization in C# and some other JVM-based languages like Scala.

4.1.1 Additional Language Features and Guarantees. Java offers several important language features that do not exist in the model described so far, but can be easily incorporated:

First, as stated previously, Java's constructors can be modeled as static methods of the same class, reflecting the fact that invoking a constructor will trigger initialization of the new object's class.

Second, Java supports subclassing in a way that is relevant for initialization. In particular, its runtime system guarantees that any class' superclasses are initialized before its own initialization starts. As a result, it is possible to have indirect cyclic initialization dependencies, if for example the initializer of class `A` reads a static field of class `Sub`, and `Sub`'s superclass `Super`'s initializer reads a field of class `A`. Since the read in `A` triggers initialization of `Super` indirectly by triggering the initialization of `Sub`, there is a danger of it causing a deadlock or reading uninitialized data in a similar way as with direct cyclic initialization dependencies. While not necessary for the

soundness of our technique, we can easily prevent these indirect cyclic dependencies by enforcing an additional level constraint: *the initialization level of a subclass must be at least that of its superclass.* As a result, subclass initializers can potentially use (duplicable) invariants of their superclass but not vice versa, as one would expect.

Third, for method overrides in subclasses, as part of ensuring behavioral subtyping [37], we have to ensure that overriding methods open at most the invariants known to hold when the original overridden method is called. This constraint can again be enforced by requiring overridden method's levels to be at least that of their overrides.

Java's initialization system offers some guarantees that go beyond the minimal conditions **C1**-**C3**. In particular, Java guarantees that initialization of a class is performed lazily and will start only when encountering the static field access or method invocation of said class, or before the similarly-triggered initialization of a subclass. Thus, Java would in principle allow identifying the exact point at which an initializer will execute, and therefore, e.g., concluding more precisely what the resulting state of the module is. However, such precise reasoning is inherently non-modular, since it requires knowing what other code has or has not been executed previously.

4.1.2 Sequential Java. While Java is of course a concurrent language, plenty of existing Java code executes only sequentially. Many patterns of accessing global state that are not modularly correct in a concurrent setting are unproblematic in a sequential one. For example, in a concurrent setting, when opening some module A's non-duplicable invariant, we have to make sure that this invariant is not currently being used or broken by other threads (**IC2**). We enforce this condition by requiring the token [Token A]. As a result, initializers generally cannot open non-duplicable invariants of other modules unless they obtain the required token via a lock or other synchronization primitive.

However, in sequential Java, no such concurrent accesses are possible and condition **IC2** holds trivially: There is only one thread, and initializers are triggered at clearly defined points in the execution and running in the same thread. Thus, in this setting, assuming all other conditions are satisfied, it is always sound (including from initializers) to open any module invariants that are not already opened *by the current thread.*

We can adapt our system to this setting as follows: We no longer use tokens to ensure condition **IC2**, and instead rely on initialization levels to track which invariants may be opened. To do this, we observe that since class initialization in Java is triggered only upon the first use of a class, which in turn is required by our logic to only happen in a context whose initialization level is higher than that of the class, throughout the entire call stack of the single thread (including implicitly-triggered static initializers), the initialization level only ever decreases. We can now enforce a simple additional condition, namely that code that executes while the non-duplicable invariant of some class with level l is opened must be correct (according to the Hoare triple definition from Sec. 3.3) with a level *strictly less than* l, i.e., said code may only call methods with a level less than l, open invariants of other modules with a level less than l, and so on. Effectively, this condition is equivalent to using LevelDrop to temporarily reduce the current level whenever an invariant is opened. We also enforce that initializers close all non-duplicable invariants they have opened before they terminate. This condition suffices to maintain the invariant that *if the current level is greater or equal that of any class A, then the invariant of A holds and can be opened.*

As a result, static initializers can directly open the static invariants of classes with lower levels, which means that initializers may access static fields of (some) other classes without requiring any additional synchronization, as one would expect in a setting where no other threads can interfere. Note that this additional freedom does not require changing our core technique and instead directly results from enforcing an additional condition and incorporating the additional guarantees provided by the language (Java) and the setting (the absence of multithreading).

4.1.3 Implementation in VerCors. We have implemented our technique for Java in VerCors [17], an automated, SMT-based verifier for a number of concurrent languages including Java. Our implementation supports advanced Java features like subclassing as described in Sec. 4.1.1; additionally, we have implemented a mode (enabled via a command line flag) that further simplifies reasoning for sequential code, as described in Sec. 4.1.2.

We were able to easily integrate our technique into VerCors, adding the required specification constructs and a single encoding pass that translates specifications and adds all relevent proof obligations. In total, we added or modified a total of ca. 1000 lines of code in VerCors.

We introduced ghost [23] statements that allow users to explicitly open and close module invariants, or open duplicable invariants. Our implementation is able to automatically eliminate the initialization-guarded modality whenever initialization of the relevant class has provably finished.

We require users to annotate classes with duplicable and non-duplicable invariant assertions (with the obvious defaults), and similarly allows annotating methods, constructors, and classes with initialization levels (in the form of natural number literals) as described in Sec. 2. We choose default values for initialization levels to be 1 for classes and methods, and $l - 1$ for the initializer of a class with level l. This choice ensures that in code that does not perform or interact with module initialization or static fields, all level constraints are always satisfied, meaning that no additional annotation overhead is imposed for code that does not require it.

4.2 Application to Go

Initialization in Go works fundamentally differently from Java; Go offers additional guarantees not present in Java, but also poses a new challenge. Nevertheless, our technique applies.

4.2.1 Additional Language Features and Guarantees. First, unlike Java's lazy initialization, Go performs initialization eagerly. As a result, one may soundly assume that all *packages* (Go's notion of modules) have initialized at the beginning of the `main` function.

More importantly, the Go compiler guarantees that the package import relation forms a DAG. Thus, cyclic initialization dependencies between packages are ruled out by the compiler. We can automatically and modularly infer an assignment of initialization levels per package from the import relation: if package A imports B, the level of B must be lower than A's level. In addition, *by default*, all methods in a package may adopt the package level. With these conditions, level annotations are necessary only in methods that may be called from the package initializer, which greatly reduces the number of required annotations.

Since all packages are guaranteed to be initialized when `main` starts, there is no longer a need to open and close non-duplicable invariants; instead, conceptually, all non-duplicable invariants can be opened at the beginning of `main` and remain open, which also fully eliminates the need for module tokens. Finally, package initialization in Go is guaranteed to run sequentially, package by package, in reverse order of imports. Thus, similarly to sequential programs in Java (Sec. 4.1.2), package initializers in Go may open the invariants of imported packages, as long as they close them by the end of the initializer. As a result, when applying our logic to Go, we end up with significantly simpler reasoning just by incorporating language guarantees into the existing technique.

On the other hand, Go also poses a new challenge by allowing multiple initializers in the same package, spread across different files of the package. This leads to the infamous *Static Initialization Order Fiasco* [5], originally identified in C++, where the order in which global variables are initialized is determined by the order in which files are passed to the compiler. This may cause unexpected problems as programs evolve. Files `file2.go` (Listing 5) and `file3.go` (Listing 6) provide a contrived, yet demonstrative example. Together, these files constitute the package `main`, and each of these files has an initializer (function `init`). If we follow the Go team's recommendation

of passing files in the lexical name order to the compiler [10], the initializer in `file2.go` runs before the one in `file3.go` and the program runs successfully to completion. If we rename `file3.go` to `file1.go` and pass the files in lexicographic order, the two initializers run in the reversed order, and a `panic` is caused in line 4 of `file1.go`, as we try to put a key/value pair in a `nil` map.

```
1 package main
2
3 var m map[int]int
4
5 func init() {
6   m = make(map[int]int)
7 }
```

Listing 5. File `file2.go`

```
1 package main
2
3 func init() {
4   m[0] = 1
5 }
6
7 func main() {}
```

Listing 6. File `file3.go`

A simple solution (in the spirit of CSL [44]) is to over-approximate all possible orders in which the initializers of a package may run by modeling each of them as a thread that executes in parallel with the other initializers in the same package. This treatment requires that the initializer of a file does not access global variables declared in other files, which can be enforced by defining the precondition of the initializer of each file to contain only the permissions to the global variables declared in that file. One can then simply ensure that the package invariant of a package is established after all its initializer threads conclude.

4.2.2 Implementation in Gobra. We implemented our technique in Gobra [55], a deductive program verifier for Go. In total, our extension comprises around 900 new lines of code. Our implementation incorporates the simplifications afforded by the additional guarantees listed in Sec. 4.2.1.

For every package, we introduce two initialization levels: the package level and the level of the package initializer. The latter is constrained to be lower than the former and higher than all the package levels of the imported packages. We introduce an annotation `mayInit` for methods that should be verified with the level of the initializer, as they may be called during initialization. All other methods are verified at the package level.

We tested our implementation in Gobra to guarantee it implements the Go language specification correctly. In the process, we identified a mismatch between the intended semantics for initialization and the compiled program, which led us to find a previously unknown bug in the Go compiler [2]. This bug was caused by an optimization that rewrites the initializing expression for a global variable x without taking into account the effect that functions called during initialization might have on the value of x. In some cases, this bug caused the compiled program to store unexpected values in global variables after package initialization.

5 Evaluation

To demonstrate both the flexibility and the real-world applicability of our technique, we have evaluated it in two steps. First, we verified a number of small case studies representing important usage patterns for global state and initialization, shown in Table 1, using our implementations in Gobra and VerCors. Second, we applied our technique to a large-scale verification project of a real-world Go code base [47], for which we specified and verified all usages of global state using our technique as implemented in Gobra.

5.1 Case Studies

We collected a set of 11 examples representing important coding patterns or showcasing specific aspects of initialization to demonstrate that our technique is sufficiently flexible to verify different

Table 1. Verified case studies with total lines of code (LOC), lines of proof annotations (Ann.), the number of those lines that is specific to the declaration of module invariants, levels and opening and closing module invariants (MInv.), as well as the verification time in seconds (Time).

	Go				Java			
Name	LOC	Ann.	MInv.	Time [s]	LOC	Ann.	MInv.	Time [s]
Byte	22	27	8	1.77	18	17	7	1.91
Client*	15	26	9	1.02	13	18	11	1.58
Contexts	35	20	5	1.17	38	21	5	1.34
Counter	24	55	8	1.44	43	59	15	2.03
Fib	14	21	6	0.88	19	18	6	1.41
Fib Conc.	21	29	9	1.37	25	28	9	1.79
Finalizer	47	75	21	33.46	57	59	21	21.35
Names*	42	25	4	2.25	35	24	11	2.26
Singleton*	21	24	2	0.61	24	20	10	4.24
Subclass	-	-	-	-	26	10	9	1.35
Theory*	-	-	-	-	27	23	11	1.69

idiomatic uses of global state and initializers. We manually translated all applicable examples to both Java and Go. Since, for example, Go does not have support for subclassing, we only verified a Java version of that example. For examples marked with an asterisk, we verified the Java code using the version of our technique adapted for sequential Java discussed in Sec. 4.1.2, since the code is not thread-safe and thus not correct in a concurrent setting.

Fib is the example from Listing 1, specified and verified exactly as shown, and **Fib Conc.** is a thread-safe version of `Fib`. **Counter** is the example from Listing 2, adapted so that all calls to `getAndInc` are specified to return a fresh value. **Byte** and **Finalizer** are (simplified) versions of code found in the Java standard library and the OpenJDK Java runtime, respectively. The `Byte` class allocates a global pool of (immutable) `Byte` objects (one for every possible `byte` value) upon initialization that can then be used by all clients via a duplicable invariant. The `Finalizer` class maintains a global doubly-linked list of `Finalizer` instances for objects that are being garbage collected and whose `finalize` methods are to be invoked before they are deleted, which we verify using a non-duplicable invariant. **Singleton** is a generic implementation of (a non-thread-safe version of) the singleton pattern, using a static method to initialize a static field on first access. We use a duplicable invariant to give all clients read access. The **Subclass** example declares several classes and subclasses, and illustrates that our technique correctly rejects (using initialization levels) classes which introduce cyclic dependencies between static initializers even when they are indirect via superclass initialization, but allows similar uses that do not lead to cycles. **Client** and **Theory** are examples from Leino and Müller's work [35]. The former uses a global counter that is read and incremented every time a new instance of the class is allocated, thus providing a unique identifier for each instance. The latter shows that their system can be used in a setting with subclassing where subclasses must satisfy behavioral subtyping [37]; we show that we can support the same setting. **Contexts** and **Names** are two examples from the evaluation of Liu et al.'s static analysis [39]. They are simplified extracts from the Scala compiler which are rejected by their static analysis because they do not satisfy its initialization-time irrelevance property, but which (as stated by their paper) constitute cases where the program can nevertheless be regarded as correct. We verify both examples, i.e., prove that module invariants are established and maintained correctly, showing that the presence of user-defined invariants allows us to be more precise than a static analysis.

Table 1 shows the size, number of overall proof annotations, and number of annotations specific to initialization (i.e., module invariants, levels, and ghost code for opening and closing module invariants) for each example, as well as the verification time. Verification time was measured on a laptop with an Apple M2 Max and 64GB of RAM and averaged over 10 runs. The amount of initialization-specific annotations is low throughout: for our case studies, which emphasize complex uses of global state, they are, at most, 22% of the total code for the case studies in Go, and at most 25% for those in Java, with the exception of the `Client` class, where 35% of the total code consists of initialization-specific annotations. The higher annotation count for this example results from having to manually provide levels to this class and all its methods, as the default levels do not satisfy the level requirements. For the other case studies in Java, a substantial amount of the proof can be performed using the default values for levels.

Our Go case studies tend to require fewer initialization-specific annotations than the ones written in Java: Gobra requires fewer level annotations and no annotations to open and close non-duplicable invariants. Apart from these differences, the overall proof structure and the module invariants for the Go and Java versions of the same code are generally similar, in spite of the different initialization semantics of both languages, showing that our technique offers a general way of reasoning about global state and initialization that effectively abstracts over semantic details.

For the most part, our examples verify in a few seconds, which is in line with the verification time in both VerCors and Gobra for programs of similar size and complexity that do not use global state and initialization. This suggests that the proof obligations imposed by our technique are not more difficult to prove than those that are not related to module initialization. In our evaluation, the `Finalizer` example stands out for taking multiple tens of seconds to verify in both VerCors and Gobra. The cause of the long verification time is a method using a module invariant that makes heavy use of quantifiers and sequences, which are challenging for SMT solvers to reason about. This is a known issue unrelated to initialization-specific proof obligations.

5.2 Real-World Go Code Base

To demonstrate that our technique scales beyond small case studies, we used it to specify and verify *all uses* of global state in the Go implementation of the router for the SCION internet architecture [48]. This code base comprises around 4.7k lines of Go code that was not written with verification in mind, and uses complex features of the Go language such as concurrency, interfaces and closures. We integrated our technique into an existing large-scale verification effort to verify the entire router code [47], proving memory safety properties as well as protocol compliance.

The code base uses global state mainly for two purposes: (1) to store, in a mutable slice, the metadata about the types of network paths the router is aware of, and (2) to store immutable values that are reused throughout the codebase, like error messages. We capture properties about the former using non-duplicable invariants and properties about the later using duplicable invariants.

In total, we specified properties of global state and package initializers for eight packages, using duplicable invariants for all eight and non-duplicable invariants for three of them. The resulting annotation overhead was low, with a total of 67 lines of duplicable invariants and 27 lines of non-duplicable invariants. Due to the simplifications described in Section 4.2.2, little additional ghost code was required, with only 30 additional lines of initialization-related ghost code or annotations.

Applying our technique to this code base allowed us to replace previously unjustified assumptions of properties of the global state by sound, principled reasoning. Compared to the previous version of the code base, our changes did not slow down verification significantly. In our experiments, verifying the Go router took on average 45m46s without verifying the proof obligations related to initialization and opening module invariants, and 47m05s with them enabled, an increase of less than 3%. This demonstrates our approach scales to larger verification efforts.

6 Related Work

Module invariants have been suggested as a way to specify the global variables of modules in various projects. The Java Modeling Language (JML), a specification language for Java adopted by various deductive program verifiers including KeY [12], Krakatoa [40] and OpenJML [21], proposes the use of *static invariants* to specify the expected values in static fields [1]. However, they do not prescribe a way to modularly check the establishment of these invariants or state at what program points they may be soundly assumed. To the best of our knowledge, no deductive verifier for mainstream programming languages supported module invariants prior to this work, including the aforementioned ones based on JML, other Java verifiers like VerCors [17] and VeriFast [27], and verifiers for other languages like Gobra [55] for Go, and Leon [15] for Scala.

Leino and Müller [35] propose using static invariants to reason about static fields in a programming language based on a *sequential* subset of Java and C#. They combine a strict ownership model for objects (see below) with a partial order on classes, the *validity ordering*, which prescribes the order in which classes are initialized. A notion of transitive validity allows them to write concise specifications. Unfortunately, this validity ordering requires that the initialization of a class runs *before* that of its superclasses. This strong restriction is not satisfied by the semantics of languages like Java and C#. Their approach does not handle concurrency and uses a restrictive ownership model that rules out referencing an object from two classes, even when sharing is safe, e.g., because the object is immutable. In contrast, our approach handles subclassing as it occurs in mainstream languages, supports concurrency, and leverages the full flexibility of separation logic to allow sharing when it is safe.

Jacobs et al. [28] propose a technique for modular verification of static invariants in multi-threaded programs written in a Java-like language. Their approach requires that *all accesses* to static fields are guarded by a class-level lock, unless *all fields* of that class are immutable. Our technique, by contrast, supports a wide variety of synchronization primitives and requires none when unsynchronized accesses are safe. Jacobs et al.'s approach prevents deadlocks, including those caused by cyclic initialization dependencies, by imposing an acquisition order on all locks. Their approach requires that module imports are acyclic, which is enforced by runtime checks. In contrast, we use initialization levels to prevent cyclic initialization dependencies statically, and allow for cyclic module imports.

Summers et al. [51] propose an extension to the *Visibility Technique* from Müller et al. [43] for verifying static invariants in a *sequential* Java-like programming language. Their approach imposes *effect annotations* on methods, consisting of all classes whose static methods may be transitively called from that method. To alleviate the need for these annotations, they introduce levels, which are similar to our initialization levels, but only apply to classes. In their approach, classes at a lower level may not mention others at a higher level, which prevents mutually defined classes.

All approaches mentioned so far [28, 35, 51] are based on restrictive ownership disciplines [14, 34, 43]. To the best of our knowledge, our technique is the first based on concurrent separation logic [30, 44], which is more expressive than the aforementioned disciplines, and for which there is an extensive body of work on reusable reasoning principles for different concurrency primitives like locks [20, 24, 26] and channels [25, 36, 54].

The Boogie verifier [13] supports global variables, which may be declared with `where`-clauses [3] that specify properties that are assumed whenever the variable is havocked. However, these clauses play a different role than our module invariants. They are used to encode trusted assumptions, rather than to express verified properties. Mutations of global variables are tracked via `modifies`-clauses in method specifications, whereas we rely on separation logic to reason about side effects.

Liu et al. [39] propose a static analysis to identify common issues in the initialization of Scala's singleton objects based on the principle of *initialization-time irrelevance*, which expresses that the state of a global singleton object at the end of its initialization should not depend on when that object is initialized. In contrast, we allow a module's state after initialization to depend on the initialization time *only if* its invariant is guaranteed to hold. Their technique does not consider user-provided invariants and imposes cumbersome restrictions, e.g., the static initializer of a singleton object must not perform any (even synchronized!) accesss to mutable global state of another.

Internally, Liu et al.'s analysis makes use of ideas commonly used when reasoning about object initialization and instance invariants, for which various solutions have been proposed [16, 38, 52]. These works do not address module initialization and instead identify conditions under which objects may safely escape constructors and non-initialized objects may be accessed. While there are commonalities with module initialization, reasoning about the latter poses distinct challenges: Module initialization starts and ends at modularly unknown execution points (unlike constructors, which are invoked explicitly and eventually return), and global state can be accessed from anywhere in the code (unlike objects under construction, which must be passed around explicitly).

7 Conclusion

We proposed a practical technique for specifying and verifying programs that make use of global data. Our technique can be applied to concurrent programs, and to a wide range of programming languages with varying semantics, including lazy and eager initialization. It allows specifying modules with invariants, and uses initialization levels, a novel specification construct that can be used to prevent cyclical initialization cycles and to identify which module invariants are guaranteed to hold. Furthermore, we show it can be automated in state-of-the-art deductive program verifiers.

Acknowledgments

The authors would like to thank Ralf Jung for his helpful input, especially on the use of Iris.

Data-Availability Statement

The implementation of our technique in VerCors and Gobra, our case studies, instructions on how to reproduce our evaluation, and our formalization in Iris are provided as an artifact accompanying this paper [45].

References

[1] 2013. JML Reference Manual - Type Specifications. Retrieved 25 November 2024 from https://www.cs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_8.html#SEC68
[2] 2022. cmd/compile: global variable initialization done in unexpected order. Retrieved 16 October, 2024 from https://github.com/golang/go/issues/51913
[3] 2023. The Boogie IVL Language Reference - Global Variables. Retrieved 25 November, 2024 from https://boogie-docs.readthedocs.io/en/latest/LangRef.html#global-variables
[4] 2023. Crate `lazy_static`. Retrieved 16 December, 2023 from https://docs.rs/lazy_static/latest/lazy_static/
[5] 2023. Static Initialization Order Fiasco. Retrieved 22 December, 2023 from https://en.cppreference.com/w/cpp/language/siof
[6] 2024. Java Language Specification. Chapter 12: Execution. Retrieved 11 October, 2024 from https://docs.oracle.com/javase/specs/jls/se7/html/jls-12.html
[7] 2024. Java™ Platform, Standard Edition 8 API Specification - Class Logger. Retrieved 31 October, 2024 from https://docs.oracle.com/javase/8/docs/api/java/util/logging/Logger.html
[8] 2024. OpenJDK's implementation of Java's Byte class. Retrieved 7 October, 2024 from https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/lang/Byte.java
[9] 2024. Static Constructors (C# Programming Guide). Retrieved 11 October, 2024 from https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-constructors

[10] 2025. The Go Programming Language Specification, Package initialization. Retrieved 8 March, 2025 from https://go.dev/ref/spec#Program_initialization

[11] 2025. HeapLang overview. Retrieved 24 January, 2025 from https://gitlab.mpi-sws.org/iris/iris/-/blob/master/docs/heap_lang.md

[12] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification - The KeY Book - From Theory to Practice*. Lecture Notes in Computer Science, Vol. 10001. Springer. https://doi.org/10.1007/978-3-319-49812-6

[13] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. 2006. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures 4*. Springer, 364–387. https://doi.org/10.1007/11804192_17

[14] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. 2004. Verification of Object-Oriented Programs with Invariants. *Journal of Object Technology* 3, 6 (June 2004), 27–56. https://doi.org/10.5381/jot.2004.3.6.a2 Workshop on Formal Techniques for Java-like Programs (FTfJP), ECOOP 2003.

[15] Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. 2013. An overview of the Leon verification system: verification by translation to recursive functions. In *Proceedings of the 4th Workshop on Scala* (Montpellier, France) *(SCALA '13)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2489837.2489838

[16] Clément Blaudeau and Fengyun Liu. 2022. A conceptual framework for safe object initialization: a principled and mechanized soundness proof of the Celsius model. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 151 (Oct. 2022), 29 pages. https://doi.org/10.1145/3563314

[17] Stefan Blom and Marieke Huisman. 2014. The VerCors Tool for Verification of Concurrent Programs. In *FM 2014: Formal Methods*, Cliff Jones, Pekka Pihlajasaari, and Jun Sun (Eds.). Springer International Publishing, Cham, 127–131. https://doi.org/10.1007/978-3-319-06410-9_9

[18] Egon Börger and Wolfram Schulte. 2000. Initialization problems for Java. *Software-Concepts & Tools* 19 (2000), 175–178. https://doi.org/10.1007/s003789900003

[19] John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis*, Radhia Cousot (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 55–72. https://doi.org/10.1007/3-540-44898-5_4

[20] Alexandre Buisse, Lars Birkedal, and Kristian Støvring. 2011. Step-Indexed Kripke Model of Separation Logic for Storable Locks. *Electronic Notes in Theoretical Computer Science* 276 (2011), 121–143. https://doi.org/10.1016/j.entcs.2011.09.018 Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII).

[21] David R. Cok. 2011. OpenJML: JML for Java 7 by Extending OpenJDK. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 472–479. https://doi.org/10.1007/978-3-642-20398-5_35

[22] Werner Dietl and Peter Müller. 2005. Universes: Lightweight Ownership for JML. *J. Object Technol.* 4, 8 (2005), 5–32. https://doi.org/10.5381/JOT.2005.4.8.A1

[23] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. 2014. The Spirit of Ghost Code. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 1–16. https://doi.org/10.1007/978-3-319-08867-9_1

[24] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. 2007. Local Reasoning for Storable Locks and Threads. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 19–37. https://doi.org/10.1007/978-3-540-76637-7_3

[25] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2019. Actris: session-type based reasoning in separation logic. *Proc. ACM Program. Lang.* 4, POPL, Article 6 (Dec. 2019), 30 pages. https://doi.org/10.1145/3371074

[26] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle semantics for concurrent separation logic. In *Proceedings of the Theory and Practice of Software, 17th European Conference on Programming Languages and Systems* (Budapest, Hungary) *(ESOP'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 353–367. https://doi.org/10.1007/978-3-540-78739-6_27

[27] Bart Jacobs, Jan Smans, and Frank Piessens. 2010. A Quick Tour of the VeriFast Program Verifier. In *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6461)*, Kazunori Ueda (Ed.). Springer, 304–311. https://doi.org/10.1007/978-3-642-17164-2_21

[28] Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte. 2007. A Simple Sequential Reasoning Approach for Sound Modular Verification of Mainstream Multithreaded Programs. *Electronic Notes in Theoretical Computer Science* 174, 9 (2007), 23–47. https://doi.org/10.1016/j.entcs.2007.04.005 Proceedings of the Thread Verification Workshop (TV 2006).

[29] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[30] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 637–650. https://doi.org/10.1145/2676726.2676980

[31] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.* 2, ICFP, Article 77 (July 2018), 30 pages. https://doi.org/10.1145/3236772

[32] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 205–217. https://doi.org/10.1145/3009837.3009855

[33] K. Rustan M. Leino. 2023. *Program Proofs*. MIT Press.

[34] K. Rustan M. Leino and Peter Müller. 2004. Object Invariants in Dynamic Contexts. In *ECOOP 2004 – Object-Oriented Programming*. https://doi.org/10.1007/978-3-540-24851-4_22

[35] K. Rustan M. Leino and Peter Müller. 2005. Modular Verification of Static Class Invariants. In *FM 2005: Formal Methods*. Springer Berlin Heidelberg, Berlin, Heidelberg, 26–42. https://doi.org/10.1007/11526841_4

[36] K. Rustan M. Leino, Peter Müller, and Jan Smans. 2010. Deadlock-Free Channels and Locks. In *Programming Languages and Systems*, Andrew D. Gordon (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 407–426. https://doi.org/10.1007/978-3-642-11957-6_22

[37] Barbara Liskov and Jeannette M. Wing. 1994. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (1994), 1811–1841. https://doi.org/10.1145/197320.197383

[38] Fengyun Liu, Ondřej Lhoták, Aggelos Biboudis, Paolo G. Giarrusso, and Martin Odersky. 2020. A type-and-effect system for object initialization. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 175 (Nov. 2020), 28 pages. https://doi.org/10.1145/3428243

[39] Fengyun Liu, Ondřej Lhoták, David Hua, and Enze Xing. 2023. Initializing Global Objects: Time and Order. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 268 (Oct. 2023), 28 pages. https://doi.org/10.1145/3622844

[40] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. 2004. The KRAKATOA tool for certificationof Java/Java Card programs annotated in JML. *J. Log. Algebraic Methods Program.* 58, 1-2 (2004), 89–106. https://doi.org/10.1016/J.JLAP.2003.07.006

[41] Albert R. Meyer and Adam Chlipala. 2015. Partial Orders. In *Mathematics for Computer Science—MIT Course No. 6.042J*. MIT OpenCourseWare: Massachusetts Institute of Technology, 336–338. https://ocw.mit.edu/courses/6-042j-mathematics-for-computer-science-spring-2015/ Accessed March 25, 2025.

[42] Bertrand Meyer. 1992. *Eiffel: The Language*. Prentice Hall.

[43] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. 2006. Modular invariants for layered object structures. *Sci. Comput. Program.* 62, 3 (Oct. 2006). https://doi.org/10.1016/j.scico.2006.03.001

[44] Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR 2004 - Concurrency Theory*, Philippa Gardner and Nobuko Yoshida (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 49–67. https://doi.org/10.1007/978-3-540-28644-8_4

[45] João Pereira, Isaac van Bakel, Patricia Firlejczyk, Marco Eilers, and Peter Müller. 2025. Artifact for "Modular Reasoning about Global Variables and Their Initialization". https://doi.org/10.5281/zenodo.15756716

[46] João Pereira, Isaac van Bakel, Patricia Firlejczyk, Marco Eilers, and Peter Müller. 2025. Modular Reasoning about Global Variables and Their Initialization (Extended Version).

[47] João C. Pereira, Tobias Klenze, Sofia Giampietro, Markus Limbeck, Dionysios Spiliopoulos, Felix A. Wolf, Marco Eilers, Christoph Sprenger, David Basin, Peter Müller, and Adrian Perrig. 2024. Protocols to Code: Formal Verification of a Next-Generation Internet Router. arXiv:2405.06074 [cs.CR] https://arxiv.org/abs/2405.06074

[48] Adrian Perrig, Pawel Szalachowski, Raphael M. Reischuk, and Laurent Chuat. 2017. *SCION: A Secure Internet Architecture*. Springer. https://doi.org/10.1007/978-3-319-67080-5

[49] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*. IEEE Computer Society, USA, 55–74. https://doi.org/10.1109/LICS.2002.1029817

[50] Alexander J. Summers and Sophia Drossopoulou. 2013. A Formal Semantics for Isorecursive and Equirecursive State Abstractions. In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7920)*, Giuseppe Castagna (Ed.). Springer, 129–153. https://doi.org/10.1007/978-3-642-39038-8_6

[51] A. J. Summers, S. Drossopoulou, and P. Müller. 2009. Universe-Type-Based Verification Techniques for Mutable Static Fields and Methods. *Journal of Object Technology (JOT)* 8, 4 (June 2009). https://doi.org/10.5381/jot.2009.8.4.a4

[52] Alexander J. Summers and Peter Mueller. 2011. Freedom before commitment: a lightweight type system for object initialisation. *SIGPLAN Not.* 46, 10 (Oct. 2011), 1013–1032. https://doi.org/10.1145/2076021.2048142

[53] The Rocq Development Team. 2025. The Rocq Prover Reference Manual – Release 9.0.0. https://rocq-prover.org/doc/V9.0.0/refman.

[54] Jules Villard, Étienne Lozes, and Cristiano Calcagno. 2009. Proving Copyless Message Passing. In *Programming Languages and Systems*, Zhenjiang Hu (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 194–209. https://doi.org/10.1007/978-3-642-10672-9_15

[55] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, and Peter Müller. 2021. Gobra: Modular Specification and Verification of Go Programs. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 367–379. https://doi.org/10.1007/978-3-030-81685-8_17

Received 2025-03-26; accepted 2025-08-12