Initializing Global Objects: Time and Order

FENGYUN LIU, Oracle Labs, Switzerland
ONDŘEJ LHOTÁK, University of Waterloo, Canada
DAVID HUA, University of Waterloo, Canada
ENZE XING, University of Waterloo, Canada

Object-oriented programming has been bothered by an awkward feature for a long time: *static members*. Static members not only compromise the conceptual integrity of object-oriented programming, but also give rise to subtle initialization errors, such as reading non-initialized fields and deadlocks.

The Scala programming language eliminated static members from the language, replacing them with *global objects* that present a unified object-oriented programming model. However, the problem of global object initialization remains open, and programmers still suffer from initialization errors.

We propose *partial ordering* and *initialization-time irrelevance* as two fundamental principles for initializing global objects. Based on these principles, we put forward an effective static analysis to ensure safe initialization of global objects, which eliminates initialization errors at compile time. The analysis also enables static scheduling of global object initialization to avoid runtime overhead. The analysis is modular at the granularity of objects and it avoids whole-program analysis. To make the analysis explainable and tunable, we introduce the concept of *regions* to make context-sensitivity understandable and customizable by programmers.

CCS Concepts: • **Software and its engineering** → **Object oriented languages**; **Classes and objects**.

Additional Key Words and Phrases: initialization safety, initialization-time irrelevance, region context

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

1 INTRODUCTION

For ease of programming, most programming languages make it possible to declare global variables. Global variables may be either mutable or immutable, and they can hold either primitive values (e.g., integers) or non-primitive values (e.g., pointers). This is the case not only for C, but also for the latest languages such as Rust[1] and WebAssembly.[2]

To align with its design philosophy of modularity, object-oriented programming introduced *static fields* to support the need for global variables, as we have seen in most object-oriented languages, such as Java, C#, C++ and Swift, but also in dynamically-typed languages, such as Python and Ruby. The idea of static fields naturally extends to *static methods*. Some languages go even further. For example, Swift supports overriding static properties (similar to fields) but not static methods, and Ruby supports overriding static methods.

[1]See 6.10 Static items in *The Rust Reference* and the macro lazy_static.
[2]See Globals in *WebAssemply Specification* [W3C 2022].

Authors' addresses: Fengyun Liu, fengyun.liu@oracle.com, Oracle Labs, Zurich, Switzerland; Ondřej Lhoták, olhotak@uwaterloo.ca, University of Waterloo, Waterloo, Canada; David Hua, david.hua@uwaterloo.ca, University of Waterloo, Waterloo, Canada; Enze Xing, e2xing@uwaterloo.ca, University of Waterloo, Waterloo, Canada.

2475-1421/2023/10-ART268
https://doi.org/10.1145/3622844

Static fields not only compromise the conceptual integrity of object-oriented programming, but also give rise to subtle initialization errors, such as reading non-initialized fields, deadlocks and unsound compiler optimizations [Börger and Schulte 2000].

The Scala programming language [Odersky 2019] improves the situation by presenting a unified object-oriented programming model with the introduction of *global objects*, which eliminate static fields and static methods from the language. Semantically, a global object can be thought of as a lazily initialized singleton instance of its underlying class with the same body.

Despite the improvement in conceptual integrity, Scala programmers still suffer from subtle initialization errors. The Scala bug tracker contains bug reports about reading uninitialized fields and deadlocks related to global objects, which have been open for more than 8 years (Appendix C).

Our work tries to address the initialization safety problem related to global objects. The following code illustrates the problem of initialization errors:[3]

```
1 object A:
2   val a: Int = B.b + 10
3 object B:
4   def foo(): Int = A.a * 2
5   val b: Int = foo()
```

According to the semantics of Scala, the objects `A` and `B` are lazily initialized the first time they are accessed. Therefore, the initialization of the object `A` would trigger the initialization of `B` due to the code `B.b` at line 2, and vice versa. As a result, either at line 2 or at line 4, the code reads an uninitialized field.

In the presence of concurrency, to make sure that an object is not initialized twice, the language semantics usually require a lock to be acquired when initialization of an object starts and released when initialization completes. If the initializations of objects `A` and `B` are triggered from two different threads simultaneously, a *deadlock* happens!

The contributions of our work are as follows:

(1) We propose *partial ordering* and *initialization-time irrelevance* as fundamental principles for safe initialization of global objects. More generally, the principles can be used in the design of modules in programming languages (Section 2.2 and Section 2.3).
(2) We develop an effective static analysis to ensure safe initialization of global objects (Section 3 and Section 4). The algorithm tracks *ownership of mutable state* to enforce initialization-time irrelevance. It introduces the concept of *regions* to make *context-sensitivity* understandable and tunable by programmers for complex initialization code (Section 4.3).
(3) We implement the static analysis based on the Scala 3 compiler. Our implementation resolves all existing bug reports related to initialization of global objects (Section 5).
(4) We conduct a case study on the practicality of the principles of partial ordering and initialization-time irrelevance based on a large real-world Scala project (Section 6).

Note that while we present the core ideas and the algorithms of our work in a formal language for clarity, we leave the development of meta-theories to future work.

2 CHALLENGES AND IDEAS

In this section we survey the challenges in statically checking the safe initialization of global objects. Then, we introduce the main ideas of our solution informally.

[3]Unless noted otherwise, code examples will be in Scala. We use the Scala 3 indented syntax for clarity and to save space.

2.1 Challenges

There are several challenges to developing a sound and practical system to ensure safe initialization of global objects.

Challenge 1: Avoid whole-program analysis. Whole-program analysis should be avoided: it is slow and it is not easy to explain errors to end users due to the long context from the program entry point to the place where the error happens. Instead, the analysis should examine only the parts of the program that may be called from the initializers of global objects. However, the following example shows that whole-program analysis cannot be avoided unless we enforce some restrictions on global object initialization code.

Suppose we have a project that defines a class `X` and objects `A` and `B`:

```
1 // project 1
2 class X { def foo(): Int = 10 }
3 object A { var a: X = new X() }
4 object B { val b: Int = A.a.foo() } // B -> A
```

In the code above, the initialization of object `B` depends on initialization of object `A` due to the code `A.a` at line 4. It seems that regardless of whether `A` or `B` is accessed first in the program, they can be initialized without any issue according to the lazy initialization semantics.

Similarly, suppose there is another project that defines a class `Y` and object `C`, where `C` depends on the object `B` due to the code `B.b`, as the following code shows:

```
1 // project 2
2 class Y extends X { override def foo() = C.c }
3 object C { val c: Int = B.b }     // C -> B
```

If objects `A` and `B` can be initialized without problem, object `C` should also be initialized without any issue, because now we have the dependency $C \longrightarrow B \longrightarrow A$: there are no cyclic dependencies.

However, suppose in another project, which is an application, we have the following code in the main method:

```
1 // project 3
2 def main = { A.a = new Y(); C } // C -> B -> C
```

Now, suddenly, the initialization of object `B` depends on object `C` ($B \longrightarrow C$). This is because the field `A.a` points to an instance of class `Y`; consequently, the call `A.a.foo()` in object `B` executes the method `foo` defined in the class `Y`, which depends on class `C`. But we also have $C \longrightarrow B$ due to the code `B.b` in the body of object `C`. This creates a cycle in the dependency graph!

The takeaway of this example is that if we stay with the current design of global objects, we cannot check global objects individually — whole-program analysis cannot be avoided.

Challenge 2: Explainability. While it is well-known that static analysis can be used to enhance expressiveness of a system without incurring syntactic overhead on programmers, making the analysis *explainable* is a challenge. This is due to the fact that the rules of static analysis are prone to optimization changes. The complexity and instability of the rules make them difficult for programmers to learn and reason about in daily programming. This is not a problem if the analysis is used to justify compiler optimizations, but it is a big usability issue when the analysis is used to help programmers construct better programs by checking and reporting errors.

Explainability requires that the abstractions and rules used by the analysis should be simple and stable, just like types and typing rules in type systems. This way, we can avoid the syntactic verbosity of type systems while reaping the same benefits of safety.

One obscure concept in program analysis is *context-sensitivity* [Smaragdakis et al. 2011]. Context-sensitivity is also one of the major reasons which make an analysis less explainable to programmers. We introduce a more user-friendly concept, called *regions*, to make context-sensitivity more explainable and tunable for complex use cases (Section 4.3).

2.2 Maintaining Order in Initialization

The problems in the initialization of global objects are related to ordering. The first ordering problem involves reading a field before its initialization, as the following code demonstrates:

```
1 object A:
2   def foo(): Int = b * 2
3   val b   : Int = foo() + 10
```

We detect this kind of error by abstractly evaluating the initialization code in the global object with respect to its initialization state. If an uninitialized field may be read, the checker reports an error. Due to the fact that the global object may be accessed anywhere in the program, all code reachable from a global object's initializer needs to be abstractly evaluated.

The second ordering problem involves cyclic object initialization, as the following example demonstrates:

```
1 object A:
2   val a: Int = 10
3   val n: Int = B.b + 10
4 object B:
5   val b: Int = A.a * 2
```

In the above, the initialization of object `A` would trigger initialization of object `B`, and vice versa, a good recipe for introducing deadlocks in the presence of concurrency. The deadlock happens if one thread holds the lock for the initialization of object `A`, while the other thread holds the lock for the initialization of object `B`. Neither can progress as each waits for the lock owned by the other.

Another consideration to disallow such code patterns is *information hiding*. The code above exposes the partial initialization state of one global object to another, which creates a subtle and brittle contract between them.

To prevent such cycles, we impose that *initialization of global objects must follow a partial order*. This implies that the initialization dependencies of global objects form a *directed acyclic graph* (DAG). To enforce the partial ordering, the checker takes each not-yet checked global object as an entry point, and then proceeds with the abstract evaluation of initialization code in the object. It is in essence a depth-first search of all other global objects that it depends on. The checker maintains a list of objects currently under initialization to detect cycles in the dependency graph.

However, the checker takes care to allow trivial self-cycles, as the following code shows:

```
1 class A(x: Int) { def foo(): Int = B.m }
2 object B:
3   val m: Int = 20
4   val n: Int = new A(10).foo()
```

In the code above, the initialization of the object `B` would need to access itself in the method `A.foo`. In Scala, this code pattern is quite common, where the class is nested in the object and its instances are created as value members of the object. In such cases, there is no need to worry about deadlocks because the cycle only involves one object. Therefore, such trivial self-cycles are allowed.

To clarify, here partial ordering only requires that the initialization of global objects form a partial order. It does not mean that the compilation order of global objects needs to form a partial order. The global objects may refer to each other in their member methods as long as there are no cycles in the initialization code. In fact, *acyclic compilation dependencies in languages such as OCaml are neither necessary nor sufficient* for safe initialiation of global objects.

2.3 Initialization-Time Irrelevance and Ownership

Reflecting on the principles of safe initialization for global objects, we find that it is natural to assume the following invariant:

> **The state of a global object at the end of its initialization should not depend on when the global object is initialized**.

We call this principle *initialization-time irrelevance*. To enforce initialization-time irrelevance, we need to track *ownership* of mutable state. This can be demonstrated with the following program:

```
1 class Box(var value: Int)
2 object A:
3   val box: Box = new Box(4)
4 object B:
5   val boxB: Box = new Box(5)
6   val boxA: Box = A.box    // ok
7   val m: Int = boxB.value // ok
8   val n: Int = boxA.value // error
```

In the program above, aliasing at line 6 is fine, as it does not read any mutable state. Meanwhile, reading the mutable state via `boxB.value` at line 7 is also safe because the box referred to by `boxB` was only created during the initialization of object `B`, so the value of its field is not affected by any other code that ran before the initialization of object `B`. In contrast, reading the mutable state via `boxA.value` at line 8 violates initialization-time irrelevance — the field could have been changed by the main program, thus it is dependent on when object `B` is initialized.

From the example, it is obvious that we need to track the ownership of mutable objects in order to tell the difference between `boxA` and `boxB` in the code above. Reading mutable state owned by a global object is fine, while reading mutable state owned by other objects is problematic.

A mutable class instance is *owned* by a global object if its instantiation is triggered by the initialization of that global object. We track the ownership of class instances by recording the owner in the abstract domain (see Section 4.4). This way, we may check the ownership when the mutable state is read or written:

> **During the initialization of a global object, it is forbidden to read or write the mutable state owned by another global object.**

Strictly speaking, writing to another global object is harmless. However, in an empirical study, we did not discover such need in practice (see Section 6). Therefore, as a rule, it's better to disallow such writes together with reads for simplicity.

This principle is not only natural, but it also enables us to check each object individually. This is because global object initialization does not have any parameter passing mechanism, so the only way that the main program could communicate values to the global object initialization code is by storing them in mutable state owned by another global object, but initialization-time irrelevance forbids reading such state during object initialization.

The principle of initialization-time irrelevance is quite flexible. For example, the following code pattern found in the Scala 3 compiler is allowed (Section 6):

```
1 object Trees:
2   private var counter = 0
3   class Tree { counter += 1 }
4   class EmptyTree extends Tree
5   val theEmptyTree = new EmptyTree
```

In the code above, the initialization of the global object `Trees` creates an instance of the class `EmptyTree` at line 5. The super class of `EmptyTree` reads and mutates the field `counter` — since the state is owned by the global object `Trees`, it is allowed to do so.

Aliasing an object owned by other global objects is also supported, as the following code shows:

```
1 object A:
2   class Box(var value: Int) { val initial: Int = value }
3   val box: Box = new Box(0)
4 object B:
5   val box: A.Box = A.box
6   val a: Int = box.initial // OK
7   val b: Int = box.value   // error
```

In the code above, reading the field `A.box` at line 5 is OK, as the read does not observe mutable state owned by `A`. Reading the field `box.initial` at line 6 is also fine, as the field is immutable. Reading the field `box.value` at line 7 is illegal, as the value of that field depends on the order of initialization, which breaks initialization-time irrelevance. Our analysis has the ability to track the ownership of mutable state, thus it reports an error at line 7.

More generally, it is allowed to create complex data structures by mixing mutable objects owned by different global objects, as the following example shows:

```
1 object A:
2   class Box(var value: Int)
3   val box: Box = new Box(0)
4 object B:
5   val boxes: Array[A.Box] = new Array(1)
6   boxes(0) = A.box            // ok
7   val box: Box = boxes(0)     // ok
8   val x: Int = box.value      // error
```

In the above, we put the mutable box created in object `A` into an array in object `B`. Reading its own mutable state at line 7 is OK. However, reading the mutable state owned by `A` at line 8 is an error.

Side effects. If global objects are allowed to perform side effects during initialization, e.g., read and write files, the property of initialization-time irrelevance cannot be guaranteed. As our analysis (Section 3 and Section 4) tracks all method calls during the intialization of an object, in theory it may flag all such system calls as warnings. In practice, side effects during initialization of global objects are of a lesser concern for programmers. We therefore leave the handling of side effects during object initialization for future work.

2.4 Static Scheduling of Global Object Initialization

Global objects usually have lazy semantics — they are initialized the first time the object is used in the program. The lazy semantics requires that access of a global object be guarded with a check

to ensure initialization of the object before usage, which incurs runtime overhead.[4] To eliminate the runtime overhead, Native Image, an ahead-of-time compiler for Java, initializes static fields at compile time (when it is safe to do so) [Wimmer et al. 2019], to eliminate the initialization check when accessing a class.[5] However, while such optimizations are very helpful in practice, a theoretical foundation is missing to justify the practice.

The principle of initialization-time irrelevance provides a theoretical foundation for optimizations related to initialization of global objects. If all objects in a program conform to the principle, then we know that changing the initialization time of those objects will not impact program semantics. In the extreme, we may initialize all of them at the entry point or even at build time[6] to remove the overhead of a run-time initialization check.

As a concrete example, consider the following program:

```
1 object A:
2   val a: Int = 10
3   val b: Int = 20
4 object B:
5   var n: Int = A.a * A.b
6 @main def entry() = println(B.n)
```

The code `B.n` at line 6 will trigger initialization of the object `B`, which in turn triggers the initialization of the object `A` as stipulated by the usual lazy semantics of global objects. At lines 5 and 6, the code needs to check whether objects `A` or `B` are initialized or not, which is an overhead at run time. The principle of initialization-time irrelevance enables us to transform the program to the following:

```
1 object A:
2   val a: Int = 10
3   val b: Int = 20
4 object B:
5   var n: Int = A.a * A.b
6 @main def entry() = { init(A); init(B); println(B.n) }
```

In the code above, we initialize the objects `A` and `B` at the beginning of the entry method. As all objects in the program are initialized according to the their inter-dependencies, there is no need to check whether a global object is initialized or not at run time.

Initializing all global objects at the beginning of the entry point is just one possible design. It is conceptually possible to initialize objects and their dependencies just before the first usage. Our current work only provides a theoretical framework for justifying such optimizations, and we leave it open as regards to which particular scheme is optimal in a particular setting.

Kozen and Stillerman [2002] also foresee the possibility of scheduling class initialization at compile time based on topological sorting of the acyclic dependency graph. However, they do not realize that doing so is unsound in the absence of initialization-time irrelevance.

[4]Runtime overhead is only important in Ahead-of-Time (AOT) compilation, because in the case of Just-in-Time (JIT) compilation, the virtual machine can initialize the static fields when a class is loaded.

[5]Strictly speaking, build-time class initialization in Native Image serves two different purposes: (1) as a compiler optimization; (2) as a programming model, e.g., to read immutable configuration files and partially evaluate program, which is a form of staged programming.

[6]This requires persistence of the heap as in Wimmer et al. [2019].

Expressions

$$
\begin{array}{rr}
e ::= x & \text{(Local variable)} \\
\mid O & \text{(Object access)} \\
\mid \text{this} & \text{(Self-reference)} \\
\mid e.f & \text{(Field access)} \\
\mid e.m\,(\overline{e}) & \text{(Method call)} \\
\mid \text{new}\, C\,(\overline{e}) & \text{(Instance creation)} \\
\mid e.f = e & \text{(Assignment)} \\
\mid e; e & \text{(Sequence)}
\end{array}
$$

Type

$$T ::= C \mid \overline{T} \Rightarrow T \mid \{\, \overline{\mathbb{I}} \,\}$$

$$\mathbb{I} ::= \text{var}\, f : T \mid \text{val}\, f : T \mid \text{def}\, m(\overline{x : T}) : T$$

Class and Object

$$\mathbb{D} ::= \mathbb{C} \mid \mathbb{O}$$

$$\mathbb{C} ::= \text{class}\, C(\overline{\text{var}\, x : T})\ \{\, \overline{\mathbb{M}} \,\}$$

$$\mathbb{O} ::= \text{object}\, O\ \{\, \overline{\mathbb{F}}\ \overline{\mathbb{M}} \,\}$$

Field and Method

$$\mathbb{F} ::= \text{val}\, f : T = e$$

$$\mathbb{M} ::= \text{def}\, m(\overline{x : T}) : T = e$$

Program

$$\mathbb{E} ::= \text{object}\, E\ \{\, \text{def}\ main() : T = e\ \}$$

$$\mathbb{P} ::= \{\text{defs} = \overline{\mathbb{D}}, \text{entry} = \mathbb{E}\}$$

Fig. 1. Surface language definition

3 THE IMMUTABLE CALCULUS

In this section, we formalize the essence of our analysis using a small calculus. For readability, we first present the analysis for an immutable variant of the calculus, and then show how to deal with mutation in the next section (Section 4).

3.1 The Surface Language

The surface language is presented in Figure 1. Its syntax and semantics are shared between the immutable calculus and mutable calculus with slight adaptations. The language introduces top-level object definitions, which may contain both fields and methods. A program $\mathbb{P}$ consists of a list of definitions $\mathbb{D}$ (class $\mathbb{C}$ or object $\mathbb{O}$) and a unique entry object $\mathbb{E}$. The semantics is defined such that the program starts by executing the main method of the entry object.

We restrict classes to not have field members to avoid dealing with issues related to safe initialization of class instances, which are addressed by other works [Blaudeau and Liu 2022; Liu et al. 2020, 2021; Qi and Myers 2009; Summers and Mueller 2011]. To simplify presentation, we avoid the need to distinguish the syntactic categories of expressions and statements by making assignment an expression that returns the value of the right-hand side ($e.f = e$). The semantics of the language is standard. We refer the reader to Appendix A for more details.

To make the small calculus more interesting, we introduce structural types of the form $\{\, \overline{\mathbb{I}} \,\}$, where each member $\mathbb{I}$ can be either a field declaration like $\text{var}\, f : T$ or a method declaration $\text{def}\, m(\overline{x : T}) : T$. A function type $\overline{T} \Rightarrow T$ is syntactic sugar for the structural type $\{\, \text{def}\, \text{apply}(\overline{x : T}) : T \,\}$. Introducing a type system for this small calculus is straightforward and we omit the details here. Such type systems, although simple and useful in practice, are incapable of handling initialization errors for global objects. In contrast, our analysis will work both for typed and untyped languages: it does not depend on the type system. Nevertheless, a type system is given in Appendix B.

3.2 The Immutable Variant

To present the analysis in steps, we will deal with an *immutable variant* of the surface language where all fields are immutable. In syntax, we replace **`var`** with **`val`** and disallow assignment. The immutable variant already enables us to write programs that cause initialization errors, as the following code demonstrates:

```
object A { val a: Int = B.b }
object B { val b: Int = A.a }
```

According to the semantics defined in Figure 4, the code above will result in reading an uninitialized field, no matter whether object A or B is accessed first in the program. The analysis should be able to detect and report such errors.

The immutable variant also allows invented programs like the following:

```
class A(val a: A)
object B:
  val a: A = loop()
  def loop(): A = new A(loop())
```

Such programs can serve as tests for the analysis, since a naive analysis for the language is prone to either non-termination or unsoundness.

3.3 Abstract Domain

The analysis defines the following abstract values:

$$\hat{v} \in \text{AValue} ::= \text{Cold} \mid \hat{o} \mid \text{Bot}$$
$$\hat{o} \in \text{AObject} ::= \text{Object}(O) \mid \text{Instance}(C, \Omega)$$
$$\Sigma \in \text{AHeap} ::= \overline{O \mapsto \Omega}$$
$$\Omega \in \text{AMap} ::= \overline{f \mapsto \hat{v}}$$

The abstract value `Cold` represents cold aliases, which is the top of the lattice. A cold alias cannot be actively used. That is, it is forbidden to call methods or access fields of a cold alias. However, they can be used as a black box to create complex data structures. The introduction of the abstraction is motivated by the observation that programmers usually create and prepare values in global objects, but do not actively use them. The abstraction is inspired by similar abstractions in previous works, where a class instance possibly under initialization should not be used [Blaudeau and Liu 2022; Liu et al. 2020; Summers and Mueller 2011].

The bottom of the lattice is represented by the abstract value Bot, which corresponds to an empty set in the concrete domain. It is allowed to call any method and access any field on a bottom value to support programs like `loop().m()` and `loop().a`, where `loop` is a defined by itself, i.e., `def loop():T=loop()`.

The analysis employs an abstract heap Σ to track the initialization state of global objects. As a result, an object reference in the code can be given the abstract value $\text{Object}(O)$ — the object name serves as an index into the abstract heap. For class instances, they are represented by the abstract value $\text{Instance}(C, \Omega)$, where C is the class name of the instance, and Ω is a map from field names to abstract field values.

For a real-world language, we can make the abstract domain more complex. In particular, the abstract domain should have an abstraction for a set of abstract values to allow more precise abstraction for `if`-expressions. Given that such extensions are easy to carry out in practice and they add more notational complexity, we keep the abstract domain simple for the sake of presentation.

The reader might have noticed that the domain for abstract values is infinite. This is because we can define a program like `class A(val a: A)`. The class can be instantiated with the help of an infinite loop: `def loop():A=new A(loop())`. Handling the infinite loop naively in the anlaysis would give rise to the ever-growing abstract value: $\text{Instance}(A, a \mapsto \text{Instance}(A, a \mapsto \ldots))$.

As the abstract values have a tree-like structure, a simple way to make the domain finitary is to restrict the *height* of values. This is implemented with the helper function *widen*, defined as follows:

$$
\begin{aligned}
\mathit{widen}(\hat{v}, 0) &= \mathsf{Cold} \\
\mathit{widen}(\mathsf{Cold}, k) &= \mathsf{Cold} \\
\mathit{widen}(\mathsf{Bot}, k+1) &= \mathsf{Bot} \\
\mathit{widen}(\mathsf{Object}(O), k+1) &= \mathsf{Object}(O) \\
\mathit{widen}(\mathsf{Instance}(C, \overline{f \mapsto \hat{v}}), k+1) &= \mathsf{Instance}(C, \overline{f \mapsto \mathit{widen}(\hat{v}, k)})
\end{aligned}
$$

To make the analysis explainable, we make the following design choice: *By default, method arguments and constructor arguments are widened with a height of* 1 *unless specified otherwise.* In Scala, programmers may write `e.m(e: @widen(k))` to restrict the argument to the specified height.

3.4 Analysis

The analysis is presented as a set of declarative rules in Figure 2. As in the concrete semantics, we assume that there exists a mapping from names to their definitions and that there are no duplicate names in method parameters, class members and object members. We use several helpers: $\mathit{rhs}(\mathbb{F})$ returns the right-hand side of a field definition; $\mathit{definition}(O)$ returns the definition of the object named O, $\mathit{lookup}(\hat{o}, m)$ returns the definition of the method m associated with the abstract object $\hat{o}$; $\mathit{params}(C)$ returns the class parameters of the class C.

Program Check (Rule A-Prog). To check that global objects in a program can be initialized safely with partial ordering, written $\Vdash (\overline{\mathbb{D}}, \mathbb{E})$, we check that each object $\mathbb{O}$ in the definitions $\overline{\mathbb{D}}$ can be initialized safely.

The check of a global object takes the form $\varnothing; \varnothing \Vdash \mathbb{O} \rightarrow \Sigma$. On the left-hand side, the first parameter $\varnothing$ signifies that there are no other objects under initialization. The second parameter $\varnothing$ represents the abstract heap for global objects — initially it is empty. After checking, we get an updated abstract heap Σ for global objects.

Object Check (Rule A-Obj-Init). To ensure safe initialization of an object in a given context, written $\Delta; \Sigma \Vdash \mathbb{O} \rightarrow \Sigma'$, we check each field initializer in the object, with $\mathsf{Object}(O)$ as the value for **this**, $\varnothing$ for local variables. Σ_i is the abstract heap after running the initializer for field i. In particular, it makes sure that the abstract heap Σ_i is consistent with already initialized field values and updates the corresponding field-value map of the object $(\Sigma_{i+1} = [O \mapsto \Omega_i]\,\Sigma_i')$. Note that evaluating the field initializer might trigger the initialization of other global objects. Therefore, abstractly evaluating the initializer of a field $\mathbb{F}_i$ in Σ_i gives rise to an updated abstract heap Σ_i'.

Expression Check. Expressions are checked with the judgement $\Delta; \hat{\psi}; \Theta; \Sigma \Vdash e \rightarrow \hat{v}; \Sigma'$, which means that the expression e evaluates to the abstract value $\hat{v}$ in the given context, where (1) Δ represents the global objects currently under initialization; (2) $\hat{\psi}$ represents the abstract value for **this**; (3) Θ is the environment for local variables; (4) Σ holds the field values for global objects; (5) Σ' holds the field values for global objects after evaluation of the expression — evaluating an expression may cause more objects to become initialized. The updated abstract heap Σ' is always an extension of Σ, because the variant of the calculus we consider here is immutable.

Normally, an object access will trigger the initialization check of the object as defined in the rule A-Obj-Acc. The rule checks that the object is not already under initialization ($O \notin \Delta$). This condition ensures that the initialization of global objects follows a partial order. However, if the object being accessed is the most recent object under initialization ($\mathit{head}(\Delta) = O$), we simply return

Program check $\boxed{\Vdash (\overline{\mathbb{D}}, \mathbb{E})}$

$$\frac{\forall \mathbb{O} \in \overline{\mathbb{D}},\ \varnothing; \varnothing \Vdash \mathbb{O} \to \Sigma}{\Vdash (\overline{\mathbb{D}}, \mathbb{E})} \text{ (A-Prog)}$$

Object initialization check $\boxed{\Delta; \Sigma \Vdash \mathbb{O} \to \Sigma}$

$$\frac{\begin{array}{c} O :: \Delta; \mathsf{Object}(O); \varnothing; \Sigma_i \Vdash \mathit{rhs}(\mathbb{F}_i) \to \hat{v}_i; \Sigma_i' \\ \Omega_0 = \varnothing \qquad \Omega_{i+1} = \Omega_i \cup \{f_i \mapsto \hat{v}_i\} \qquad \Sigma_{i+1} = [O \mapsto \Omega_i]\,\Sigma_i' \end{array}}{\Delta; \Sigma_0 \Vdash \mathbb{O} \to \Sigma_n} \text{ (A-Obj-Init)}$$

Expression check $\boxed{\Delta; \hat{\psi}; \Theta; \Sigma \Vdash e \to \hat{v}; \Sigma}$

$$\Delta; \hat{\psi}; \Theta; \Sigma \Vdash x \to \Theta(x); \Sigma \quad \text{(A-Var)} \qquad \Delta; \hat{\psi}; \Theta; \Sigma \Vdash \mathsf{this} \to \hat{\psi}; \Sigma \quad \text{(A-This)}$$

$$\frac{\mathit{head}(\Delta) = O}{\Delta; \hat{\psi}; \Theta; \Sigma \Vdash O \to \mathsf{Object}(O); \Sigma} \text{ (A-Obj-Cyc)} \qquad \frac{\Delta; \hat{\psi}; \Theta; \Sigma \Vdash e \to \mathsf{Bot}; \Sigma'}{\Delta; \hat{\psi}; \Theta; \Sigma \Vdash e.f \to \mathsf{Bot}; \Sigma'} \text{ (A-Sel-Bot)}$$

$$\frac{O \notin \Delta \qquad \mathbb{O} = \mathit{definition}(O) \qquad \Delta; \Sigma \Vdash \mathbb{O} \to \Sigma'}{\Delta; \hat{\psi}; \Theta; \Sigma \Vdash O \to \mathsf{Object}(O); \Sigma'} \text{ (A-Obj-Acc)}$$

$$\frac{\Delta; \hat{\psi}; \Theta; \Sigma \Vdash e \to \mathsf{Object}(O); \Sigma' \qquad \hat{v} = \Sigma(O)(f)}{\Delta; \hat{\psi}; \Theta; \Sigma \Vdash e.f \to \hat{v}; \Sigma'} \text{ (A-Sel-Obj)}$$

$$\frac{\Delta; \hat{\psi}; \Theta; \Sigma \Vdash e \to \mathsf{Instance}(C, \Omega); \Sigma' \qquad \hat{v} = \Omega(f)}{\Delta; \hat{\psi}; \Theta; \Sigma \Vdash e.f \to \hat{v}; \Sigma'} \text{ (A-Sel-Ins)}$$

$$\frac{\begin{array}{c} \Delta; \hat{\psi}; \Theta; \Sigma \Vdash e_0 \to \hat{o}; \Sigma_0 \qquad \overline{\Delta; \hat{\psi}; \Theta; \Sigma_i \Vdash e_a \to \hat{v}_a; \Sigma_{i+1}} \qquad \Theta' = \overline{x \mapsto \mathit{widen}(\hat{v}_a, k)} \\ \mathit{lookup}(\hat{o}, m) = \langle \mathsf{def}\, m(\overline{x : T}) : T_r = e_1 \rangle \qquad \Delta; \hat{o}; \Theta'; \Sigma_n \Vdash e_1 \to \hat{v}; \Sigma_n' \end{array}}{\Delta; \hat{\psi}; \Theta; \Sigma \Vdash e_0.m(\overline{e_a}) \to \hat{v}; \Sigma_n'} \text{ (A-Call)}$$

$$\frac{\Delta; \hat{\psi}; \Theta; \Sigma \Vdash e_0 \to \mathsf{Bot}; \Sigma_0 \qquad \overline{\Delta; \hat{\psi}; \Theta; \Sigma_i \Vdash e_a \to \hat{v}_a; \Sigma_{i+1}}}{\Delta; \hat{\psi}; \Theta; \Sigma \Vdash e_0.m(\overline{e_a}) \to \mathsf{Bot}; \Sigma_n} \text{ (A-Call-Bot)}$$

$$\frac{\overline{\Delta; \hat{\psi}; \Theta; \Sigma_i \Vdash e_a \to \hat{v}_a; \Sigma_{i+1}} \qquad \mathit{params}(C) = \langle \overline{x : T} \rangle \qquad \Omega = \overline{x \mapsto \mathit{widen}(\hat{v}_a, k)}}{\Delta; \hat{\psi}; \Theta; \Sigma_0 \Vdash \mathsf{new}\, C\, (\overline{e_a}) \to \mathsf{Instance}(C, \Omega); \Sigma_n} \text{ (A-New)}$$

Fig. 2. Declarative check rules for the immutable calculus. Both class and object fields are immutable.

the object reference, as stipulated by the rule A-Obj-Cyc. This allows trivial self cycles during initialization.

At the high level, field selection and method call on a cold alias is forbidden — there are no rules for selection and method call when the receiver is cold. Otherwise, the rules follow the concrete semantics for each syntactic form.

- A-Var. A local variable takes the abstract value defined in the environment, i.e., $\Theta(x)$.
- A-This. The self reference `this` takes the provided abstract value $\hat{\psi}$.
- A-Sel-Obj. A selection on a global object is valid if the corresponding field is already initialized. The rule A-Obj-Init ensures that Σ only contains currently initialized fields of the objects under initialization when evaluating field initializers.
- A-Sel-Ins. A selection on a class instance simply returns the abstract value bound to the immutable field of the class instance.
- A-Call. In a method call, we require the receiver to be either a global object or class instance $\hat{o}$. Widening is performed on the evaluated abstract values of method arguments according to a user specification or the default. Finally, the body of the method is evaluated in the abstract environment.
- A-New. In a new expression, we widen the abstract values of constructor arguments according to user specification or default. Then we create an abstract value for the class instance with the widened arguments.
- A-Sel-Bot and A-Call-Bot. In a selection, if the abstract value for the receiver is Bot, then the selection is also Bot. The same goes for method calls. These two rules are intended to support programs like `loop().m()` and `loop().a`, where `loop` is a defined by itself, i.e., `def loop():T=loop()`.

The rules are declarative (not algorithmic) in the sense that we do not specify how to handle recursive method calls. We discuss that co-inductive interpretation is needed and show how to make the rules algorithmic in Section 5.1.

3.5 Examples

Let us look at the two examples to illustrate how the rules work. Consider the following program:

```
1 object A { val a: Int = B.b }
2 object B { val b: Int = A.a }
```

This program will be rejected regardless of whether object `A` or `B` is initialized first. This is because the check in the rule A-Obj-Acc would detect cycles in Δ.

The second example is more interesting:

```
1 class A(val a: A)
2 object B:
3   val a: A = loop().a
4   def loop(): A = new A(loop())
```

This program will be accepted by the analysis. The recursive call `loop()` at line 4 can be abstracted by the value $\mathsf{Instance}(A, a \mapsto \mathsf{Instance}(A, a \mapsto \mathsf{Cold}))$. The choice is consistent with the default widening of constructor arguments to be of height 1. Consequently, selection of the field `a` on such a value at line 3 would return the abstract value $\mathsf{Instance}(A, a \mapsto \mathsf{Cold})$.

4 THE MUTABLE CALCULUS

While it is recommended to avoid mutable state in global objects, even new programming languages (e.g., Rust) support mutable global variables. In this section, we show how to track ownership of mutable state to enforce initialization-time irrelevance. We also present a novel design to make context sensitivity explainable and customizable by programmers.

The syntax and semantics of the mutable variant is exactly as presented in Section 3.1 (see Figure 1 and Figure 4). Therefore, we omit the introduction here.

4.1 Ownership of Mutable Objects

We mentioned in Section 2.3 that to enforce the principle of initialization-time irrelevance, we forbid reading or writing mutable state owned by other global objects during the initialization of one global object.

A mutable class instance is *owned* by a global object if its instantiation is triggered by the initialization of that global object. We can track the ownership of abstract class instances by recording the owner in the abstract domain (see Section 4.4). This way, when the analysis encounters a read or write of mutable state, it can allow it if it concerns a class instance that is owned by the global object currently being initialized. Cold aliases lose information about instance owners — this does not pose a problem because cold aliases cannot be used as receivers of reads or writes.

As an example, we draw the reader's attention again to the following program:

```
1 class Box(var value: Int)
2 object A:
3   val box: Box = new Box(4)
4 object B:
5   val boxA: Box = A.box    // ok
6   val boxB: Box = new Box(5)
7   val m: Int = boxB.value // ok
8   val n: Int = boxA.value // error
```

In the program above, the instance referred to by `A.box` and `boxA` is owned by object `A`, and that of `boxB` is owned by object `B`. The aliasing at line 5 is fine, as it does not read any mutable state. Meanwhile, while reading its own mutable state via `boxB.value` at line 7 is safe, reading the mutable state of object `A` via `boxA.value` at line 8 violates initialization-time irrelevance.

4.2 Context Sensitivity

The following example illustrates why context sensitivity is needed in the analysis when we allow mutable state:[7]

```
abstract class Foo { def foo(): Int }
class C(var x: Int) extends Foo { def foo(): Int = 20 }
class D(var y: Int) extends Foo { def foo(): Int = A.m }
class Box(var value: Foo)
object A:
  val box1: Box = new Box(new C(5))
  val box2: Box = new Box(new D(10))
  val m: Int = box1.value.foo()
```

[7]We write the code in Scala instead of the calculus for readability. It can be translated to the calculus by replacing the abstract class *Foo* with the corresponding structural type and removing the abstract class and inheritance.

When it runs, this program creates two boxes, one containing an instance of class `C` and the other an instance of class `D`. Then, the program calls the method `box1.value.foo()` at line 8. Is it safe to perform this method call?

In the concrete semantics, we allocate a new heap address for each new mutable class instance. Complex aliasing and mutation may happen via the heap address. To reason about aliasing and mutation in the analysis, we need to do the same — to allocate abstract addresses for **new**-expressions. However, in the presence of the potentially large or even infinite number of objects in the concrete domain, the static analysis has to approximate a potentially infinite set of concrete objects with a single abstract object in the abstract heap. In the abstract domain, the aliases that refer to the same abstract object share the same abstract addresses.

Context sensitivity is a way to finitize the abstract addresses (thus abstract objects) in the abstract heap. If the domain of the abstract heap and abstract values are finite, then the set of possible configurations for abstractly evaluating an expression is finite, which is necessary for the analysis to terminate. Different context sensitivity policies can achieve different degrees of precision and performance [Sharir and Pnueli 1981; Smaragdakis et al. 2011].

For the immutable calculus, we do not need context sensitivity because we can abstract immutable objects with abstract values directly without resorting to abstract addresses and an abstract heap — the field values of immutable objects cannot change. We only need to finitize the domain for abstract values to make sure that the analysis terminates.

Returning to the example above, how do we decide whether `box1` and `box2` have the same abstract address or not? That matters because if they share the same abstract address, then the code `box1.value.foo()` at line 8 may reach the method `D.foo` at line 2, which leads to initialization errors — a false positive. If they do not share the same abstract address, we can avoid such false positives.

4.3 Regions

We follow the design philosophy advocated by Alan Kay that *simple things should be simple, complex things should be possible*. When using program analysis to perform static checks, the complex use cases usually demand better precision in the analysis, which involves tuning context sensitivity.

The concept of context sensitivity is one of the most obscure concepts in program analysis [Smaragdakis et al. 2011]. While a practical static analysis usually has reasonable defaults so that it balances the performance and the precision of the analysis, there should be a mechanism to tune context sensitivity to handle complex cases.

As we mentioned in Section 2.1, explainability is critical for a static checker of global objects to be successful. We approach the problem by reversing the design: Instead of fixing a particular design of context sensitivity into the analysis, we expose it to programmers in the form of a more understandable concept — *regions*.

Intuitively, instances of the same class in the same region are represented by the same abstract object in the abstract domain. A program may introduce a new region for evaluating an expression e by writing region { e }. The construct has the same semantics as e at run time.

At the high level, a region context consists of a stack of explicitly marked program locations encountered while abstractly evaluating an expression from the entry point of the analysis. Cycles in the region context will be detected by the checker and reported as errors. This does not prevent the usage of region annotations in recursive functions — it only requires that the region annotation does not enclose recursive calls.

By default, the region context is empty — all mutable objects are in the same region, which works reasonably well for practical code in the wild. To support complex initialization code in global objects, programmers can explicitly introduce regions to tune the analysis.

For example, by default the checker would report an error for the invented example in Section 4.2. This is because by default, `box1` and `box2` will be in the same region, so they share the same abstract representation because they are of the same class `Box` — this means that the field `box1.value` is considered to possibly point to either an instance of class `C` or an instance of class `D`. Therefore, the checker would report an error at line 8 — a false positive, because the method call `box1.value.foo()` will be resolved as possibly calling both `C.foo()` and `D.foo()`, and the latter reads the uninitialized field `A.m`.

To pass the check, the programmer has to introduce explicit region annotations:

```
1 abstract class Foo { def foo(): Int }
2 class C(var x: Int) extends Foo { def foo(): Int = 20 }
3 class D(var y: Int) extends Foo { def foo(): Int = A.m }
4 class Box(var value: Foo)
5 object A:
6   val box1: Box = region { new Box(new C(5)) }     // explicit region
7   val box2: Box = region { new Box(new D(10)) }    // explicit region
8   val m: Int = box1.value.foo()
```

With the explicit region annotations at lines 6 and 7, now `box1` and `box2` will have different abstract representations. Therefore, `box1.value` will point to instances of class `C` and `box2.value` to instances of class `D`. Consequently, the method call `box1.value.foo()` will be considered to call only the method `C.foo()`, so no false positive warnings will be reported.

The concept of regions not only makes the obscure concept of context sensitivity more intuitive, it also enables flexibility in tuning the precision of the analysis. For simple initialization code, programmers usually do not need to do anything. For complex initialization code, programmers may need to tune the region context to make the analysis more precise.

Discussion. In the terminology of Smaragdakis et al. [2011], regions are neither *call-site-sensitivity* nor *object-sensitivity* but can subsume both. In call-site-sensitivity, contexts are strings of call sites, while in object-sensitivity, contexts are strings of allocation sites. In our abstractions, objects are represented by strings of static locations of region annotations. A programmer can place region annotations on call sites, on allocation sites, or on any other expression in the program.

The key distinction between the various forms of context sensitivity in Smaragdakis et al. [2011] and regions is that the former builds contexts out of all call sites (resp. allocation sites) encountered, while regions give the programmer control to determine which expressions (which sites) are relevant and are to be used for context sensitivity. This is important because the complexity of call-site-sensitivity and object-sensitivity grows exponentially in the length of the context strings, so those context strings must be kept short in practice. Thus, it is important to choose carefully and precisely which sites to include, and regions give the programmer fine-grained control to do that.

4.4 Abstract Domain

The analysis defines the following abstract values:

$$\hat{v} \in \text{AValue} ::= \text{Cold} \mid \hat{l} \mid \text{Bot}$$
$$\hat{l} \in \text{ALoc} ::= \text{Object}(O) \mid \text{Instance}(C, O, \vec{r})$$
$$\Sigma \in \text{AHeap} ::= \overline{\hat{l} \mapsto \Omega}$$
$$\Omega \in \text{AMap} ::= \overline{f \mapsto \hat{v}}$$

Cold and Bot are the top and bottom of the lattice respectively, as in the immutable variant. Now the object and class instances are addresses of the abstract heap. The field values of class instances are now stored in the abstract heap instead of directly in the abstract class instance because they are mutable. What is interesting lies in the address for class instances, written as $\mathsf{Instance}(C, O, \vec{r})$:

(1) C represents the concrete class of the instance.
(2) O represents the global object that owns the current abstract class instance.
(3) $\vec{r}$ represents the region context for the creation of the instance.

By default, the region context is empty. Programmers can write region { e } to introduce regions to have a more fine-grained approximation of runtime behavior for complex initialization code.

A region context $\vec{r}$ consists of a stack of locations in the program, and the analysis checks that the context may not contain cycles. Therefore, the number of region contexts is finite. Consequently, the domains of abstract addresses and abstract values are also finite.

As in the immutable variant, we could introduce a set of addresses as an abstract value to allow a more precise abstraction for the joins of two values. Given that such extensions are easy to carry out in practice and they add more notational complexity, we keep the abstract domain simple for the sake of presentation.

4.5 Analysis

The analysis is presented as a set of declarative rules in Figure 3. The notable differences from the immutable variant are highlighted.

The rule for checking a program $(\overline{\mathbb{D}}, \mathbb{E})$ is exactly the same as in the immutable calculus (Rule A-Prog in Figure 2) — we simply check that each object in the definitions $\overline{\mathbb{D}}$ is well-formed. We therefore omit it from the figure for space reasons.

Expressions are checked with the judgement $\Delta; \hat{\psi}; \Theta; \Sigma; \vec{r} \Vdash e \longrightarrow \hat{v}; \Sigma'$, which means that the expression e evaluates to the abstract value $\hat{v}$ in the given context, where (1) Δ represents the global objects currently under initialization; (2) $\hat{\psi}$ represents the abstract value for **this**; (3) Θ is the environment for local variables; (4) Σ is the abstract heap before evaluation of the expression; (5) $\vec{r}$ is the region context for evaluating the expression; (6) Σ' is the updated abstract heap after evaluation of the expression — mutation might happen during evaluation of an expression.

The object initialization check (Rule A-Obj-Init) is the same as in the immutable calculus, except that in checking the field initializer, we use the empty region context $\varnothing$.

Most of the rules are similar to those in the immutable calculus. Therefore, we only emphasize the differences here.

- A-Sel-Ins. When accessing a field of a mutable class instance, we check that the mutable instance is owned by the object currently being initialized with the condition $head(\Delta) = O$. This is how we enforce initialization-time irrelevance in the analysis.
- A-Region. This is how we allow programmers to specify a region context. We require that the region context is acyclic with the condition $\mathsf{loc} \notin \vec{r}$ to ensure that (1) abstract addresses are finite; (2) the structure of region context is easy to understand.
- A-New. When creating a new class instance, we first create a new abstract heap address $\hat{l}$ with the region context and owner of the instance, i.e., $\hat{l} = \mathsf{Instance}(C, O, \vec{r})$. Note that the abstract object corresponding to the address might already exist in the abstract heap. Therefore, we need to do a join with the existing object when updating the abstract heap.
- A-Mut. In mutation, we check that the class instance to be mutated is owned by the object currently being initialized with the condition $head(\Delta) = O$. When updating the field in the heap entry, we need to do a join with the existing value of the field because the abstract

Object initialization check $\boxed{\Delta; \Sigma \Vdash \mathbb{O} \to \Sigma}$

$$\frac{\begin{array}{c} O :: \Delta; \mathsf{Object}(O); \varnothing; \Sigma_i; \ \varnothing \ \Vdash \mathit{rhs}(\mathbb{F}_i) \to \hat{v}_i; \Sigma_i' \\ \Omega_0 = \varnothing \qquad \Omega_{i+1} = \Omega_i \cup \{f_i \mapsto \hat{v}_i\} \qquad \Sigma_{i+1} = [O \mapsto \Omega_i]\, \Sigma_i' \end{array}}{\Delta; \Sigma_0 \Vdash \mathbb{O} \to \Sigma_n} \text{(A-Obj-Init)}$$

Expression check $\boxed{\Delta; \hat{\psi}; \Theta; \Sigma; \ \vec{r} \ \Vdash e \to \hat{v}; \Sigma}$

$$\Delta; \hat{\psi}; \Theta; \Sigma; \vec{r} \Vdash x \to \Theta(x); \Sigma \quad \text{(A-Var)} \qquad \Delta; \hat{\psi}; \Theta; \Sigma; \vec{r} \Vdash \mathsf{this} \to \hat{\psi}; \Sigma \quad \text{(A-This)}$$

$$\frac{\mathit{head}(\Delta) = O}{\Delta; \hat{\psi}; \Theta; \Sigma; \vec{r} \Vdash O \to \mathsf{Object}(O); \Sigma} \text{(A-Obj-Cyc)} \qquad \frac{\Delta; \hat{\psi}; \Theta; \Sigma; \vec{r} \Vdash e \to \mathsf{Bot}; \Sigma}{\Delta; \hat{\psi}; \Theta; \Sigma; \vec{r} \Vdash e.f \to \mathsf{Bot}; \Sigma} \text{(A-Sel-Bot)}$$

$$\frac{O \notin \Delta \qquad \mathbb{O} = \mathit{definition}(O) \qquad \Delta; \Sigma \Vdash \mathbb{O} \to \Sigma'}{\Delta; \hat{\psi}; \Theta; \Sigma; \vec{r} \Vdash O \to \mathsf{Object}(O); \Sigma'} \text{(A-Obj-Acc)}$$

$$\frac{\Delta; \hat{\psi}; \Theta; \Sigma; \vec{r} \Vdash e \to \hat{l}; \Sigma' \qquad \hat{l} = \mathsf{Object}(O) \qquad \hat{v} = \Sigma'(\hat{l})(f)}{\Delta; \hat{\psi}; \Theta; \Sigma; \vec{r} \Vdash e.f \to \hat{v}; \Sigma'} \text{(A-Sel-Obj)}$$

$$\frac{\Delta; \hat{\psi}; \Theta; \Sigma; \vec{r} \Vdash e \to \hat{l}; \Sigma' \quad \hat{l} = \mathsf{Instance}(C, O, \vec{r}') \quad \mathit{head}(\Delta) = O \quad \hat{v} = \Sigma'(\hat{l})(f)}{\Delta; \hat{\psi}; \Theta; \Sigma; \vec{r} \Vdash e.f \to \hat{v}; \Sigma'} \text{(A-Sel-Ins)}$$

$$\frac{\begin{array}{c} \Delta; \hat{\psi}; \Theta; \Sigma; \vec{r} \Vdash e_0 \to \hat{l}; \Sigma_1 \quad \overline{\Delta; \hat{\psi}; \Theta; \Sigma_i; \vec{r} \Vdash e_a \to \hat{v}_a; \Sigma_{i+1}} \quad \Theta' = \overline{x \mapsto \hat{v_a}} \\ \mathit{lookup}(\hat{l}, m) = \langle \mathsf{def}\, m(\overline{x : T}) : T_r = e_1 \rangle \qquad \Delta; \hat{l}; \Theta'; \Sigma_n; \vec{r} \Vdash e_1 \to \hat{v}; \Sigma' \end{array}}{\Delta; \hat{\psi}; \Theta; \Sigma; \vec{r} \Vdash e_0.m(\overline{e_a}) \to \hat{v}; \Sigma'} \text{(A-Call)}$$

$$\frac{\mathsf{loc} \notin \vec{r} \qquad \Delta; \hat{\psi}; \Theta; \Sigma; \mathsf{loc} :: \vec{r} \Vdash e \to \hat{v}; \Sigma'}{\Delta; \hat{\psi}; \Theta; \Sigma; \vec{r} \Vdash (\mathsf{region}\ \{\ e\ \})^{\mathsf{loc}} \to \hat{v}; \Sigma'} \text{(A-Region)}$$

$$\frac{\Delta; \hat{\psi}; \Theta; \Sigma; \vec{r} \Vdash e_0 \to \mathsf{Bot}; \Sigma_1 \qquad \overline{\Delta; \hat{\psi}; \Theta; \Sigma_i; \vec{r} \Vdash e_a \to \hat{v}_a; \Sigma_{i+1}}}{\Delta; \hat{\psi}; \Theta; \Sigma; \vec{r} \Vdash e_0.m(\overline{e_a}) \to \mathsf{Bot}; \Sigma_n} \text{(A-Call-Bot)}$$

$$\frac{\begin{array}{c} \overline{\Delta; \hat{\psi}; \Theta; \Sigma_i; \vec{r} \Vdash e_a \to \hat{v}_a; \Sigma_{i+1}} \qquad \mathit{params}(C) = \langle \overline{x : T} \rangle \qquad O = \mathit{head}(\Delta) \\ \Omega = \overline{x \mapsto \hat{v}_a} \qquad \hat{l} = \mathsf{Instance}(C, O, \vec{r}) \qquad \Sigma' = \left[\hat{l} \mapsto \Omega \cup \Sigma_n(\hat{l})\right] \Sigma_n \end{array}}{\Delta; \hat{\psi}; \Theta; \Sigma; \vec{r} \Vdash (\mathsf{new}\, C\, (\overline{e_a})) \to \hat{l}; \Sigma'} \text{(A-New)}$$

$$\frac{\begin{array}{c} \Delta; \hat{\psi}; \Theta; \Sigma; \vec{r} \Vdash e_1 \to \hat{l}_1; \Sigma_1 \quad \hat{l}_1 = \mathsf{Instance}(C, O, \vec{r}') \\ \Delta; \hat{\psi}; \Theta; \Sigma_1; \vec{r} \Vdash e_2 \to \hat{v}_2; \Sigma_2 \quad \Omega = \Sigma_2(\hat{l}_1) \\ \mathit{head}(\Delta) = O \quad \Omega' = [f \mapsto \Omega(f) \cup \hat{v}_2]\, \Omega \quad \Sigma_3 = [l_1 \mapsto \Omega']\, \Sigma_2 \end{array}}{\Delta; \hat{\psi}; \Theta; \Sigma; \vec{r} \Vdash (e_1.f = e_2) \to \hat{v}_2; \Sigma_3} \text{(A-Mut)}$$

Fig. 3. Declarative check rules for the mutable calculus. Class fields are mutable, object fields are immutable.

address corresponds to a group of concrete objects and we cannot distinguish the individual concrete objects of the group in the abstract domain.

Note that in the calculus, the fields of global objects are immutable and can only be initialized at their definition sites. Therefore, in the rule A-Obj-Init, we can be sure that the corresponding field f_i is absent in Ω_i, hence we can simply add the field value to the abstract heap object without worrying about overwriting. If the fieds of global objects can be mutable (as is the case in Scala), we need to do a join with the existing values for mutable fields, as it is done in A-New and A-Mut.

4.6 Examples

Consider the following program:

```
1 class Box(var value: Int)
2 object A:
3   val box: Box = new Box(4)
4 object B:
5   val boxB: Box = new Box(5)
6   val boxA: Box = A.box    // ok
7   val m: Int = boxB.value // ok
8   val n: Int = boxA.value // error
```

The checker will report an error at line 8, because `boxA` is abstracted by $\mathsf{Instance}(Box, A, \varnothing)$ whose owner is different from the object `B` currently being initialized. In contrast, `boxB` is abstracted by $\mathsf{Instance}(Box, B, \varnothing)$, thus the field access `boxB.value` at line 7 is valid.

The following program will be rejected:

```
1 abstract class Foo { def foo(): Int }
2 class C(var x: Int) extends Foo { def foo(): Int = 20 }
3 class D(var y: Int) extends Foo { def foo(): Int = A.m }
4 class Box(var value: Foo)
5 object A:
6   val box1: Box = new Box(new C(5))
7   val box2: Box = new Box(new D(10))
8   val m: Int = box1.value.foo()
```

The reason is that both `box1` and `box2` are in the same region, so they thus share the same abstract address $\mathsf{Instance}(Box, A, \varnothing)$. Consequently, `box1.value` evaluates to a value that is the join of the abstract values $\mathsf{Instance}(C, A, \varnothing)$ and $\mathsf{Instance}(D, A, \varnothing)$.[8] Calling the method `foo()` on the field `box1.value` at line 7 is therefore illegal.

To make the code above pass the check, we can introduce explicit region contexts:

```
1 abstract class Foo { def foo(): Int }
2 class C(var x: Int) extends Foo { def foo(): Int = 20 }
3 class D(var y: Int) extends Foo { def foo(): Int = A.m }
4 class Box(var value: Foo)
5 object A:
6   val box1: Box = region { new Box(new C(5)) } // explicit region
7   val box2: Box = region { new Box(new D(10)) } // explicit region
8   val m: Int = box1.value.foo()
```

[8] The result of the join is Cold for the calculus due to its simplicity. In our actual implementation, it is a set that consists of the two aforementioned abstract values. The method call at line 7 is still invalid in the actual implementation.

Now box1 is abstracted by Instance($Box, A, [5]$)[9] and box2 is abstracted by Instance($Box, A, [6]$). Therefore, the values contained in the boxes are kept separate by the analysis: `box1.value` is abstracted by Instance($C, A, [5]$) and `box2.value` is abstracted by Instance($D, A, [6]$). Consequently, the method call `box1.value.foo()` resolves to only `C.foo()`, which is safe.

4.7 Properties

To formulate the property of initialization-time irrelevance, we introduce the following program transform, which removes trivial object accesses in the program:

$$\mathcal{T}(\langle O; e\rangle) = e$$

In the above, the transform $\mathcal{T}$ recursively rewrites the code pattern $\langle O; e\rangle$ (which evaluates to the address of object O, forcing its initialization, then discards that address and evaluates expression e) to just $\langle e\rangle$ in the program — effectively removing all trivial object accesses in the original program. For a removed object access, there are three scenarios:

(1) Its initialization time will *not change* if the access happens after the first real use of object O.
(2) Its initialization will be *delayed* if the access precedes the first real use of object O.
(3) Its initialization will be *removed* if there are no real uses of object O.

Obviously, the first scenario will not change program semantics. We expect the second scenario to not change program semantics either if the principle of initialization-time irrelevance is observed. The third scenario might change program semantics, e.g., the removal of the initialization of an object might remove non-termination in the program. We capture the intuition with the following property:

Property 1 (Initialization-Time Irrelevance)**.** *If a program passes the initialization check, then the removal of trivial object accesses will maintain semantics of terminating programs. Formally, given a well-formed program* $\mathbb{P} = (\overline{\mathbb{D}}, \mathbb{E})$*, i.e.* $\vdash \mathbb{P}$*, and* $\mathbb{P}' = \mathcal{T}(\mathbb{P}) = (\overline{\mathbb{D}}', \mathbb{E}')$*, if* $[\![\mathbb{P}]\!] = (l, \sigma)$*, then* $[\![\mathbb{P}']\!] = (l, \sigma')$ *and* $\sigma' \subseteq \sigma$*.*

The property above only captures equivalence for terminating programs — removing a trivial object access may remove non-termination as well. Technically, the heap σ' is a subset of σ because some global objects (and instances they create) might not be initialized after the removal.

This property enables us to schedule the initialization of global objects by the topological order in the initialization dependence graph. For example, a simple scheme would be to initialize them at the beginning of the entry method in ahead-of-time (AOT) compilation. It is also possible to initialize the global objects at build time, as is done in Native Image of the GraalVM project [Wimmer et al. 2019]. This way, we can reduce the overhead of initialization checking at run time as required by the lazy semantics. Our current work provides a theoretical justification for such optimizations. Previous work also foresees the possibility of scheduling class initialization at compile time based on topological sorting of the acyclic dependency graph [Kozen and Stillerman 2002]. However, they do not realize that doing so is unsound in the absence of initialization-time irrelevance.

Discussion. A technical subtlety is related to memory locations. Usually, the fresh memory location is implemented in mechanized proofs by using the size of the current heap [Blaudeau and Liu 2022]. With this scheme, the removal of trivial object accesses might result in fewer objects being created and thus lead to a mismatch in memory locations. The proposition $\sigma' \subseteq \sigma$ fails to hold when such a mismatch happens.

In formal semantics, we can overcome this difficulty by changing the scheme for addresses. Instead of using just a number, we can use a pair (O, n) as locations: Always use $(O, 0)$ as the

[9]We use line numbers instead of exact source positions to approximate region context.

address of object O, and when allocating a new class instance owned by an object O, use the address $(O, n+1)$ where the number n is the number of instances currently owned by the object O.

Given the subtlety of the theorem, we think a mechanized proof will be necessary to validate the theorem, which we leave for future work.

5 IMPLEMENTATION

In this section, we show how to implement the declarative checking rules and scale from the small language to Scala. Our implementation is integrated in the Scala 3 compiler, Dotty, and can be enabled via the compiler option `-Ysafe-init-global`.

5.1 Fixed-Point Computation

The checking rules presented in Figure 2 and Figure 3 are declarative (not algorithmic). The rules should be read co-inductively. The co-inductive reading can be illustrated by the following example:

```
1 class C(var x: Int)
2 object A:
3   var f: C = foo()
4   def foo(): C = foo()
```

In order to prove that the method call `foo()` is safe at line 3, the analysis will encounter exactly the same sub-goal to prove that the call `foo()` is safe. A coherent interpretation of the rules needs to assume that the call `foo()` returns a fixed-point value for the recursive call `foo()`. In this example, both the top and bottom of the value lattice (Cold and Bot) are fixed points for the expression `foo()`. We are only interested in *least fixed points*, which allow more programs to be accepted. The *greatest fixed point* Cold is always safe, but it rejects too many programs (even though it accepts the program above), so it is not useful.

We use an abstract definitional interpreter to compute least fixed points based on the technique of co-inductive caching [Darais et al. 2017]:

$$\hat{\zeta} \in \text{ACache} \quad = \quad \overline{(e, \hat{\psi}, \Theta, \Sigma, \vec{r}) \mapsto (\hat{l}, \Sigma')}$$

The cache ζ maps the expression e, the value $\hat{\psi}$ for **this**, Θ for the abstract environment, Σ for the abstract heap and $\vec{r}$ for the current region context to the value $\hat{l}$ of the expression and the updated heap Σ'.

As in Darais et al. [2017], we need to employ both an input cache ζ_{in} and output cache ζ_{out}. In the abstract evaluation of an expression, we check whether the corresponding key exists in the output cache ζ_{out}. If it is in ζ_{out}, we return the cached value immediately. Otherwise, we retrieve the value from the input cache ζ_{in} (using the bottom value Bot if the key is not yet in the cache), put it in the output cache ζ_{out}, and evaluate the expression by evaluating its sub-expressions. The output cache ζ_{out} is then updated with the actual value for the expression. The co-inductive caching ensures that cache values are only used in recursive calls but not eagerly.

The iterative algorithm works at the granularity of global objects. After each iteration, it checks whether ζ_{in} and ζ_{out} are the same. If not, it will use ζ_{out} as the new ζ_{in}, reset ζ_{out} to empty, and check the object again until a fixed point is reached. The fixed point always exists because the abstract definitional interpreter is monotone with respect to the abstract cache.

5.2 Handling Scala Features

Scala [Odersky 2019] is a language with rich features: path-dependent types, higher-kinded types, traits, local classes, enums, first-class functions, etc. For an analysis to be practical for Scala, it has to handle all of the language features.

Theoretically, the analysis only needs to handle the language model for Scala after type erasure and flattening — assuming that the compiler preserves language semantics. This language model is much simpler because most language features are elaborated to classes and global objects, which are at the top level. The non-trivial complications compared to our small language are inheritance, traits, closures[10] and arrays.

For inheritance and traits, the analysis needs to follow the *linearization* ordering in the initialization semantics [Odersky 2019]. Virtual method calls do not pose a problem, as the abstract domain records the concrete classes of abstract values. Therefore, the analysis may resolve virtual method calls according to the language semantics.

Conceptually, closures are immutable classes — the fields are the variable values that they capture.[11] Therefore, it is easy to handle functions with a special form of abstract values similar to immutable classes. Calling a function can be emulated by a method call in the abstract interpreter.

We handle arrays as if they were class instances with a single mutable field — we do not distinguish the array elements at different indexes. Similar to mutable class instances, we need to record both the owner and the region context in the abstract domain.

In the actual implementation, however, we define the abstract interpreter before flattening and type erasure. This design decision is motivated by two considerations: (1) For separately compiled libraries, the intermediate representation, called TASTY, is in a form before flattening and type erasure; (2) The analysis may report better error messages when the input language is closer to the surface language.

To handle this language model, we ignore type parameters and type arguments in the abstract interpreter. Meanwhile, we augment the abstract values with the environment that local/inner classes and functions capture, which achieves the same convenience that flattening provides.

The abstract domains we presented in Section 3.3 and Section 4.4 modeled only a single object at a time for simplicity of presentation. In the implementation, an abstract value can be a set of such abstract objects in order to more precisely represent the join of two abstract values.

The analysis is implemented as a compiler phase after type checking. The implementation does not require any changes to the type system in the compiler. The implementation resolves all existing bug reports related to initialization of global objects (Appendix C).

If only immutable classes are reachable in the initialization of global objects, the analysis is the same as the immutable calculus in essence (Section 3). To uniformly handle mutable and immutable fields in classes, we implement context sensitivity at the level of mutable fields instead of objects. This design also allows us to unify mutable local variables in closures and mutable fields in objects. For programmers, the same mutable field in the same region context has the same abstract representation.

6 CASE STUDY

In this section, we discuss the practicality of the principles of partial ordering and initialization-time irrelevance based on a case study of a large real-world Scala project. We would like to remind the

[10] Functions used to be represented by anonymous classes in earlier versions of the Scala compiler. Now there are special AST trees for closures to reduce code size since Java 8 introduced lambdas.

[11] As a technical detail, the capture of mutable variables is not an issue — the compiler desugars a captured mutable variable to an immutable variable which points to an object containing the mutable state.

reader that the findings discussed below are true positives: they were actual violations of the two principles in the code.

6.1 Initialization-Time Irrelevance

We have been justifying the principle of initialization-time irrelevance from the semantic point of view. The reader might be wondering whether the principle works with existing Scala code. To answer this question, we performed a case study on one of the most complex open source Scala applications: the Scala 3 compiler.

The Scala 3 compiler, named Dotty, contains about 123K lines of code for the compiler itself. It defines 1017 classes and 567 non-synthesized objects.[12] Among the objects, we find 4 violations of the principle, which we discuss below.

The first violation is arguably a valid use case, as the following code illustrates (adapted):

```
object Names:
  class Name(val start: Int, val length: Int)
  var chrs: Array[Char] = new Array[Char](0x20000)
  def name(s: String): Name = Name(0, chrs.length) // simplified for readability

object StdNames:
  val AnyRef: Names.Name = Names.name("AnyRef")
  val Array: Names.Name  = Names.name("Array")
  val List: Names.Name   = Names.name("List")
```

In the code above, the global object `Names` contains a mutable name table, which is shared across different compiler runs. The object `StdNames` creates predefined names that are used in the compiler. Creating a new name requires reading (and possibly writing) the mutable state owned by the object `Names`, thus it violates initialization-time irrelevance.

This seems to be a justified violation of the principle. However, instead of capitulating by classifying such use cases as an exception, it is possible to support the usage with a slight modification of the code, making `StdNames` a class instance rather than a global object:

```
object Names:
  val StdNames = new StdNames
class StdNames:
  val List: Names.Name   = Names.name("List")
```

The second violation is related to diagnostic code in the compiler, as shown below:

```
object Stats { var monitored: Boolean = false }
class UncachedGroundType { if (Stats.monitored) println("record stats") }
class LazyType extends UncachedGroundType
object NoCompleter extends LazyType
```

In the code above, the mutable field `Stats.monitored` could be set to **`true`** by programmers via a compiler flag. But at the time when the object `NoCompleter` is initialized, it is uncertain whether the user input has been written to the field already or not. This could result in slight imprecision in diagnostic data, but this is negligible in practice. For this case, we believe it is justified to use the annotation `@unchecked` to suppress the warning from the analysis.

The third violation comes from the following code (adapted for readability):

[12] Synthesized companion objects for case classes do not contain member fields, so their initialization is trivial.

```
class SourceFile
object Contexts:
  val NoContext: Context = new Context
  class Context:
    private var _source: SourceFile = null
    final def source: SourceFile = _source
    def setSource(source: SourceFile) = { this._source = source }
object Implicits:
  import Contexts.*
  case class SearchFailure(tag: Int, source: SourceFile)
  val NoMatchingFailure: SearchFailure = SearchFailure(1, NoContext.source)
```

In the code above, initialization of the object `Implicits` would read the mutable field `source` of `Contexts.NoContext`, which is owned by the object `Contexts`. In fact, both `NoContext.source` and `NoMatchingFailure.source` will be **`null`** at run time. However, the fields will never be used by the compiler, thus they do not pose a problem. Given the fact that the value `NoMatchingFailure` is only used as a tag in the compiler, we think a reasonable refactoring would be to make `NoMatchingFailure` an object extending a shared base class with `SearchFailure`. This way, it would avoid reading the uninitialized field `NoContext.source` in the first place.

The fourth violation again involves diagnostic code, which can be illustrated as follows:

```
object Positioned { var debug: Boolean = false }
abstract class Positioned:
  if (Positioned.debug) { println("do debugging") }
object Trees:
  class Tree extends Positioned
  val emptyTree = new Tree
```

In the code above, initialization of the object `Trees` would read the mutable state `debug` in the object `Positioned`, which can be set via compiler flags. It is uncertain whether the field is set or not when the object `Trees` is initialized. That should not matter in practice, because programmers are not supposed to debug the empty trees which are usually used as placeholders.

For this reason, we believe it is better to move the diagnostic code from the object `Positioned` to the object `Trees`. This way, initialization of the object `Trees` will only read its own state, which is allowed by initialization-time irrelevance.

6.2 Partial Ordering

While the principle of partial ordering is justified based on deadlock freedom and information hiding, there is concern whether the principle is expressive enough for real-world programs.

In the Dotty project, we encounter only one violation of partial ordering. The violation can be illustrated with the following code:

```
object Names:
  val ctorString = "<init>"
  val ctorName: MethodName = MethodName.apply(ctorString)
class MethodName(encoded: String)
object MethodName:
  val ctor: MethodName = new MethodName(Names.ctorString)
  def apply(name: String): MethodName =
    if (name == Names.ctorString) ctor else new MethodName(name)
```

In the code above, initialization of the object `Names` triggers the initialization of the object `MethodName` at line 3; the latter in turn depends on the former at line 6. The cycle might lead to a deadlock in the presence of concurrency.

To avoid the cycle, we can simply move the field `ctorString` at line 2 to the object `MethodName`.

6.3 Discussion

As it is explained above, the violation of initialization principles in the Scala 3 compiler can be fixed with code refactoring. For the particular case study, we do not need to resort to regions to make the analysis more expressive. To some extent, it meets our expectation that regions are only needed for rare and complex cases, for which programmers pay the tax for the complexity. We expect such complex use cases to arise as the checker becomes more widely used in practice.

The reader might be wondering: how long does it take to check the Scala 3 compiler? We ran 5 iterations of compiling Dotty with the checker enabled and disabled and observed the following compilation times:

- Enabled: 59s, 55s, 55s, 54s, 53s
- Disabled: 57s, 56s, 56s, 52s, 53s

The execution time of the current implementation is not a significant fraction of total compilation time. Given the acceptable result, we think it is premature to do performance tuning on the implementation, and we delay a carefully controlled experiment to obtain a precise measurement of the execution times as future work when performance becomes a concern.

7 RELATED WORK

Börger and Schulte [2000] report several problems with Java's lazy initialization semantics of classes, such as portability, deadlocks and unsound compiler optimizations.

Kozen and Stillerman [2002] propose a graph-based algorithm to ensure that there are no cycles when running static initializers of classes. For this purpose, they build an initialization dependency graph with edges $A \Rightarrow B$ to mean that running the static initializer of class `A` would lead to running the static initializer of class `B`. They enforce that the dependency graph is acyclic except for benign self-cycles (see Figure 2 and Figure 4), similar to *partial ordering* in our design. Their algorithm, however, needs to perform whole-program analysis due to the absence of the *initialization-time irrelevance* proposed in our work.

They also foresee the possibility of scheduling class initialization at compile time based on topological sorting of the acyclic dependency graph. However, they do not realize that doing so is unsound in the absence of initialization-time irrelevance.

Hubert and Pichardie [2009] propose a system for tracking which static fields of a class have been initialized at any given point to ensure that a static field is only read after initialization. The work is based on bytecode and it depends on whole-program analysis.

Leino and Müller [2004] consider the problem of modular verification of global module invariants in object-oriented programs. Interestingly, to support reasoning about invariants of modules, they also arrive at the design of *ownership*: The invariant of module A may depend on fields of objects that are (transitively) owned by `A`. They introduce syntax to specify owners explicitly for **`new`**-expressions when creating new objects. The ownership structure forms a tree in their work, while it is flat in ours: only global objects may be owners.

In a following work, Leino and Müller [2005] extend the technique to static class invariants. They impose a stronger concept of partial ordering on classes — compilation partial ordering. In contrast, our work only requires that the initialization dependencies of global objects form a partial order — global objects are allowed to mutually reference each other as long as they do not form

initialization cycles. Leino and Müller [2005] did not realize that compilation partial ordering is neither necessary nor sufficient for safe initialization of global objects because compilation partial ordering does not imply initialization-time irrelevance.

There is a lineage of works on safe initialization of class instances [Blaudeau and Liu 2022; Liu et al. 2020, 2021; Qi and Myers 2009; Summers and Mueller 2011]. These works do not handle global objects or static fields. In contrast, our work sidesteps the problem of safe initialization of class instances in the syntax of the calculus. A language designer might be interested in reading both, given that they are complementary.

The abstraction *Cold* in the abstract domains of our analyses is inspired by similar concepts in previous work on safe initialization of class instances (*cold* in [Blaudeau and Liu 2022; Liu et al. 2020] and *free* in [Summers and Mueller 2011]). In both cases, it is forbidden to access fields or call methods on cold values. However, they are motivated slightly differently. For class instances potentially under initialization, it is dangerous to access fields or call methods on them. For global objects, programmers usually prepare values in global objects and do not intend those values to be used. Our analysis currently enables programmers to use any values prepared in global objects — albeit with some annotation overhead for complex use cases. We also envision the introduction of the annotation "@Cold" to enable programmers to control the usage of values during initialization.

Meanwhile, in the aforementioned previous works, the systems maintain a heap separation between cold and non-cold objects: the static system guarantees that cold objects are not reachable from non-cold objects at run time to observe the transitivity of the *initialized* state. The invariant is usually enforced by forbidding the assignment of a cold value to the field of a non-cold object. Our analysis explicitly models the heap, therefore it does not need such a rule.

8 CONCLUSION

In this paper, we propose two principles for safe initialization of global objects: *partial ordering* and *initialization-time irrelevance*. The principles (1) enable modular reasoning about initialization; (2) avoid deadlocks in initializing global objects; (3) allow static scheduling of global object initialization.

To enforce the principles, we present a static analysis algorithm to ensure safe initialization of global objects. The algorithm tracks *ownership of mutable state* to enforce initialization-time irrelevance. It introduces the concept of *regions* to make *context-sensitivity* understandable and tunable by programmers for complex initialization code.

ACKNOWLEDGMENTS

We sincerely thank OOPSLA 2023 reviewers for their helpful feedback. We are also grateful to our shepherd Philipp Haller for the many improvement suggestions. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

A SEMANTICS

The big-step semantics of the language is presented in Figure 4. An evaluation relation takes the form $[\![e]\!](\sigma, \rho, \psi) \longrightarrow (l, \sigma')$, which means that the expression e evaluates to the value l with the updated heap σ', given the heap σ before the evaluation, the environment ρ for method parameters, and the value ψ for this.

We assume a few helpers: $lookup(\tau, m)$ looks up the method m in a corresponding class or object definition depending on whether τ is a class name or object name; $params(C)$ returns the class parameters of class C; $field(O, i)$ returns the i-th field of the object O; $addr(O)$ returns the heap address bound with the object O.

$$[\![x]\!](\sigma,\rho,\psi) \longrightarrow (\rho(x),\sigma) \quad \text{(E-Var)} \qquad [\![\mathsf{this}]\!](\sigma,\rho,\psi) \longrightarrow (\psi,\sigma) \quad \text{(E-This)}$$

$$\frac{[\![e]\!](\sigma,\rho,\psi) \longrightarrow (l,\sigma') \qquad \sigma'(l) = (\_,\omega) \qquad \omega(f) = l'}{[\![e.f]\!](\sigma,\rho,\psi) \longrightarrow (l',\sigma')} \quad \text{(E-Select)}$$

$$\frac{l = addr(O) \qquad l \in dom\,(\sigma)}{[\![O]\!](\sigma,\rho,\psi) \longrightarrow (l,\sigma)} \quad \text{(E-Obj1)}$$

$$\frac{\begin{array}{c} l = addr(O) \qquad l \notin dom\,(\sigma) \qquad field(O,i) = \langle \mathsf{val}\ f_i : T_i = e_i \rangle \\ [\![e_i]\!](\sigma_i,\varnothing,l) \longrightarrow (v_i,\sigma_i') \qquad \sigma_0 = \sigma \cup \{l \mapsto (O,\varnothing)\} \qquad \sigma_{i+1} = \left[l \mapsto \left(O, \overline{f_i \to v_i}\right)\right]\sigma_i' \end{array}}{[\![O]\!](\sigma,\rho,\psi) \longrightarrow (l,\sigma_n)} \quad \text{(E-Obj2)}$$

$$\frac{\begin{array}{c} [\![e]\!](\sigma,\rho,\psi) \longrightarrow (l_1,\sigma_1) \qquad \sigma_1(l_1) = (\tau,\_) \qquad [\![\overline{e_a}]\!](\sigma_1,\rho,\psi) \longrightarrow (\overline{l_a},\sigma_2) \\ lookup(\tau,m) = \langle \mathsf{def}\ m(\overline{x:T}) : T_r = e_m \rangle \qquad [\![e_m]\!](\sigma_2,\overline{x \mapsto l_a},l_1) \longrightarrow (l_3,\sigma_3) \end{array}}{[\![e.m\,(\overline{e_a})]\!](\sigma,\rho,\psi) \longrightarrow (l_3,\sigma_3)} \quad \text{(E-Call)}$$

$$\frac{\begin{array}{c} [\![\overline{e_a}]\!](\sigma,\rho,\psi) \longrightarrow (\overline{l_a},\sigma_1) \qquad l_{fresh} \notin dom\,(\sigma_1) \qquad \forall O.l_{fresh} \neq addr(O) \\ params(C) = \overline{(x:T)} \qquad \sigma_2 = \sigma_1 \cup \left\{l_{fresh} \mapsto \left(C, \overline{x \mapsto l_a}\right)\right\} \end{array}}{[\![\mathsf{new}\ C\,(\overline{e_a})]\!](\sigma,\rho,\psi) \longrightarrow (l_{fresh},\sigma_2)} \quad \text{(E-New)}$$

$$\frac{\begin{array}{c} [\![e_1]\!](\sigma,\rho,\psi) \longrightarrow (l_1,\sigma_1) \qquad [\![e_2]\!](\sigma_1,\rho,\psi) \longrightarrow (l_2,\sigma_2) \\ \sigma_2(l_1) = (C,\omega) \qquad \sigma_3 = [l_1 \mapsto (C,[f \mapsto l_2]\,\omega)]\,\sigma_2 \end{array}}{[\![e_1.f = e_2]\!](\sigma,\rho,\psi) \longrightarrow (l_2,\sigma_3)} \quad \text{(E-Assign)}$$

$$\frac{[\![e_1]\!](\sigma,\rho,\psi) \longrightarrow (l_1,\sigma_1) \qquad [\![e_2]\!](\sigma_1,\rho,\psi) \longrightarrow (l_2,\sigma_2)}{[\![e_1;e_2]\!](\sigma,\rho,\psi) \longrightarrow (l_2,\sigma_2)} \quad \text{(E-Seq)}$$

Fig. 4. Big-step semantics for surface language

The rules E-Obj1 and E-Obj2 show that global objects have lazy semantics. The other rules are straightforward, so we omit a detailed explanation here. The semantics of a program, written $[\![\mathbb{P}]\!]$, is defined by executing the main method of the entry object.

As usual, big-step semantics only defines the behavior of terminating programs. It can be instrumented with a fuel to define the behavior of non-terminating programs [Amin and Rompf 2017; Blaudeau and Liu 2022].

B TYPE SYSTEM

Here we present a simple type system for the language. Our analyses (Section 3 and Section 4) do not depend on the type system. However, our analyses only check that global objects can be initialized safely and the principle of initialization-time irrelevance is observed. They do not check that the main program is well-formed. To ensure that the main program will not get stuck at runtime, the type system can be used.

For simplicity, we intentionally keep the subtyping of structural types simple: It is straightforward to make the subtyping more expressive by extending subtyping to members of structural types. The typing rules are standard, we therefore omit a detailed explanation.

Program Typing $\boxed{\vdash (\overline{\mathbb{D}}, \mathbb{E})}$

$$\frac{\forall \mathbb{C} \in \overline{\mathbb{D}},\ \vdash \mathbb{C} \qquad \forall \mathbb{O} \in \overline{\mathbb{D}},\ \vdash \mathbb{O} \qquad \vdash \mathbb{E}}{\vdash (\overline{\mathbb{D}}, \mathbb{E})} \text{ (T-Prog)}$$

Class Typing $\boxed{\vdash \mathbb{C}}$

$$\frac{\forall \mathbb{M} \in \mathit{methods}(\mathbb{C}),\ \mathbb{M} = \langle \text{def } m(\overline{x : T_i}) : T_r = e \rangle \qquad \overline{x{:}T_i}; C \vdash e : T_r}{\vdash \mathbb{C}} \text{ (T-Class)}$$

Object Typing $\boxed{\vdash \mathbb{O}}$

$$\frac{\begin{array}{c}\forall \mathbb{F} \in \mathit{fields}(\mathbb{O}),\ \mathbb{F} = \langle \text{val } f : T = e \rangle \qquad \varnothing; \mathit{structType}(O) \vdash e : T \\ \forall \mathbb{M} \in \mathit{methods}(\mathbb{O}),\ \mathbb{M} = \langle \text{def } m(\overline{x : T_i}) : T_r = e \rangle \qquad \overline{x{:}T_i}; \mathit{structType}(O) \vdash e : T_r\end{array}}{\vdash \mathbb{O}} \text{ (T-Object)}$$

Subtyping $\boxed{T <: T}$

$$\frac{\mathit{structType}(C) <: T}{C <: T} \text{ (S-Class)} \qquad \frac{\overline{\mathbb{I}_1} \subseteq \overline{\mathbb{I}_2}}{\{\,\overline{\mathbb{I}_1}\,\} <: \{\,\overline{\mathbb{I}_2}\,\}} \text{ (S-Struct)}$$

Expression Typing $\boxed{\Gamma; T \vdash e : T}$

$$\Gamma; T \vdash \text{this} : T \text{ (T-This)} \qquad \frac{x : U \in \Gamma}{\Gamma; T \vdash x : U} \text{ (T-Var)} \qquad \frac{T_O = \mathit{structType}(O)}{\Gamma; T \vdash O : T_O} \text{ (T-Obj)}$$

$$\frac{\Gamma; T \vdash e : T_1 \qquad T_1 <: T_2}{\Gamma; T \vdash e : T_2} \text{ (T-Sub)} \qquad \frac{\Gamma; T \vdash e : T_e \qquad T_f = \mathit{fieldType}(T_e, f)}{\Gamma; T \vdash e.f : T_f} \text{ (T-Sel)}$$

$$\frac{\overline{T_i} = \mathit{constrType}(C) \qquad \Gamma; T \vdash e_i : T_i}{\Gamma; T \vdash \text{new } C(\overline{e}) : C} \text{ (T-New)} \qquad \frac{\Gamma; T \vdash e_1 : T_1 \qquad \Gamma; T \vdash e_2 : T_2}{\Gamma; T \vdash e_1; e_2 : T_2} \text{ (T-Seq)}$$

$$\frac{\Gamma; T \vdash e_1 : T_1 \quad T_f = \mathit{fieldType}(T_1, f) \wedge \mathit{mutable}(T_1, f) \quad \Gamma; T \vdash e_2 : T_f}{\Gamma; T \vdash e_1.f = e_2 : T_f} \text{ (T-Assign)}$$

$$\frac{\Gamma; T \vdash e : T_e \qquad (\overline{T_i}, T_r) = \mathit{methodType}(T_e, m) \qquad \Gamma; T \vdash e_i : T_i}{\Gamma; T \vdash e.m(\overline{e}) : T_r} \text{ (T-Invoke)}$$

Fig. 5. Type system for the calculus

C SCALA OPEN BUGS

Scala programmers have being bothered by problems related to initialization of global objects for a long time. The following language bugs were reported about 8 years ago: #9312 #9115 #9261 #5366 #9360 at github.com/scala/bug. And programmers continue to report such issues for the Scala 3 compiler: #16152 #9176 #11262 at github.com/lampepfl/dotty. Our checker manages to resolve all the bug reports.

REFERENCES

Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 666–679. https://doi.org/10.1145/3009837.3009866

Clement 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 (2022), 729–757. https://doi.org/10.1145/3563314

Egon Börger and Wolfram Schulte. 2000. Initialization problems for Java. *Software-Concepts & Tools* 19 (2000), 175–178.

David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *Proc. ACM Program. Lang.* 1, ICFP (2017), 12:1–12:25. https://doi.org/10.1145/3110256

Laurent Hubert and David Pichardie. 2009. Soundly handling static fields: Issues, semantics and analysis. *Electronic Notes in Theoretical Computer Science* 253, 5 (2009), 15–30.

Dexter Kozen and Matt Stillerman. 2002. Eager class initialization for Java. In *Formal Techniques in Real-Time and Fault-Tolerant Systems: 7th International Symposium, FTRTFT 2002 Co-sponsored by IFIP WG 2.2 Oldenburg, Germany, September 9–12, 2002 Proceedings 7*. Springer, 71–80.

K Rustan M Leino and Peter Müller. 2004. Modular verification of global module invariants in object-oriented programs. *Technical Report/ETH Zurich, Department of Computer Science* 459 (2004).

K. Rustan M. Leino and Peter Müller. 2005. Modular Verification of Static Class Invariants. In *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3582)*, John S. Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki (Eds.). Springer, 26–42. https://doi.org/10.1007/11526841_4

Fengyun Liu, Ondřej Lhoták, Aggelos Biboudis, Paolo G. Giarrusso, and Martin Odersky. 2020. A type-and-effect system for object initialization. 4 (2020), 1–28. Issue OOPSLA. https://doi.org/10.1145/3428243

Fengyun Liu, Ondřej Lhoták, Enze Xing, and Nguyen Cao Pham. 2021. Safe object initialization, abstractly. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Scala*. Association for Computing Machinery, 33–43. https://doi.org/10.1145/3486610.3486895

Martin Odersky. 2019. Scala Language Specification. https://scala-lang.org/files/archive/spec/2.13/.

Xin Qi and Andrew C. Myers. 2009. Masked types for sound object initialization. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 53–65. https://doi.org/10.1145/1480881.1480890

Micha Sharir and Amir Pnueli. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, Steven S Muchnick and Neil D Jones (Eds.). Prentice-Hall, Chapter 7, 189–233.

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 17–30. https://doi.org/10.1145/1926385.1926390

Alexander J. Summers and Peter Mueller. 2011. Freedom before commitment: a lightweight type system for object initialisation. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2011-10-22) *(OOPSLA '11)*. Association for Computing Machinery, 1013–1032. https://doi.org/10.1145/2048066.2048142

W3C. 2022. WebAssembly Core Specification. https://www.w3.org/TR/wasm-core-2/

Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize once, start fast: application initialization at build time. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 184:1–184:29. https://doi.org/10.1145/3360610

Received 2023-04-14; accepted 2023-08-27