Towards Safe Initialization of Scala Global Objects

by

Enze Xing

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Enze Xing 2025

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contribution

This thesis is a result of my collaboration with Fengyun Liu, Ondřej Lhoták and David Hua. The work in this thesis originates from the material presented in the following publications:

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

Based on the safe initialization principles presented in the previous material, this thesis presents the author's work of designing a global object initialization checker (Chapters 2, 3, 4, and 5), implementing the initialization checker in the Scala compiler (Chapter 6), and evaluating the checker (Chapter 7).

Abstract

This thesis focuses on safe initialization of global objects in Scala. Global objects encapsulate global information in Scala, and their initialization is susceptible to causing run-time errors. Moreover, global objects are initialized by demand (i.e. on their first access). The initialization safety of a global object is brittle if it depends on the initialization point of the object, because the initialization point is the first access in the entire program. This motivates the idea of automatically detecting potential initialization errors during compilation.

The main contribution of this thesis is designing and implementing a global object initialization checker in the Scala compiler. Theoretically, we identified run-time errors caused by unsafe initialization patterns of global objects and organized three static principles to enforce on Scala programs: Prohibiting accesses to uninitialized fields, which prevents null pointer exceptions; partial ordering of global object initialization order, which prevents deadlocks between locks that guard the initialization of global objects; and initialization-time irrelevance, which ensures that initialization safety of the global object is independent of the initialization point. We then designed the global object initialization checker by proposing the formal initialization semantics of a Scala initialization calculus, and the initialization checker is presented as an abstract interpreter of the initialization calculus. The initialization checker also checks the initialization process of each global object individually rather than conducting a whole-program analysis.

Practically, we have integrated the abstract interpreter into the Scala compiler after extending the initialization semantics with more Scala features. The initialization checker can be turned on when compiling Scala programs, and we evaluated the initialization checker during the compilation of many widely-used open-source Scala projects which form a test suite. The initialization checker reports warnings in several projects that are verified to be true positives. The result highlights the necessity of checking the initialization safety of Scala projects and the utility of the global object initialization checker in this thesis.

Acknowledgements

First, I must express my sincere gratitude to my supervisor, Professor Ondřej Lhoták. You taught me how to construct the structure of this thesis from scratch, and you also spent so much time checking the details of this thesis. Moreover, you always accommodate my needs during my master's studies and let me explore new possibilities. I have become a beginner researcher under your supervision thanks to your dedication and kindness.

I also cannot thank Dr. Fengyun Liu enough for collaborating with me throughout my masters studies. You always offer insights in every meeting that lead to new and exciting directions. I especially thank you for all the specific guidance when I got stuck in understanding the compiler framework or writing proofs. In addition, I thank all undergraduate students who contributed to this research project, including David, Kavin, and Daisy. This thesis is not an accomplishment of my own, but an accomplishment made by a team.

Next, there are so many people that make me enjoy my six-year studies at the University of Waterloo. I became interested in the design of programming languages and compilers as I took the courses offered by Prof. Lhoták and other professors in the PLG lab, including Prof. Buhr, Prof. Richards, and Prof. Zhang. They also provided valuable feedback for this thesis. I also remembered all the happy memories with my friends Steven, Wendy, and Klaus, especially when we stayed together during the difficult Covid times.

Finally, this thesis means so much to my parents, who sacrificed a lot but still encourage me to explore my path in a foreign country. I express my deepest love for them through writing, in order to compensate for not expressing them enough in life.

Table of Contents

Author's Declaration ii

Statement of Contribution iii

Abstract iv

Acknowledgements v

List of Figures ix

List of Tables xiv

1 Introduction 1

1.1 Motivation and goals 1

1.2 Contribution and overview 6

1.2.1 Run-time guarantee 6

1.2.2 Design 7

1.2.3 Implementation and Evaluation 9

2 An initialization calculus of Scala 11

2.1 Motivation and goals 11

2.2 The immutable initialization calculus 13

2.3 The concrete semantics of the immutable calculus 15
2.3.1 The concrete domain 16
2.3.2 Inheritance and overriding 18
2.3.3 The concrete interpreter 20
2.4 Maintaining run-time guarantees 26

3 Converting to Abstract Initialization Semantics **31**
3.1 Motivation and goals 31
3.2 Handling simple recursive methods 33
3.2.1 Including a cache and Bottom value in the semantics 33
3.2.2 Understanding Bottom 35
3.3 Finitizing the concrete domain 39
3.4 The Abstract Initialization Semantics 42
3.5 Termination and soundness 49
3.6 Producing warnings 56
3.7 Summary 56

4 Checking Initialization Locally **59**
4.1 Motivation and goals 59
4.2 Initialization-time irrelevance 60
4.3 Initialization-time irrelevance for free 62
4.4 Designing a local analysis 67

5 The Mutable Initialization Calculus **70**
5.1 Motivation and goals 70
5.2 Syntax and concrete semantics 71
5.3 Mutation and initialization-time irrelevance 74
5.3.1 Violation of initialization-time irrelevance 74
5.3.2 Restricting access to mutable states 75

6 Implementing the Global Object Initialization Checker for Scala 85
6.1 Motivation and goals 85
6.2 Scalars in abstract domain 86
6.3 Array values in abstract domain 88
6.4 Function values in abstract domain 90
6.5 Inner classes 93

7 Evaluating the Scala Global Object Initialization Checker 103
7.1 Test suite 103
7.2 Statistics 104
7.3 Verification of warnings 105
7.4 Summary 111

8 Related Work and Future Work 112
8.1 Checking initialization safety 112
8.2 The abstract interpretation framework 113
8.3 Optimizing abstract interpreters 114
8.4 Tuning the precision of abstract interpreters 116

9 Conclusion 119

References 121

List of Figures

1.1 A class template and a global object template with fields and methods . . 2
1.2 Accessing uninitialized fields through `this` in Scala 2
1.3 Accessing uninitialized field in the method of another object 3
1.4 Accessing uninitialized fields of global object in method of class instance . 4
1.5 Cyclic initialization of global objects in Scala 5
1.6 A program with mutable fields that may lead to unsoundness 6
1.7 A program without run-time errors that violates Principle 1 7
1.8 A program that violates Principle 2 regardless of the concurrent environment 8
1.9 A program without run-time error that violates Principle 3 8

2.1 Unsafe initialization related to inheritance and polymorphism 12
2.2 Accessing uninitialized fields of global object through class parameters . . 13
2.3 The syntax of the immutable initialization calculus of Scala 14
2.4 Representation of class instances and global objects in the concrete domain 16
2.5 Representation of frames in the concrete domain 17
2.6 The concrete domain of the initialization semantics 17
2.7 The relation between parent and child classes 18
2.8 Algorithm to find the overridden method definition of the call 19
2.9 The evaluation of a parameter accesses 21
2.10 The evaluation rule of direct accesses of global objects 21
2.11 The rules of evaluating `new` expresions 22

2.12 The rules of field selection and method invocation 23
2.13 The initialization process of a global object 24
2.14 The initialization process within a class template 25
2.15 The initialization process of each field 26
2.16 The function of program execution 26
2.17 Evaluating field selection with run-time guarantee 27
2.18 Updated concrete domain to track the current object under initialization . 28
2.19 The updated initialization function according to the domain 28
2.20 The algorithm to detect initialization cycles when referring to a global object 30

3.1 Self-recursive method in the initialization calculus 32
3.2 The repeated EVAL function call when evaluating `foo` 32
3.3 Mutual-recursive methods in the initialization calculus 33
3.4 Cache in the domain 34
3.5 The CACHEEVAL1 when evaluating methods in the initialization calculus . 34
3.6 The domain with cache and Bottom 35
3.7 The recursive rule when evaluating recursion in the abstract semantics . . 36
3.8 The CACHEEVAL function with `Bottom` 37
3.9 Dereferencing the result of infinite recursion in Scala 38
3.10 The trace of cached evaluation starting at Figure 3.2 38
3.11 A program in the initialization calculus that creates infinite number of instances 40
3.12 Another program in the initialization calculus that creates infinite number of instances 40
3.13 The first attempt to define an abstract finitized heap 41
3.14 The abstract domain of the global object initialization checker 42
3.15 A program with two instances with different bodies represented by the same abstract instance 43
3.16 The abstract heap after initializing `O` in Figure 3.15 43

3.17 The abstract evaluation of a parameter accesses 44
3.18 The abstract evaluation of direct accesses of global objects 45
3.19 The abstract evaluation of field selections 45
3.20 The abstract evaluation of method invocation 47
3.21 Infinite recursions of `new` expressions 48
3.22 The abstract evaluation of `new` expressions 48
3.23 The abstract initialization functions 50
3.24 The abstract function of executing programs in the initialization calculus . 51
3.25 The lattice of subsets of finite sets 52
3.26 Illustration of concrete trace and abstract trace 56
3.27 Evaluating field selections with warnings 57
3.28 Preventing deadlocks when evaluating direct references to global objects . . 57

4.1 Deep levels of global object initialization in a project 60
4.2 The initialization point of global objects 61
4.3 A dependent initialization order 65
4.4 Locally checking one global object 67
4.5 Warning to the program in Figure 4.1 68
4.6 Checking global objects at arbitrary order 68
4.7 Checking global objects at arbitrary order with initialization-time irrelevance 69

5.1 A program defining mutable states and mutating them 71
5.2 The syntax of the mutable initialization calculus 72
5.3 The semantics of assignment in the concrete interpreter 73
5.4 The semantics of invoking methods with assignments 75
5.5 A program without initialization-time irrelevance 76
5.6 Another program without initialization-time irrelevance 76
5.7 Extending the concrete domain with the owner of mutable states 78

5.8 Concretely enforcing Principle 5.2 79
5.9 Extending the abstract domain with the owner of mutable states 80
5.10 Extending the abstract domain with the owner of mutable states (continue) 81
5.11 Abstractly enforcing Principle 5.2 82
5.12 The assignment semantics in the abstract interpreter 83

6.1 Definition of class `Int` in Scala 2 library 87
6.2 The abstract semantics of scalar constants 87
6.3 Definition of `Array` class in Scala 88
6.4 Accessing uninitialized field through array access 89
6.5 A program allocating infinite arrays at run-time 89
6.6 The syntax and the abstract domain with arrays 90
6.7 The abstract interpretation of arrays and array operations 91
6.8 The abstract domain and semantics extended with closure values 93
6.9 The abstract interpretation of closure creations and invocations 94
6.10 Initialization calculus with nested class 95
6.11 Initializing inner class instances 95
6.12 Accessing the outer instance in Scala 96
6.13 The relation between outer and inner classes 97
6.14 The first attempt of interpreting C`.this` 98
6.15 Accessing the outer instance of the parent class 98
6.16 The correct interpretation of C`.this` 99
6.17 The interpretation of instantiation of inner instances 100
6.18 The initialization of templates related to outer and inner classes 101
6.19 The abstract interpretation of C`.this` 102

7.1 Accessing uninitialized field in Scala 2 standard library 107
7.2 Cyclic initialization in the Dotty compiler 108

7.3 Deadlock caused by cyclic initialization of global objects 109
7.4 Violation of initialization-time irrelevance in parboiled2 110

8.1 Unreachable instances in the initialization calculus 115
8.2 A program with false positive warning 118
8.3 Scala expressions with regions 118

List of Tables

7.1 Evaluation of the global object initialization checker on Scala open-source projects . . . 105

7.2 Summary of distinct warnings in the test suite . . . 106

Chapter 1

Introduction

1.1 Motivation and goals

Almost every programming language introduces and initializes compound data structures. In many object-oriented programming languages, compound data structures appear in the form of classes, and a program initializes multiple instances of different classes. Each class instance records a list of fields, and the initialization of fields is described in the class definition. However, class instances are inconvenient for recording global information because the lifetime of fields in each instance ends when the instance is destroyed. In order to record global states, earlier object-oriented languages like Java introduce static fields in classes, whose lifetime persists during the execution of the entire program. Recent object-oriented programming languages like Scala and Kotlin organize global information in global objects. Figure 1.1 shows a global object template `O` in Scala and a class template `C` in Scala. Similarly to a class, a global object template introduces fields and methods as members. Unlike classes, a global object encapsulates global information in each field, and a global object is a single entity and can only be created and initialized once. This thesis closely investigates the subtle initialization semantics of global objects in a Scala program, as it is different from the initialization semantics of class instances, and initialization bugs related to global objects may lead to run-time errors that are hard to debug.

Since each global object represents a singleton instance, programmers can directly refer to an object by its name, and the initialization process of a global object starts when it is first directly accessed within the program. This is different from the initialization of class instances, which starts immediately after each `new` expression instantiates a new instance. Each global object then initializes all its fields in the order in which they appear in the

```
class C(x: Int) {
    val f1 = 5
    val f2 = foo()
    def foo(): Int = {
        val y = 6
        x + y
    }
}

object O {
    val f1 = new C(5)
    val f2 = bar()
    def bar(): Int = f1.foo()
}
```

Figure 1.1: A class template and a global object template with fields and methods

```
object O:
    val f1 = this.f2.hashCode() // error
    val f2: Object = ...
```

Figure 1.2: Accessing uninitialized fields through `this` in Scala

```scala
object A {
    val a: Int = B.foo()
}

object B {
    def foo(): Int = A.a.hashCode() // error
}
```

Figure 1.3: Accessing uninitialized field in the method of another object

global object template, and the initialization process ends when all fields are initialized. A global object under initialization is vulnerable because it contains uninitialized fields. Within a global object, accessing a field before its initialization always returns 0 or `null`, and `null` can lead to run-time exceptions. Figure 1.2 shows a simple example where object `O` introduces two fields `f1` and `f2`, and the initializer of `f1` accesses `f2` through `this` before `f2` is initialized. The call to `hashCode` will always lead to an exception regardless of when the initialization process of `O` starts. Accessing uninitialized fields directly within the defining global object usually can be easily detected by programmers or by the compiler with a symbol table, but Figures 1.3 and 1.4 show more complex examples. In Figure 1.3, the initializer of `a` calls the method `foo` defined in `B`, but the body of `foo` then accesses `a` before its initialization by directly referring to `A`. In Figure 1.4, class `C` contains a direct reference to `O` in the method `foo` and then selects the field `f2`. The program is unsafe when it creates an instance of `C` on line 6, before the initialization of `f2`, and line 7 will lead to a null pointer exception. In general, as a Scala project scales, detecting accesses to uninitialized fields of a global object becomes more challenging because it is necessary to visit numerous definitions that are not within the global object itself, and these definitions could be scattered in the project, possibly in different source files.

Reading uninitialized fields is related to the sequential initialization order of fields in global objects, but errors can also be related to the initialization order of global objects themselves in the whole program. A complete Scala program can be seen as a collection of global objects, which includes an entry object named `Main`, and the execution of the program is equivalent to the initialization process of `Main`. During program execution, a global object `A` under initialization will trigger the initialization of `B` if `B` is directly accessed for the first time. The initialization process of `A` then halts and waits for the initialization of `B` to finish. This requires synchronization in the presence of concurrency, so each global object is accompanied by a lock that is owned by the thread that initializes the object, and other threads have to wait for the release of the lock. However, the presence of locks

```
class C {
    def foo() = O.f2
}

object O {
    val f1 = new C
    val f2 = f1.foo().hashCode() // error
}
```

Figure 1.4: Accessing uninitialized fields of global object in method of class instance

can lead to deadlocks, as illustrated in Figure 1.5. In this example, object `Main` creates two threads that initialize `O1` and `O2`, respectively, but the initialization of `O1` requests an access to `O2` on line 3, and the initialization of `O2` requests an access to `O1` on line 7. Therefore, no matter which thread starts first, they will eventually wait for each other and deadlock happens! It is vital to detect such patterns during compilation, since deadlocks are notoriously hard to debug.

The unsafe initialization of `A` in Figure 1.2 and Figure 1.4 will always lead to run-time errors after `A` starts its initialization process. Similarly, unsafe initializations in Figure 1.3 and Figure 1.5 will lead to run-time errors after `A` or `B` starts its initialization process. However, in many cases, whether a run-time error occurs may depend on *when* a global object starts its initialization. Consider the program in Figure 1.6. In this example, classes `X` and `Y` provide two distinct definitions of the method `foo`, and the initialization of the field `b` in object `B` will call one of them. However, the precise version to call depends on the value of the field `A.a` at the initialization point of `B`, and the value of `a` is re-assigned in the `Main` object at some point from an instance of `X` to an instance of `Y`. Remember that global objects are initialized on demand when first accessed. Therefore, the initialization of `B.b` is safe if the first access to `B` occurs before line 10, but the initialization of `B.b` will trigger a null pointer exception if the first access to `B` occurs after line 10. In order to verify the initialization safety of the previous examples, the programmer or compiler needs significant effort to go through the whole project and find the first access of all objects. This motivates software engineering principles to regulate the input project to avoid initialization errors related to the precise initialization point.

To summarize, programmers can rarely detect all potential initialization errors related to initialization of global objects, which demands help from the Scala compiler. The goal of this thesis is to implement a global object initialization checker in the Scala compiler

```
1  object O1 {
2      val a = 5
3      val b = O2.c
4  }
5  object O2 {
6      val c = 10
7      val d = O1.a
8  }
9  def main() = {
10   val createO1 = new Runnable {
11     def run = {
12       val a = O1;
13     }
14   }
15   val createO2 = new Runnable {
16     def run = {
17       val b = O2;
18     }
19   }
20   val t1 = new Thread(createPredef)
21   val t2 = new Thread(createVector)
22   t1.start()
23   t2.start()
24 }
```

Figure 1.5: Cyclic initialization of global objects in Scala

```
class X { def foo(): X = this }
class Y extends X { override def foo(): X = B.b}
object A { var a: X = new X() }
object B {
    val b: X = A.a.foo()
    b.hashCode()
}
def main() = {
    ...
    A.a = new Y()
    ...
}
```

Figure 1.6: A program with mutable fields that may lead to unsoundness

that automatically detects potential run-time errors related to global object initialization when compiling programs.

1.2 Contribution and overview

1.2.1 Run-time guarantee

Based on the previous examples, it is necessary for the global object initialization checker to ensure the following run-time guarantees of the input program:

- **Guarantee 1: Accesses to uninitialized fields are prohibited.** During program execution, there can be no access to uninitialized fields of global objects. This prevents null pointer exceptions caused by dereferencing uninitialized fields of global objects. Note that this principle does not cover dereferencing uninitialized fields of class instances. Liu et al. have developed a class instance initialization checker in the Scala compiler that prohibits accessing uninitialized fields of class instances [15]. Detecting accesses to uninitialized fields of global objects is more challenging because of the unique lazy initialization semantics of global objects in Scala.
- **Guarantee 2: The wait-for graph is acyclic.** The wait-for graph characterizes the waiting orders between global objects during program execution. The input

```
object O {
  val f1 = O.f2 // f1 has value null, but f1 is not deferenced
  val f2 = ...
}
```

Figure 1.7: A program without run-time errors that violates Principle 1

program must form an acyclic wait-for graph regardless of the concurrent setting. Specifically, the initialization order of global objects cannot form cycles. This prevents the initialization processes of different global objects from mutually waiting for each other, causing deadlocks.

- **Guarantee 3: Initialization time is irrelevant.** The initialization process of every global object should be the same regardless of its precise initialization time. Specifically, if the initialization of a global object violates any previous principles during program execution, then it will always violate such principles regardless of the time when it starts its initialization.

Each of the above guarantees is a sufficient condition to prevent specific kinds of run-time errors, though they are not necessary conditions. Based on the first guarantee, the global object initialization checker will reject programs that access uninitialized fields, even if such a program does not dereference the uninitialized field, as illustrated in Figure 1.7. Similarly, the program in Figure 1.8 will be rejected because it contains cycles between `A` and `B`, even if `A` and `B` are initialized by the same thread. Additionally, the program in Figure 1.6 violates initialization-time irrelevance because the safety of the initialization of `B` depends on the initialization point of `B`. The global object initialization will reject a program even if `B` is initialized safely at run-time, as illustrated in Figure 1.9.

1.2.2 Design

The global object initialization checker is implemented as an abstract interpreter. Abstract interpreters are derived from concrete interpreters that execute the program based on its formal concrete semantics. However, abstract interpreters execute the program based on abstract semantics. Compared with concrete interpreters, abstract interpreters leave out some concrete details but achieve additional important properties. Chapters 2, 3, 4, and 5 describe the basis of the abstract interpreter that checks global object initialization in a subset of the Scala language. Specifically:

```
object A {
    val f1 = 5
    val f2 = B.f1
}

object B {
    val f1 = 6
    val f2 = A.f1
}
```

Figure 1.8: A program that violates Principle 2 regardless of the concurrent environment

```
class X { def foo(): Int = 10 }
class Y extends X { override def foo(): Int = B.b}
object A { var a: X = new X() }
object B { val b: Int = A.a.foo().hashCode() }
object Main {
    val f = B.b
    A.a = new Y()
    ...
}
```

Figure 1.9: A program without run-time error that violates Principle 3

- Chapter 2 defines an immutable initialization calculus that includes global objects, class instances, and their members. In the initialization calculus, direct references to global objects are the only channels that violate the run-time guarantees in the initialization calculus. Chapter 2 then defines the formal concrete semantics of the initialization calculus as a definitional interpreter of the input program. The definitional interpreter starts at the program entry point and only accepts programs that do not violate Guarantee 1 and Guarantee 2.

- Chapter 3 defines the abstract interpreter of any input program in the immutable initialization calculus, based on the concrete interpreter in Chapter 2. The key property of the abstract interpreter, different from the concrete interpreter, is its termination on all programs. It is proved to be terminating and to still provide an over-approximation of the concrete interpreter.

- In order to avoid the abstract interpreter in Chapter 3 that starts at the program entry point and visits the whole program, Chapter 4 formally defines initialization-time irrelevance as an extra guarantee. We prove that for any program accepted by the concrete interpreter in Chapter 2, all objects also follow initialization-time irrelevance automatically. By avoiding initialization errors related to the precise initialization points of global objects, the initialization checker can be presented as a local analysis that checks each object individually, in arbitrary order.

- Chapter 5 then extends the initialization calculus with mutable fields and class parameters. Assignments to mutable fields and class parameters after their initialization may break initialization-time irrelevance. In order to maintain initialization-time irrelevance, we propose that all mutable states should only be accessible during the initialization of their owner global object. We then update the initialization checker to verify that each global object follows initialization-time irrelevance before checking its initialization safety.

1.2.3 Implementation and Evaluation

This thesis implements a global object initialization checker that locally checks individual objects based on the abstract interpreter in Chapter 5. Chapter 6 describes how to incorporate the checker into Dotty, the official Scala compiler, as a static analysis pass. The initialization checker reports unsafe global object initialization with a warning designed to be explainable and useful for debugging. The warnings only show the local portion of

the initialization code that violates any static principles, and the initialization code is an excerpt of the source program instead of the intermediate representation in the compiler.

In Chapter 7, the initialization checker is applied to a test suite formed by open-source Scala projects that include complex initialization patterns. The evaluation statistics show that large portions of code have been analyzed by the initialization checker, and 18 projects are flagged with warnings. We verify that some true positive warnings may lead to potential run-time errors and show that they can be refactored to follow the proposed regulations. The evaluation shows that global object initialization errors may occur in real-world Scala projects, and the initialization checker is an effective tool for finding and preventing them.

Chapter 2

An initialization calculus of Scala

This chapter presents an immutable initialization calculus that models global object initialization in Scala. Section 2.1 discusses the object-oriented features in Scala that are relevant to the safety of global object initialization in Scala. Section 2.2 presents the syntax of the initialization calculus that covers all essential features in Section 2.1, and also excludes Scala features that are irrelevant to global object initialization at this point. Section 2.3 then presents a concrete semantics of programs in the initialization calculus with a definitional interpreter that models the program execution.

2.1 Motivation and goals

Scala is an object-oriented language that supports the functional programming paradigm. The initialization calculus presented in this chapter does not aim to model functional programming features that are not relevant to the initialization semantics of global objects. In fact, Scala implements most of its functional features by transforming them into equivalent object-oriented programs. For example, each function value can be transformed into a global object with a special `apply` method. Therefore, the initialization calculus in this chapter is expressive enough to model real-world Scala programs with functional programming features.

A program written in the initialization calculus should be a collection of global objects with fields, and the program directly refers to global objects by name. To express the various unsafe initialization patterns in Chapter 1, the initialization calculus should also at least include the following features:

```scala
class C {
    def foo() = 5
}

class D extends C {
    def foo() = O.f4
}

object O {
    val f1 = new C
    val f2 = f1.foo() // safe
    val f3 = new D
    val f4 = f3.foo() // accessing uninitialized field f4
}
```

Figure 2.1: Unsafe initialization related to inheritance and polymorphism

- Methods of global objects. Figure 1.3 suggests including direct references in the methods of global objects.
- Classes and methods of class instances. Class instances can also have methods with direct references to global objects, and class instances can be created in field initializers or method invocations of global objects, as illustrated in Figure 1.4.
- Inheritance and polymorphism. Inheritance and polymorphism are key features of object-oriented languages. In Scala, each object or class can indicate another class as the parent class, and the global object or child class inherits all fields and methods of its parent class. Polymorphism is achieved by overriding methods of parent classes in child classes. Figure 2.1 shows an example of unsafe initialization related to inheritance and polymorphism. On line 11, calling `foo` on an instance of class `C` is safe. However, class `D` inherits class `C` and overrides the method `foo`, so on line 13, the call to `foo` on an instance of `D` will call the overridden version, which includes direct references to `O` and is unsafe.
- Aliasing. If an initializer of a field evaluates to a direct reference to a global object under initialization, then the field becomes an alias of the reference and also becomes a channel to access uninitialized fields. Moreover, classes and methods in Scala have parameters, and a direct reference to a global object can be provided as an argument when creating an instance or invoking a method, and the parameters become aliases

```
class C(p) {
    def foo() = p.f2
}

object O {
    val f1 = new C(O)
    val f2 = f1.foo() // accessing uninitialized field f2
}
```

Figure 2.2: Accessing uninitialized fields of global object through class parameters

of the direct reference. Figure 2.2 shows an example of aliasing a direct reference to a class parameter and then using the alias to access uninitialized fields.

2.2 The immutable initialization calculus

The syntax of the initialization calculus is presented in Figure 2.3. A program written in this calculus is a collection of class and global object templates, plus a main expression $\mathfrak{e}_{main}$ as the program entry point. Every template must refer to a parent class, represented by $\mathbb{PC}$. `Object` is a class that is not defined in the program and has no parameters or members, so a class that refers to `Object` as the parent does not inherit anything. Global objects also have parent classes but cannot be parents of others and cannot have parameters.

The members defined in a global object include fields and methods, and the members defined in a class include its parameters and methods. Every field in templates comes with an expression as an initializer. Similarly, the bodies of Scala methods in templates are also expressions. Expressions come in several forms, which include parameter access by name (p), direct references to global object by name (O), field selections of global objects ($\mathfrak{e}.f$), `new` expressions that instantiate class instances, and method invocation. When instantiating a class instance, initializing the parameters of the parent class, or invoking a method, the caller needs to provide an expression as argument for each class parameter or method parameter.

The syntax of the initialization calculus includes all essential object-oriented features mentioned in the previous section that are relevant to the initialization process of global objects. There are also some features that are not supported in the syntax for various reasons:

Names:

$$
\begin{array}{rr}
O \in & \mathit{ObjectName} \\
C \in & \mathit{ClassName} \\
f \in & \mathit{FieldName} \\
m \in & \mathit{MethodName} \\
p \in & \mathit{ParamName}
\end{array}
$$

Syntax:

$$
\begin{array}{rrr}
\mathbb{P} ::= & \overline{\mathbb{C}}\ \overline{\mathbb{O}}\ \mathfrak{e}_{main} \in & \mathit{Program} \\
\mathbb{C} ::= & \texttt{class}\ C\texttt{(}\overline{p}\texttt{)}\ \texttt{extends}\ \mathbb{PC}\texttt{(}\overline{\mathfrak{e}}\texttt{)}\ \{\ \overline{\mathbb{M}}\ \} \in & \mathit{ClassTemplate} \\
\mathbb{O} ::= & \texttt{object}\ O\ \texttt{extends}\ \mathbb{PC}\texttt{(}\overline{\mathfrak{e}}\texttt{)}\ \{\ \overline{\mathbb{F}}\ \overline{\mathbb{M}}\ \} \in & \mathit{ObjectTemplate} \\
\mathbb{PC} ::= & \texttt{Object} \mid C \in & \mathit{ParentClassName} \\
\mathbb{F} ::= & \texttt{val}\ f\ \texttt{=}\ \mathfrak{e} \in & \mathit{Field} \\
\mathbb{M} ::= & \texttt{def}\ m\texttt{(}\overline{p}\texttt{)}\ \texttt{=}\ \mathfrak{e} \in & \mathit{Method} \\
\mathfrak{e} ::= & p \mid O \mid \texttt{new}\ C\texttt{(}\overline{\mathfrak{e}}\texttt{)} \mid \mathfrak{e}\texttt{.}f \mid \mathfrak{e}\texttt{.}m\texttt{(}\overline{\mathfrak{e}}\texttt{)} \in & \mathit{Expression}
\end{array}
$$

Notation:

$$
\overline{x} ::= \overbrace{x, x, \cdots, x}^{\geq 0}
$$

Figure 2.3: The syntax of the immutable initialization calculus of Scala

- Fields in classes and local variables in methods. Their initialization processes have already been checked by the compiler and are not the focus of this thesis. Moreover, they do not limit the expressiveness of the existing calculus (local variables can be inlined into the method body, and fields of instances can be computed first and then passed as parameters during instantiation).
- The pointer `this`. It also provides a channel for referring to a global object under initialization. However, detecting unsafe initialization caused by deferencing `this` has been studied in previous works [15]. This thesis focuses on unsafe initialization of global objects caused by direct references, so direct references are the only way to trigger the initialization of global objects and to refer to a global object in this calculus.
- Abstract classes, fields, methods. A Scala class is abstract if and only if it contains some abstract fields without initializers or some abstract methods without bodies. In this calculus, fields only exist in global objects, which cannot be abstract. Abstract classes and methods are also not included in the calculus since the instances of an abstract class can never be instantiated, and an abstract method can never be invoked. Therefore, every field in the calculus contains an initializer and every method in the calculus contains a body.
- The value `null`. Null pointer exceptions can be triggered by deferencing an uninitialized field, but can also be triggered by deferencing a field that is explicitly assigned `null`. This thesis focuses on preventing the first kind of exception, and the second kind of exception is handled by other works [21].
- Mutation. Note that every parameter and every field in this calculus is initialized, but never mutated (fields are qualified by `val`). Scala allows mutable fields, but they add subtle caveats to the safety of initialization of global objects, so they are introduced in Chapter 5.

The supported features in the immutable initialization calculus already form complex semantics, so the next section formally describes the concrete semantics of programs in the initialization calculus.

2.3 The concrete semantics of the immutable calculus

This section will define the formal semantics of executing a program written in the initialization calculus. The semantics is big-step and is presented using a definitional interpreter.

$$
\begin{aligned}
& & \mathit{Addr} &::= & \mathbb{N} \\
v \in & & \mathit{Value} &::= & \mathit{Addr} \cup \{\texttt{null}\} \\
\sigma \in & & \mathit{Heap} &::= & \mathit{Addr} \rightarrow \mathit{Instance} \\
& & \mathit{Instance} &::= & (\mathit{ClassName} \cup \mathit{ObjectName}) \times ((\mathit{FieldName} \cup \mathit{ParamName}) \rightarrow \mathit{Value})
\end{aligned}
$$

Figure 2.4: Representation of class instances and global objects in the concrete domain

The semantics has two components. The first component is a concrete domain that models the run-time environment of the initialization calculus using data structures. The second component is the pseudo-algorithm of the interpreter.

2.3.1 The concrete domain

Scala programs typically run on the Java Virtual Machine (JVM), so the concrete domain should aim to model the run-time environment of the JVM [12]. The key component of the JVM is a heap, which is the run-time data area from which the memory for all class instances is allocated, as illustrated in Figure 2.4. Each valid address can be represented as a natural number. An instance is a class instance created from a `new` expression or a global object. After modeling the heap using a map, the instances should also be modeled using data structures. The JVM specification does not define any standard representations for instances, so any data structure works as long as it records the essential information stored in each instance. Each global object captures the value of fields during its lifetime; similarly each class instance captures the value of the class parameters provided when instantiating the instance. Therefore, the modeling of instances presented in Figure 2.4 contains a map as the second component of each instance. Each key of the map is the name of a field or a parameter, and the value of each key is an address in the heap or the special value `null` as the result of accessing an uninitialized field. This illustrates that instances are passed by reference in Scala. Note that each instance also stores the name of the defining class or the global object in the first component. The name is necessary to find the correct method to call when the method is overridden. Method definitions are not associated with each instance but are associated with each defining template, so they are not stored in the heap.

Apart from the heap, another component of the JVM is the frames. A frame is created

$$\tau \in \quad Frame ::= \quad ParamName \to Value$$

Figure 2.5: Representation of frames in the concrete domain

$$\begin{aligned} \psi \in \quad & Addr \\ (\sigma, \tau, \psi) \in \quad & Conf ::= \quad Heap \times Frame \times Addr \end{aligned}$$

Figure 2.6: The concrete domain of the initialization semantics

for each method invocation and stores the values of local variables during invocation. In the initialization calculus, the frame should only store the values of method parameters, and their values should also be references to instances, as illustrated in Figure 2.5. The execution of programs also involves method call chains, and all frames of the invocations in the chain form the run-time stack in JVM. However, the run-time stack will not be included in the concrete domain. This is because the concrete semantics is presented as a definitional interpreter, and Reynolds made the famous remark that the defined language inherently inherits the run-time stack of the defining language of the definitional interpreter [23]. The concrete domain only needs to record the frame at the top of the stack, which is created for the current method invocation to evaluate.

The complete concrete domain is then presented in Figure 2.6. The domain records the current configuration of the run-time environment, which includes the current heap and the frame of the current method invocation. The heap is updated throughout the program execution as more instances are created. A new frame is created when evaluating a method invocation, and the frame can be destroyed after the method invocation returns, because the method parameters are never captured after the method invocation in the initialization calculus. The remaining component of the configuration is another reference to an instance (ψ). This reference always points to the current instance during the initialization process of any class instance or global object. However, when evaluating a method invocation, ψ switches to point to the receiver instance, because the invoked method can access the members of the receiver instance. In this case, ψ is equivalent to the pointer `this` in the method.

$$\boxed{B \xrightarrow{parent} A}$$

$$\text{PARENT1}\quad \frac{B \text{ extends } A}{B \xrightarrow{parent} A}$$

$$\boxed{B \xrightarrow{ancestor} A}$$

$$\text{ANCESTOR1}\quad \frac{B \xrightarrow{parent} A}{B \xrightarrow{ancestor} A} \qquad \text{ANCESTOR2}\quad \frac{B \xrightarrow{parent} A \qquad C \xrightarrow{ancestor} B}{C \xrightarrow{ancestor} A}$$

Figure 2.7: The relation between parent and child classes

2.3.2 Inheritance and overriding

Every class and object in the initialization calculus identifies a parent class and inherits all members of the parent class. This section provides an informal description of the initialization semantics related to inheritance. Figure 2.7 defines the parent-child relationship and the ancestor relationship between classes and global objects. Note that all ancestors must be classes and that global objects can only appear as children. Each creation of a child instance inherits and records the parameter values of the ancestor classes in the child instance itself, and a child instance can directly access the parameters of its ancestors. This indicates that members of each global object also include the parameters of the ancestors of the global object, so a global object instance should capture the values of all parameters and fields. The initialization order of the parameters of the ancestors is important. After a child instance receives the arguments for the parameters of the child class, it evaluates the argument for the parameters in the parent class according to the `extends` clause of the child class, and then the arguments for the parameters in the higher-order ancestors following the `extends` clause of the parent class. Therefore, the parameters of B are always initialized before the parameters of A if $B \xrightarrow{ancestor} A$.

Apart from accessing parameters of the ancestors, the program can call any method of the ancestors on a child instance, provided that the method is not overridden. In general, when calling a method on an instance of class C, the procedure in Figure 2.8 finds the correct definition among all the overridden definitions in the inheritance chain. The same procedure also applies when calling a method m on a global object named O.

Algorithm 1: $\textsc{BodyOf}(m, C)$

1 Input: The name of the invoked method, and the class of the receiver.
2 Output: The correct definition for invoking method m on an instance of class C.
3 **if** C *has a definition of* m **then**
4 | return the definition of m in C
5 **else if** C *`extends Object`* **then**
6 | cannot find any definition; report error
7 **else if** C *`extends`* B **then**
8 | return $\textsc{BodyOf}(m, B)$

Figure 2.8: Algorithm to find the overridden method definition of the call

After introducing the semantics of initializing parent parameters and calling overridden methods, we define several auxiliary requirements on the input program:

- All names of classes, global objects, fields, and parameters are unique. Then the semantics can find a unique definition when referring to a name. All Scala programs can be manually rewritten to rename all non-unique names. A method name can be non-unique if and only if it overrides the method of the same name in one of its ancestors. An overridden method must also have the same list of parameters as the method defined in any ancestor.

- Every inheritance hierarchy chain of the form $A \xrightarrow{extends} B \xrightarrow{extends} C \cdots$ must have finite length, cannot contain any cycles, and must end with the class `Object`.

We also define several auxiliary helpers that will be used by the concrete interpreter of the initialization calculus:

- $\textsc{ParamsOf}(C)$ returns the parameters of the template of class C.

- $\textsc{ParamsOf}(m)$ returns the parameters of method m.

- $\textsc{FieldsOf}(O)$ returns the fields defined in the template of the global object O.

- $\textsc{BodyOf}(m, C)$ is defined to return the definition of the overridden method to call on instances of C, as defined in Figure 2.8. The same applies for $\textsc{BodyOf}(m, O)$.

- $\textsc{TemplateOf}(C)$ returns the template of the class named C.

2.3.3 The concrete interpreter

This section defines the big-step semantics of the initialization calculus by presenting a definitional interpreter that models the execution of programs in the initialization calculus. The definitional interpreter contains functions that define the initialization process of global objects and class instances or describe the evaluation of each kind of expression in a concrete configuration. The initialization process of one instance may then trigger the initialization of others, and the evaluation of a compound expression involves the evaluation of the sub-expressions, so most of the functions in the interpreter are recursive or mutually recursive. The definitional interpreter can be implemented in any programming language that supports recursive function calls with run-time stacks.

Parameter access. The definitional interpreter can start by defining the evaluation process of every expression in a concrete configuration. Figure 2.9 illustrates the signature of the function EVAL that evaluates expressions. The evaluation depends on all components in a concrete configuration. Evaluating an expression may change the heap by creating more instances, so the output also includes a heap. The frame is an input but is not changed during the evaluation of any expression because all parameters are immutable. Finally, the result of evaluating the expression is always an element of $Value$ in the concrete domain, which is an address of an instance or `null`. Figure 2.9 then illustrates the procedure of evaluating a parameter access. If the parameter is a method parameter, then the method parameter must be a key in the frame, and the value is looked up from the frame. Otherwise, the parameter must be a member of the instance with address ψ. The parameter could be defined in the class of the underlying instance or inherited from the ancestors of the underlying instance. In either case, the parameter is the key of the map recorded in the instance and the function returns the corresponding value by looking it up in the heap. Note that parameter accesses will never change the heap.

Direct references of global objects. Figure 2.10 then defines the evaluation of direct references to global objects. Suppose the object O has already been initialized or is under initialization, then the instance of O must exist in the heap under some address n. The evaluation function then returns the address of the global object instance with the heap intact. If the heap does not contain any instance of O, then the instance is created in a fresh address, which is the size of the domain of the heap. Every new instance will also increase the heap size by 1, so the size of the domain of the heap should always be the address of a fresh instance. The heap then records the address with the fresh instance with name tag O. The fresh instance does not record the value of any parameters or fields at this point.

$$\textsc{Eval} :: Heap \times Frame \times Addr \times Expression \to Value \times Heap$$

Algorithm 2: $\textsc{Eval}(\sigma, \tau, \psi, p)$

```
1 if p ∈ dom(τ) then
2 │ return (τ(p), σ)
3 else
4 │ (_, ρ) ← σ(ψ)
5 │ assert(p ∈ dom(ρ))
6 │ return (ρ(p), σ)
```

Figure 2.9: The evaluation of a parameter accesses

Algorithm 3: $\textsc{Eval}(\sigma, \tau, \psi, O)$

```
1 if ∃n, σ(n) = (O, _) then
2 │ return (n, σ)
3 else
4 │ ψ' ← |dom(σ)|
5 │ σ ← σ ∪ [ψ' ↦ (O, [])]
6 │ 𝕆 ← TemplateOf(O)
7 │ σ ← initializeObject(𝕆, σ, ψ')
8 │ return (ψ', σ)
```

Figure 2.10: The evaluation rule of direct accesses of global objects

Finally, the initialization process of the object is triggered by calling $\textsc{initializeObject}$, which is defined later. $\textsc{initializeObject}$ will record parameter values of the ancestors of O as well as fields in O, and the evaluation returns the address of the fresh instance and the updated heap.

Class instantiation. Figure 2.11 then defines the evaluation of `new` expressions. The first step is to evaluate the arguments of the `new` expression. Every `new` expression always creates a fresh instance at a fresh address which equals to the size of the heap. Then the heap associates the fresh address with the fresh instance, containing only the values of the parameters defined in C at this time. Finally, the initialization process of the class of the instance is defined in $\textsc{initializeClassTemplate}$, which will record the ancestor

Algorithm 4: $\text{EVAL}(\sigma, \tau, \psi, \texttt{new}\ C(\overline{\mathfrak{e}_i}))$

1 $\rho \leftarrow []$
2 $\overline{p_i} \leftarrow \text{PARAMSOF}(C)$
3 **foreach** i **do**
4   $(v_i, \sigma_i) \leftarrow \text{EVAL}(\sigma, \tau, \psi, \mathfrak{e}_i)$
5   $\sigma \leftarrow \sigma_i$
6   $\rho \leftarrow \rho \cup [p_i \mapsto v_i]$
7 $\psi' \leftarrow |dom(\sigma)|$
8 $\sigma \leftarrow \sigma \cup [\psi' \mapsto (C, \rho)]$
9 $\sigma \leftarrow \text{INITIALIZECLASSTEMPLATE}(C, \sigma, \psi')$
10 return (ψ', σ)

Figure 2.11: The rules of evaluating `new` expresions

parameters of the instance. The evaluation of the `new` expression then returns the address of the fresh instance.

Field selections and method invocations. The final set of rules in Figure 2.12 is related to field selections and method invocations. When evaluating a field selection, the receiver is first evaluated. The value of the receiver cannot be `null`, otherwise an exception is thrown due to dereferencing `null`. In fact, the value of the receiver must be a global object instance, and we look up the initialized fields recorded in the instance. The value of the field is returned if it is initialized, otherwise `null` is returned as the result of accessing an uninitialized field. When evaluating a method invocation, the receiver and all arguments are evaluated first. The value of the receiver cannot be `null`, otherwise an exception is thrown due to dereferencing `null`. A new frame is created for this invocation that associates each parameter of the method with the value of the argument. Finally, the interpreter looks up the correct body of the method based on the class tag of the receiver, and the body is evaluated in the updated heap and frame, with the address of the receiver.

Initialization of global objects. The definitional interpreter then defines the initialization process of a global object after its initialization is triggered, which is introduced in Figure 2.13. When the initialization process starts, the instance should already be created and added to the heap when calling INITIALIZEOBJECT, and the function takes the location ψ of the fresh global object instance in the heap. The initialization of global objects does not need any frames because it does not depend on any method parameter, but it

Algorithm 5: $\textsc{Eval}(\sigma, \tau, \psi, \mathfrak{e}\,.\,f)$

```
1  (v, σ') ← Eval(σ, τ, ψ, 𝔢)
2  if v = null then
3  |  throw NullPointerException
4  (O, ρ) ← σ'(v)
5  if f ∈ dom(ρ) then
6  |  return (ρ(f), σ')
7  else
8  |  return (null, σ')
```

Algorithm 6: $\textsc{Eval}(\sigma, \tau, \psi, \mathfrak{e}\,.\,m(\overline{\mathfrak{e}_i}))$

```
1  (v, σ_0) ← Eval(σ, τ, ψ, 𝔢)
2  if v = null then
3  |  throw NullPointerException
4  (C/O, _) ← σ_0(v)
5  σ ← σ_0
6  p̄_i ← paramsOf(m)
7  τ' = []
8  foreach i do
9  |  (v_i, σ_i) ← Eval(σ, τ, ψ, 𝔢_i)
10 |  σ ← σ_i
11 |  τ' ← τ' ∪ [p_i ↦ v_i]
12 𝔢' ← BodyOf(C/O, m)
13 return Eval(σ, τ', v, 𝔢')
```

Figure 2.12: The rules of field selection and method invocation

$$\textsc{initializeObject} :: (ObjectTemplate, Heap, Addr) \to Heap$$

Algorithm 7: $\textsc{initializeObject}(\mathbb{O}, \sigma, \psi)$

1 `object` O `extends` $\mathbb{PC}$`(`$\overline{\mathfrak{e}_i}$`)` { $\overline{\mathbb{F}_j}$ $\overline{\mathbb{M}}$ } $\leftarrow \mathbb{O}$
2 $\overline{p_i} \leftarrow \textsc{paramsOf}(\mathbb{PC})$
3 **foreach** i **do**
4 $\quad (v_i, \sigma_i) \leftarrow \textsc{eval}(\sigma, \{\}, \psi, \mathfrak{e}_i)$
5 $\quad (O, \rho) \leftarrow \sigma_i(\psi)$
6 $\quad \sigma \leftarrow \sigma_i[\psi \mapsto (O, \rho[p_i \mapsto v_i])]$
7 $\sigma \leftarrow \textsc{initializeClassTemplate}(\sigma, \psi, \mathbb{PC})$
8 **foreach** j **do**
9 $\quad \sigma \leftarrow \textsc{initializeField}(\sigma, \psi, \mathbb{F}_j)$
10 return σ

Figure 2.13: The initialization process of a global object

may create more class instances and trigger the initialization of more global objects, so the heap is constantly updated. The global object initialization process consists of evaluating the arguments of the parent class, which calls the EVAL function, then following the initialization process of the parent template, which calls the INITIALIZECLASSTEMPLATE function defined later. Note that the values of the parent arguments are also recorded in the global object instance at location ψ, and the initialization process of the parent class also operates on the global object instance at location ψ. Finally, the fields in the object are initialized in order by calling the INITIALIZEFIELD function defined later. When the initialization process ends, the latest heap is returned.

Initialization of class templates. Next, the function INITIALIZECLASSTEMPLATE defines the initialization process within a parent class template, as illustrated in Figure 2.14. Similarly to the function INITIALIZEOBJECT, INITIALIZACLASSTEMPLATE has an input heap, does not depend on any frame, and operates on an existing instance that has been added to the heap at location ψ. In fact, the underlying instance at the location ψ may not be of class C, but could be of another class C' where $C' \xrightarrow{ancestor} C$, or a global object O where $O \xrightarrow{ancestor} C$. The function also assumes that the arguments passed to the template are already evaluated and recorded in the heap. The special class `Object` has an empty

$\textsc{initializeClassTemplate} :: (ParentClassName, Heap, Addr) \rightarrow Heap$

Algorithm 8: $\textsc{initializeClassTemplate}(C, \sigma, \psi)$

```
1 if C = Object then
2 |  return σ
3 class C(p̄_i) extends ℙℂ(ē_j) { M̄ } ← TemplateOf(C)
4 p̄_j ← paramsOf(ℙℂ)
5 foreach j do
6 |  (v_j, σ_j) ← Eval(σ, {}, ψ, e_j)
7 |  (C′/O, ρ) ← σ_j(ψ)
8 |  σ ← σ_j[ψ ↦ (C′/O, ρ[p_j ↦ v_j])]
9 return initializeClassTemplate(ℙℂ, σ, ψ)
```

Figure 2.14: The initialization process within a class template

template with no parameters, so the initialization process of `Object` does nothing. For a regular class C defined in the program, initialization within the template of C also involves evaluating the arguments of its parent class. Each argument is still recorded in the instance at the location ψ. Finally, the function recursively follows the initialization process of the parent class template.

Initialization of fields. Figure 2.15 then defines the initialization process of each field in a global object. The function also takes an input heap and the location of the underlying global object instance in the heap and does not depend on any frames. The initialization process simply evaluates the initializer and associates the field with the value of the initializer in the resulting heap. Note that the global object instance records the arguments of the parent class template and the values of each field, because the lifetimes of parameters and fields are bound with the underlying instance.

Program execution. Finally, we can define the function that executes an entire program in the initialization calculus. The execution of the program is equivalent to evaluating the main expression in an empty configuration, as illustrated in Execute in Figure 2.16. The main expression initializes other global objects and the program returns the result of the main expression. This completes the interpreter that models the concrete execution of the programs in the immutable initialization calculus.

$$\textsc{initializeField} :: (Field, Heap, Addr) \rightarrow Heap$$

Algorithm 9: $\textsc{initializeField}(\mathbb{F}, \sigma, \psi)$

1 val f = $\mathfrak{e} \leftarrow \mathbb{F}$
2 $(v, \sigma') \leftarrow \textsc{Eval}(\sigma, \{\}, \psi, \mathfrak{e})$
3 $(O, \rho) \leftarrow \sigma'(\psi)$
4 return $\sigma'[\psi \mapsto (O, \rho \cup [f \mapsto v]]$

Figure 2.15: The initialization process of each field

$$\textsc{Execute} :: List[ClassTemplate] \times List[ObjectTemplate] \times Expression \rightarrow Value \times Heap$$

Algorithm 10: $\textsc{Execute}(\overline{\mathbb{C}_i}, \overline{\mathbb{O}_j}, \mathfrak{e}_{main})$

1 return $\textsc{Eval}([], [], 0, \mathfrak{e}_{main})$

Figure 2.16: The function of program execution

2.4 Maintaining run-time guarantees

At this point, the concrete interpreter truthfully executes programs in the initialization calculus according to how they are executed on JVM. Specifically, certain programs may throw null pointer exceptions caused by reading and dereferencing uninitialized fields. One difference is that the concrete interpreter does not model the concurrent environment in the JVM due to its complexity, but the initialization calculus can represent programs that may result in deadlocks caused by global object initialization in a specific JVM concurrent environment, just like the program in Figure 1.5. As a first step to avoiding both kinds of run-time errors, we can slightly modify the concrete interpreter to reject the programs that may fail at run-time due to null pointer exceptions and deadlocks. For each kind of run-time error, the concrete interpreter will maintain a run-time guarantee of the input program:

For null exceptions. Considering that programmers rarely intentionally read uninitialized fields, it is therefore beneficial to prohibit all reads of uninitialized fields during concrete execution. This leads to a slight modification to the evaluation function on field

Algorithm 11: EVAL'$(\sigma, \tau, \psi, \mathfrak{e} \,.\, f)$

1 $(v, \sigma') \leftarrow$ EVAL'$(\sigma, \tau, \psi, \mathfrak{e})$
2 $(O, \rho) \leftarrow \sigma'(v)$
3 $assert(f \in dom(\rho))$
4 return $(\rho(f), \sigma')$

Figure 2.17: Evaluating field selection with run-time guarantee

selections, illustrated in Figure 2.17. The function ensures that the selected field must be recorded in the global object instance in the heap, which means the field is initialized. Without reading uninitialized fields, `null` can be excluded from the set of possible values, and the evaluation function for field selections and method invocations will never throw null pointer exceptions.

For deadlocks. Deadlocks exist because of locks that guard the initialization of global objects. The deadlock problem is often illustrated in terms of wait-for graphs [24], where each directed edge $p_1 \to p_2$ suggests that the process p_1 depends on the resource held by process p_2 and may wait for the lock of p_2. In terms of the initialization process of global objects, each node can be represented by the name of a global object. If a global object O_1 directly refers to a different object O_2 during the initialization of O_1, then O_1 depends on the resources that are allocated during the initialization of O_2, so O_1 has an edge to O_2 in the wait-for graph. If this edge corresponds to the first access to O_2 in the program, then O_1 will trigger the initialization of O_2 by switching to the initialization process of O_2.

However, the initialization process of O_1 may also span the instantiation of multiple class instances, and the current concrete domain loses track of the current global object under initialization when instantiating class instances. This leads to a slight modification to the concrete domain, as illustrated in Figure 2.18. The concrete configuration always includes an additional component, $O_{current}$, to keep track of the current global object under initialization. With the updated concrete domain, we also modify the concrete interpreter to follow the updated domain, as illustrated in Figure 2.19. During the initialization of an object O represented by a call to INITIALIZEOBJECT', all evaluations set the value of $O_{current}$ to O, replacing the previous value of $O_{current}$. The evaluation function will also be updated to EVAL' based on the updated concrete domain, but in most cases, EVAL' will pass the extra argument $O_{current}$ to the sub-processes without changing it, so we omit the full definition of EVAL' here. Now we can formally define the wait-for graph for global object initialization:

$$(\sigma, \tau, \psi, O_{current}) \in \quad Conf ::= \quad Heap \times Frame \times Addr \times ObjectName$$

Figure 2.18: Updated concrete domain to track the current object under initialization

Algorithm 12: INITIALIZEOBJECT'($\mathbb{O}, \sigma, \psi$)

1 `object` O `extends` $\mathbb{PC}(\overline{\mathfrak{e}_i})$ { $\overline{\mathbb{F}_j}$ $\overline{\mathbb{M}}$ } $\leftarrow \mathbb{O}$
2 $\overline{p_i} \leftarrow$ PARAMSOF($\mathbb{PC}$)
3 **foreach** i **do**
4 　$(v_i, \sigma_i) \leftarrow$ EVAL'($(\sigma, \{\}, \psi, O), \mathfrak{e}_i$)
5 　$(O, \rho) \leftarrow \sigma_i(\psi)$
6 　$\sigma \leftarrow \sigma_i[\psi \mapsto (O, \rho[p_i \mapsto v_i])]]$
7 $\sigma \leftarrow$ INITIALIZECLASSTEMPLATE'($\sigma, \psi, O, \mathbb{PC}$)
8 **foreach** j **do**
9 　$\sigma \leftarrow$ INITIALIZEFIELD'($\sigma, \psi, O, \mathbb{F}_j$)
10 return σ

Algorithm 13: INITIALIZECLASSTEMPLATE'($C, \sigma, \psi, O_{current}$)

1 **if** $C =$ *`Object`* **then**
2 　return σ
3 `class` $C(\overline{p_i})$ `extends` $\mathbb{PC}(\overline{\mathfrak{e}_j})$ { $\overline{\mathbb{M}}$ } $\leftarrow$ TEMPLATEOF(C)
4 $\overline{p_j} \leftarrow$ PARAMSOF($\mathbb{PC}$)
5 **foreach** j **do**
6 　$(v_j, \sigma_j) \leftarrow$ EVAL'($(\sigma, \{\}, \psi, O_{current}), \mathfrak{e}_j$)
7 　$(C'/O, \rho) \leftarrow \sigma_j(\psi)$
8 　$\sigma \leftarrow \sigma_j[\psi \mapsto (C'/O, \rho[p_j \mapsto v_j])]]$
9 return INITIALIZECLASSTEMPLATE'($\mathbb{PC}, \sigma, \psi, O_{current}$)

Algorithm 14: INITIALIZEFIELD'($\mathbb{F}, \sigma, \psi, O_{current}$)

1 `val` f `=` $\mathfrak{e} \leftarrow \mathbb{F}$
2 $(v, \sigma') \leftarrow$ EVAL'($(\sigma, \{\}, O_{current}, \psi), \mathfrak{e}$)
3 $(O_{current}, \rho) \leftarrow \sigma'(\psi)$
4 return $\sigma'[\psi \mapsto (O, \rho \cup [f \mapsto v])]$

Figure 2.19: The updated initialization function according to the domain

Definition 2.1: $O_1 \to O_2$ in the wait-for graph if and only if there exists a call

$$\textsc{Eval}((\sigma, \tau, \psi, O_{current}), O_2)$$

illustrated in Algorithm 13 during the concrete program execution.

With the definition of the wait-for graph, a classic solution to prevent deadlocks caused by mutual waiting is to ensure an acyclic wait-for graph. For the concrete interpreter, one approach is to construct the wait-for graph during the concrete interpretation and then detect the cycles. The other approach relies on a further insight: If O_2 is fully initialized during the initialization of O_1, then O_2 does not wait for the initialization of O_1. Conversely, if O_2 is partially initialized during the initialization of O_1, then O_2 waits for the initialization of O_1 and there is a path from O_2 to O_1 in the wait-for graph. Using this approach to prevent deadlocks leads to a modification to the object access semantics, which is presented in Figure 2.20. When evaluating an object access, EVAL' detects an edge from $O_{current}$ to O, where $O_{current}$ is the current object under initialization, and checks the status of O:

- If the object O to be accessed is not yet in the heap, then this is the first access to O that will trigger the initialization of O. The interpreter still draws the edge from $O_{current}$ to O, but may detect cycles after triggering the initialization of O.
- If O is in the heap and is also fully initialized (i.e., all fields are in the heap), then the initialization lock of O is unlocked and O is not in any initialization cycles. In this case, the interpreter safely draws the edge from $O_{current}$ to O and returns the address of O.
- If O is in the heap but is partially initialized (i.e., not all fields are in the heap), then it is locked and waiting for the initialization of $O_{current}$ to finish. In this case, drawing the edge from $O_{current}$ to O will result in a cycle that reflects the mutual wait between $O_{current}$ and O, and the interpreter rejects the program unless $O_{current}$ is accessing itself (i.e., $O_{current} = O$).

Now we can define another concrete interpreter that calls EVAL' on $\mathfrak{e}_{main}$, and we can prove that this interpreter will reject all programs whose wait-for graph contains a cycle of length more than 1.

Theorem 2.2: If the wait-for graph of a program contains a cycle involving more than one global object, then EXECUTE' of the interpreter in this section will reject the program.

Algorithm 15: EVAL'$((\sigma, \tau, \psi, O_{current}), O)$

```
1  if ∃n, σ(n) = (O, ρ) then
2  |   if O_current ≠ O then
3  |   |   assert(|dom(ρ)| = |FieldsOf(O)|)
4  |   return (n, σ)
5  else
6  |   ψ' ← |dom(σ)|
7  |   σ ← σ ∪ [ψ' ↦ (O, [])]
8  |   𝕆 ← TemplateOf(O)
9  |   σ ← initializeObject(𝕆, σ, ψ')
10 |   return (ψ', σ)
```

Figure 2.20: The algorithm to detect initialization cycles when referring to a global object

Proof: Suppose that the wait-for graph contains a cycle involving $\{O_1, \cdots, O_n\}(n > 1)$. Let O_i be the first object in $\{O_1, \cdots, O_n\}$ whose initialization process starts during EXECUTE'. Then the initialization of O_i will trigger the initialization of O_{i+1}, etc. Following the cycle, the concrete interpreter will eventually trigger the initialization of O_{i-1} (or O_n if $i = 1$), during which all objects in $\{O_1, \cdots, O_n\}$ are partially initialized. Then, the program will be rejected when O_{i-1} accesses O_i.

Note that the concrete interpreter only checks for cycles that involve more than one object. It always permits direct references to the current object under initialization (i.e. self-loops in the wait-for graphs), since this is the only pattern to access the initialized fields in $O_{current}$. Self-loops in wait-for graphs also do not cause deadlock because we assume that the initialization process of a single object occurs in one single thread, and the thread cannot wait for itself.

In conclusion, the concrete interpreter based on the updated concrete domain in this section rejects programs with run-time errors caused by global object initialization, and the global object initialization checker defined in this thesis must also achieve this. However, this concrete interpreter still follows the concrete execution of Scala programs. The following chapter will over-approximate the concrete interpreter in this section and lead to an abstract whole program interpreter, which fits as a static analysis.

Chapter 3

Converting to Abstract Initialization Semantics

This chapter converts the concrete interpreter in Section 2.4 into an abstract interpreter that always terminates. Section 3.1 identifies the potential non-termination in the concrete semantics caused by recursive function calls. Section 3.2 then proposes a fixed point evaluation model with cache to ensure termination on simple programs with recursion. To ensure termination on all programs, Section 3.3 presents an abstract domain with finite states, and the abstract initialization semantics is formally defined in Section 3.4 on the abstract domain. Section 3.5 provides a sketch proof that the abstract initialization semantics will always terminate and claims that it is a sound over-approximation of the concrete semantics, and Section 3.6 discusses how to report warnings for unsafe initialization patterns in the abstract semantics before they cause run-time errors.

3.1 Motivation and goals

The concrete interpreter in Section 2.4 ensures that the input program will not cause any run-time errors due to unsafe global object initialization. However, we cannot directly adopt the concrete interpreter of the initialization calculus due to an undesirable property: Non-termination. The initialization calculus in the previous chapter contains programs with infinite recursion. When interpreting such programs, the concrete interpreter does not terminate, but the global object initialization checker is required to terminate on all programs as part of the compiler framework.

```
1   object O {
2       def foo() = O.foo()
3       val b = O.foo()
4   }
```

Figure 3.1: Self-recursive method in the initialization calculus

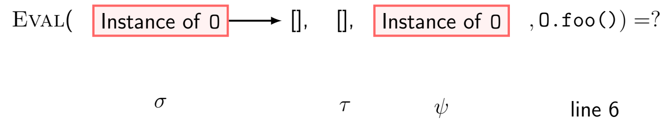


Figure 3.2: The repeated EVAL function call when evaluating `foo`

The syntax of the initialization calculus in Section 2.1 is capable of expressing recursive methods in two forms: a self-recursive method contains a call to itself in its body, and mutual recursive methods are composed of multiple methods with a cyclic call chain. Figure 3.1 illustrates a self-recursive method `foo` in object `O`. This self-recursive method is infinitely recursive and cannot terminate because the initialization calculus does not support conditional expressions. When evaluating `foo` in Figure 3.1, the concrete semantics repeatedly tries to evaluate `O.foo()` in the concrete configuration illustrated in Figure 3.2. Even if we extend the initialization calculus with conditional expressions, infinitely recursive methods still exist, and the initialization checker should terminate on them.

Similarly, Figure 3.3 illustrates the mutually-recursive methods `foo` and `bar`. The concrete interpreter will alternatively evaluate the call to `foo` and `bar`, resulting in non-termination. However, repetition is meaningless because every call to `foo` or `bar` is evaluated in the same concrete configuration. Since the evaluation function only depends on the configuration and the expression, the interpreter should realize when it evaluates the same expression in the same configuration for the second time and exit early.

This chapter aims to define an abstract initialization semantics of the initialization calculus that terminates on all programs in the initialization calculus. In order to ensure termination, the abstract semantics must over-approximate the result of an expression evaluation that does not terminate. The over-approximation will introduce imprecision compared with the concrete interpretation result, so the abstract semantics must be sound: If the input program suffers from unsafe initialization in the concrete semantics, then the unsafe initialization must be exposed by the abstract semantics.

```
1    class C {}
2    object O {
3        def foo(c: C) = O.bar(c)
4        def bar(c: C) = O.foo(c)
5        val f = O.foo(new C)
6    }
```

Figure 3.3: Mutual-recursive methods in the initialization calculus

3.2 Handling simple recursive methods

3.2.1 Including a cache and Bottom value in the semantics

The global object initialization checker must terminate on all programs with infinite recursion, in order to integrate the checker into the Scala compiler. The previous observation hints that the global object initialization checker can always terminate early when evaluating a recursive call in the same concrete state that has appeared before in the call chain, provided that the previous evaluation remembers its output. Darais et al. are inspired by this observation and suggest caching the result of previous evaluations when designing a terminating interpreter for the lambda calculus [4]. Therefore, Figure 3.4 introduces a cache for the initialization calculus, which maps the input of the EVAL function to the output of the EVAL function. With a cache that remembers the result of evaluating certain expressions, we can define a CACHEEVAL function that takes advantage of the cache. As illustrated in Figure 3.5, the CACHEEVAL1 function takes the input of the EVAL function, plus a cache that records previous evaluation results. CACHEEVAL1 starts by checking whether the input is already in the input cache and directly returns the result in the cache in that case. If the input configuration is not in the cache, then CACHEEVAL1 follows the procedure of the regular EVAL function, and the result of the evaluation is cached for future use. The cache is also included in the output of CACHEEVAL1 because the current evaluation process modifies the cache. CACHEEVAL1 is designed to handle infinite recursion, so currently it only caches the evaluation of method calls, which may lead to infinite recursion.

However, CACHEEVAL1 still does not terminate when evaluating the call to `foo` on line 3 of Figure 3.1, because the cache never contains any previous results of evaluating `foo`. The second recursive call to `foo` on line 3 is expected to terminate when the cache contains the result of the first recursive call, but it raises a chicken-and-egg problem: The second call

$$
\begin{aligned}
((\sigma, \tau, \psi, O_{current}), \mathfrak{e}) \in \quad & Conf \times Expression ::= \quad & Input \\
(v, \sigma) \in \quad & Value \times Heap ::= \quad & Output \\
\Gamma \in \quad & Input \rightarrow \text{Output} ::= \quad & Cache
\end{aligned}
$$

Figure 3.4: Cache in the domain

$$\text{CACHEEVAL1} :: (Input \times Cache) \rightarrow (Output \times Cache)$$

Algorithm 16: CACHEEVAL1$((\sigma, \tau, \psi, O_{current}), \mathfrak{e} . m (\overline{\mathfrak{e}_i}), \Gamma)$

1 **if** $((\sigma, \tau, \psi, O_{current}), \mathfrak{e} . m (\overline{\mathfrak{e}_i})) \in dom(\Gamma)$ **then**
2   return $(\Gamma((\sigma, \tau, \psi, O_{current}), \mathfrak{e} . m (\overline{\mathfrak{e}_i})), \Gamma)$
3 **else**
4     $(v, \sigma_0, \Gamma_0) \leftarrow$ CACHEEVAL1$((\sigma, \tau, \psi, O_{current}), \mathfrak{e}, \Gamma)$
5     $(C/O, _) \leftarrow \sigma_0(v)$
6     $\sigma \leftarrow \sigma_0$
7     $\Gamma \leftarrow \Gamma_0$
8     $\overline{p_i} \leftarrow$ PARAMSOF(m)
9     $\tau' \leftarrow []$
10   **foreach** i **do**
11         $(v_i, \sigma_i, \Gamma_i) \leftarrow$ CACHEEVAL1$((\sigma, \tau, \psi, O_{current}), \mathfrak{e}_i, \Gamma)$
12         $\sigma \leftarrow \sigma_i$
13         $\Gamma \leftarrow \Gamma_i$
14         $\tau' \leftarrow \tau' \cup [p_i \mapsto v_i]$
15     $\mathfrak{e}' \leftarrow$ BODYOF$(C/O, m)$
16     $(v', \sigma', \Gamma') \leftarrow$ CACHEEVAL1$((\sigma, \tau', v, O_{current}), \mathfrak{e}', \Gamma)$
17     $\Gamma'' \leftarrow \Gamma' \cup [((\sigma, \tau, \psi, O_{current}), \mathfrak{e} . m (\overline{\mathfrak{e}_i})) \mapsto (v', \sigma')]$
18   return (v', σ', Γ'')

Figure 3.5: The CACHEEVAL1 when evaluating methods in the initialization calculus

$$
\begin{aligned}
v \in \quad & \mathit{Value} ::= \quad && \mathit{Addr} \cup \{\texttt{Bottom}\} \\
((\sigma, \tau, \psi, O_{current}), \mathfrak{e}) \in \quad & \mathit{Conf} \times \mathit{Expression} ::= \quad && \mathit{Input} \\
(v, \sigma) \in \quad & \mathit{Value} \times \mathit{Heap} ::= \quad && \mathit{Output} \\
\Gamma \in \quad & \mathit{Input} \rightarrow \text{Output} ::= \quad && \mathit{Cache}
\end{aligned}
$$

Figure 3.6: The domain with cache and Bottom

to `foo` occurs during the evaluation of the first call to `foo`, so what value should be placed in the cache associated with the first call before the evaluation of the first call finishes? The solution is to introduce a special value `Bottom`, which is the default value in the cache for every key before evaluating the first recursive call. The domain with the addition of cache and `Bottom` is illustrated in Figure 3.6. During the evaluation of parameters and the body of the recursive call to `foo`, the cache then associates `Bottom` as the hypothesis result of evaluating `foo`, which allows the second recursive call to terminate. Figure 3.7 then introduces the function CacheEval2. The only difference between CacheEval1 and CacheEval2 is the inclusion of the step of associating `Bottom` before continuing to evaluate the body of the method call. Finally, CacheEval2 still only caches the evaluation of method calls. The evaluation of the remaining expression should not modify the input cache.

It is worth noting that `Bottom` is an abstract value of a concretely non-terminating expression that initializes nothing. All other values are references to instances and can be dereferenced during field selection and method invocation, but `Bottom` should not be dereferenced. Now we need to specially handle the case when the cached evaluation of the receiver returns `Bottom`. Since all field selections and method invocations on `Bottom` are unreachable at run-time, the interpreter should ignore any such operations on `Bottom` and return `Bottom` again, which is illustrated in Figure 3.8. For example, the abstract interpreter will ignore the call to the method `bar` in Figure 3.9, because the receiver evaluates to `Bottom`, and the call is concretely unreachable.

3.2.2 Understanding Bottom

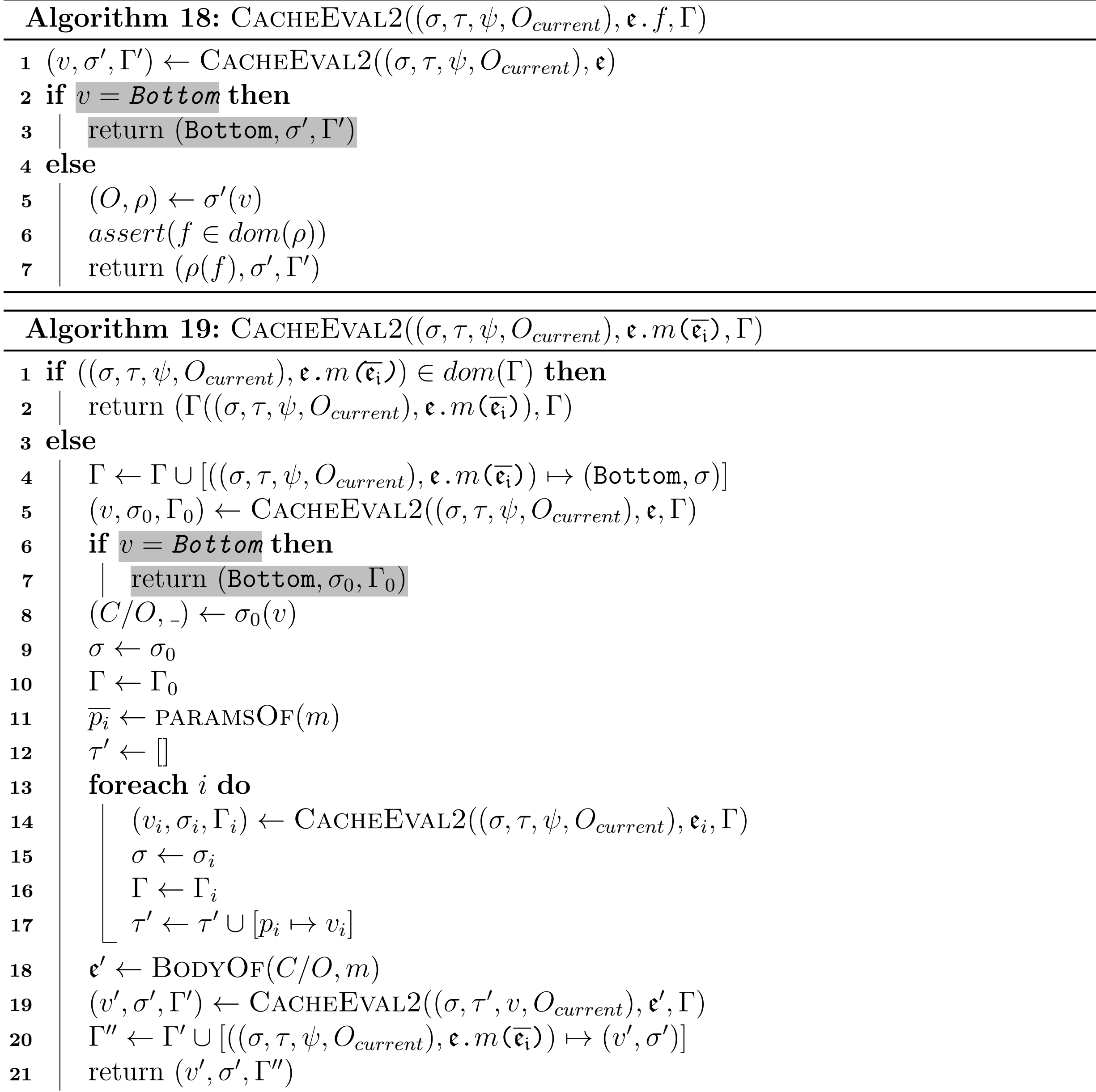
The addition of the cache and the `Bottom` value marks the first distinction between the concrete semantics and the abstract semantics designed in this chapter. The cache in the

$$\textsc{CacheEval2} :: (Input \times Cache) \rightarrow (Output \times Cache)$$

Algorithm 17: $\textsc{CacheEval2}((\sigma, \tau, \psi, O_{current}), \mathfrak{e}\,.\,m\,(\overline{\mathfrak{e}_i}), \Gamma)$

```
1  if ((σ, τ, ψ, O_current), e.m(ē_i)) ∈ dom(Γ) then
2  |   return (Γ((σ, τ, ψ, O_current), e.m(ē_i)), Γ)
3  else
4  |   Γ ← Γ ∪ [((σ, τ, ψ, O_current), e.m(ē_i)) ↦ Bottom]
5  |   (v, σ_0, Γ_0) ← CacheEval2((σ, τ, ψ, O_current), e, Γ)
6  |   (C/O, _) ← σ_0(v)
7  |   σ ← σ_0
8  |   Γ ← Γ_0
9  |   p̄_i ← paramsOf(m)
10 |   τ' ← []
11 |   foreach i do
12 |   |   (v_i, σ_i, Γ_i) ← CacheEval2((σ, τ, ψ, O_current), e_i, Γ)
13 |   |   σ ← σ_i
14 |   |   Γ ← Γ_i
15 |   |_  τ' ← τ' ∪ [p_i ↦ v_i]
16 |   e' ← BodyOf(C/O, m)
17 |   (v', σ', Γ') ← CacheEval2((σ, τ', v, O_current), e', Γ)
18 |   Γ'' ← Γ' ∪ [((σ, τ, ψ, O_current), e.m(ē_i)) ↦ (v', σ')]
19 |   return (v', σ', Γ'')
```

Figure 3.7: The recursive rule when evaluating recursion in the abstract semantics



Algorithm 18: $\textsc{CacheEval2}((\sigma, \tau, \psi, O_{current}), \mathfrak{e}\,.\,f, \Gamma)$

1 $(v, \sigma', \Gamma') \leftarrow \textsc{CacheEval2}((\sigma, \tau, \psi, O_{current}), \mathfrak{e})$
2 **if** $v = \texttt{Bottom}$ **then**
3   return $(\texttt{Bottom}, \sigma', \Gamma')$
4 **else**
5     $(O, \rho) \leftarrow \sigma'(v)$
6     $assert(f \in dom(\rho))$
7   return $(\rho(f), \sigma', \Gamma')$

Algorithm 19: $\textsc{CacheEval2}((\sigma, \tau, \psi, O_{current}), \mathfrak{e}\,.\,m(\overline{\mathfrak{e}_i}), \Gamma)$

1 **if** $((\sigma, \tau, \psi, O_{current}), \mathfrak{e}\,.\,m(\overline{\mathfrak{e}_i})) \in dom(\Gamma)$ **then**
2   return $(\Gamma((\sigma, \tau, \psi, O_{current}), \mathfrak{e}\,.\,m(\overline{\mathfrak{e}_i})), \Gamma)$
3 **else**
4     $\Gamma \leftarrow \Gamma \cup [((\sigma, \tau, \psi, O_{current}), \mathfrak{e}\,.\,m(\overline{\mathfrak{e}_i})) \mapsto (\texttt{Bottom}, \sigma)]$
5     $(v, \sigma_0, \Gamma_0) \leftarrow \textsc{CacheEval2}((\sigma, \tau, \psi, O_{current}), \mathfrak{e}, \Gamma)$
6   **if** $v = \texttt{Bottom}$ **then**
7   return $(\texttt{Bottom}, \sigma_0, \Gamma_0)$
8     $(C/O, _) \leftarrow \sigma_0(v)$
9     $\sigma \leftarrow \sigma_0$
10     $\Gamma \leftarrow \Gamma_0$
11     $\overline{p_i} \leftarrow \textsc{paramsOf}(m)$
12     $\tau' \leftarrow []$
13   **foreach** i **do**
14         $(v_i, \sigma_i, \Gamma_i) \leftarrow \textsc{CacheEval2}((\sigma, \tau, \psi, O_{current}), \mathfrak{e}_i, \Gamma)$
15         $\sigma \leftarrow \sigma_i$
16         $\Gamma \leftarrow \Gamma_i$
17         $\tau' \leftarrow \tau' \cup [p_i \mapsto v_i]$
18     $\mathfrak{e}' \leftarrow \textsc{BodyOf}(C/O, m)$
19     $(v', \sigma', \Gamma') \leftarrow \textsc{CacheEval2}((\sigma, \tau', v, O_{current}), \mathfrak{e}', \Gamma)$
20     $\Gamma'' \leftarrow \Gamma' \cup [((\sigma, \tau, \psi, O_{current}), \mathfrak{e}\,.\,m(\overline{\mathfrak{e}_i})) \mapsto (v', \sigma')]$
21   return (v', σ', Γ'')

Figure 3.8: The CacheEval function with `Bottom`

```scala
class C { def bar() = ... }
object O {
    def foo(): C = foo()
    val f = foo()
    val f2 = f.bar()
}
```

Figure 3.9: Dereferencing the result of infinite recursion in Scala

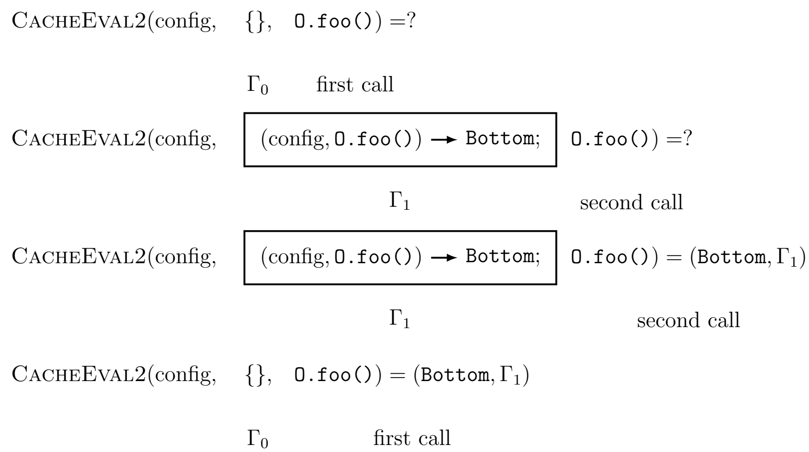


Figure 3.10: The trace of cached evaluation starting at Figure 3.2

abstract semantics aims to skip the evaluation of recursive method calls when the cache contains the result of evaluating the same call in the same configuration. The addition of `Bottom` should be scrutinized because the meaning of `Bottom` as a value in the domain can be confusing. To understand the role of `Bottom`, let us realize how CacheEval2 terminates on the program in Figure 3.1 with the addition of `Bottom`. Figure 3.2 shows the configuration to evaluate each recursive call on line 2, and Figure 3.10 shows the process of evaluating the recursive call to `foo` in the specific configuration, starting with the empty cache. Since `Bottom` is added to the cache before evaluating the first call, CacheEval2 will terminate without evaluating the second recursive call because the same input to CacheEval2 is associated with `Bottom` in the cache. The abstract interpreter knows that the second recursive call will lead to infinite other recursive calls in the same configuration, so it returns `Bottom` as a value that shows that the interpreter has given up to avoid non-termination. Therefore, `Bottom` should be considered as the result of every recursive call to `foo` starting at the second recursive call. In fact, the cached evaluation can be modified to put `Bottom` in the cache before evaluating the third call to `foo`, then the interpreter will evaluate the body of `foo` twice before giving up, and `Bottom` represents the value of each call to `foo` starting from the third recursive call. No matter when `Bottom` is added to the cache, the result of the first call evaluation will also be `Bottom`. This means that `Bottom` is a fixed point result of evaluating `foo`: Every recursive call in an infinite recursive call chain returns the value `Bottom`.

3.3 Finitizing the concrete domain

The domain presented in Figure 3.6 terminates infinitely recursive methods that do not initialize any instances because at one point, the recursive chain always finds the configuration as the key in the cache. However, in the example illustrated in Figure 3.11, each recursive call instantiates a fresh instance of `C`. Therefore, the heap in the configuration changes for each recursive call, and the abstract interpreter keeps adding new keys to the cache and never terminates. For another program in Figure 3.12, the abstract interpreter can terminate the infinite call chain of `foo`, but it cannot find the fixed point of the infinite call chain of `foo` in the current initialization domain. If we assume that the infinite recursion terminates at some recursive calls and converges to a fixed point value, the previous call always instantiates a fresh instance and returns a fresh address that is different from the fixed point. In both examples, the root cause of non-termination is that the heap allocates a fresh address for every fresh instance and can accommodate an arbitrary number of instances, so the total possible configurations is infinite. In general, if the set of possible

```
class C {}

object O {
    def foo(c) = foo(new C)
    val f = foo(new C)
}
```

Figure 3.11: A program in the initialization calculus that creates infinite number of instances

```
class C(x: C)

object O {
    def foo(): C = new C(foo())
    val f = foo()
}
```

Figure 3.12: Another program in the initialization calculus that creates infinite number of instances

configurations is infinite, then there always exist programs that do not have a fixed-point configuration as the termination point. In order to design an abstract interpreter that always terminates, it is essential to design an abstract domain with a finite set of possible configurations.

In order to finitize the set of configurations of an abstract interpreter, the first step is to finitize the set of all possible heaps. The heap definition in Figure 2.4 is an infinite set because the set of all possible addresses is infinite. The finitized heap in the abstract domain should start by defining a set of finite abstract addresses, $\widehat{Addr}$. However, if the set of addresses becomes finite, then multiple instances will share the same abstract address. Figure 3.13 illustrates the first attempt to define a finitized domain. The set of abstract addresses simply becomes a finite subset of natural numbers, and each abstract address maps to a set of multiple abstract instances. The remaining components of the abstract domain correspond to the concrete domain. The set of abstract configurations, $\widehat{Conf}$, is indeed finite because every component is a finite set (note that every set of class, object, field, and parameter names in the domain is finite with respect to the input program). We also define an abstract address allocation function, which was introduced by Van Horn and Might in their framework of abstracting abstract machines [27]. The allocation function

$$
\begin{array}{rrr}
 & \widehat{Addr} ::= & \{0, \cdots, 9\} \\
\widehat{v} \in & \widehat{Value} ::= & P(\widehat{Addr}) \\
\widehat{\sigma} \in & \widehat{Heap} ::= & \widehat{Addr} \to P(\widehat{Instance}) \\
 & \widehat{Instance} ::= & (ClassName \cup ObjectName) \times ((FieldName \cup ParamName) \to \widehat{Value}) \\
\widehat{\tau} \in & \widehat{Frame} ::= & ParamName \to \widehat{Value} \\
\widehat{\psi} \in & \widehat{Addr} & \\
 & \widehat{Conf} ::= & \widehat{Heap} \times \widehat{Frame} \times \widehat{Addr} \times ObjectName
\end{array}
$$

Figure 3.13: The first attempt to define an abstract finitized heap

is of the form $\widehat{alloc} : Instance \times \widehat{Conf} \to \widehat{Addr}$, which describes the abstract address to associate with a fresh instance allocated in the given abstract configuration. We can define the allocation function as follows, which chooses the abstract address in cyclic order regardless of the concrete instance:

$$\widehat{alloc}((_, _), \widehat{\sigma}, \widehat{\tau}, \widehat{\psi}) = |dom(\widehat{\sigma})| \mod 10$$

The design of the abstract addresses in Figure 3.13 and the allocation function is too coarse-grained. In the worst case, each abstract address maps to a set of instances with completely different class tags, fields, and parameter information. This makes the abstract interpreter less understandable and precise. A heuristic of designing the abstract addresses is to include the metrics that may decide the value associated with each address. In the context of the global object initialization checker, we need to define a set of metrics that is computable during the analysis and affects the information encoded in each abstract instance. The motivation of such a heuristic is that when the set of instances associated with each address shares the same set of metrics, then they are more likely to record the same information. This leads to fewer distinct abstract instances in the heap. An essential metric of an instance is its class tag, and instances that share the same class tag enclose the same list of fields or parameters. Therefore, we define the abstract address allocation function to be based on class tags:

$$\begin{cases} \widehat{alloc}((C, \rho), \widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}) = C \\ \widehat{alloc}((O, \rho), \widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}) = O \end{cases}$$

$$
\begin{array}{rrr}
\widehat{\psi} \in & \widehat{Addr} ::= & ClassName \cup ObjectName \\
\widehat{v} \in & \widehat{Value} ::= & P(\widehat{Addr}) \cup \{\texttt{Bottom}\} \\
\widehat{\sigma} \in & \widehat{Heap} ::= & \widehat{Addr} \rightarrow \widehat{Instance} \\
 & \widehat{Instance} ::= & (FieldName \cup ParamName) \rightarrow \widehat{Value} \\
\widehat{\tau} \in & \widehat{Frame} ::= & ParamName \rightarrow \widehat{Value} \\
(\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}) \in & \widehat{Conf} ::= & \widehat{Heap} \times \widehat{Frame} \times \widehat{Addr} \times ObjectName \\
\widehat{\Gamma} \in & \widehat{Cache} ::= & (\widehat{Conf} \times Expression) \rightarrow (\widehat{Value} \times \widehat{Heap})
\end{array}
$$

Figure 3.14: The abstract domain of the global object initialization checker

Under this allocation strategy, each abstract instance maps a parameter or a field to a set of abstract addresses, combining the values of this field or parameter among all existing instances that share the same class tag. This abstract domain is adopted by the global object initialization checker developed in this thesis, and readers can verify that the set of abstract configurations is still finite.

The program in Figure 3.15 is an example that illustrates the allocation strategy based on the current abstract domain. The program contains five concrete instances. The instance of `O`, the instance of `A1` allocated on line 5 and the instance of `A2` allocated on line 6 are associated with different addresses because they have distinct class tags. The two instances of `C` allocated on line 5 and line 6 will share the same abstract address, because they have the same class tag and are both owned by `O`. Therefore, the value of `x` in the abstract instance of `C` owned by `O` should contain references to two possible arguments. The final abstract heap after initializing `O` is shown in Figure 3.16.

3.4 The Abstract Initialization Semantics

This section presents the formal abstract initialization semantics based on the abstract domain in Figure 3.14. The abstract semantics still consists of an evaluation function with a cache for expressions. The signature of the $\widehat{\text{CacheEval}}$ function is presented in Figure 3.17. The function operates on the abstract components in the abstract configuration. The evaluation function also returns a set of abstract addresses since each value in $\widehat{Value}$

```
class A1 {}
class A2 extends A1 {}
class C(x) {}
object O {
    val c1 = new C(new A1)
    val c2 = new C(new A2)
}
```

Figure 3.15: A program with two instances with different bodies represented by the same abstract instance

$$\widehat{\sigma} = \begin{cases} addrOf(\texttt{O}) \mapsto \begin{cases} \texttt{c1} \mapsto \{addrOf(\texttt{C}, \text{owner} = \texttt{O})\} \\ \texttt{c2} \mapsto \{addrOf(\texttt{C}, \text{owner} = \texttt{O})\} \end{cases} \\ addrOf(\texttt{C}, \text{owner} = \texttt{O}) \mapsto \Big\{ \texttt{x} \mapsto \{addrOf(\texttt{A1}, \text{owner} = \texttt{O}), addrOf(\texttt{A2}, \text{owner} = \texttt{O})\} \\ addrOf(\texttt{A1}, \text{owner} = \texttt{O}) \mapsto \{\} \\ addrOf(\texttt{A2}, \text{owner} = \texttt{O}) \mapsto \{\} \end{cases}$$

Figure 3.16: The abstract heap after initializing O in Figure 3.15

$$\widehat{\text{CACHEEVAL}} :: \widehat{Conf} \times Expression \times \widehat{Cache} \to \widehat{Value} \times \widehat{Heap} \times \widehat{Cache}$$

Algorithm 20: $\widehat{\text{CACHEEVAL}}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), p, \widehat{\Gamma})$

1 **if** $p \in dom(\widehat{\tau})$ **then**
2 $\quad \widehat{\tau}(p)$
3 **else**
4 $\quad \rho \leftarrow \widehat{\sigma}(\widehat{\psi})$
5 $\quad assert(p \in dom(\rho))$
6 $\quad$ return $(\rho(p), \widehat{\sigma}, \widehat{\Gamma})$

Figure 3.17: The abstract evaluation of a parameter accesses

represents a set of abstract addresses.

Parameter access, global object access, and field selection. In Figure 3.17, the evaluation of parameter accesses is similar to the concrete semantics, except that $\widehat{\sigma}(\widehat{\psi})$ now directly returns the map to the parameter values. Figure 3.18 then illustrates the procedure for abstractly evaluating a direct reference to object O during the initialization of $O_{current}$. The abstract address of the instance of O is represented by the name O itself, and the evaluation always returns a singleton set containing this address. The function $\widehat{\text{INITIALIZEGLOBALOBJECT}}$ will be defined later. Figure 3.19 then defines the abstract evaluation of field selections. The receiver evaluation now may return a set of abstract addresses due to the finitization, or `Bottom`. However, if the receiver is not `Bottom`, the set of addresses must contain only the defining object of f, since the field names are unique. Moreover, the field must be initialized.

Method invocation. Next, we define the abstract evaluation of method invocations, which involves finding the fixed point of recursive calls. The purpose of finitizing the abstract domain is to ensure the existence of fixed points when evaluating all method invocations, and Figure 3.12 is an example whose fixed point does not exist in the infinite domain. With the current finite abstract domain, if `Bottom` is the fixed point result starting from the second recursive call to `foo` in Figure 3.12, then the first recursive call returns the abstract address of the instances of `C` owned by object `O`, because the first recursive call instantiates a fresh instance. Therefore `Bottom` is not the fixed point. However, if the

Algorithm 21: $\widehat{\text{CacheEval}}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), O, \widehat{\Gamma})$

1 **if** $O \in dom(\widehat{\sigma})$ **then**
2   **if** $O \neq O_{current}$ **then**
3         $\rho \leftarrow \widehat{\sigma}(O)$
4         $assert(|dom(\rho)| = |\text{FieldsOf}(O)|)$
5   return $(\{O\}, \widehat{\sigma}, \widehat{\Gamma})$
6 **else**
7     $\widehat{\psi}' \leftarrow O$
8     $\sigma' \leftarrow \sigma \cup [\widehat{\psi}' \mapsto []]$
9     $\mathbb{O} \leftarrow \text{TemplateOf}(O)$
10     $(\widehat{\sigma}'', \widehat{\Gamma}') \leftarrow \widehat{\text{initializeObject}}(\mathbb{O}, \widehat{\sigma}', \widehat{\psi}', \widehat{\Gamma})$
11   return $(\{O\}, \widehat{\sigma}'', \widehat{\Gamma}')$

Figure 3.18: The abstract evaluation of direct accesses of global objects

Algorithm 22: $\widehat{\text{CacheEval}}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), \mathfrak{e}\,.\,f, \widehat{\Gamma})$

1 $(\widehat{v}, \widehat{\sigma}', \widehat{\Gamma}') \leftarrow \widehat{\text{CacheEval}}(\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}, \mathfrak{e}, \widehat{\Gamma})$
2 **if** $\widehat{v} = \mathit{Bottom}$ **then**
3   return $(\texttt{Bottom}, \widehat{\sigma}', \widehat{\Gamma}')$
4 $assert(\widehat{v} = \{O\} \wedge f \in \text{FieldsOf}(O) \wedge f \in dom(\widehat{\sigma}'(O)))$
5 $\widehat{v}' \leftarrow \widehat{\sigma}'(O)(f)$
6 return $(\widehat{v}', \widehat{\sigma}', \widehat{\Gamma}')$

Figure 3.19: The abstract evaluation of field selections

abstract address $(\texttt{C}, \texttt{O})$ is the result of the second recursive call, then the instance allocated in the first recursive call shares the same abstract address. This means that $(\texttt{C}, \texttt{O})$ is the fixed point in evaluating the recursion in Figure 3.9, and finding the fixed point requires two iterations of evaluating the body of the recursive call.

Figure 3.20 presents the overall procedure of cached evaluation of method calls. The process in each **do while** loop corresponds to evaluating the method body once. The receiver is a set of possible addresses, so the second **foreach** loop unions the result of evaluating the proper method body for each possible receiver value. The key feature of the **do while** loop is recording the result of two iterations of evaluating the method body via $(\widehat{v}_{last}, \widehat{\sigma}_{last})$ and $(\widehat{v}_{current}, \widehat{\sigma}_{current})$. Before the first iteration of the **do while** loop, $\widehat{v}_{last}$ is set to `Bottom`. Each iteration of the **do while** loop evaluates one recursive call and records the output in $(\widehat{v}_{current}, \widehat{\sigma}_{current})$. If $(\widehat{v}_{current}, \widehat{\sigma}_{current}) \neq (\widehat{v}_{last}, \widehat{\sigma}_{last})$, then the **do while** loop evaluates another recursive call based on $(\widehat{v}_{current}, \widehat{\sigma}_{current})$ returned in the previous loop (by setting $(\widehat{v}_{last}, \widehat{\sigma}_{last}) = (\widehat{v}_{current}, \widehat{\sigma}_{current})$). The **do while** loop terminates when the output of the current iteration matches the output of the last iteration, which means the cache converges to a fixed point.

Class instantiation. When defining the abstract evaluation of `new` expressions, note that `new` expressions may also result in infinite recursions in the initialization calculus, as illustrated in Figure 3.21. Each `new` expression that instantiates `D` attempts to instantiate another instance of `D` to initialize the parameter `c` in the instance of `D`. Therefore, Figure 3.22 also calculates the fixed point when evaluating `new` expressions, which uses the cache similar to the process of evaluating method calls. The abstract address of the fresh instance is represented by the class tag. Recall that this abstract address is shared by all existing concrete instances of the class tag. Therefore, when initializing the parameter of the fresh instance, the value of the arguments must be combined with the values of the arguments of other existing instances that share the same abstract address, which is illustrated on line 19.

Initialization and program execution. Finally, the abstract functions describing the initialization processes of objects, classes, and templates are similar to their concrete counterparts. Figure 3.23 describes the abstract initialization functions. Note that $\textsc{Initialize}\widehat{\textsc{Class}}\textsc{Template}$ may combine the values of the arguments in different concrete instances that share the same abstract address as the new instance. $\textsc{Initi}\widehat{\textsc{alize}}\textsc{Field}$ does not need to combine the values of different fields because addresses of global objects only accommodate one instance. The function of abstractly executing programs in the

Algorithm 23: $(\widehat{CacheEval}(\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), \mathfrak{e}\texttt{.}m\texttt{(}\overline{\mathfrak{e}_i}\texttt{)}, \widehat{\Gamma})$

1 **if** $(\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}, \mathfrak{e}\texttt{.}m\texttt{(}\overline{\mathfrak{e}_i}\texttt{)}) \in dom(\widehat{\Gamma})$ **then**
2 $\quad$ return $(\widehat{\Gamma}(\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}, \mathfrak{e}\texttt{.}m\texttt{(}\overline{\mathfrak{e}_i}\texttt{)}), \widehat{\Gamma})$
3 $(\widehat{v}_{last}, \widehat{\sigma}_{last}) \leftarrow (\texttt{Bottom}, \widehat{\sigma})$
4 $(\widehat{v}_{current}, \widehat{\sigma}_{current}) \leftarrow (\widehat{v}_{last}, \widehat{\sigma}_{last})$
5 **do**
6 $\quad (\widehat{v}_{last}, \widehat{\sigma}_{last}) \leftarrow (\widehat{v}_{current}, \widehat{\sigma}_{current})$
7 $\quad \widehat{\Gamma} \leftarrow \widehat{\Gamma} \cup [(\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}, \mathfrak{e}\texttt{.}m\texttt{(}\overline{\mathfrak{e}_i}\texttt{)}) \mapsto (\widehat{v}_{last}, \widehat{\sigma}_{last})]$
8 $\quad (\widehat{v}, \widehat{\sigma}', \widehat{\Gamma}') \leftarrow \widehat{CacheEval}(\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}, \mathfrak{e}, \widehat{\Gamma})$
9 $\quad$ **if** $\widehat{v} =$ *`Bottom`* **then**
10 $\quad\quad$ return $(\texttt{Bottom}, \widehat{\sigma}', \widehat{\Gamma}')$
11 $\quad \overline{p_i} \leftarrow$ PARAMSOF(M)
12 $\quad \widehat{\tau}' \leftarrow []$
13 $\quad \widehat{\sigma} \leftarrow \widehat{\sigma}'$
14 $\quad \widehat{\Gamma} \leftarrow \widehat{\Gamma}'$
15 $\quad$ **foreach** i **do**
16 $\quad\quad (\widehat{v}_i, \widehat{\sigma}_i, \widehat{\Gamma}_i) \leftarrow \widehat{CacheEval}(\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}, \mathfrak{e}_i)$
17 $\quad\quad \widehat{\tau}' \leftarrow \widehat{\tau}' \cup [p_i \mapsto v_i]$
18 $\quad\quad \widehat{\sigma} \leftarrow \widehat{\sigma}_i$
19 $\quad\quad \widehat{\Gamma} \leftarrow \widehat{\Gamma}_i$
20 $\quad \widehat{v}_{current} \leftarrow \{\}$
21 $\quad$ **foreach** $\nu \in \widehat{v}$ **do**
22 $\quad\quad$ **if** $\nu = (C, _)$ **then**
23 $\quad\quad\quad \mathfrak{e}' \leftarrow$ BODYOF(C, m)
24 $\quad\quad$ **else**
25 $\quad\quad\quad \nu = O$
26 $\quad\quad\quad \mathfrak{e}' \leftarrow$ BODYOF(O, m)
27 $\quad\quad (\widehat{v}', \widehat{\sigma}', \widehat{\Gamma}') \leftarrow \widehat{CacheEval}(\widehat{\sigma}, \widehat{\tau}', \nu, O_{current}, \mathfrak{e}', \widehat{\Gamma})$
28 $\quad\quad \widehat{v}_{current} \leftarrow \widehat{v}_{current} \cup \widehat{v}'$
29 $\quad\quad \widehat{\sigma} \leftarrow \widehat{\sigma}'$
30 $\quad\quad \widehat{\Gamma} \leftarrow \widehat{\Gamma}'$
31 $\quad \widehat{\sigma}_{current} \leftarrow \widehat{\sigma}$
32 **while** $(\widehat{v}_{current}, \widehat{\sigma}_{current}) \neq (\widehat{v}_{last}, \widehat{\sigma}_{last})$;
33 return $(\widehat{v}_{current}, \widehat{\sigma}_{current}, \widehat{\Gamma})$

Figure 3.20: The abstract evaluation of method invocation

```
class C(c) {}
class D extends C(new D) {}
object O {
    val f = new D
}
```

Figure 3.21: Infinite recursions of `new` expressions

Algorithm 24: $\widehat{\text{CacheEval}}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), \texttt{new } C(\overline{\mathfrak{e}_i}), \widehat{\Gamma})$

1 **if** $(\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}, \texttt{new } C(\overline{\mathfrak{e}_i})) \in dom(\widehat{\Gamma})$ **then**
2   return $(\widehat{\Gamma}(\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}, \texttt{new } C(\overline{\mathfrak{e}_i})), \widehat{\Gamma})$
3 $(\widehat{v}_{last}, \widehat{\sigma}_{last}) \leftarrow (\texttt{Bottom}, \widehat{\sigma})$
4 $(\widehat{v}_{current}, \widehat{\sigma}_{current}) \leftarrow (\widehat{v}_{last}, \widehat{\sigma}_{last})$
5 **do**
6     $(\widehat{v}_{last}, \widehat{\sigma}_{last}) \leftarrow (\widehat{v}_{current}, \widehat{\sigma}_{current})$
7     $\widehat{\Gamma} \leftarrow \widehat{\Gamma} \cup [(\widehat{\Gamma}(\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}, \texttt{new } C(\overline{\mathfrak{e}_i})) \mapsto (\widehat{v}_{last}, \widehat{\sigma}_{last})]$
8     $\overline{p_i} \leftarrow \text{paramsOf}(C)$
9     $\rho \leftarrow []$
10   **foreach** i **do**
11         $(\widehat{v}_i, \widehat{\sigma}_i, \widehat{\Gamma}_i) \leftarrow \widehat{CacheEval}(\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}, \mathfrak{e}_i)$
12         $\rho \leftarrow \rho \cup [p_i \leftarrow v_i]$
13         $\widehat{\sigma} \leftarrow \widehat{\sigma}_i$
14         $\widehat{\Gamma} \leftarrow \widehat{\Gamma}_i$
15     $\widehat{\psi}' \leftarrow C$
16   **if** $\widehat{\psi}' \notin dom(\widehat{\sigma})$ **then**
17         $\widehat{\sigma} \leftarrow \widehat{\sigma}[\widehat{\psi}' \mapsto \rho]$
18   **else**
19         $\widehat{\sigma} \leftarrow \widehat{\sigma}[\widehat{\psi}' \mapsto \rho \cup \widehat{\sigma}(\widehat{\psi}')]$
20     $(\widehat{\sigma}', \widehat{\Gamma}') \leftarrow \widehat{\text{initializeClassTemplate}}(C, \widehat{\sigma}, \widehat{\psi}', \widehat{\Gamma})$
21     $\widehat{v}_{current} \leftarrow \{\widehat{\psi}'\};\ \widehat{\sigma}_{current} \leftarrow \widehat{\sigma}';\ \widehat{\Gamma} \leftarrow \widehat{\Gamma}'$
22 **while** $(\widehat{v}_{current}, \widehat{\sigma}_{current}) \neq (\widehat{v}_{last}, \widehat{\sigma}_{last})$;
23 return $(\widehat{v}_{current}, \widehat{\sigma}_{current}, \widehat{\Gamma})$

Figure 3.22: The abstract evaluation of `new` expressions

initialization calculus is presented in Figure 3.24. It treats $\mathfrak{e}_{main}$ as if it is being evaluated in a main object named `Main`. This completes the formal abstract initialization semantics.

3.5 Termination and soundness

This chapter identifies two non-terminating patterns in the initialization calculus. The first is infinite recursion in method calls, and the second is infinite recursion in `new` expressions. The abstract interpreter uses the fixed point of the cache to approximate the result of the infinite recursion, so this section needs to prove that the abstract interpreter always finds the fixed point of the cache. Specifically, we need to prove that the **do while** loops in Algorithm 26 and Algorithm 27 always terminate.

The reason for the termination is based on the fact that the set of abstract configurations, $\widehat{Conf}$, is finite. Moreover, all the finite components in an abstract configuration form a lattice, whose properties are frequently applied in the static analysis framework [9].

Definition 3.1: A lattice consists of a set X and a partial order $\sqsubseteq$ on X, such that: for every $S \subseteq X$, the least upper bound $\bigsqcup S$ and the greatest lower bound $\bigsqcap S$ exist.

For $S \subseteq X, y \in X$, we use the notation $S \sqsubseteq y$ to indicate that $\forall x \in S, x \sqsubseteq y$, and $y \sqsubseteq S$ indicates that $\forall x \in S, y \sqsubseteq x$. The least upper bound $\bigsqcup S$ is defined as an element in X such that $S \sqsubseteq \bigsqcup S$, and $\forall y \in X, S \sqsubseteq y \Rightarrow \bigsqcup S \sqsubseteq y$. Similarly, the greatest lower bound $\bigsqcap S$ is defined as an element in X such that $\bigsqcap S \sqsubseteq S$, and $\forall y \in X, y \sqsubseteq S \Rightarrow y \sqsubseteq \bigsqcap S$.

The existence of the least upper bound and the greatest lower bound of every subset of X allows us to draw a lattice structure when X is finite. Each node in the lattice structure represents a least upper bound $\bigsqcup S$ of some $S \subseteq X$, and every element $x \in X$ appears as a node since $\bigsqcup\{x\} = x$. Finally, for two elements x, x', find the largest subset S, S' of X such that $x = \bigsqcup S, x' = \bigsqcup S'$. If $S' \subset S$, an edge is drawn from x' to x, and x is placed higher than x'. The resulting lattice structure must include an element on top and an element at bottom, which are $\bigsqcup X$ and $\bigsqcup \emptyset$, respectively. One can easily verify the following observations:

Observation 3.2:

- $\bigsqcup \emptyset = \bigsqcap X$
- $\forall x \in X, x \sqsubseteq \bigsqcup X$ and $\bigsqcap X \sqsubseteq x$

Algorithm 25: $\text{INITIALIZE}\widehat{\text{OBJECT}}(\mathbb{O}, \widehat{\sigma}, \widehat{\psi}, \widehat{\Gamma})$

```
1  object O extends ℙℂ(𝔢ᵢ) { 𝔽ⱼ 𝕄 } ← 𝕆
2  pᵢ ← paramsOf(ℙℂ)
3  foreach i do
4  |  (v̂ᵢ, σ̂ᵢ) ← EVAL^(σ̂, {}, ψ̂, O, 𝔢̂ᵢ, Γ̂)
5  |  ρ ← σᵢ(ψ)
6  |  σ ← σᵢ[ψ ↦ ρ[pᵢ ↦ vᵢ]]
7  |  Γ̂ ← Γ̂ᵢ
8  (σ̂, Γ̂) ← initializeClassTemplate^(σ̂, ψ̂, O, ℙℂ, Γ̂)
9  foreach j do
10 |  (σ̂, Γ̂) ← initializeField^(σ̂, ψ̂, O, 𝔽ⱼ), Γ̂
11 return (σ̂, Γ̂)
```

Algorithm 26: $\text{INITIALIZE}\widehat{\text{CLASS}}\text{TEMPLATE}(C, \widehat{\sigma}, \widehat{\psi}, O_{current}, \widehat{\Gamma})$

```
1  if C = Object then
2  |  return (σ̂, Γ̂)
3  class C(pᵢ) extends ℙℂ(𝔢ⱼ) { 𝕄 } ← TemplateOf(C)
4  pⱼ ← paramsOf(ℙℂ)
5  foreach j do
6  |  (v̂ⱼ, σ̂ⱼ, Γ̂ⱼ) ← EVAL^(σ̂, {}, ψ̂, O_current, 𝔢ⱼ, Γ̂)
7  |  ρ ← σ̂ⱼ(ψ̂)
8  |  if pⱼ ∉ dom(ρ) then
9  |  |  ρ ← ρ ∪ [pⱼ ↦ vⱼ]
10 |  else
11 |  |  ρ ← ρ[pⱼ ↦ vⱼ ∪ ρ(pⱼ)]
12 |  Γ̂ ← Γ̂ⱼ
13 return initializeClassTemplate^(ℙℂ, σ̂, ψ̂, O_current, Γ̂)
```

Algorithm 27: $\text{INITIA}\widehat{\text{LIZE}}\text{FIELD}(\mathbb{F}, \widehat{\sigma}, \widehat{\psi}, O_{current}, \widehat{\Gamma})$

```
1  val f = 𝔢 ← 𝔽
2  (v̂, σ̂′, Γ̂′) ← Eval(σ̂, {}, ψ̂, O_current, 𝔢, Γ̂)
3  ρ ← σ̂′(ψ̂)
4  return (σ̂′[ψ̂ ↦ ρ ∪ [f ↦ v̂]], Γ̂)
```

Figure 3.23: The abstract initialization functions

$\widehat{\text{EXECUTE}} :: List[ClassTemplate] \times List[ObjectTemplate] \times MainObject \to \widehat{Heap} \times \widehat{Cache}$

Algorithm 28: $\widehat{\text{EXECUTE}}(\overline{\mathbb{C}_i}, \overline{\mathbb{O}_j}, \mathfrak{e}_{main}$

---

1 return $\text{CACHE}\widehat{\text{EVAL}}(([], [], \texttt{Main}, \texttt{Main}), \mathfrak{e}_{main}, [])$

Figure 3.24: The abstract function of executing programs in the initialization calculus

The main work done in Section 3.3 is to finitize the abstract domain. Specifically, $\widehat{Conf}$ in Figure 3.14 is finite, but $Conf$ in Figure 2.6 is not. The result of evaluating an expression is represented by an element in $Value$ in the concrete calculus, but in the abstract calculus it is represented by an element in $\widehat{Value}$. In fact, all elements in $\widehat{Value}$ form a lattice structure according to the following observation:

Observation 3.3: for every finite set X, $(P(X), \subseteq)$ forms a lattice. Specifically, for every $S = \{s_1, \cdots, s_n\} \subseteq P(X)$, $\bigsqcup S = \bigcup_{i=1}^{n} s_i$, and $\bigsqcap S = \bigcap_{i=1}^{n} s_i$.

In fact, if $X = \{x_i : 1 \leq i \leq 3\}$, the lattice constructed by the elements in $P(X)$ can be drawn as the following lattice in Figure 3.25. An edge between nodes on two levels shows the partial order between them. Also, if $x \sqsubset x'$, then x' is placed higher than x in the lattice.

Recall that $\widehat{Value} := P(\widehat{Addr}) \cup \{Bottom\}$. Since $\widehat{Addr}$ is finite, observation 3.4 shows that $\widehat{Value}$ forms a lattice. In order to integrate `Bottom` into the lattice, we can define `Bottom` $\sqsubseteq s$ for all $s \in P(\widehat{Addr})$. This partial order suggests the fact that any operations on `Bottom` never trigger exceptions in the abstract semantics, while operations on other elements may trigger exceptions. After identifying the lattice of abstract values, one can in turn prove that all maps from a specific finite domain to $\widehat{Value}$ also form a lattice.

Observation 3.4: Let X be a finite set, and let $(Y, \sqsubseteq)$ be a lattice, one can define the set of complete maps from X to Y, Y^X, and a partial order on Y^X : $\{f_1 \sqsubseteq' f_2 : \forall x \in X, f_1(x) \sqsubseteq f_2(x)\}$. Then $(Y^X, \sqsubseteq')$ forms a lattice. Specifically, let $S = \{f_i : 1 \leq i \leq n\}$, then $\bigsqcup S = [x \mapsto \bigsqcup_{i=1}^{n} f_i(x)]$, and $\bigsqcap S = [x \mapsto \bigsqcap_{i=1}^{n} f_i(x)]$.

With observation 3.4, one can verify that all components of an abstract configuration are elements of separate lattices. For example, the set of abstract heaps forms a lattice because each heap is a map from field names to abstract values, and field names are finite in

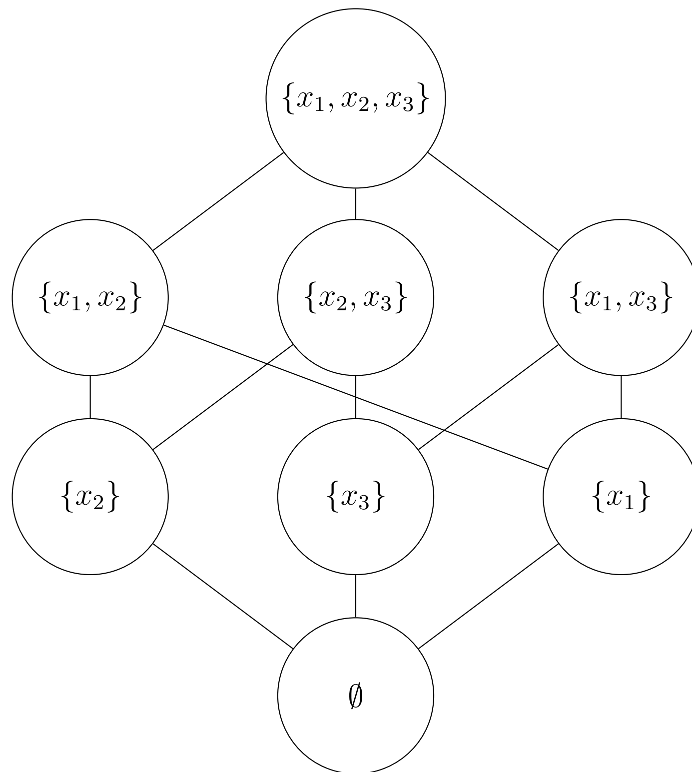


Figure 3.25: The lattice of subsets of finite sets

the given program. The same holds for abstract frames. Finally, the following observation shows that if every component of the abstract configuration comes from separate lattices, then the set of abstract configurations also forms a lattice.

Observation 3.5: Let $\{(X_i, \sqsubseteq_i) : 1 \leq i \leq n\}$ be a finite collection of lattices, then one can define the cartesian products of these components, $X_1 \times \cdots \times X_n$, and a partial order on it, $\{(x_1, \cdots, x_n) \sqsubseteq' (x'_1, \cdots, x'_n) : \forall 1 \leq i \leq n, x_i \sqsubseteq_i x'_i\}$. Then the set of cartesian products and the partial order forms a lattice. Specifically, let $S = \{(x_{j1}, \cdots, x_{jn}) : 1 \leq j \leq m\}$, then $\bigsqcup S = (\bigsqcup_{j=1}^{m} x_{j1}, \cdots, \bigsqcup_{j=1}^{m} x_{jn})$, and $\bigsqcap S = (\bigsqcap_{j=1}^{m} x_{j1}, \cdots, \bigsqcap_{j=1}^{m} x_{jn})$

With observation 3.5, one can verify that the set of abstract configurations forms a lattice, and the set of abstract caches also forms a lattice. Finally, we can prove the following observation:

Observation 3.6: For each iteration of **do while** loop in $\widehat{\text{CACHEDEVAL}}$ in Algorithm 26 and Algorithm 27, $\widehat{v}_{last} \sqsubseteq \widehat{v}_{current}$ and $\widehat{\sigma}_{last} \sqsubseteq \widehat{\sigma}_{current}$.

Proof: Each iteration of **do while** loop starts with $\widehat{\sigma}_{last}$ as the input heap. Each iteration of the **do while** loop involves separate evaluation of receivers, arguments, and method bodies. Note that during every call of $\widehat{CacheEval}$, the heap can combine the information of new instances and can include global objects whose initialization is triggered, but the existing information in the input heap is never lost. Therefore, we must have $\widehat{\sigma}_{last} \sqsubseteq \widehat{\sigma}_{current}$. Similarly, the values of the receivers, arguments, and method bodies are evaluated based on a less precise heap, so their results also become less precise, so $\widehat{v}_{last} \sqsubseteq \widehat{v}_{current}$.

With observation 3.6, we can prove that Algorithms 26 and 27 always terminate.

Theorem 3.7: Algorithm 26 in Figure 3.20 and Algorithm 27 in Figure 3.22 will always terminate after a finite number of iterations.

Proof: Notice that before the first iteration of the **do while** loop, both algorithms set $\widehat{v}_{last}$ to `Bottom`, which is the bottom element of the lattice. After each iteration, $\widehat{v}_{last} \sqsubseteq \widehat{v}_{current}$ by observation 3.7. Since the lattice of $\widehat{Value}$ has a finite height, it is impossible for the **do while** loop to run infinitely with $\widehat{v}_{last} \neq \widehat{v}_{current}$ in every iteration.

Since Algorithms 26 and 27 always terminate, we can verify that all functions that form the abstract interpreter always terminate.

Apart from termination, abstractly executing a program using the abstract interpreter should always over-approximate the program behaviour with respect to using the concrete interpreter. Each step of the abstract interpreter or the concrete interpreter evaluates an expression in an abstract or concrete configuration, respectively. The prerequisite of a sound abstract interpreter is that the abstract configuration contains the information in the concrete information, albeit with imprecision. This is characterized by the following definition:

Definition 3.8 (Abstraction function based on concrete semantics in Section 2.4):

- First, we define an abstraction function from concrete to abstract values, given the concrete heap:

$$\alpha_v(\psi, \sigma) = \{\widehat{\psi}\}$$

  Here $\widehat{\psi}$ is the corresponding abstract address of the concrete instance at address ψ based on the class tag.

- Next, we define an abstraction function from a concrete map of parameter and field value to an abstract map of parameter and field value, given the concrete heap:

$$\begin{aligned} \alpha_\rho(\rho, \sigma) &= \widehat{\rho} \\ \textit{where } \widehat{\rho}(p) &= \alpha_v(\rho(p), \sigma) \\ \text{and } \widehat{\rho}(f) &= \alpha_v(\rho(f), \sigma) \end{aligned} \qquad (\forall p, f \in dom(\rho))$$

- Then, we define an abstraction function from concrete to abstract heap $\alpha_\sigma : Heap \rightarrow \widehat{Heap}$, where

$$\begin{aligned} \alpha_\sigma(\sigma) &= \widehat{\sigma} \\ \textit{where } \widehat{\sigma}(\widehat{\psi}) &= \alpha_\rho(\sigma(\psi)._2, \sigma) \end{aligned} \qquad (\forall \psi \in dom(\sigma))$$

- Next, we define an abstraction function from concrete to abstract frames, given the concrete heap

$$\alpha_\tau(\tau, \sigma) = \widehat{\tau}$$
$$where\ \widehat{\tau}(p) = \alpha_v(\tau(p), \sigma) \qquad (\forall p \in dom(\tau))$$

- Finally, we can define an abstraction function from concrete to abstract configurations $\alpha : Conf \to \widehat{Conf}$, where

$$\alpha(\sigma, \tau, \psi, O_{current}) = (\alpha_\sigma(\sigma), \alpha_\tau(\tau, \sigma), \alpha_\psi(\psi, \sigma), O_{current})$$

All the abstraction functions return the most precise abstract component corresponding to the input concrete counterpart. Considering the imprecision, we say that an abstract configuration over-approximates a concrete configuration, i.e. $(\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}) \xrightarrow{approx} (\sigma, \tau, \psi, O_{current})$, if $\alpha(\sigma, \tau, \psi, O_{current}) \sqsubseteq (\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current})$. Then we can prove that an evaluation step in the abstract interpreter still over-approximates the concrete evaluation result.

Lemma 3.9: Suppose $(\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}) \xrightarrow{approx} (\sigma, \tau, \psi, O_{current})$, and $(\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}, \mathfrak{e}) \notin dom(\Gamma)$, then $\widehat{CacheEval}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), \mathfrak{e}) \xrightarrow{approx}$ EVAL'$((\sigma, \tau, \psi, O_{current}), \mathfrak{e})$, given that EVAL'$((\sigma, \tau, \psi, O_{current}), \mathfrak{e})$ terminates.

The previous lemma is not sufficient to prove the soundness of the abstract interpreter. The main theorem considers the sequence of stack traces of EVAL' during the concrete program execution, and the sequence of stack traces of $\widehat{\text{CACHEEVAL}}$ during the abstract program execution. Note that the concrete sequence of stack traces can be infinite, while the abstract sequence must be finite. The main theorem then shows that each abstract stack trace in the abstract sequence over-approximates a sub-sequence of the concrete stack traces in order.

Theorem 3.10: For every input program, let $\{\delta_i : i \in \mathbb{N}, \delta \in (Conf \times Expression)*\}$ be its concrete sequence of stack traces of EVAL' (possibly infinite), and let $\{\widehat{\delta}_j : 1 \leq j \leq n, \delta \in (\widehat{Conf} \times Expression)*\}$ be its abstract sequence of stack traces of $\widehat{\text{CACHEEVAL}}$. Then there exist breakpoints $i_1 < i_2 < i_{n-1}$, where for all δ_i between i_j and i_{j+1}, $\widehat{\delta}_j \xrightarrow{approx} \delta_i$. The $\xrightarrow{approx}$ relation on stack traces is defined component-wise.

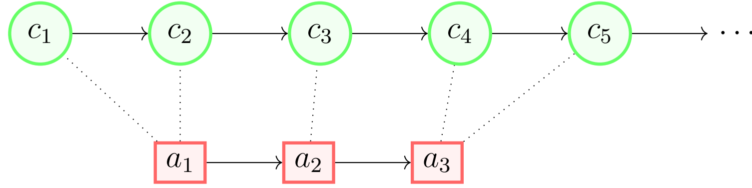


Figure 3.26: Illustration of concrete trace and abstract trace

We omit the full proof of Lemma 3.9 and Theorem 3.10, which can be proved by structural induction on $\mathfrak{e}$. Theorem 3.10 can be illustrated by figure 3.26. The circled nodes represent concrete stack traces, and the squared nodes represent abstract stack traces. The dotted lines between an abstract trace and concrete trace illustrate the over-approximation relation between them. Figure 3.26 hints that the process from c_1 to c_2 finishes the evaluation of an expression, but that evaluation can be finished in one call to $\widehat{CacheEval}$. Similarly, the processes from c_4 onward represent the evaluation of another expression, which may result in infinite recursion, but the abstract interpreter terminates the process in one call to $\widehat{\textsc{CacheEval}}$.

3.6 Producing warnings

The abstract interpreter presented in this chapter rejects programs with unsafe global object initialization. However, as a global object initialization checker, it should produce warnings for these programs to help programmers debug. We can modify the abstract interpreter to produce warnings at all program points that violate any run-time guarantee of safe initialization. Specifically, Figure 3.27 makes the checker produce warnings when it detects reads of uninitialized fields, and Figure 3.28 illustrates the warnings when the program contains cyclic initialization order of global objects. Note that when reading an uninitialized field, the checker returns `Bottom` to keep interpreting. The abstract interpreter also emits its stack trace when producing the warnings, which aids programmers to verify the warning and avoid it.

3.7 Summary

The abstract interpreter defined in this chapter forms the basis of the global object initialization checker. It is a valid over-approximation of the concrete interpreter, and it

Algorithm 29: $\widehat{\text{CacheEval}}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), \mathfrak{e}\,.\,f, \widehat{\Gamma})$

```
1  (v̂, σ̂′, Γ̂′) ← CacheEval^(σ̂, τ̂, ψ̂, O_current, 𝔢, Γ̂)
2  if v̂ = Bottom then
3  |   return (Bottom, σ̂′, Γ̂′)
4  assert(v̂ = {O} ∧ f ∈ FieldsOf(O))
5  if f ∉ dom(σ̂′(O)) then
6  |   report_warning("Reading uninitialized field!", stackTrace)
7  |   return (Bottom, σ̂′, Γ̂′)
8  else
9  |   v̂′ ← σ̂′(O)(f)
10 |   return (v̂′, σ̂′, Γ̂′)
```

Figure 3.27: Evaluating field selections with warnings

Algorithm 30: $\widehat{\text{CacheEval}}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), O, \widehat{\Gamma})$

```
1  if O ∈ dom(σ̂) then
2  |   if O ≠ O_current then
3  |   |   ρ ← σ̂(O)
4  |   |   if |dom(ρ)| < |FieldsOf(O)| then
5  |   |   |   report_warning("Cyclic initialization order!", stackTrace)
6  |   return ({O}, σ̂, Γ̂)
7  else
8  |   ψ̂′ ← O
9  |   σ′ ← σ ∪ [ψ̂′ ↦ []]
10 |   𝕆 ← TemplateOf(O)
11 |   (σ̂″, Γ̂′) ← initializeObject^(𝕆, σ̂′, ψ̂′, Γ̂)
12 |   return ({O}, σ̂″, Γ̂′)
```

Figure 3.28: Preventing deadlocks when evaluating direct references to global objects

terminates on all input programs. With the modified algorithms in the previous section, the abstract interpreter also reports warnings for all patterns that may result in run-time errors: It prohibits accessing uninitialized fields and prohibits cyclic wait-for order of the initialization processes of global objects. The programs without warnings maintain two run-time guarantees: they never throw null pointer exceptions related to uninitialized fields (`null` is excluded from the abstract domain) and they never result in deadlocks due to mutual waiting on the initialization of global objects.

Chapter 4

Checking Initialization Locally

This chapter investigates how to check the initialization safety of each global object individually, rather than interpreting the entire program. Section 4.1 identifies the weaknesses of the whole-program analysis presented in Chapter 3, which checks the program by starting with the global object `Main`. Section 4.2 then formally defines the property of initialization-time irrelevance, which states that the concrete initialization process of a global object during program execution can be replaced by the local initialization process, which ignores the initialization point. Section 4.3 then proves that, in most cases, all global objects in the immutable initialization calculus inherently possess initialization-time irrelevance, given that local initialization always succeeds. Section 4.4 then designs the local initialization checker. Instead of visiting the whole program starting at `Main`, it is sound to check the local initialization process of each object in arbitrary order and reuse previous results.

4.1 Motivation and goals

Section 2.4 defines a concrete interpreter that dynamically maintains initialization safety, and Chapter 3 over-approximates it in order to statically maintain initialization safety. However, the interpreter in Section 2.4 requires the complete program with all global object and class definitions, which makes the global object initialization checker a whole-program analysis. Whole-program analysis generally suffers from analyzing deep levels of initialization code in one go, which increases the risk of stack overflow of the checker and leaves a confusing trace of history when reporting the warning to the user. Consider the program in Figure 4.1. The stack trace of the global object initialization checker must

```
object Main {
    O1
}
object O1 {
    val f1 = O2
}
object O2 {
    val f2 = O3
}
// omit O3 to O99 which follow the same pattern
object O100 {
    val f100 = O99.f99
}
```

Figure 4.1: Deep levels of global object initialization in a project

start from `Main`, then initialize `O1`, `O2`, $\cdots$, `O100`, before finding a cyclic initialization order between only `O99` and `O100`. When the checker reports the warning with the full stack trace of the checker, the initialization processes from `O1` to `O98` are irrelevant to the actual errors. The actual error is irrelevant to the initialization point of `O99` and `O100` and can be detected by checking only the templates of these two objects.

This chapter aims to avoid whole-program analysis by adopting a local analysis of each individual global object initialization process. However, locally checking global objects will ignore all the global states of the program interpreter when the initialization starts. Is it always safe to do so? This chapter will define a property that characterizes the objects that can be safely initialized without the information of the whole-program analysis, known as initialization-time irrelevance.

4.2 Initialization-time irrelevance

It is important to understand the global states of the program interpreter when it concretely initializes an object. In Chapter 2, the initialization of a global object O corresponds to a call to initializeObject during the concrete execution. The input of the initialization contains (O, σ, ψ). Specifically, the input heap σ contains all relevant objects and instances whose initialization starts before O. Therefore, the initialization point of an object O can be represented by the input heap σ. For example, when the concrete interpreter initializes

```
object Main { O1.foo() }
object O1 {
    val f1 = 5
    def foo() = O2.f2
}
object O2 {
    val f = O1.f1
}
```

Figure 4.2: The initialization point of global objects

O2 in Figure 4.2, the input heap contains O1 that is already fully initialized, and the initialization process of O2 accesses O1. However, a local analysis of O2 will ignore the input heap and may initialize O2 in an empty heap, which corresponds to the following call:

$$\textsc{InitializeObject}(\texttt{O2}, [\texttt{O2} \mapsto [(0, \texttt{O2}) \mapsto []]], (0, \texttt{O2})).$$

This initialization process treats O2 as the first object to be initialized in the program. This also modifies the initialization point of O1, which is now initialized during the initialization of O2. Since it ignores the input heap and the evaluations before the concrete initialization of O2, the interpreter can keep a shorter trace of O2. However, the interpreter must also ensure that initializing O2 first and then executing the program does not affect initialization safety. Therefore, we want a definition that precisely defines the following property of a program: The interpreter can conveniently interpret the local initialization process of a global object O instead of interpreting its initialization during concrete program execution, and the program semantics stays the same. Here, the program semantics refers to the semantics in Section 2.4 that maintains initialization safety. Formally:

Definition 4.1 (Initialization-time irrelevance:) Given a program $\mathbb{P} =$

$$(\overline{\mathbb{C}}, \overline{\mathbb{O}}, \mathfrak{e}_{main})$$

Let O be a global object in $\overline{\mathbb{O}}$ that is initialized in $\mathbb{P}$. We define the following program which initializes O first before executing the program:

$$\mathbb{P}'_O = (\overline{\mathbb{C}}, \overline{\mathbb{O}}, O; \mathfrak{e}_{main})$$

.

We say that $\mathbb{P}'_O$ conducts the local initialization of O before executing the program. Then, with respect to the concrete interpreter defined in Section 2.4, the initialization of O is initialization-time irrelevant in $\mathbb{P}$ if both EXECUTE'($\mathbb{P}$) and EXECUTE'($\mathbb{P}'_O$) fail in the same way (i.e., both detect unsafe initialization or both do not terminate), or if both succeed and return the same result and output heap.

4.3 Initialization-time irrelevance for free

In this section, we will prove that all objects in the current initialization calculus are initialization-time irrelevant.

Theorem 4.2: Let $\mathbb{P}$ be a program written in the immutable initialization calculus in Chapter 2. Then all objects initialized in $\mathbb{P}$ (excluding `Main`) are initialization-time irrelevant.

Sketch Proof: We will mostly present the structure of the proof without going into too much technical detail. The proof will focus on the following two evaluation processes:

$$\begin{aligned}
&\text{EVAL'}(\delta_\emptyset = (\sigma = [], \tau = [], \psi = (0, \texttt{Main}), O_{current} = \texttt{Main}), \mathfrak{e}_{main}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\text{(corresponding to EXECUTE'}(\mathbb{P})) \\
&\text{EVAL'}(\delta_O = (\sigma = \sigma_O, \tau = [], \psi = (0, \texttt{Main}), O_{current} = \texttt{Main}), \mathfrak{e}_{main}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\text{(corresponding to EXECUTE'}(\mathbb{P}_O)) \\
&\text{where } (_, \sigma_O) \leftarrow \text{EVAL'}((\sigma = [], \tau = [], \psi = (0, \texttt{Main}), O_{current} = \texttt{Main}), O)
\end{aligned}$$

We know that O is initialized in $\mathbb{P}$ and the second process initializes O ahead of time. Do they always return the same result or fail the same way? Intuitively, σ_O contains the result of initializing O ahead of time as well as some other objects triggered by the initialization of O, so when the second evaluation process accesses O or any other objects in σ_O, it can directly get the result instead of initializing them. Then, the second evaluation process should evaluate to the same result but takes more steps for the interpreter due to

initializing multiple objects ahead of time. To formally define this property, we introduce the following definition of fuel:

Definition 4.3: For every call to an evaluation function $\textsc{Eval}'(\delta, \mathfrak{e})$ where $\delta \in Conf$, we define:

$$fuel(\textsc{Eval}'(\delta, \mathfrak{e})) = \begin{cases} \infty, \text{ if } \textsc{Eval}\text{' does not terminate or detects unsafe initialization} \\ \text{The total number of recursive calls of the interpreter during } \textsc{Eval}\text{'} \end{cases}$$

With the definition of the fuel, we can prove the following important lemma:

Lemma 4.4: For all $\delta \in Conf, \mathfrak{e}, \sigma_1, O, f$, if $\textsc{Eval}\text{'}(\delta, \mathfrak{e}) \rightsquigarrow_f (v, \sigma_1)$ (i.e. $\textsc{Eval}\text{'}(\delta, \mathfrak{e})$ successfully evaluates to (v, σ_1) and costs fuel f) and O is not initialized in δ, then:

- $O \in \sigma_1 \Longrightarrow \textsc{Eval}\text{'}(\delta, O; \mathfrak{e}) \rightsquigarrow_g (v, \sigma_1)$ for some fuel $g > f$
- $O \notin \sigma_1 \wedge \textsc{InitializeObject}'(O, \sigma_1 \cup [O \mapsto [(0, O) \mapsto (O, [])]], (0, O)) \rightsquigarrow_g \sigma_2 \Longrightarrow \textsc{Eval}\text{'}(O; \mathfrak{e}) \rightsquigarrow_{f+g} (v, \sigma_2)$

Proof: By strong induction on fuel f.

Base case: $f = 1$. According to the concrete semantics, $\mathfrak{e}$ must be one of the following:

- $\mathfrak{e} = x$ for some parameter x. In this case, $\textsc{Eval}'$ does not change the input heap. Since O is not in the input heap, $O \notin \sigma_1$. With the additional assumption that $\textsc{InitializeObject}'(O, \sigma_1 \cup [O \mapsto [(0, O) \mapsto (O, [])]], (0, O)) \rightsquigarrow_g \sigma_2$, consider evaluating $O; x$ in the original configuration. Evaluating $O; x$ will immediately call $\textsc{InitializeObject}'(O, \sigma_1 \cup [O \mapsto [(0, O) \mapsto (O, [])]], (0, O))$ since σ_1 is the input heap. Next it takes an additional step to select x in σ_2. In the immutable calculus, evaluating O cannot change the value of x in the heap. Therefore, evaluating $O; x$ returns (v, σ_2).
- $\mathfrak{e} = O'$ and $O' = O_{current}$ in the input configuration. Again, $\textsc{Eval}'$ does not change the input heap, so $O \notin \sigma_1$. Using a similar argument as above, evaluating $O; O'$ will immediately initialize O in the input configuration, which returns σ_2. In the immutable calculus, O' will stay initialized in σ_2 without any mutation during the initialization of O, so evaluating $O; O'$ returns (v, σ_2).

Inductive case: We still do case analysis on $\mathfrak{e}$.

- $\mathfrak{e} = \mathfrak{e}_1 . f$. We know $\textsc{Eval}'(\delta, \mathfrak{e}_1 . f) \rightsquigarrow_f (v, \sigma_1)$, and this evaluation process first evaluates $\mathfrak{e}_1$, and selecting f does not modify the heap, so we know $\textsc{Eval}'(\delta, \mathfrak{e}_1) \rightsquigarrow_{f-1} (v', \sigma_1)$.
  - $O \in \sigma_1$. Then consider $\textsc{Eval}'(\delta, O; \mathfrak{e}_1 . f)$. This evaluation process first evaluates $\textsc{Eval}'(\delta, O; \mathfrak{e}_1)$. Now the inductive hypothesis indicates that $\textsc{Eval}'(\delta, O; \mathfrak{e}_1) \rightsquigarrow_g (v', \sigma_1)$ for some $g > f - 1$. Then evaluating $O; \mathfrak{e}_1 . f$ must still return (v, σ_1) with fuel $g + 1 > f$.
  - $O \notin \sigma_1$, and $\textsc{InitializeObject}'(O, \sigma_1 \cup [O \mapsto [(0, O) \mapsto (O, [])]], (0, O)) \rightsquigarrow_g \sigma_2$. Also, in the immutable calculus, initializing O in σ_1 does not modify any existing instances in σ_1. Then evaluating $O; \mathfrak{e}_1 . f$ still first evaluates $O; \mathfrak{e}_1$. The inductive hypothesis indicates that evaluating $O; \mathfrak{e}_1$ returns (v', σ_2), then selecting f will return (v, σ_2).

- $\mathfrak{e} = \mathfrak{e}_1 . m(\overline{\mathfrak{e}_i})$. Then the evaluation process of $\mathfrak{e}$ is broken down into the following sequence of sub-processes: First, evaluating $\mathfrak{e}_1$ returns (v', σ'), then evaluating each of $\mathfrak{e}_i$ based on σ' returns (v_i, σ_i), finally evaluating the method body $\mathfrak{e}_m$ returns (v, σ_1). We are interested in evaluating $O; \mathfrak{e}_1 . m(\overline{\mathfrak{e}_i})$.
  - $O \in \sigma_1$. Then there are several possibilities. If $O \in \sigma'$, the inductive hypothesis states that evaluating $(O; \mathfrak{e}_1)$ still returns (v', σ'), and the rest of the sub-processes proceed with the same steps, so evaluating $O; \mathfrak{e}_1 . m(\overline{\mathfrak{e}_i})$ still returns (v, σ_1). If there exists k such that $O \in \sigma_{k'}$ for all $k' \geq k$ but $O \notin \sigma_{k'}$ for all $k' < k$. Then the inductive hypothesis on the evaluation of $\mathfrak{e}_k$ states that evaluating $\mathfrak{e}_k$ is equivalent to evaluating $O; \mathfrak{e}_k$ based on σ_{k-1}. However, evaluating $O; \mathfrak{e}_k$ based on σ_{k-1} is equivalent to first initializing O in σ_{k-1}, returning σ', then evaluating $\mathfrak{e}_k$ based on σ'. The inductive hypothesis of the evaluation of $\mathfrak{e}_{k-1}$ then states that evaluating $O; \mathfrak{e}_{k-1}; \mathfrak{e}_k$ is equivalent to evaluating $\mathfrak{e}_{k-1}; O; \mathfrak{e}_k$, which is equivalent to evaluating $\mathfrak{e}_{k-1}; \mathfrak{e}_k$. We can continually apply inductive hypotheses and prove that evaluating $O; \mathfrak{e}_1 . m(\overline{\mathfrak{e}_i})$ is equivalent to evaluating $\mathfrak{e}_1 . m(\overline{\mathfrak{e}_i})$. The same argument applies if O is only initialized during the evaluation of method body.
  - $O \notin \sigma_1$, and evaluating $\mathfrak{e}_1 . m(\overline{\mathfrak{e}_i}); O$ returns σ_2. Continually applying inductive hypotheses, we can show that evaluating $\mathfrak{e}_1 . m(\overline{\mathfrak{e}_i}); O$ is equivalent to evaluating $\mathfrak{e}_1; O; \overline{\mathfrak{e}_i}; \mathfrak{e}_m$, which is transitively equivalent to evaluating $O; \mathfrak{e}_1 . m(\overline{\mathfrak{e}_i})$.

```
object O1 {
    val f1 = 5
    val f2 = 6
}

object O {
    val f = O1.f2
}
```

Figure 4.3: A dependent initialization order

- $\mathfrak{e} = \texttt{new}\ C(\overline{\mathfrak{e}_i})$. We also break the evaluation process into a sequence of sub-processes: First, the evaluation of $\overline{\mathfrak{e}_i}$; next, the sequence of parameters of parent classes of C following the inheritance hierarchy. Then we can apply a similar argument to the case of method invocations.

- $\mathfrak{e} = O'$ and O' is not initialized in δ. We also break the evaluation process into a sequence of sub-processes: First, the sequence of parameters of parent classes of O' following the inheritance hierarchy; next, the sequence of fields in O'. Then we also consider whether O is in the heap after initializing O', say $\sigma_{O'}$.

  - $O \in \sigma_{O'}$. This means that the initialization of O' depends on O and the initialization of O and O' is safe. Therefore, the initialization of O does not depend on O', and we can push the initialization point of O upwards along the evaluation process of O' until O is initialized before O', similar to the case of method invocations.
  - $O \notin \sigma_{O'}$ and initializing O in $\sigma_{O'}$ returns σ_O. This means that the initialization of O' does not depend on O and the initialization of O is safe. However, we must be careful if we use a similar argument in the case of method invocations and push the initialization point of O upwards along the evaluation process of O', because the initialization of O may contains an access to O'. Consider the program in Figure 4.3. Suppose O is initialized after O' and we push the initialization point of O before the initialization of `f2`, then `O1` will be initialized during the initialization of `O` as it accesses `O1`. We need to prove that when pushing the initialization point of a global object ahead-of-time, it does not matter to change the initialization point of other objects with the following proposition:

Proposition 4.5: If $\textsc{Eval}'(\sigma, \tau, \psi, O_{current}, O')$ returns $\sigma_{O'}$ and initializing O in $\sigma_{O'}$ returns σ_O for some $O \notin \sigma_{O'}$, then $\textsc{Eval}'((\sigma, \tau, \psi, O_{current}), O; O')$ also returns σ_O.

Proof: We only consider the case when the initialization of O contains access to O', and pushing the initialization of O before O' changes the initialization point of O', since the other case is trivial using a similar argument to method invocations in the proof of Lemma 4.4. We need to show that for each object triggered by the initialization of O' (including O'), its initialization process behaves the same regardless of whether O' is initialized before or during the initialization of O. If the initialization process of every object behaves the same, then because there is no mutation in the immutable calculus, the final heap must be the same. Since the initialization of O is safe, it induces a wait-for graph that is a DAG based on Definition 2.1. Then we can prove by induction on the position of O' in such a DAG.

Base case: O' is a leaf. This means that the initialization O' does not depend on any other objects, then the initialization of O' must be the same regardless of when O' is initialized.

Inductive case: O' directly accesses other objects, say $\overline{O_i}$. Every object reachable from some O_i must also be reachable from O, so they do not access O. Then we can apply the inductive hypothesis on all of O_i, which indicates that their initialization process stay the same if they are initialized before O. Therefore, when initializing O' during O, if it accesses and triggers the initialization process of O_i, the induction hypothesis states that the initialization process of O_i is the same as if it is initialized before O and O' accesses it later (again, immutability ensures this). Since the initialization of O' is decided by the result of initializing $\overline{O_i}$, we know that initializing O' ahead of O is the same as initializing O' during O.

□

Proposition 4.5 finished the proof of Lemma 4.4. Now we can prove Theorem 4.2. If $\mathbb{P}$ succeeds and O is initialized in $\mathbb{P}$, then O is in the final heap, then Lemma 4.4 directly show that evaluating $O; \mathfrak{e}_{main}$ also returns the same value and final heap. If $\mathbb{P}$ does not terminate, then evaluating $O; \mathfrak{e}_{main}$ does not terminate either since it requires more fuel. In the other direction, if $\mathbb{P}'_O$ succeeds, then initializing O ahead of time returns σ_O. Then, evaluating e_{main} in σ_O is equivalent to initializing an extra object `Main` with a field e_{main}. Since `Main` $\notin \sigma_O$ and the program execution successfully returns (v, σ), Lemma 4.4 shows

Algorithm 31: LocallyCheckOneObject($\overline{\mathbb{C}}, \overline{\mathbb{O}}, \mathbb{E}$)

1 $\mathbb{O}_j \leftarrow$ any object in $\overline{\mathbb{O}}$
2 $\sigma \leftarrow [\mathbb{O}_j \mapsto [(0, \mathbb{O}_j) \mapsto []]]$
3 $\sigma_O \leftarrow$ InitializeObject'$(\mathbb{O}_j, \sigma, (0, \mathbb{O}_j))$
4 return Eval'$((\sigma_O[\texttt{Main} \mapsto (0, \texttt{Main}) \mapsto []], [], (0, \texttt{Main}), \texttt{Main}), \mathfrak{e}_{main})$

Figure 4.4: Locally checking one global object

that evaluating $\texttt{Main}; O$ still returns (v, σ). However, since O is initialized in $\mathbb{P}$, evaluating $\texttt{Main}; O$ is exactly evaluating $\mathbb{P}$ and it returns the same result. Also, if $\mathbb{P}'_O$ does not terminate, then evaluating $\texttt{Main}; O$ does not terminate either. So does evaluating $\mathfrak{e}_{main}$, which only reduces a finite number of steps compared to evaluating $\texttt{Main}; O$. Therefore, we have proved Theorem 4.2, which states that all initialized objects in the immutable calculus are initialization-time irrelevant. Chapter 5 will discuss how to maintain initialization-time irrelevance when a program includes mutable states.

□

4.4 Designing a local analysis

Initialization-time irrelevance enables the design of a local analysis for the global object initialization checker. Consider the analysis of the LocallyCheckOneObject function in Figure 4.4. It follows the same concrete domain and concrete semantics in Section 2.4, but locally initializes one arbitrary object before executing the whole program. As a natural consequence of Theorem 4.2, we know that if LocallyCheckOneObject succeeds, then Execute' also succeeds with the same output.

The advantage of LocallyCheckOneObject over Execute′ is that errors can be detected without deep levels of visiting. For example, LocallyCheckOneObject can quickly detect the cyclic initialization order in Figure 4.1 when starting at `O99` and return the warning in Figure 4.5, which only shows the template of related objects instead of the trace from the `Main` object to the related objects. Additionally, for the objects in σ_O, the concrete interpreter never initializes them twice during program execution.

We also believe that the interpreter can locally check all objects sequentially. Figure 4.7 then illustrates an improvement of the local analysis in Figure 4.6. Compared to LocallyCheckOneObject, CheckObjects triggers the initialization process of each

```
|Cyclic initialization: object O99 -> object O100 -> object O99.
|Calling trace:
|├── object O99 {
||   ^
|├── val f99 = O100
||             ^^^^
|├── object O100 {
||   ^
|└── val f100 = O99.f99
```

Figure 4.5: Warning to the program in Figure 4.1

Algorithm 32: CHECKOBJECTS($\overline{\mathbb{C}}, \overline{\mathbb{O}}, \mathfrak{e}_{main}$)

1 $\sigma \leftarrow [\texttt{Main} \mapsto (0, \texttt{Main}) \mapsto []]$
2 **foreach** $\mathbb{O}_i \in \overline{\mathbb{O}}$ *(in arbitrary order)* **do**
3   **if** $O_i \notin dom(\sigma)$ **then**
4         $\sigma' \leftarrow \sigma[\mathbb{O}_i \mapsto [(0, \mathbb{O}_i) \mapsto []]]$
5         $\sigma \leftarrow$ INITIALIZEOBJECT$'(\mathbb{O}_i, \sigma', (0, \mathbb{O}_i))$
6 return EVAL'$((\sigma, [], (0, \texttt{Main}), \texttt{Main}), \mathfrak{e}_{main})$

Figure 4.6: Checking global objects at arbitrary order

global object in a flexible order. In the first iteration, O_1 is initialized with an empty heap. After the first iteration, the heap contains O_1 and other global objects that O_1 depend on. Later objects can directly access the objects that have been initialized. Therefore, each object is still only initialized once. We can prove that $\widehat{\text{CHECKOBJECTS}}2$ is still sound with respect to the immutable initialization calculus because of Theorem 4.2, by viewing the program as the local initialization of the object `Main`. Finally, we adopt CHECKOBJECTS as the main concrete interpretation function, and we can equip the abstract interpreter with the abstract function $\widehat{\text{CHECKOBJECTS}}$, as illustrated in Figure 4.7.

Algorithm 33: $\widehat{\text{CHECKOBJECTS}}2(\overline{\mathbb{C}}, \overline{\mathbb{O}}, \mathfrak{e}_{main})$

1 $\widehat{\sigma} \leftarrow []$
2 **foreach** $\mathbb{O}_i \in \overline{\mathbb{O}}$*(in arbitrary order)* **do**
3   **if** $O_j \notin dom(\widehat{\sigma})$ **then**
4         $\widehat{\psi} \leftarrow O_i$
5         $\widehat{\sigma} \leftarrow \widehat{\sigma} \cup [\widehat{\psi} \mapsto []]$
6         $\widehat{\Gamma} \leftarrow []$
7         $(\widehat{\sigma}', \widehat{\Gamma}') \leftarrow \widehat{\text{INITIALIZEOBJECT}}(\mathbb{O}_i, \widehat{\sigma}, \widehat{\psi}, \widehat{\Gamma})$
8         $\widehat{\sigma} \leftarrow \widehat{\sigma}'$

9 return $\widehat{\text{CACHEEVAL}}((\widehat{\sigma}, [], \texttt{Main}, \texttt{Main}), \mathfrak{e}_{main}, \widehat{\Gamma})$

Figure 4.7: Checking global objects at arbitrary order with initialization-time irrelevance

Chapter 5

The Mutable Initialization Calculus

This chapter extends the initialization calculus presented in Chapter 2 with mutable fields and class parameters as well as assignments to them. Section 5.2 presents the syntax and concrete semantics of the mutable initialization calculus. Section 5.3 then points out that mutating global object fields or class parameters after their initialization may break initialization-time irrelevance. In order to maintain a local initialization checker, Section 5.3 proposes another static principle that can be ensured by a local initialization checker, which limits the access scope of all mutable fields and class parameters.

5.1 Motivation and goals

The expressiveness of the immutable calculus presented in Chapter 2 is limited because it restricts fields and parameters to be immutable. In contrast, Scala allows mutable fields of global objects and mutable class parameters. The value of these mutable states can be re-assigned after their initialization through any reference to their defining global object or class instance. For example, the method `foo` in Figure 5.1 mutates the field `f1` of `O1` as well as the mutable parameter of the instance of `C` through `f2`. This chapter aims to model mutations in the initialization checker. Although mutating a field or parameter occurs after its initialization, mutation could affect the initialization process of other global objects. In order to develop a checker that maintains initialization safety in the presence of mutation, this chapter must achieve the following goals:

- Propose an initialization calculus with mutable fields and class parameters.

```
class C(var p)
object O1 {
    var f1 = new C(5)
    val f2 = new C(5)
}

object O2 {
    def foo() {
        O1.f1 = new C(6)
        O1.f2.p = 6
        O1.f2
    }
}
```

Figure 5.1: A program defining mutable states and mutating them

- Propose the concrete and abstract initialization semantics of the mutable initialization calculus.
- Investigate how mutation affects the local analysis of the initialization of other global objects.

5.2 Syntax and concrete semantics

In the initialization calculus in Chapter 2, the fields of global objects are immutable since they are qualified by `val`. In Scala, mutable fields and mutable parameters are qualified by `var`. Therefore, Figure 5.2 presents the syntax of a modified calculus updated to allow global object fields and class parameters to be qualified by `var`. Note that method parameters are still immutable and are not qualified. Additionally, each method in the mutable initialization calculus can include a list of assignments before the body. The first form of assignment mutates fields of global objects, and the second form of assignment mutates class parameters of the receiver of the method. The rest of the syntactic components are identical to the immutable initialization calculus.

In order to present the concrete semantics of the mutable calculus, we assume an auxiliary helper function $\text{isMutable}(f/p)$, which returns true only if the input f or p is qualified by `var`. The auxiliary function $\text{BodyOf}(C/O, m)$ is modified to also return the

Names:

$$
\begin{array}{rl}
O \in & ObjectName \\
C \in & ClassName \\
f \in & FieldName \\
m \in & MethodName \\
p \in & ParamName
\end{array}
$$

Syntax:

$$
\begin{array}{rrr}
\mathbb{P} ::= & \overline{\mathbb{C}}\ \overline{\mathbb{O}}\ \mathbb{O}_{main} \in & Program \\
\mathbb{C} ::= & \texttt{class}\ C\texttt{(}\overline{\mathbb{CP}}\texttt{)}\ \texttt{extends}\ \mathbb{PC}\texttt{(}\overline{\mathfrak{e}}\texttt{)}\ \{\ \overline{\mathbb{M}}\ \} \in & ClassTemplate \\
\mathbb{CP} ::= & \texttt{val}\ p \mid \texttt{var}\ p \in & ClassParameter \\
\mathbb{O} ::= & \texttt{object}\ O\ \texttt{extends}\ \mathbb{PC}\texttt{(}\overline{\mathfrak{e}}\texttt{)}\ \{\ \overline{\mathbb{F}}\ \overline{\mathbb{M}}\ \} \in & ObjectTemplate \\
\mathbb{PC} ::= & \texttt{Object} \mid C \in & ParentClassName \\
\mathbb{F} ::= & \texttt{val}\ f\ \texttt{=}\ \mathfrak{e} \mid \texttt{var}\ f\ \texttt{=}\ \mathfrak{e} \in & Field \\
\mathbb{M} ::= & \texttt{def}\ m\texttt{(}\overline{p}\texttt{)}\ \texttt{=}\ \{\ \overline{\mathbb{A}}\ \mathfrak{e}\ \} \in & Method \\
\mathbb{A} ::= & \mathfrak{e}\texttt{.}f\ \texttt{=}\ \mathfrak{e} \mid p\ \texttt{=}\ \mathfrak{e} \in & Assignment \\
\mathfrak{e} ::= & p \mid O \mid \texttt{new}\ C\texttt{(}\overline{\mathfrak{e}}\texttt{)} \mid \mathfrak{e}\texttt{.}f \mid \mathfrak{e}\texttt{.}m\texttt{(}\overline{\mathfrak{e}}\texttt{)} \in & Expression
\end{array}
$$

Notation:

$$
\overline{x} ::= \overbrace{x, x, \cdots, x}^{\geq 0}
$$

Figure 5.2: The syntax of the mutable initialization calculus

$\textsc{Assign} :: Conf \times Assignment \rightarrow Heap$

Algorithm 34: $\textsc{Assign}((\sigma, \tau, \psi, O_{current}), \mathfrak{e}_0 \,.\, f \texttt{ = } \mathfrak{e}_1)$

1. $(v_0, \sigma_0) \leftarrow \textsc{Eval'}((\sigma, \tau, \psi, O_{current}), \mathfrak{e}_0)$
2. $(0, O) \leftarrow v_0$
3. $(O, \rho) \leftarrow \sigma_0(O)(v_0)$
4. $assert(\text{is}\textsc{Mutable}(f) \wedge f \in dom(\rho))$
5. $(v_1, \sigma_1) \leftarrow \textsc{Eval'}((\sigma_0, \tau, \psi, O_{current}), \mathfrak{e}_1)$
6. $\rho' \leftarrow \rho[f \mapsto v_1]$
7. $\sigma' \leftarrow \sigma_1[O \mapsto \sigma_1(O)[v_0 \mapsto (O, \rho')]]$
8. return σ'

Algorithm 35: $\textsc{Assign}((\sigma, \tau, \psi, O_{current}), p \texttt{ = } \mathfrak{e})$

1. $(_, O) \leftarrow \psi$
2. $(C/O, \rho) \leftarrow \sigma(O)(\psi)$
3. $assert(\text{is}\textsc{Mutable}(p) \wedge p \in dom(\rho))$
4. $(v_0, \sigma_0) \leftarrow \textsc{Eval'}((\sigma, \tau, \psi, O_{current}), \mathfrak{e})$
5. $\rho' \leftarrow \rho[p \mapsto v_0]$
6. $\sigma' \leftarrow \sigma_0[O \mapsto \sigma_0(O)[\psi \mapsto (C/O, \rho')]]$
7. return σ'

Figure 5.3: The semantics of assignment in the concrete interpreter

assignments introduced in the method together with the expression of the method body. The concrete semantics of the mutable calculus operates in the same concrete domain and inherits all functions from the previous concrete interpreter in Section 2.4, which maintains initialization safety dynamically. In addition, the mutable initialization calculus includes another function for the concrete interpreter that describes the re-assignment of mutable fields and class parameters, as illustrated in Figure 5.3. The assignment takes the concrete configuration of the current method invocation as the input, but it only modifies the heap and does not return any values. When assigning to mutable fields of global objects, the interpreter first evaluates $\mathfrak{e}_0$, whose result must be the object O that defines f. Then, the interpreter ensures that the field is mutable and initialized, since assignment must occur after initialization. Finally, the interpreter evaluates the right-hand side of the assignment and changes the value of the field stored in the heap to the right-hand value. Assignments to class parameters follow a similar procedure. The address of the method receiver ψ must point to an instance with class parameter p, and the interpreter modifies the value of p in the heap. The function in Figure 5.4 then describes the process of invoking methods with assignments. The assignments are evaluated before evaluating the body of the method. During the evaluation of assignments and body expression, ψ always points to the address of the receiver. The resulting concrete interpreter always correctly records the values of mutable fields and parameters in the concrete instances in the heap.

5.3 Mutation and initialization-time irrelevance

5.3.1 Violation of initialization-time irrelevance

Mutating global object fields or class parameters does not seem to affect the initialization process, which takes place before mutation occurs. However, we also need to consider whether mutation affects the soundness of the local analysis presented in the previous chapter. When proving the soundness of the local interpreter, immutability is a key feature that fixes the initialization behaviour of each global object, regardless of its initialization point. However, the value of a mutable field or parameter may change after its initialization, via assignment. This means that the initialization process of specific objects depends on their initialization point. Consider the program in Figure 5.5, which is also presented in Chapter 1. The local initialization of `B` starts with an empty heap and triggers the local initialization of `A`, which initializes `a` to an instance of class `X`. The local initialization of `B` will then call `foo` defined in `X`, which is safe. However, in global execution, A.a is assigned an instance of Y before B is initialized. Consequently, the initialization of `B` will call `foo`

Algorithm 36: $\text{EVA'L}((\sigma, \tau, \psi, O_{current}), \mathfrak{e}\,.\,m(\overline{\mathfrak{e}_i}))$

1 $(v, \sigma_0) \leftarrow \text{EVAL'}((\sigma, \tau, \psi, O_{current}), \mathfrak{e})$
2 $(_, O) \leftarrow v$
3 $(C/O, _) \leftarrow \sigma_0(O)(v)$
4 $\sigma \leftarrow \sigma_0$
5 $\overline{p_i} \leftarrow \text{PARAMSOF}(m)$
6 $\tau' = []$
7 **foreach** i **do**
8 $\quad (v_i, \sigma_i) \leftarrow \text{EVAL}((\sigma, \tau, \psi, O_{current}), \mathfrak{e}_i)$
9 $\quad \sigma \leftarrow \sigma_i$
10 $\quad \tau' \leftarrow \tau' \cup [p_i \mapsto v_i]$
11 $(\overline{\mathbb{A}_j}, \mathfrak{e}') \leftarrow \text{BODYOF}(C/O, m)$
12 **foreach** j **do**
13 $\quad \sigma \leftarrow \text{ASSIGN}((\sigma, \tau', v, O_{current}), \mathbb{A}_j)$
14 return $\text{EVAL}(\sigma, \tau', v, \mathfrak{e}')$

Figure 5.4: The semantics of invoking methods with assignments

defined in `Y`, which reads the uninitialized field `B.b` and is unsafe. Figure 5.6 is a similar example, but the initialization safety of `B` depends on the value of `p` in the instance that `A.a` points to. The local initialization of `B` is safe because the local initialization of `A` initializes `p` to be an instance of `X`. Although `A.a` is immutable, the `Main` object can still mutate the value of `p` to an instance of `Y` through `A.a`. This changes the initialization status of `A` and makes the initialization of `B` unsafe during program execution. In both examples, the behaviour of the local initialization process of `B` is inconsistent with the concrete initialization process, because the initialization behaviour depends on the values of mutable states in the concrete initialization point.

5.3.2 Restricting access to mutable states

With mutation, whole-program analysis seems inevitable unless we ensure the initialization-time irrelevance of the input program with static principles. Based on the observation of Figure 5.5, it seems natural that if O is initialization-time irrelevant, then its initialization process cannot depend on the mutable states created during the initialization of other objects, because the values of such mutable states are inconsistent at different initialization points of O. Therefore, we propose the following definition and principle.

```
1   class X { def foo() = new X }
2   class Y extends X { def foo() = B.b }
3   object A { var a = new X }
4   object B { val b = A.a.foo() }
5   object Main {
6       def bar() = { A.a = new Y; B.b }
7       Main.bar()
8   }
```

Figure 5.5: A program without initialization-time irrelevance

```
1   class X { def foo() = new X }
2   class Y extends X { def foo() = B.b }
3   class Z(var p) {
4       def getP() = p
5       def mutateAndGetP() = { p = new Y; p }
6   }
7   object A { val a = new Z(new X) }
8   object B { val b = A.a.getP().foo() }
9   object Main {
10      A.a.mutateAndGetP(); B.b
11  }
```

Figure 5.6: Another program without initialization-time irrelevance

Definition 5.1 (Owner object): Every mutable state (mutable field or class parameter) has an owner global object (equivalently, every mutable state is owned by a global object). A mutable global object field is owned by the defining object itself, and a mutable class parameter is owned by the current object under initialization ($O_{current}$) when its defining instance is instantiated.

Principle 5.2 (Restriction of mutable states): During the initialization of a global object O, it can only read the value mutable states owned by O. It cannot read the value of any mutable state owned by other global objects.

It turns out that Principle 5.2 is a sufficient condition to maintain initialization-time irrelevance.

Theorem 5.3: If a program in the mutable initialization calculus satisfies Principle 5.2, then all objects initialized in the program are initialization-time irrelevant.

The proof of Theorem 5.3 is omitted but we can prove that Principle 5.2 ensures that Lemma 4.4 still holds using a similar proof, which ensures that all initialized objects are initialization-time irrelevant. Moreover, the local interpreter can enforce Principle 5.2 when the domain records owner object of each mutable state. For example, the concrete heap in Figure 5.7 records the owner object of each mutable state together with the defining instance of the mutable state. This is necessary to enforce Principle 5.2, which needs to check the owner object of every mutable state, even after the initialization of the mutable state finishes. Figure 5.8 then defines $\textsc{Eval}''$ to check the owner object when reading mutable fields or class parameters and ensures that the owner object matches the current object under initialization. Now, all programs in the mutable calculus can be checked locally by enforcing Principle 5.2.

Next, we can extend the abstract interpreter with assignment and Principle 5.2. Considering that different instances of the same class may be owned by different objects, we can refine the allocation strategy to combine instances with the same class tag and owned by the same object, as illustrated in Figures 5.9 and 5.10. Figure 5.11 then defines $\widehat{CacheEval}'$ that abstractly enforces Principle 5.2. Finally, Figure 5.12 presents the abstract semantics of assignments. When mutating class parameters, the abstract interpreter does not update the value of the mutable class parameters to the right-hand-side value. This is because multiple concrete instances may share the same abstract address. Updating the parameter of one concrete instance in $\widehat{\psi}$ cannot affect the values of the parameters of other instances in $\widehat{\psi}$, so the abstract interpreter unions the right-hand-side value into the set of

$$
\begin{aligned}
& Addr ::= && \mathbb{N} \\
v \in \quad & Value ::= && Addr \\
\sigma \in \quad & Heap ::= && Addr \to ObjectName \times Instance \\
& Instance ::= && (ClassName \cup ObjectName) \times ((FieldName \cup ParamName) \to Value)
\end{aligned}
$$

Algorithm 37: $\text{EVAL"}((\sigma, \tau, \psi, O_{current}), O)$

1. **if** $\exists n, \sigma(n) = (O, (O, \rho))$ **then**
2.   **if** $O_{current} \neq O$ **then**
3. $assert(|dom(\rho)| = |\text{FIELDSOF}(O)|)$
4.   return (n, σ)
5. **else**
6. $\psi' \leftarrow |dom(\sigma)|$
7. $\sigma \leftarrow \sigma \cup [\psi' \mapsto (O, (O, []))]$
8. $\mathbb{O} \leftarrow \text{TEMPLATEOF}(O)$
9. $\sigma \leftarrow \text{INITIALIZEOBJECT'}(\mathbb{O}, \sigma, \psi')$
10.   return (ψ', σ)

Algorithm 38: $\text{EVAL"}((\sigma, \tau, \psi, O_{current}), \texttt{new}\ C\texttt{(}\overline{\mathfrak{e}_i}\texttt{)})$

1. $\rho \leftarrow []$
2. $\overline{p_i} \leftarrow \text{PARAMSOF}(C)$
3. **foreach** i **do**
4. $(v_i, \sigma_i) \leftarrow \text{EVAL"}((\sigma, \tau, \psi, O_{current}), \mathfrak{e}_i)$
5. $\sigma \leftarrow \sigma_i$
6. $\rho \leftarrow \rho \cup [p_i \mapsto v_i]$
7. $\psi' \leftarrow |dom(\sigma)|$
8. $\sigma \leftarrow \sigma \cup [\psi' \mapsto (O_{current}, (C, \rho))]$
9. $\sigma \leftarrow \text{INITIALIZECLASSTEMPLATE'}(C, \sigma, \psi')$
10. return (ψ', σ)

Figure 5.7: Extending the concrete domain with the owner of mutable states

Algorithm 39: $\text{EVAL''}((\sigma, \tau, \psi, O_{current}), \mathfrak{e} \,.\, f)$

1 $(v, \sigma') \leftarrow \text{EVAL''}(\sigma, \tau, \psi, \mathfrak{e})$
2 $(O, (O, \rho)) \leftarrow \sigma'(v)$
3 **if** $\text{isMutable}(f)$ **then**
4   $assert(O = O_{current})$
5 $assert(f \in dom(\rho))$
6 return $(\rho(f), \sigma')$

Algorithm 40: $\text{EVAL''}((\sigma, \tau, \psi, O_{current}), p)$

1 **if** $p \in dom(\tau)$ **then**
2   return $(\tau(p), \sigma)$
3 **else**
4   $(O, (_, \rho)) \leftarrow \sigma(\psi)$
5   **if** $\text{isMutable}(p)$ **then**
6     $assert(O = O_{current})$
7   $assert(p \in dom(\rho))$
8   return $(\rho(p), \sigma)$

Figure 5.8: Concretely enforcing Principle 5.2

$$
\begin{aligned}
\widehat{\psi} \in \quad & \widehat{Addr} ::= & (ClassName \cup ObjectName) \times ObjectName \\
\widehat{v} \in \quad & \widehat{Value} ::= & P(\widehat{Addr}) \\
\widehat{\sigma} \in \quad & \widehat{Heap} ::= & \widehat{Addr} \rightarrow \widehat{Instance} \\
& \widehat{Instance} ::= & (FieldName \cup ParamName) \rightarrow \widehat{Value} \\
(\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}) \in \quad & \widehat{Conf} ::= & \widehat{Heap} \times \widehat{Frame} \times \widehat{Addr} \times ObjectName
\end{aligned}
$$

Algorithm 41: $\widehat{\text{CACHEEVAL}}$'$((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), O, \widehat{\Gamma})$

1 **if** $O \in dom(\widehat{\sigma})$ **then**
2 | **if** $O \neq O_{current}$ **then**
3 | | $\rho \leftarrow \widehat{\sigma}(O)$
4 | | **if** $|dom(\rho)| < |\text{FIELDSOF}(O)|$ **then**
5 | | | $report_warning$("Cyclic initialization order!", $stackTrace$)
6 | return $(\{O\}, \widehat{\sigma}, \widehat{\Gamma})$
7 **else**
8 | $\widehat{\psi}' \leftarrow (O, O)$
9 | $\sigma' \leftarrow \sigma \cup [\widehat{\psi}' \mapsto []]$
10 | $\mathbb{O} \leftarrow \text{TEMPLATEOF}(O)$
11 | $(\widehat{\sigma}'', \widehat{\Gamma}') \leftarrow \widehat{\text{INITIALIZEOBJECT}}(\mathbb{O}, \widehat{\sigma}', \widehat{\psi}', \widehat{\Gamma})$
12 | return $(\{O\}, \widehat{\sigma}'', \widehat{\Gamma}')$

Figure 5.9: Extending the abstract domain with the owner of mutable states

existing possible values of the mutable class parameter. This is known as a weak update. In contrast, the abstract interpreter can apply strong updates to mutable fields of global objects and rewrite the value of the field in the heap, just as the concrete interpreter. This is because each global object address only abstracts one concrete global object instance. Weak updates are always sound but lose precision compared with strong updates.

To summarize, programs in the mutable calculus can still be checked locally by the concrete or abstract interpreter by enforcing Principle 5.2, which maintains initialization-time irrelevance. The strictness of Principle 5.2 may become a concern as it may reject too many real-world Scala programs. However, Principle 5.2 only enforces on mutable fields and parameters, whereas most fields and parameters are immutable in Scala. We conjecture

Algorithm 42: $\widehat{\text{CacheEval}}$'$((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), \texttt{new } C\,(\overline{\mathfrak{e}_i}), \widehat{\Gamma})$

```
1  if (σ̂, τ̂, ψ̂, O_current, new C (e̅ᵢ)) ∈ dom(Γ̂) then
2  |   return (Γ̂(σ̂, τ̂, ψ̂, O_current, new C (e̅ᵢ)), Γ̂)
3  (v̂_last, σ̂_last) ← (Bottom, σ̂)
4  (v̂_current, σ̂_current) ← (v̂_last, σ̂_last)
5  do
6  |   (v̂_last, σ̂_last) ← (v̂_current, σ̂_current)
7  |   Γ̂ ← Γ̂ ∪ [(Γ̂(σ̂, τ̂, ψ̂, O_current, new C (e̅ᵢ)) ↦ (v̂_last, σ̂_last)]
8  |   p̅ᵢ ← paramsOf(C)
9  |   ρ ← []
10 |   foreach i do
11 |   |   (v̂ᵢ, σ̂ᵢ, Γ̂ᵢ) ← CacheEval'(σ̂, τ̂, ψ̂, O_current, eᵢ)
12 |   |   ρ ← ρ ∪ [pᵢ ← vᵢ]
13 |   |   σ̂ ← σ̂ᵢ
14 |   |   Γ̂ ← Γ̂ᵢ
15 |   ψ̂' ← (C, O_current)
16 |   if ψ̂' ∉ dom(σ̂) then
17 |   |   σ̂ ← σ̂[ψ̂' ↦ ρ]
18 |   else
19 |   |   σ̂ ← σ̂[ψ̂' ↦ ρ ∪ σ̂(ψ̂')]
20 |   (σ̂', Γ̂') ← initializeClassTemplate(C, σ̂, ψ̂', Γ̂)
21 |   v̂_current ← {ψ̂'}; σ̂_current ← σ̂'; Γ̂ ← Γ̂'
22 while (v̂_current, σ̂_current) ≠ (v̂_last, σ̂_last);
23 return (v̂_current, σ̂_current, Γ̂)
```

Figure 5.10: Extending the abstract domain with the owner of mutable states (continue)

Algorithm 43: $\widehat{\text{CacheEval}}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), \mathfrak{e}\,.\,f, \widehat{\Gamma})$

```
1  (v̂, σ̂′, Γ̂′) ← CacheEval^(σ̂, τ̂, ψ̂, O_current, 𝔢, Γ̂)
2  if v̂ = Bottom then
3  |  return (Bottom, σ̂′, Γ̂′)
4  else if isMutable(f) then
5  |  if v̂ = {(O_current, O_current)} then
6  |  |  if f ∈ dom(σ̂′(O_current)) then
7  |  |  |  return (σ̂′(O_current)(f), σ̂′, Γ̂′)
8  |  |  else
9  |  |  |  report_warning(Reading uninitialized field!)
10 |  |  |  return (Bottom, σ̂′, Γ̂′)
11 |  else
12 |  |  report_warning(Violating initialization-time irrelevance!)
13 |  |  return (Bottom, σ̂′, Γ̂′)
14 assert(v̂ = {O} ∧ f ∈ FieldsOf(O) ∧ f ∈ dom(σ̂′(O)))
15 v̂′ ← σ̂′(O)(f)
16 return (v̂′, σ̂′, Γ̂′)
```

Algorithm 44: $\widehat{\text{CacheEval}}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), p, \widehat{\Gamma})$

```
1  if p ∈ dom(τ̂) then
2  |  τ̂(p)
3  else if isMutable(p) then
4  |  if ψ̂ = (_, O_current) then
5  |  |  ρ ← σ̂(ψ̂)
6  |  |  assert(p ∈ dom(ρ))
7  |  |  return (ρ(p), σ̂, Γ̂)
8  |  else
9  |  |  report_warning(Violating initialization-time irrelevance!)
10 |  |  return (Bottom, σ̂, Γ̂)
11 else
12 |  ρ ← σ̂(ψ̂)
13 |  assert(p ∈ dom(ρ))
14 |  return (ρ(p), σ̂, Γ̂)
```

Figure 5.11: Abstractly enforcing Principle 5.2

$$\widehat{\text{Assign}} :: \widehat{Conf} \times Assignment \times \widehat{Cache} \rightarrow \widehat{Heap} \times \widehat{Cache}$$

Algorithm 45: $\widehat{\text{Assign}}(\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}, \mathfrak{e}_0.f = \mathfrak{e}_1, \widehat{\Gamma})$

```
1  (v̂0, σ̂0, Γ̂0) ← CacheEval^(σ̂, τ̂, ψ̂, O_current, e0, Γ̂)
2  if v̂0 = Bottom then
3  |  return (σ̂0, Γ̂0)
4  else if v̂0 = {O_current} then
5  |  (v̂1, σ̂1, Γ̂1) ← CacheEval^(σ̂0, τ̂, ψ̂, O_current, e1, Γ̂0)
6  |  σ̂ ← σ̂1
7  |  ρ ← σ̂(O_current)
8  |  assert(isMutable(f) ∧ f ∈ dom(ρ))
9  |  ρ' ← ρ[f ↦ v̂1]
10 |  σ̂ ← σ̂[O_current ↦ ρ']
11 |  return (σ̂, Γ̂1)
12 else
13 |  report_warning(Violating initialization-time irrelevance!)
14 |  return (Bottom, σ̂0, Γ̂0)
```

Algorithm 46: $\widehat{\text{Assign}}(\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}, p = \mathfrak{e}, \widehat{\Gamma})$

```
1  if ψ̂ = O_current ∨ ψ̂ = (_, O_current) then
2  |  ρ ← σ̂(ψ̂)
3  |  assert(isMutable(p) ∧ p ∈ dom(ρ))
4  |  (v̂', σ̂', Γ̂') ← CacheEval^(σ̂, τ̂, ψ̂, O_current, e, Γ̂)
5  |  ρ' ← ρ[p ↦ ρ(p) ∪ v̂']
6  |  σ̂' ← σ̂'[ψ̂ ↦ ρ']
7  |  return (σ̂', Γ̂')
8  else
9  |  report_warning(Violating initialization-time irrelevance!)
10 |  return (Bottom, σ̂, Γ̂)
```

Figure 5.12: The assignment semantics in the abstract interpreter

that violations of Principle 5.2 are not very common in real-world Scala programs, and we will empirically evaluate the conjecture in Chapter 7.

Chapter 6

Implementing the Global Object Initialization Checker for Scala

The previous chapters describe the theoretical foundation of the global object initialization checker as an abstract interpreter in the initialization calculus. This chapter focuses on implementing the global object initialization checker in the Scala compiler. Section 6.1 discusses the goal of producing explainable error messages for programmers. The global object initialization checker needs to model more Scala features in order to achieve this, so Sections 6.2 to 6.5 describe how to extend the initialization calculus with scalar values, arrays, closures, and inner classes and how to abstractly interpret them.

6.1 Motivation and goals

In the previous chapters, the global object initialization checker is presented as an isolated abstract interpreter on an initialization calculus. It is beneficial to integrate the global object initialization checker into the Scala compiler so that Scala developers can receive warnings about global object initialization when compiling their projects. Furthermore, the Scala compiler provides the auxiliary information required in the abstract interpreter, such as BodyOf.

The Scala compiler consists of three phases. The front-end constructs and types the syntax tree of the Scala program. The middle-end transforms the syntax tree by desugaring some Scala features into equivalent lower-level constructs. The backend then generates the target code in the run-time environment, such as JVM bytecode, based on the syntax tree.

The initialization calculus in the previous chapters aims to represent Scala programs with few native constructs that only include classes, global objects, and their related members, so it should align with the syntax trees at the later stages of the middle-end, after functional features are transformed into equivalent classes and global objects. However, the global object initialization checker will generate warnings showing the interpretation traces on the syntax trees that are different from the source program. This reduces the *explainability* of the warnings, and programmers cannot effectively debug the initialization errors based on the error messages.

In order to produce explainable warnings, the global object initialization checker will interpret the syntax tree right after the front-end. This indicates that the global object initialization checker should also interpret additional Scala features not included in the initialization calculus, and this chapter will describe the interpretation of some significant Scala features.

6.2 Scalars in abstract domain

Almost every programming language includes native constructs for scalar constants and string literals. In Scala, scakar constants are categorized into several class definitions, such as `Int`, `Long`, `Byte`, `Float`, `Double`, etc. All of these classes are defined in the Scala standard library without any fields, and most of the mathematical operations are represented as methods. For example, Figure 6.1 shows a snippet of the definition of class `Int` in the Scala standard library. Note that the `Int` class is abstract, so users cannot explicitly instantiate integers, and the only instances of `Int` are scalar constants and the results of scalar expressions. The implementations of mathematical operations are not defined explicitly, but the compiler will generate the corresponding operations in JVM bytecode.

In the abstract domain, we need to find a proper representation for scalar constants and the result of scalar expressions. An option is to treat each scalar constant as abstract instances of class `Int`, and to interpret all operations on scalar constants as method invocations. This follows the definitions in the standard library, but the global object initialization checker cannot find the body of the arithmetic operations since they are not defined. A better option is to treat each scalar constant as `Bottom` in order to skip all operations on them, as illustrated in Figure 6.2.

```
final abstract class Int private extends AnyVal {
  def toByte: Byte
  def toShort: Short
  def toChar: Char
  def toInt: Int
  def toLong: Long
  def toFloat: Float
  def toDouble: Double
  def ==(x: Int): Boolean
  def <(x: Int): Boolean
  def +(x: Int): Int
  def +(x: Float): Float
}
```

Figure 6.1: Definition of class `Int` in Scala 2 library

Algorithm 47: $\widehat{\textsc{CacheEval}}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), 1, \widehat{\Gamma})$

1 return $(\texttt{Bottom}, \widehat{\sigma}, \widehat{\Gamma})$

Figure 6.2: The abstract semantics of scalar constants

```
1   final class Array[T](_length: Int) extends java.io.Serializable with
        java.lang.Cloneable {
2       def length: Int
3       /** ‘xs(i)‘ is a shorthand for ‘xs.apply(i)‘. */
4       def apply(i: Int): T
5       /** ‘xs(i) = x‘ is a shorthand for ‘xs.update(i, x)‘. */
6       def update(i: Int, x: T): Unit
7   }
```

Figure 6.3: Definition of `Array` class in Scala

6.3 Array values in abstract domain

Apart from scalar constants, arrays in Scala are also represented by a class with undefined operations. All arrays in Scala will be compiled to native code in the runtime system that allocates memory for the elements. The `Array` class in Scala also specifies methods that can be called on Scala arrays but leaves them undefined, as illustrated in Figure 6.3. Moreover, arrays are the underlying representation for almost all containers defined in the Scala standard library, so almost every program that uses the standard library will allocate arrays at some point. Therefore, the abstract interpreter should also treat arrays specifically.

There are several concerns that relate to global object initialization when modeling arrays. Firstly, reading elements from arrays may cause unsafe global object initialization later. For example, in the program in Figure 6.4, the first entry of the array is allocated with an instance of `Box`, and then accessed as the receiver of the call to `foo`, which then accesses the uninitialized field `f`. This example shows that the contents in an array must be recorded and any accesses to the array must return an over-approximation of the contents in the abstract interpreter. Secondly, a non-terminating program may concretely allocate an infinite number of arrays, as illustrated in Figure 6.5. Therefore, arrays must be finitized into the finite abstract heap in the global object initialization checker. Finally, arrays encapsulate mutable states, so to maintain initialization-time irrelevance, each object can only read or update the arrays owned by itself.

Figure 6.6 extends the initialization calculus with arrays and their operations. Now the initialization calculus corresponds to a subset of syntax trees constructed by the Scala front-end. Since arrays need to record the state of all elements, they can be represented by a special form of abstract instances. There are multiple ways to categorize all abstract

```scala
class Box {
    def foo(): Int = O.f
}
object O {
    val array = new Array[Box](2)
    array(0) = new Box()
    val f = array(0).foo()
}
```

Figure 6.4: Accessing uninitialized field through array access

```scala
object O {
    var arr = new Array[C](2)
    val f = foo()
    def foo() = {
        arr = new Array[C](2)
        arr(0) = foo()
        arr(0)
    }
}
```

Figure 6.5: A program allocating infinite arrays at run-time

Initialization calculus with arrays

$$\begin{array}{lll} \mathfrak{e} ::= & \cdots \mid \texttt{new Array}[C]\texttt{(}\mathfrak{e}\texttt{)} \mid \mathfrak{e}\texttt{(}\mathfrak{e}\texttt{)} \in & \textit{Expression} \\ \mathbb{A} ::= & \mathfrak{e}\texttt{.}f \texttt{ = } \mathfrak{e} \mid p \texttt{ = } \mathfrak{e} \mid \mathfrak{e}\texttt{(}\mathfrak{e}\texttt{)} \texttt{ = } \mathfrak{e} \in & \textit{Assignment} \end{array}$$

Abstract domain with array instances

$$\begin{array}{lrr} & \widehat{ArrayAddr} ::= & \texttt{Array}[ClassName \times ObjectName] \\ \widehat{v} \in & \widehat{Value} ::= & P(\widehat{Addr} \cup \widehat{ArrayAddr}) \cup \{\texttt{Bottom}\} \\ & \widehat{ArrayInstance} ::= & \widehat{Value} \\ \widehat{\sigma} \in & \widehat{Heap} ::= & \widehat{Addr} \cup \widehat{ArrayAddr} \to \widehat{Instance} \cup \widehat{ArrayInstance} \end{array}$$

Figure 6.6: The syntax and the abstract domain with arrays

arrays into finite abstract addresses. Figure 6.6 also illustrates one specific design. The address of an abstract array instance is determined by the type of the elements provided in the allocation site and the global object owner. This means that all arrays of the same element type and owned by the same object share the same address, and the total array addresses are finite. In addition to abstract class instances, the heap also maps array addresses to abstract array instances. Finally, functions in Figure 6.7 illustrate how the abstract interpreter models array creations, array accesses, and array updates. For array creations, the corresponding array address is determined by C in the allocation site and $O_{current}$. For array accesses and array updates, the checker ensures that the owner of every possible array in the receiver matches $O_{current}$, then returns or updates the combination of all element values in each array address. This ensures that an array access will always return an over-approximation of the concrete element value.

6.4 Function values in abstract domain

Scala incorporates both the object-oriented paradigm and the functional programming paradigm, and the initialization calculus in the previous chapters only includes class and

Algorithm 48: $\widehat{\text{CacheEval}}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), \texttt{new Array[}C\texttt{](}\mathfrak{e}\texttt{)}, \widehat{\Gamma})$

1 $(_, \widehat{\sigma}', \widehat{\Gamma}') \leftarrow \widehat{\text{CacheEval}}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), \mathfrak{e}, \widehat{\Gamma})$
2 **if** *Array[*$(C, O_{current})$*]* $\notin dom(\widehat{\sigma}')$ **then**
3 | $\widehat{\sigma}' \leftarrow \widehat{\sigma}' \cup [\texttt{Array[}(C, O_{current})\texttt{]} \mapsto \{\}]$
4 return $(\{\texttt{Array[}(C, O_{current})\texttt{]}\}, \widehat{\sigma}', \widehat{\Gamma}')$

Algorithm 49: $\widehat{\text{CacheEval}}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), \mathfrak{e}_1\texttt{(}\mathfrak{e}_2\texttt{)}, \widehat{\Gamma})$

1 $(\widehat{v}, \widehat{\sigma}_1, \widehat{\Gamma}_1) \leftarrow \widehat{\text{CacheEval}}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), \mathfrak{e}_1, \widehat{\Gamma})$
2 $(_, \widehat{\sigma}_2, \widehat{\Gamma}_2) \leftarrow \widehat{\text{CacheEval}}((\widehat{\sigma}_1, \widehat{\tau}, \widehat{\psi}, O_{current}), \mathfrak{e}_2, \widehat{\Gamma}_1)$
3 $\widehat{v}' \leftarrow \{\}$
4 **foreach** $\nu \in \widehat{v}$ **do**
5 | $assert(\nu \in \widehat{ArrayAddress})$
6 | $\texttt{Array[}(C, O)\texttt{]} \leftarrow \nu$
7 | **if** $O_{current} \neq O$ **then**
8 | | *report_warning*(Violating initialization-time irrelevance!)
9 | | return $(\texttt{Bottom}, \widehat{\sigma}_2, \widehat{\Gamma}_2)$
10 | $\widehat{v}'' \leftarrow \widehat{\sigma}_2(\nu)$
11 | $\widehat{v}' \leftarrow \widehat{v}' \cup \widehat{v}''$
12 return $(\widehat{v}', \widehat{\sigma}_2, \widehat{\Gamma}_2)$

Algorithm 50: $\widehat{\text{Assign}}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), \mathfrak{e}_1\texttt{(}\mathfrak{e}_2\texttt{)=}\mathfrak{e}_3, \widehat{\Gamma})$

1 $(\widehat{v}, \widehat{\sigma}_1, \widehat{\Gamma}_1) \leftarrow \widehat{\text{CacheEval}}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), \mathfrak{e}_1, \widehat{\Gamma})$
2 $(_, \widehat{\sigma}_2, \widehat{\Gamma}_2) \leftarrow \widehat{\text{CacheEval}}((\widehat{\sigma}_1, \widehat{\tau}, \widehat{\psi}, O_{current}), \mathfrak{e}_2, \widehat{\Gamma}_1)$
3 $(\widehat{v}', \widehat{\sigma}_3, \widehat{\Gamma}_3) \leftarrow \widehat{\text{CacheEval}}((\widehat{\sigma}_2, \widehat{\tau}, \widehat{\psi}, O_{current}), \mathfrak{e}_3, \widehat{\Gamma}_2)$
4 **foreach** $\nu \in \widehat{v}$ **do**
5 | $assert(\nu \in \widehat{ArrayAddress})$
6 | $\texttt{Array[}(C, O)\texttt{]} \leftarrow \nu$
7 | **if** $O_{current} \neq O$ **then**
8 | | *report_warning*(Violating initialization-time irrelevance!)
9 | | return $(\widehat{\sigma}_3, \widehat{\Gamma}_3)$
10 | $\widehat{\sigma}_3 \leftarrow \widehat{\sigma}_3[\nu \mapsto \widehat{\sigma}_3(\nu) \cup \widehat{v}']$
11 return $(\widehat{\sigma}_3, \widehat{\Gamma}_3)$

Figure 6.7: The abstract interpretation of arrays and array operations

object constructs in the object-oriented paradigm. Scala also supports the foundational feature of the functional programming paradigm: creating first-class function values, also known as closures. The syntax of creating closures is of the form `(`*paramSignature*`) =>` *body*, and closures can be invoked with provided arguments just like methods, but without the receiver. The middle-end of the Scala compiler transforms closures to other constructs, such as instances that belong to special classes that define function values. All such classes have a method `apply`, so the syntax of invoking a closure f, $f(\overline{\mathfrak{e}})$, will be transformed to f`.apply`$(\overline{\mathfrak{e}})$. However, the stage of transforming closures happens late in the middle-end, and the initialization checker is placed at an earlier stage to produce explainable warnings. Therefore, it is necessary to include representations of closures in the abstract domain.

Based on the semantics of the lambda calculus, a closure needs to record the code of its body as well as all of the context of its creation point. The context would be used to find all variables that are accessible to the closure, and the closure body can refer to any such variables. In the initialization calculus, the closure body should be able to refer to the parameters of the instance at its creation point, as well as the parameters of the method at its creation point. In the abstract domain, the context of the closure can include the components of the abstract configuration at its creation point, specifically $\widehat{\tau}$ and $\widehat{\psi}$. It does not need to include $\widehat{\sigma}$ and $\widehat{\Gamma}$ because the abstract heap and cache is a global state constantly changing, and the closure will be provided with the global heap and cache at the evaluation point. $O_{current}$ can then be used to determine and finitize the address of abstract function values. This is reflected in the abstract domain defined in Figure 6.8, which extends the initialization calculus with the closure syntax and extends the abstract domain with closure addresses and contexts. Note that one closure address may combine the contexts of multiple closures. Finally, Figure 6.9 illustrates the abstract interpretation of closure creations and invocations. For closure creations, the abstract interpreter combines the context of the fresh closure into its corresponding address. For closure invocations, the abstract interpreter separately invokes all closure contexts in the corresponding closure address, provided that the number of arguments matches the number of parameters of the closure. Figure 6.9 assumes that paramsOf and BodyOf are extended for closure trees.

With the addition of function values, we have modeled the core functional programming feature in Scala in the abstract domain. Other functional programming features, such as pattern matching, can be directly modeled using the abstract domain values and following the specification, so their implementations are omitted here.

Syntax of closure creation and invocation

$$\mathfrak{e} ::= \cdots \mid \texttt{(}\overline{Id}\texttt{)} \texttt{ => } \mathfrak{e} \mid \mathfrak{e}\texttt{(}\overline{\mathfrak{e}}\texttt{)}$$

Abstract domain with closure values

$$\begin{aligned}
& \widehat{ClosureAddr} ::= && \texttt{Fun[}ObjectName\texttt{]} \\
\widehat{v} \in \quad & \widehat{Value} ::= && P(\widehat{Addr} \cup \widehat{ClosureAddr}) \cup \{\texttt{Bottom}\} \\
& \widehat{ClosureContext} ::= && P(\widehat{Frame} \times \widehat{Addr} \times Tree) \\
\widehat{\sigma} \in \quad & \widehat{Heap} ::= && \widehat{Addr} \cup \widehat{ClosureAddr} \to \widehat{Instance} \cup \widehat{ClosureContext}
\end{aligned}$$

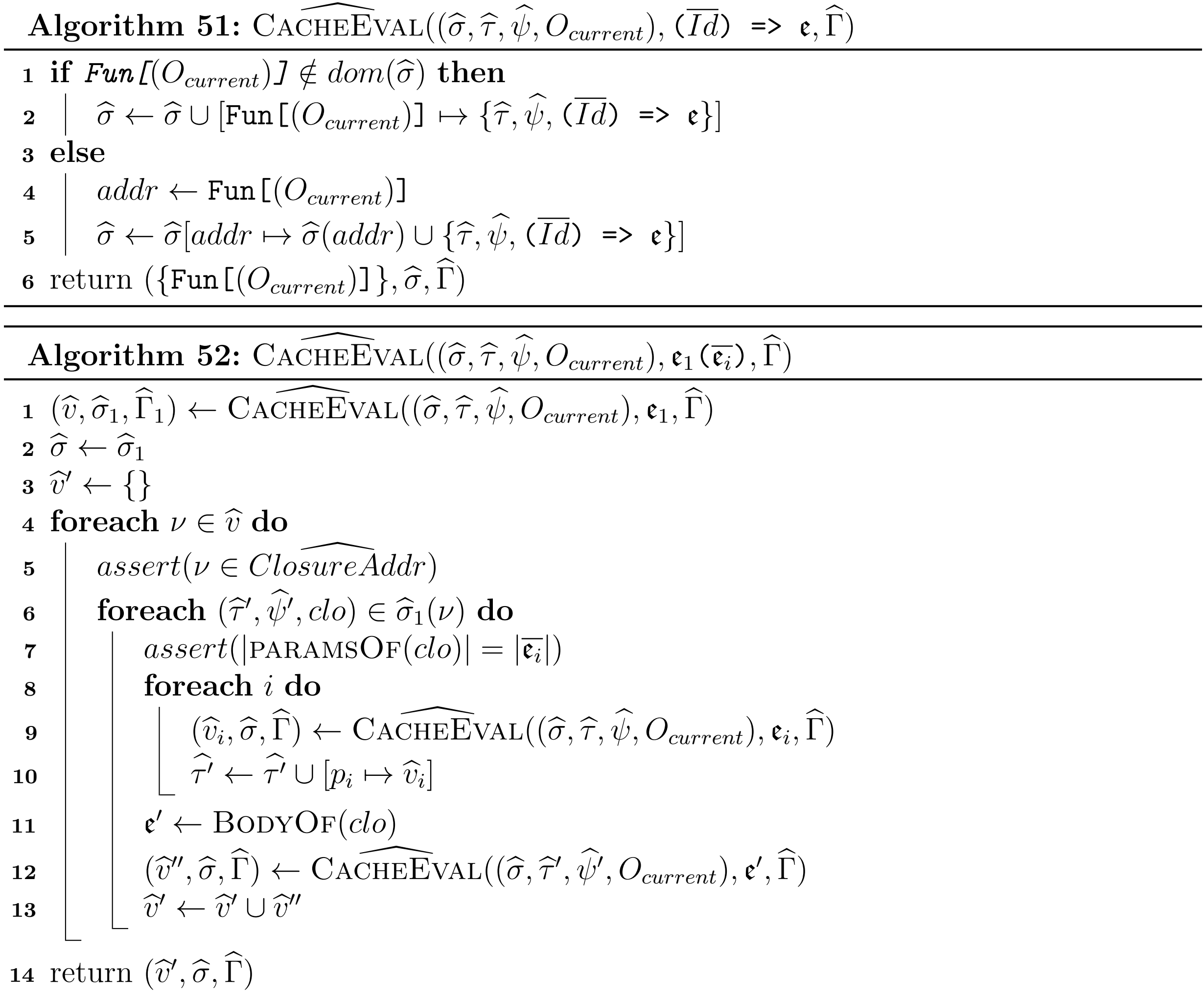
Figure 6.8: The abstract domain and semantics extended with closure values

6.5 Inner classes

Apart from parent-child relationships, classes in Scala also form nesting relationships, which means classes can be nested in the templates of other classes or global objects. This feature is present in the syntax tree after front-end but not in the initialization calculus. Figure 6.10 extends the initialization calculus with nested classes in both class templates and object templates. Note that global objects cannot be nested in other templates.

Nested classes provide a new form of class reference, $\mathfrak{e}.C$. This form of reference can be used when referring to a parent class or when referring to the class in `new` expressions. This reflects that each instance of class C nested in D must be accompanied by an outer instance of class D or a child class of D. The outer instance is implicitly passed to the instantiation of the inner instance as an argument. For example, Figure 6.11 shows that an object `O` creates an instance `a` of class `A`, and the outer instance of `a` is `O` itself. Similarly, object `O` then creates an instance `b` of class `B`, whose outer instance is `O.a`. The initialization process of `O2` is exactly equivalent to that of `O`, but all nested classes become top-level classes, and the outer instances are explicitly passed as parameters.

Since the outer instance is passed to the instantiation of the inner instance, the inner instance can access the parameters of the outer instance. In the previous example in Figure



Algorithm 51: $\widehat{\textsc{CacheEval}}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), (\overline{Id})\ \texttt{=>}\ \mathfrak{e}, \widehat{\Gamma})$

1 **if** $\texttt{Fun[}(O_{current})\texttt{]} \notin dom(\widehat{\sigma})$ **then**

2 $\quad \widehat{\sigma} \leftarrow \widehat{\sigma} \cup [\texttt{Fun[}(O_{current})\texttt{]} \mapsto \{\widehat{\tau}, \widehat{\psi}, (\overline{Id})\ \texttt{=>}\ \mathfrak{e}\}]$

3 **else**

4 $\quad addr \leftarrow \texttt{Fun[}(O_{current})\texttt{]}$

5 $\quad \widehat{\sigma} \leftarrow \widehat{\sigma}[addr \mapsto \widehat{\sigma}(addr) \cup \{\widehat{\tau}, \widehat{\psi}, (\overline{Id})\ \texttt{=>}\ \mathfrak{e}\}]$

6 return $(\{\texttt{Fun[}(O_{current})\texttt{]}\}, \widehat{\sigma}, \widehat{\Gamma})$

Algorithm 52: $\widehat{\textsc{CacheEval}}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), \mathfrak{e}_1\texttt{(}\overline{\mathfrak{e}_i}\texttt{)}, \widehat{\Gamma})$

1 $(\widehat{v}, \widehat{\sigma}_1, \widehat{\Gamma}_1) \leftarrow \widehat{\textsc{CacheEval}}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), \mathfrak{e}_1, \widehat{\Gamma})$

2 $\widehat{\sigma} \leftarrow \widehat{\sigma}_1$

3 $\widehat{v}' \leftarrow \{\}$

4 **foreach** $\nu \in \widehat{v}$ **do**

5 $\quad assert(\nu \in \widehat{ClosureAddr})$

6 $\quad$ **foreach** $(\widehat{\tau}', \widehat{\psi}', clo) \in \widehat{\sigma}_1(\nu)$ **do**

7 $\quad\quad assert(|\textsc{paramsOf}(clo)| = |\overline{\mathfrak{e}_i}|)$

8 $\quad\quad$ **foreach** i **do**

9 $\quad\quad\quad (\widehat{v}_i, \widehat{\sigma}, \widehat{\Gamma}) \leftarrow \widehat{\textsc{CacheEval}}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}), \mathfrak{e}_i, \widehat{\Gamma})$

10 $\quad\quad\quad \widehat{\tau}' \leftarrow \widehat{\tau}' \cup [p_i \mapsto \widehat{v}_i]$

11 $\quad\quad \mathfrak{e}' \leftarrow \textsc{BodyOf}(clo)$

12 $\quad\quad (\widehat{v}'', \widehat{\sigma}, \widehat{\Gamma}) \leftarrow \widehat{\textsc{CacheEval}}((\widehat{\sigma}, \widehat{\tau}', \widehat{\psi}', O_{current}), \mathfrak{e}', \widehat{\Gamma})$

13 $\quad\quad \widehat{v}' \leftarrow \widehat{v}' \cup \widehat{v}''$

14 return $(\widehat{v}', \widehat{\sigma}, \widehat{\Gamma})$

Figure 6.9: The abstract interpretation of closure creations and invocations

$$
\begin{array}{rrr}
\mathbb{P} ::= & \overline{\mathbb{C}}\ \overline{\mathbb{O}}\ \mathbb{O}_{main} \in & \mathit{Program} \\
\mathbb{C} ::= & \texttt{class}\ C(\overline{\mathbb{CP}})\ \texttt{extends}\ \mathbb{PC}(\overline{\mathfrak{e}})\ \{\ \overline{\mathbb{C}}\ \overline{\mathbb{M}}\ \} \in & \mathit{ClassTemplate} \\
\mathbb{CP} ::= & \texttt{val}\ p \mid \texttt{var}\ p \in & \mathit{ClassParameter} \\
\mathbb{O} ::= & \texttt{object}\ O\ \texttt{extends}\ \mathbb{PC}(\overline{\mathfrak{e}})\ \{\ \overline{\mathbb{C}}\ \overline{\mathbb{F}}\ \overline{\mathbb{M}}\ \} \in & \mathit{ObjectTemplate} \\
\mathbb{PC} ::= & \texttt{Object} \mid C \mid \mathfrak{e}.C \in & \mathit{ParentClassName} \\
\mathfrak{e} ::= & \cdots \mid \texttt{new}\ \mathfrak{e}.C(\overline{\mathfrak{e}}) \mid C\texttt{.this} \in & \mathit{Expression}
\end{array}
$$

Figure 6.10: Initialization calculus with nested class

```
1   object O {
2     class A(val paramA) {
3       class B {}
4     }
5     val a = new O.A(5)
6     val b = new O.a.B()
7   }
8
9   object O2 {
10    val a = new A(O, 5)
11    val b = new B(O.a)
12  }
13
14  class A2(val outerA, val paramA) {}
15  class B2(val outerB) {}
```

Figure 6.11: Initializing inner class instances

```scala
object O2 {
    val a = new A(O, 5)
    val b = new B(O.a)
    val f = b.getOuterParamA()
}

class A2(val outerA, val paramA) {
    def getParamA() = paramA
}
class B2(val outerB) {
    def getOuterParamA() = outerB.getParamA()
}

object O {
    class A(val paramA) {
        def getParamA() = paramA
        class B {
            def getOuterParamA() = A.this.getParamA()
        }
    }
    val a = new O.A(5)
    val b = new O.a.B()
    val f = b.getOuterParamA()
}
```

Figure 6.12: Accessing the outer instance in Scala

6.11, a programmer can let the instance of `B2` access the parameter `paramA` of `A2`, which is illustrated in Figure 6.12. However, when `A` and `B` are nested, then `outerB` becomes implicit. Programmers can instead access the outer instance of `b` by the expression `A.this`. This is valid because `B` is nested in `A`, so the instance of `b` must have an outer instance of `A` or a child class of `A`. `A.this` is also valid if `B` is nested in `A` separated by multiple nested levels, or using the notation $B \xrightarrow{outer} A$ defined in Figure 6.13. In this case, the instance of `b` must find an outer instance of `A` or a child class of `A` by searching through the chain of outer instances.

In order to model the semantics of the constructs related to inner classes, it is natural to extend the concrete domain in Chapter 2 so that every concrete instance also records the value of its direct outer instance. For convenience, we assume that the direct outer instance

$$\boxed{B \xrightarrow{direct_outer} A}$$

$$\textsc{DirectOuter1}\quad \frac{B \text{ is defined in the template of } A}{B \xrightarrow{direct_outer} A}$$

$$\boxed{B \xrightarrow{outer} A}$$

$$\textsc{Outer1}\quad \frac{B \xrightarrow{direct_outer} A}{B \xrightarrow{outer} A} \qquad \textsc{Outer2}\quad \frac{B \xrightarrow{direct_outer} A \qquad C \xrightarrow{outer} B}{C \xrightarrow{outer} A}$$

Figure 6.13: The relation between outer and inner classes

is recorded in the parameter value map in every instance under the special parameter name `outer`. This would lead to the design of interpreting C`.this` as illustrated in Figure 6.14, which iterates through all classes and outer instances in the outer class chain starting from C'/O until C.

However, the combination of parent classes and outer classes may create complex access patterns. Consider the program in Figure 6.15. Class `D` is nested in `C`, but `E` extends `D` with the outer instance `c`. This also indicates that all instances of `E` inherit the method `foo`. When `foo` is called on an instance of `E`, the interpretation of `C.this` will fail based on Algorithm 41 because `C` is not an outer class of `E`. As a solution, each concrete instance should not only record the direct outer instance of its class tag, but also the direct outer instances of all its ancestor classes. We assume that the direct outer instance of class `C` is associated with the special parameter name `outerOfC`. When evaluating a tree of the form C`.this`, the semantics should check that `C` encloses the class or object template where C`.this` exists textually. Suppose C`.this` exists in the template of `D`, then the current instance must be of class `D` or a child class of `D`, and it will store the direct outer instance of `D`. Figure 6.16 then formally defines the correct interpretation of C`.this`. Note that the configuration now includes another class or object name that indicates the template in which the tree exists.

Figures 6.17 and 6.18 then introduce the interpretation of other trees related to the outer and inner classes.

Algorithm 53: $\textsc{Eval}((\sigma, \tau, \psi, O_{current}), C\texttt{.this})$

1 $(C'/O, \rho) \leftarrow \sigma(\psi)$

2 $assert(C'/O \xrightarrow{outer} C)$

3 $C_{current} \leftarrow C'/O$

4 $v \leftarrow \psi$

5 **while** $C_{current} \neq C$ **do**

6 $\quad (_, \rho) \leftarrow \sigma(v)$

7 $\quad v \leftarrow \rho(\texttt{outer})$

8 $\quad C_{current} \leftarrow \textsc{DirectOuterOf}(C_{current})$

9 return (v, σ)

Figure 6.14: The first attempt of interpreting $C\texttt{.this}$

```
1    class C {
2        class D(param) {
3            def foo() = C.this
4        }
5    }
6
7    class E(c) extends c.D {}
8
9    object O {
10       val f1 = new E(new C())
11       val f2 = f1.foo()
12   }
```

Figure 6.15: Accessing the outer instance of the parent class

$$\textsc{Eval} :: Conf \times ClassName \cup ObjectName \times Expression \to Value \times Heap$$

Algorithm 54: $\textsc{Eval}((\sigma, \tau, \psi, O_{current}), C_{current}, C\texttt{.this})$

1. $assert(C_{current} \xrightarrow{outer} C)$
2. $v \leftarrow \psi$
3. **while** $C_{current} \neq C$ **do**
4. $\quad (_, \rho) \leftarrow \sigma(v)$
5. $\quad v \leftarrow \rho(\texttt{outerOf}C_{current})$
6. $\quad C_{current} \leftarrow \textsc{DirectOuterOf}(C_{current})$
7. return (v, σ)

Figure 6.16: The correct interpretation of C.`this`

Inner class instantiation. The interpreter first evaluates the direct outer instance passed as $\mathfrak{e}_1$ in the `new` expressions. It evaluates to the value of the direct outer instance of the class tag C, so it is associated with `outerOf`C. The rest is the same as evaluating `new` expressions.

Initialization process of class and object templates. The parent class of object templates may specify the direct outer instance of the parent class. This is associated with the `outerOf`C, where C is the parent class. The same goes for class templates. Each object or class template also sets $C_{current}$ to its name when evaluating any tree in it.

The rest of the evaluation functions and initialization functions should be trivial to extend with $C_{current}$, so they are not shown here. Note that invoking a method needs to set $C_{current}$ to the defining class or object of the method.

Abstract semantics. In the abstract domain, each abstract instance combines the information of multiple concrete instances, so the information of direct outer instances of each class will also be combined. This affects the search process when interpreting C.`this`, because `outer` will return a set of values. Figure 6.19 then combines all possible outer instances of all possible values again, given that every possible value must eventually reach an outer instance of C. The result must be an over-approximation of the concrete outer instance. We omit the other abstract interpretation functions since they are naturally extended from their concrete counterparts.

Algorithm 55: $\textsc{Eval}((\sigma, \tau, \psi, O_{current}), C_{current}, \texttt{new } \mathfrak{e}_1\texttt{.}C\texttt{(}\overline{\mathfrak{e}_i}\texttt{)})$

1 $(v_{outer}, \sigma) \leftarrow \textsc{Eval}(\sigma, \tau, \psi, C_{current}, \mathfrak{e}_1)$
2 $\rho \leftarrow [\texttt{outerOf}C \mapsto v_{outer}]$
3 $\overline{p_i} \leftarrow \textsc{paramsOf}(C)$
4 **foreach** i **do**
5 $\quad (v_i, \sigma_i) \leftarrow \textsc{Eval}(\sigma, \tau, \psi, \mathfrak{e}_i)$
6 $\quad \sigma \leftarrow \sigma_i$
7 $\quad \rho \leftarrow \rho \cup [p_i \mapsto v_i]$
8 $\psi' \leftarrow (|dom(\sigma(O_{current}))|, O_{current})$
9 $\sigma \leftarrow \sigma[O_{current} \mapsto \sigma(O_{current})[\psi' \mapsto (C, \rho)]]$
10 $\sigma \leftarrow \textsc{initializeClassTemplate}(C, \sigma, O_{current}, \psi')$

Figure 6.17: The interpretation of instantiation of inner instances

In conclusion, inner classes can be naturally modeled in the concrete and abstract domain introduced before. The extended concrete and abstract interpreters with inner classes can detect unsafe initialization patterns related to inner classes accessing outer instances.

Algorithm 56: $\textsc{initializeObject}(\mathbb{O}, \sigma, \psi)$

1 `object` O `extends` $\mathfrak{e}_1.C(\overline{\mathfrak{e}_i})$ `{` $\overline{\mathbb{C}}\ \overline{\mathbb{F}_j}\ \overline{\mathbb{M}}$ `}` $\leftarrow \mathbb{O}$
2 $(v_{outer}, \sigma) \leftarrow \textsc{Eval}(\sigma, [], \psi, C_{current} = O, \mathfrak{e}_1)$
3 $(O, \rho) \leftarrow \sigma(O)(\psi)$
4 $\rho \leftarrow \rho \cup [\texttt{outerOf}C \mapsto v_{outer}]$
5 $\sigma \leftarrow \sigma[O \mapsto \sigma(O)[\psi \mapsto (O, \rho)]]$
6 $\overline{p_i} \leftarrow \textsc{paramsOf}(\mathbb{PC})$
7 **foreach** i **do**
8 $\quad (v_i, \sigma_i) \leftarrow \textsc{eval}(\sigma, [], \psi, C_{current} = O, \mathfrak{e}_i)$
9 $\quad (O, \rho) \leftarrow \sigma_i(\psi)$
10 $\quad \sigma \leftarrow \sigma_i[O \mapsto \sigma_i(O)[\psi \mapsto (O, \rho[p_i \mapsto v_i])]]$
11 $\sigma \leftarrow \textsc{initializeClassTemplate}(\sigma, \psi, \mathbb{PC}, O_{current})$
12 **foreach** j **do**
13 $\quad \sigma \leftarrow \textsc{initializeField}(\sigma, \psi, \mathbb{F}_j, O_{current})$
14 return σ

Algorithm 57: $\textsc{initializeClassTemplate}(C, \sigma, \psi, O_{current})$

1 **if** $C = \mathtt{Object}$ **then**
2 $\quad$ return σ
3 `class` $C(\overline{p_i})$ `extends` $\mathfrak{e}_1.D(\overline{\mathfrak{e}_j})$ `{` $\overline{\mathbb{C}}\ \overline{\mathbb{M}}$ `}` $\leftarrow \textsc{TemplateOf}(C)$
4 $(v_{outer}, \sigma) \leftarrow \textsc{Eval}(\sigma, [], \psi, C_{current} = C, \mathfrak{e}_1)$
5 $(C'/O, \rho) \leftarrow \sigma(O_{current})(\psi)$
6 $\rho \leftarrow \rho \cup [\texttt{outerOf}D \mapsto v_{outer}]$
7 $\sigma \leftarrow \sigma[O_{current} \mapsto \sigma(O_{current})[\psi \mapsto (C'/O, \rho)]]$
8 $\overline{p_j} \leftarrow \textsc{paramsOf}(\mathbb{PC})$
9 **foreach** j **do**
10 $\quad (v_j, \sigma_j) \leftarrow \textsc{Eval}(\sigma, \{\}, \psi, \mathfrak{e}_j)$
11 $\quad (C'/O, \rho) \leftarrow \sigma_j(O_{current})(\psi)$
12 $\quad \sigma \leftarrow \sigma_j[O_{current} \mapsto \sigma(O_{current})[\psi \mapsto (C'/O, \rho[p_j \mapsto v_j])]]$
13 return $\textsc{initializeClassTemplate}(D, \sigma, \psi, O_{current})$

Figure 6.18: The initialization of templates related to outer and inner classes

Algorithm 58: $\widehat{\text{CACHEE}}\text{VAL}((\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current}, C_{current}), C\texttt{.this}, \widehat{\Gamma})$

1 $\widehat{v} \leftarrow \{\widehat{\psi}\}$
2 **while** $C_{current} \neq C$ **do**
3 $\quad assert(\forall \nu \in \widehat{v}, \texttt{outerOf}C_{current} \in dom(\widehat{\sigma}(\nu)))$
4 $\quad \widehat{v} \leftarrow \bigcup_{\nu \in \widehat{v}} \widehat{\sigma}(\nu)(\texttt{outerOf}C_{current})$
5 $\quad C_{current} \leftarrow \text{DIRECTOUTEROF}(C_{current})$
6 return $(\widehat{v}, \widehat{\sigma}, \widehat{\Gamma})$

Figure 6.19: The abstract interpretation of $C\texttt{.this}$

Chapter 7

Evaluating the Scala Global Object Initialization Checker

Based on the design in previous chapters, we have implemented a global object initialization checker and incorporated it into the Scala compiler, Dotty, as an initialization checking pass. The initialization checker can then be evaluated by compiling Scala projects using Dotty. Section 7.1 describes a set of open-source Scala projects that is used as the test suite to evaluate the effectiveness and efficiency of the abstract interpreter. Section 7.2 presents the statistics of the initialization checker when using Dotty to compile these projects. Section 7.3 conducts a case study on the true positive warnings reported by the checker and demonstrates their negative impact on the robustness of the program.

7.1 Test suite

The Dotty compiler project maintains a list of 52 open-source projects written in Scala that serves as a test suite for the compiler. It is vital to ensure the initialization safety of these projects due to the following two aspects of these projects:

- Complexity. Most of these open-source projects are large in scale with complex software design patterns. This indicates a large number of global objects and classes in these projects and complex initialization and access patterns between them. Therefore, these projects are more susceptible to unsafe global object initialization patterns. This provides opportunities for the initialization checker to detect and report them when compiling these projects.

- Popularity. Many of these open-source projects are widely used tools for the Scala community. For example, the list of projects includes munit, a Scala testing library, and Scala parallel collections, which are collections specialized for parallel execution. Therefore, any errors in these projects may potentially influence a large number of Scala projects that depend on them.

In addition to open-source projects, the Scala standard library is also written in Scala, and almost every Scala project, including every project in the test suite, initializes objects from the standard library. Therefore, the initialization checker should also ensure the initialization safety of the Scala standard library. In order to achieve this, the complete source of the standard library is included in the process of compiling every external project. The Scala source files of the standard library are preprocessed and presented in compressed TASTY format [1], and the compiler decompiles them back to AST trees at the end of the front-end transformation. The initialization checker will then have the complete input of the standard library and analyze the collections used by the external project.

7.2 Statistics

We ran the Dotty compiler with the global object initialization checker on all the projects in the test suite, as well as the bootstrapped Dotty compiler itself. Among them, the initialization checker reported warnings on 8 projects, and Table 7.1 summarizes the relevant statistics on the global object initialization checker during the compilation of these 8 projects. The first column shows the number of global objects that are analyzed, and they cover every global object defined in the projects. The projects usually define 100 to 1000 global objects, with two projects having more than 1000 global objects, reflecting their relatively large scales. The second column shows the total number of addresses of the abstract heap when the initialization checker finished. The size of the heap should always be larger than the total number of global objects since it includes abstract instances of both global objects and class instances. The table shows that the size of the heap is usually 1.5-to-3 times of the number of global objects, reflecting that class instances are frequently instantiated during the abstract interpretation. The third and fourth columns show the number of warnings generated by the initialization checker. The third column combines the numbers of two types of warnings: accessing uninitialized fields and cyclic initialization, while the fourth column shows the number of warnings related to violating initialization-time irrelevance. After excluding several duplicate warnings, we separated true positives from false positives for both kinds of warnings, as illustrated in Table 7.2. There are two

	Objects	Heap	W1	W2	Speed(s)
Dotty-bootstrapped	1691	2240	2	10	92
catsEffect3	638	1187	0	2	108
intent	96	152	1	0	26
parboiled2	369	1170	0	2	455
playJson	511	1158	3	0	134
scalaParallelCollections	279	729	1	0	148
scalaSTM	175	327	0	3	45
scissLucre	1266	2089	0	8	158

Table 7.1: Evaluation of the global object initialization checker on Scala open-source projects

warnings that we cannot verify because they are related to TASTY source files that we cannot obtain. The initialization checker does not produce too many warnings for reading uninitialized fields and cyclic initialization order, but some true positive warnings could lead to actual run-time errors, as illustrated in the next section. Table 7.2 also shows that violating initialization-time irrelevance by reading mutable fields owned by other objects is more frequent in real-world projects. Violating initialization-time irrelevance does not always lead to run-time errors, but it is a prerequisite of a local initialization checker, and the next section also illustrates the benefit of refactoring the code to maintain the principle of initialization-time irrelevance. Since the number of false positive warnings is low, it suggests that the precision of the initialization checker is satisfying for practical use under the current design of abstract domain. The final column in 7.1 shows the total time spent by the checker on each project by subtracting the time to compile the project with and without the checker. The performance of the initialization checker has substantial room for improvement, especially on parboiled2 which seems to include complex class instantiation patterns. The techniques of optimizing the performance of the abstract interpreter are discussed and left as future work in the next chapter.

7.3 Verification of warnings

We verified the validity of the warnings generated by the initialization checker in the test suites. This section conducts a case study on three warnings, one for each different type, and verifies that they are true positives and should be avoided.

The first warning is about accessing an uninitialized field in the Scala 2 standard li-

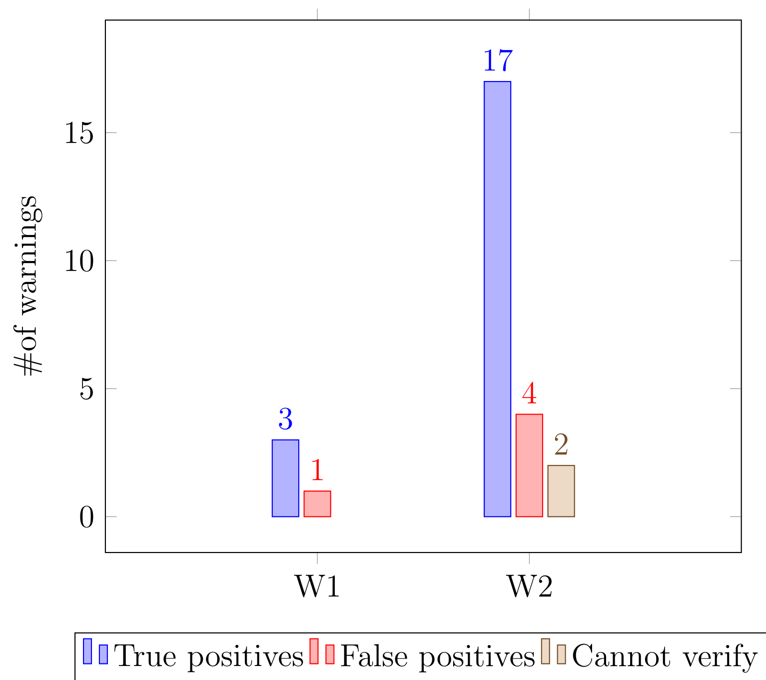


Table 7.2: Summary of distinct warnings in the test suite

```
private[concurrent] object Promise {
    private[this] final val Noop = new Transformation[Nothing,
        Nothing](Xform_noop, null, ExecutionContext.parasitic)

final class Transformation[-F, T] private[this] (
    private[this] final var _fun: Any => Any,
    private[this] final var _ec: ExecutionContext,
    private[this] final var _arg: Try[F],
    private[this] final val _xform: Int
) extends DefaultPromise[T]() with Callbacks[F] with Runnable with
    Batchable {
    final def this(xform: Int, f: _ => _, ec: ExecutionContext) =
        this(f.asInstanceOf[Any => Any], ec.prepare():
            @nowarn("cat=deprecation"), null, xform)
}

class DefaultPromise[T] private[this] (initial: AnyRef) extends
    AtomicReference[AnyRef](initial) with scala.concurrent.Promise[T]
    with scala.concurrent.Future[T] with (Try[T] => Unit) {
    final def this() = this(Noop: AnyRef)
}

}
```

Figure 7.1: Accessing uninitialized field in Scala 2 standard library

```scala
object untpd extends Trees.Instance[Untyped] with UntypedTreeInfo {...}

object Trees {
    class EmptyValDef[T <: Untyped] extends ValDef[T](
        nme.WILDCARD, genericEmptyTree[T], genericEmptyTree[T])(NoSource)
            with WithoutTypeOrPos[T] {
        myTpe = NoType.asInstanceOf[T]
        setMods(untpd.Modifiers(PrivateLocal))
    }
}
```

Figure 7.2: Cyclic initialization in the Dotty compiler

brary. Figure 7.1 shows the simplified snippet of code related to the warning. In this snippet of code, when initializing the field `Noop` in the object `Promise`, a class instance of `Transformation` is created, but it also triggers the constructor of its parent class `DefaultPromise` with parameter `Noop` again, which is not yet initialized and has value `null`. In this case, the value of `initial` in `DefaultPromise` will be `null`. `DefaultPromise` also inherits a `get` method that returns `initial`, which may expose an unexpected `null` to users and may lead to an exception if users deference it. Although all objects and classes listed here are private, it is still necessary to fix this bug to prevent any unexpected consequences.

The second example is about cyclic initialization in the Dotty compiler. Figure 7.2 shows the simplified snippet of code related to the warning. In this snippet of code, object `untpd` and object `Trees` refer to each other, so their initializations can never form a partial order. Since the initialization of global objects is guarded by locks, a deadlock can occur in this case when two threads initialize the two objects separately, as indicated by Figure 7.3.

The third example is about violating initialization-time irrelevance in the open-source Scala project parboiled2. Figure 7.4 shows the simplified snippet of code related to the warning. In this snippet of code, the object `Base64` initializes the mutable states `RFC2045` and `CUSTOM`. However, it also provides two methods to read and write these mutable states owned by itself, and another object `Base64Parsing` calling these methods is prohibited. In this case, the mutable fields can simply be rewritten to lazy fields to maintain the same semantics.

```scala


object Test {
  val createUntpd = new Runnable {
    def run = {
      val _ = dotty.tools.dotc.ast.untpd;
      Thread.sleep(1000);
    }
  }
  val createTrees = new Runnable {
    def run = {
      val _ = dotty.tools.dotc.ast.Trees;
      Thread.sleep(1000);
    }
  }
  val t1 = new Thread(createUntpd)
  val t2 = new Thread(createTrees)
  def main(args: Array[String]): Unit = {
    t1.start()
    t2.start()
    t1.join()
    t2.join()
  }
}
```

Figure 7.3: Deadlock caused by cyclic initialization of global objects

```
object Base64 {
  private var RFC2045: Base64 = _
  private var CUSTOM: Base64 = _

  def custom(): Base64 = {
    if (CUSTOM == null)
      CUSTOM = new
          Base64("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+-_")
    CUSTOM
  }

  def rfc2045(): Base64 = {
    if (RFC2045 == null)
      RFC2045 = new
          Base64("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=")
    RFC2045
  }
}

object Base64Parsing {
  val rfc2045Alphabet =
      CharPredicate(Base64.rfc2045().getAlphabet).asMaskBased
  val customAlphabet = CharPredicate(Base64.custom().getAlphabet).asMaskBased
}
```

Figure 7.4: Violation of initialization-time irrelevance in parboiled2

7.4 Summary

After incorporating the initialization checker into the Dotty compiler, the list of open-source Scala projects forms an ideal test suite for evaluating the initialization checker due to their large scale and complexity. Table 7.1 indicates that the initialization checker has covered the code triggered by the initialization of hundreds to thousands of global objects and class instances in each project. The initialization checker reports warnings for several projects, and Section 7.3 shows that some true positive warnings in Table 7.1 about accessing uninitialized fields and cyclic initialization could lead to run-time errors when exploited. Warnings about initialization-time irrelevance also indicate that the code could be refactored for better robustness. Since initialization errors occur in the Dotty compiler and the Scala 2 standard library, which is used by every Scala programmer, we believe that the global object initialization checker is necessary for checking large-scale projects before their release.

Chapter 8

Related Work and Future Work

8.1 Checking initialization safety

Existing languages. Unsafe initialization is prevalent in existing object-oriented languages and lacks universal treatment by compilers. On one hand, the standard compilers of Java and C++ do not offer any protection from reading uninitialized fields. On the other hand, Swift ensures that class properties are initialized before being used, but with complex restrictions. For example, Swift forbids users to call instance methods before all properties of the parent classes are initialized. To maintain safe initialization of instances, several theoretical initialization models were proposed with different balances between complexity and expressiveness.

Masked types. Qi and Myers proposed masked types that encode uninitialized fields of instances [22]. Each method signature will be extended with an effect signature that describes the masked type of the current instance before and after method invocation. However, encoding specific uninitialized fields requires extensive annotations in the program, which adds programming overhead. Many simple methods can be invoked in any initialization status, which requires typestate polymorphism. Therefore, supporting masked types would significantly refactor the type system in Scala.

Freedom model. Summers and Mueller proposed the freedom model that classifies instances into two categories: Under initialization (free) and transitively initialized (committed) [26]. Any chains of field selections starting from committed instances are safe. The

freedom model is light-weight and flow-insensitive, and can be extended with polymorphism. However, their work left an open challenge to model the initialized fields of free instances before transitive initialization.

Type-and-effect inference system. Liu proposed a type-and-effect inference system that further categorized instances into three categories: Partially initialized (cold), fully initialized (warm), and transitively initialized (hot) [13]. The initialization model ensures the principle of local reasoning: If every argument passed to a class template is hot, then the resulting instance must be hot. Liu et al. also implemented a class instance initialization checker in the Scala compiler as an abstract interpreter [15]. This thesis complements Liu's model by checking the initialization of global objects and sidestepping the initialization of class instances. The principle of local reasoning also inspires the design of local initialization of global objects to avoid whole-program analysis.

Initialization of static initializers. Global objects in Scala are based on the design of static initializers in earlier languages, which may cause deadlocks under unsafe initialization orders [2] [16]. Kozen and Stillerman described a static analysis method on Java bytecode to determine class initialization dependencies [10]. Hubert and Pichardie designed an analysis to compute the set of initialized static fields at each program point and prohibit reading uninitialized fields [7]. However, both analyses are whole-program analyses on Java bytecode due to the absence of initialization-time irrelevance. In contrast, our checker achieves local analysis of each object and produces explainable warnings at the early stage of compilation.

8.2 The abstract interpretation framework

Abstracting abstract machines. Cousot and Cousot formulated abstract interpretation that approximates concrete program semantics [3]. Since its proposal, the theory of abstract interpretation has been frequently applied when designing and verifying static analysis, and many papers proposed generic frameworks of designing abstract interpreters. Might described the conversion from a concrete small-step semantics to a sound abstract semantics, which often resembles the concrete counterpart [17]. Based on this approach, Van Horn and Might proposed abstract interpreters that approximated the small-step semantics of CESK machines [27]. The semantics of a CESK machine contains an environment that maps parameters to store addresses, a store that maps store addresses to closure values,

and a continuation stack that records the return location of each function call. Then the abstract domain of a CESK machine only needs to finitize store addresses and continuation stacks. The authors also showed that such abstract interpreters are equivalent to a family of environment analysis that approximates the possible arguments of each parameter in lambda calculus. The initialization semantics in this thesis is inspired by the semantics of CESK machines. The environments of the CESK machine correspond to frames in the JVM and the store of the CESK machine corresponds to the global heap of JVM. However, the initialization semantics in this thesis also defined abstract class instances and global objects and their addresses in the finite heap.

Abstracting definitional interpreters. A language can also formalize its big-step semantics presented as a definitional interpreter. This is often more concise and understandable than the small-step semantics. Moreover, Reynolds remarkably concluded that definitional interpreters inherently model the constructs of the defined language using the corresponding constructs of the defining language of the interpreter [23]. Darais et al. proposed the abstracting definitional interpreters framework as an extension to the abstracting abstract machines framework [4]. The framework includes the cache to terminate the evaluation of recursive function calls. The resulting abstract domain reflects Reynold's observation. It does not need to explicitly model the continuation stack of the CESK machine. Instead, the continuation stack is implicitly modeled by the stack of the defining language. The abstracting definitional interpreters framework was successfully applied to design analyses for existing languages such as JavaScript [5] and Haskell [28]. To our knowledge, the class instance initialization checker and the global object initialization checker are the first applications of abstract definitional interpreters in the initialization semantics of object-oriented languages.

8.3 Optimizing abstract interpreters

Abstract garbage collection. The efficiency of abstract interpreters impacts their practical uses. Specifically, the size of the heap may become the bottleneck of the checker when compiling large-scale projects because the heap is a part of the key in the cache. In the abstract initialization semantics in this thesis, the size of the heap monotonically increases, but there are optimizations to deallocate abstract instances in the heap without affecting soundness. One solution is abstract garbage collection, proposed by Might and Shivers [19], which periodically removes all abstract instances that are unreachable, similar to concrete garbage collection in the JVM. For example, we can define the set of reachable instances

```
class C { def foo() = 5 }

object O {
    def bar(p) = p.foo()
    val f = bar(new C())
}
```

Figure 8.1: Unreachable instances in the initialization calculus

at program point with abstract configuration $(\widehat{\sigma}, \widehat{\tau}, \widehat{\psi}, O_{current})$ to be the fixpoint of the transitively defined sets of addresses $\overline{\delta}_i$, where:

$$\delta_0 = \{O \in dom(\widehat{\sigma})\} \cup \bigcup_{p_i \in dom(\widehat{\tau})} \widehat{\tau}(p_i) \qquad \text{(root set)}$$

$$\delta_{i+1} = \delta_i \cup \bigcup_{\widehat{\psi} \in \delta_i} \bigcup_{f/p \in \widehat{\sigma}(\widehat{\psi})} \widehat{\sigma}(\widehat{\psi})(f/p)$$

In practice, we can easily implement a heap traversal algorithm to find all reachable and unreachable instances based on the previous definition. Each garbage collection process may remove numerous unreachable abstract instances from the heap. For example, the instance allocated on line 5 of Figure 8.1 will become unreachable after the call to `bar`. Removing unreachable instances from the heap will increase the performance and precision of the initialization checker. We leave the implementation and evaluation of the abstract garbage collector as future work.

Abstract reference counting and strong updates. Mutation is mostly modeled by weak updates in the initialization checker because multiple concrete instances are assigned to the same abstract address. Strong updates are more precise but unsound in this setting. Lhoták and Chung proposed to find abstract references that represent only one concrete instance, and updates on these references can be safely modeled by strong updates [11]. They defined such abstract addresses as singleton addresses, and they also concluded that analysis on finding singleton addresses is cheap to implement in a points-to analysis framework, which computes the possible points-to set of pointers in a constraint model. We believe that finding singleton addresses in the heap of the initialization checker is also easy because of the explicit bindings between the abstract address and the abstract instance. In

particular, abstract objects representing global objects should always be singletons. Might and Shivers also proposed an abstract domain for the lambda calculus with abstract reference counting [19]. Based on their design, we can propose a similar abstract domain for the initialization checker:

$$
\begin{aligned}
\hat{\sigma} \in \quad \widehat{Heap} &::= \widehat{Addr} \to \widehat{Instance} \\
\widehat{Instance} &::= ((FieldName \cup ParamName) \to \widehat{Value}) \times \widehat{Count} \\
\widehat{Count} &::= \{0, 1, \infty\}
\end{aligned}
$$

Recall that each abstract instance combines the information of multiple concrete instances. The new domain adds an abstract count of the number of concrete instances combined in each abstract instance. In order to finitize the domain, the set of counts must also be finite, so it only includes a count of 0, 1, or an unknown number of concrete instances (∞). Then, when a `new` expression instantiates a concrete class instance at abstract address $\hat{\psi}$, we increment the count of the abstract instance at $\hat{\psi}$. Incrementing the abstract count can be defined by:

$$
\begin{aligned}
inc(0) &= 1 \\
inc(1) &= \infty \\
inc(\infty) &= \infty
\end{aligned}
$$

Finally, when mutating the class parameters of an abstract instance with abstract count 1, it is safe to apply strong update. We leave the implementation and evaluation of abstract counting and strong updates as future work.

8.4 Tuning the precision of abstract interpreters

Contexts of abstract address allocator. The strategy of assigning abstract addresses of freshly allocated concrete instances can be customized. For every allocation function with finite range, the resulting abstract interpreter is proven to be terminating and sound [18]. However, different allocation strategies affect the performance and the precision of the resulting abstract interpreter. Many practical designs of static analyses separate abstract

entities through a finite history information that leads to the creation of the abstract entity. The historical information is often known as contexts, making such analysis context-sensitive. Gilray et al. proposed a unified abstract domain including the information that defines the context of the allocation strategy [6]. They also investigated common strategies when designing analyses for different programming paradigms and described them in the unified framework. These common strategies look at different historical information, such as:

- **Call sensitivity:** Looking at the past k call frames.
- **Argument sensitivity:** Looking at the underlying types of all arguments in the current frame.
- **Object sensitivity:** Looking at the allocation sites of the receiver of a method invocation, and the allocation sites of the ancestor receiver of the method that allocates the receiver of the current method, etc. Object sensitivity is invented by Milanova et al. [20] and formalized by Smaragdakis et al. [25] and are commonly adopted in points-to analysis.
- **Configuration sensitivity:** Looking at the summary of the configurations that lead to the current configuration, up to a given point.

The context adopted by the global object initialization checker in this thesis is reflected in the components of abstract addresses. The strategy of address allocation can be described as a combination of class-type sensitivity and owner-object sensitivity. We believe that all of the above strategies can be combined with the current strategy of the initialization checker when tuning the precision of the checker. However, it is unclear which strategy will achieve a sweet spot between performance and precision for initialization checking. We leave the implementation and evaluation of different allocation strategies for future work.

Regions. Every context loses precision in nature compared to concrete execution. If the checker produces a false-positive warning due to the loss of precision, the programmer should have an easy approach of fine-tuning the precision of the checker to avoid the warning. For example, the program in Figure 8.2 contains a false positive warning because the instances of `Box` on line 5 and line 6 share the same abstract address in the context of the checker, so their abstract instance combines the value of `p` in both instances. The programmer needs an easy way to avoid the warning without refactoring the program.

```scala
class A { def foo() = 5 }
class B extends A { def foo() = O.f3 }
class Box(p: A) { def bar() = p.foo() }
object O {
  val f1 = new Box(new A())
  val f2 = new Box(new B())
  val f3 = f1.bar() // false-positive warning: reading uninitialized
    field f3
}
```

Figure 8.2: A program with false positive warning

Syntax:

$$\mathfrak{e} ::= \quad \texttt{region} \; \{ \; \mathfrak{e} \; \}$$

Abstract domain:

$$\widehat{\psi} \in \quad \widehat{Addr} ::= \quad ((ClassName \times ObjectName) \cup ObjectName) \times Region$$

Figure 8.3: Scala expressions with regions

Liu et al. proposed to reverse the design of contexts in the initialization checker and let programmers use an explicit annotation to decide the context of abstract instances [14]. They added an annotation for Scala expressions in the form of *regions*, as illustrated in Figure 8.3. The addition of regions will not affect the concrete semantics. However, in the abstract semantics, each region annotation will be assigned with a unique name. When instantiating a class instance with a `new` expression, the name of the inner-most region of the `new` expression will also decide the abstract address. Figure 8.3 also shows the updates to abstract addresses, which add an extra component for regions. The annotation of regions lets programmers explicitly tune the precision of the checker. For example, the programmer can avoid the false-positive warning in Figure 8.2 by placing the two instances of `Box` in separate region annotations. This motivates a qualitative study on the explainability of regions, and we leave it for future work.

Chapter 9

Conclusion

The initialization process of global objects in Scala is vulnerable. This thesis identifies two kinds of run-time errors related to global object initialization: Null exceptions caused by reading uninitialized fields of global objects, and deadlocks between locks that guard the initialization of different global objects. Moreover, the safety of global object initialization may depend on its initialization point because global objects are initialized on demand.

This thesis then presents the foundation of a global object initialization checker for Scala. We propose an initialization calculus with global objects and classes and their essential members and define its concrete semantics. Based on the concrete semantics, we propose two run-time principles to avoid the run-time errors: Prohibiting reading uninitialized fields of global objects and maintaining an acyclic wait-for graph. Then we design the initialization checker as an abstract interpreter of the program based on the abstract initialization semantics. The abstract interpreter is designed to terminate on all programs and over-approximate the initialization processes in the program.

Next, in order to improve explainability of warnings, we modify the whole-program abstract interpreter to a local initialization checker that checks each global object individually. The local initialization checking for the immutable calculus is sound provided that the program satisfies initialization-time irrelevance of every object, which ensures that the initialization process of each global object is independent of its initialization point. When adding mutable states to initialization calculus, the initialization checker will verify initialization-time irrelevance by restricting the access of mutable states to the owner objects.

Finally, we integrate the global object initialization checker into the Scala compiler after modeling other Scala features during abstract interpretation. The initialization checker

reports true positive warnings in several open-source Scala projects with reasonable precision and performance. This highlights the necessity of checking the initialization safety of global objects when compiling real-world Scala projects.

References

[1] An overview of tasty. `https://docs.scala-lang.org/scala3/guides/tasty-overview.html`.

[2] Egon Börger and Wolfram Schulte. Initialization problems for Java. *Software - Concepts & Tools*, 19(4):175–178, 2000.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

[4] David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. Abstracting definitional interpreters (functional pearl). *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–25, August 2017.

[5] Kyle Dewey, Vineeth Kashyap, and Ben Hardekopf. A parallel abstract interpreter for JavaScript. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, page 34–45, USA, 2015. IEEE Computer Society.

[6] Thomas Gilray, Michael D. Adams, and Matthew Might. Allocation characterizes polyvariance: a unified methodology for polyvariant control-flow analysis. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, page 407–420, New York, NY, USA, 2016. Association for Computing Machinery.

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

[8] J. Ian Johnson, Ilya Sergey, Christopher Earl, Matthew Might, and David Van Horn. Pushdown flow analysis with abstract garbage collection. *Journal of Functional Programming*, 24(2–3):218–283, 2014.

[9] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, page 194–206, New York, NY, USA, 1973. Association for Computing Machinery.

[10] Dexter Kozen and Matt Stillerman. Eager class initialization for Java. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 71–80. Springer, 2002.

[11] Ondřej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3–16, New York, NY, USA, 2011. ACM.

[12] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[13] Fengyun Liu. *Safe initialization of objects*. PhD thesis, EPFL, Lausanne, 2020.

[14] Fengyun Liu, Ondřej Lhoták, David Hua, and Enze Xing. Initializing global objects: Time and order. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023.

[15] Fengyun Liu, Ondřej Lhoták, Enze Xing, and Nguyen Cao Pham. Safe object initialization, abstractly. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Scala*, SCALA 2021, page 33–43, New York, NY, USA, 2021. Association for Computing Machinery.

[16] Adrian Lupasc. Static Members of Classes in C#. *Annals of the University Dunarea de Jos of Galati: Fascicle: XVII, Medicine*, (3), 2017.

[17] Matthew Might. Abstract interpreters for free. In *Proceedings of the 17th International Conference on Static Analysis*, SAS'10, page 407–421, Berlin, Heidelberg, 2010. Springer-Verlag.

[18] Matthew Might and Panagiotis Manolios. A posteriori soundness for non-deterministic abstract interpretations. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI '09, page 260–274, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: abstract garbage collection and counting. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, page 13–25, New York, NY, USA, 2006. Association for Computing Machinery.

[20] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, page 1–11, New York, NY, USA, 2002. Association for Computing Machinery.

[21] Abel Nieto, Yaoyu Zhao, Ondřej Lhoták, Angela Chang, and Justin Pu. Scala with Explicit Nulls. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:26, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[22] Xin Qi and Andrew C. Myers. Masked types for sound object initialization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, page 53–65, New York, NY, USA, 2009. Association for Computing Machinery.

[23] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, page 717–740, New York, NY, USA, 1972. Association for Computing Machinery.

[24] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

[25] Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 17–30, New York, NY, USA, 2011. ACM.

[26] Alexander J. Summers and Peter Mueller. 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, OOPSLA '11, page 1013–1032, New York, NY, USA, 2011. Association for Computing Machinery.

[27] David Van Horn and Matthew Might. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, page 51–62, New York, NY, USA, 2010. Association for Computing Machinery.

[28] Bram Vandenbogaerde, Sarah Verbelen, Noah Van Es, and Coen De Roover. Monarch: A modular framework for abstract definitional interpreters in haskell. In *Static Analysis: 32nd International Symposium, SAS 2025, Singapore, Singapore, October 13–14, 2025, Proceedings*, page 357–385, Berlin, Heidelberg, 2025. Springer-Verlag.