Safe Object Initialization, Abstractly

Fengyun Liu
Oracle Labs
Switzerland

Ondřej Lhoták
Enze Xing
Nguyen Cao Pham
University of Waterloo
Canada

Abstract

Objects under initialization are fragile: some of their fields are not yet initialized. Consequently, accessing those uninitialized fields directly or indirectly may result in program crashes or abnormal behaviors at runtime.

A newly created object goes through several states during its initialization, beginning with all fields being empty until all of them are filled. However, ensuring initialization safety statically, without manual annotation of initialization states in the source code, is a challenge, due to *aliasing*, *virtual method calls* and *typestate polymorphism*.

In this work, we introduce a novel analysis based on abstract interpreters to ensure initialization safety. Compared to the previous approaches, our analysis is simpler and easier to extend, and it does not require any user annotations. The analysis is inter-procedural, context-sensitive and flow-insensitive, yet it has good performance thanks to *local reasoning* and *heap monotonicity*.

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

Keywords: Object initialization, abstract interpreter, heap monotonicity

ACM Reference Format:
Fengyun Liu, Ondřej Lhoták, Enze Xing, and Nguyen Cao Pham. 2021. Safe Object Initialization, Abstractly. In *Proceedings of the 12th ACM SIGPLAN International Scala Symposium (SCALA '21), October 17, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3486610.3486895

1 Introduction

Object-oriented programming is one of the main paradigms of software construction in industry. However, 50 years after its introduction, language designers are still at a loss about how to ensure initialization safety, and initialization still causes problems for programmers in practice [Duffy 2010]:

> Not only are partially-constructed objects a source of consternation for everyday programmers, they are also a challenge for language designers wanting to provide guarantees around invariants, immutability and concurrency-safety, and non-nullability.

The following example demonstrates the problem: [1]:

```
class Permissions:
  val ALL: Int  = READ | WRITE
  val READ: Int = 1
  val WRITE: Int = 2
```

In the code above, we would expect the field `ALL` to hold the value `3` at runtime. Instead, it holds the value `0`, because the fields `READ` and `WRITE` are not yet initialized when they are used in the second line.

A newly created object goes through several states during its initialization, beginning with all fields being empty until all of them are filled. Therefore, accessing the field of an object has to respect its initialization *typestate* [Strom and Yemini 1986]. Whereas types support detecting unsupported operations on values at compile-time, *typestates ensure that only a valid subset of the supported operations are performed for a specific state of a value.* The problem of safe initialization of objects is in essence a typestate safety problem. However, ensuring typestate safety statically, without manual annotation of typestates in the source code, is a challenge, due to *aliasing*, *virtual method calls* and *typestate polymorphism* [Liu et al. 2020].

Past research has made significant progress on the problem of object initialization, e.g., capturing initialization state as *typestates* [Fähndrich and Leino 2003; Qi and Myers 2009], using *heap-monotonic typestate* to deal with aliasing [Fähndrich and Leino 2003], employing subtyping to deal with *typestate polymorphism* [Summers and Müller 2011], and upholding *local reasoning about initialization* to avoid whole-program analysis [Liu et al. 2020].

In particular, the latter work by Liu et al. [2020] introduces a type-and-effect inference system that can perform typestate inference, and thus cut down the syntactic verbosity

[1]In the absence of special notes, the code examples are in Scala 3, which in addition supports indentation syntax and thus is cleaner and more succint.

of safe initialization systems. Our work is inspired by the type-and-effect inference system. However, we take a very different approach based on the following insight:

> A type-and-effect system is equivalent to a heap-monotonic abstract interpreter.

This insight enables us to develop a new analysis based on abstract interpreters, which is simpler, easier to extend and more regular.

We demonstrate the extensibility of the new analysis with two improvements. The inference system in [Liu et al. 2020] does not support leaking **this** to constructors without annotations on class parameters.

```
class A:
  val b = new B(this)

class B(val a: A):
  val n = 10
```

In the code above, the uninitialized object referred to by **this** is leaked at line 2. The code is safe as it does not access any uninitialized field at runtime. However, the inference system of Liu et al. [2020] would require an explicit annotation of initialization state on the parameter a of the class B. While our system may also be extended to support explicit annotations of initialization states, it removes the burden of such annotations.

Our new analysis also avoids an awkward *length-restriction* in the type-and-effect system proposed by Liu et al. [2020]. As understanding of the type-and-effect system is inessential to the main content of this paper, we do not go into technical details of the type-and-effect system here.

1.1 Contributions

Our work makes the following contributions:

1. Propose a novel analysis for safe initialization. The new analysis based on abstract interpreters improves the state of the art [Liu et al. 2020] by being simpler, easier to extend and more regular.

2. Advocate a new approach in developing practical analyses. Compiler writers face the dilemma that type systems for typestates are verbose whereas inter-procedural and context-sensitive analyses are slow. Our analysis concretely demonstrates that combining language design and analysis can yield fast and useful inter-procedural and context-sensitive analysis that can be integrated in compilers.

3. Implement the analysis in Scala 3 compiler. We implement the new analysis in the Scala 3 compiler and evaluate it on several real world projects.

2 Principles

It is well-known that type systems for typestates are verbose whereas inter-procedural and context-sensitive analyses are slow. The dilemma leaves compiler writers helpless in checking more complex properties of programs, such as initialization safety. Our analysis demonstrates that combining language design and analysis can yield fast and useful inter-procedural and context-sensitive analysis that can be integrated in compilers.

This is achieved by imposing language design rules on user programs. Our analysis inherits several core design principles from past work [Fähndrich and Leino 2003; Liu et al. 2020; Summers and Müller 2011]. In this section, we introduce the main design principles informally.

2.1 Local Reasoning

Local reasoning about initialization [Liu et al. 2020] is the reasoning principle that:

> *In a transitively initialized environment, the resulting value of an expression must be transitively initialized.*

An object is *transitively initialized* if all objects reachable from it are initialized, i.e., all fields are assigned. Intuitively, the principle implies that, for example, if a constructor is called with only transitively initialized arguments, the resulting object is transitively initialized. Similarly, if the receiver and arguments of a method call are transitively initialized, so is the result.

Local reasoning about initialization is a valuable reasoning principle, as it avoids global analysis of programs, which is the key for simple and fast initialization systems.

Liu et al. [2020] provide a modular understanding of local reasoning as three independent properties:

- **Weak monotonicity**: initialized fields continue to be initialized.
- **Stackability**: all fields of a class should be initialized at the end of the class constructor.
- **Scopability**: there are no side channels for accessing uninitialized objects.

From the perspective of language design, the property of *weak monotonicity* suggests the removal of **null** from languages, so that an initialized field cannot become uninitialized by assigning **null** to it. The property *stackability* suggests that field initializers should be mandatory. The property *scopability* suggests that a method may only access uninitialized objects through **this** or method parameters.

Global variables and control effects (e.g. exceptions and algebraic effects) pose a challenge to local reasoning because they may serve as side channels for teleporting uninitialized values. To maintain local reasoning about initialization, global variables should only contain transitively initialized objects [2], and the initialization system needs to make sure that only initialized values may travel through control effects.

[2]The safe initialization of global variables themselves pose a challenge as well, which is a different problem that the current paper does not handle.

2.2 Monotonicity

One insight of Fähndrich and Leino [2003] to deal with type-state in the presence of aliasing is *monotonicity*. Roughly, it means that objects may only become more initialized but not less initialized.

Liu et al. [2020] identify three different concepts of monotonicity: *weak monotonicity*, *strong monotonicity* and *perfect monotonicity*. Weak monotonicity requires that initialized fields continue to be initialized. Strong monotonicity additionally requires that transitively initialized objects continue to be transitively initialized. Perfect monotonicity in addition stipulates that the initialization states of the fields of objects are monotone.

The following example shows that weak monotonicity is not enough to ensure initialization safety:

```
1  trait Reporter { def report(msg: String): Unit }
2  class FileReporter(ctx: Context) extends Reporter:
3      ctx.typer.reporter = this  // problematic
4      ctx.operation()            // problematic
5      val file: File = new File("report.txt")
6      def report(msg: String) = file.write(msg)
```

In the code above, suppose `ctx` is a transitively initialized value. Now the assignment at line 3 makes **`this`**, which is not fully initialized, reachable from `ctx`. This makes the operation on `ctx` at line 4 dangerous, as it may indirectly call the method `report` of the current object and thus reach the uninitialized field `file`. Strong monotonicity would reject the assignment at line 3 and thus defend against possible runtime errors.

However, to enable safe usage of already initialized fields of an object under initialization, we need an even stronger concept, as the following example demonstrates:

```
1  trait Reporter { def report(msg: String): Unit }
2  class MyReporter(rp: Reporter) extends Reporter:
3    this.rp = this      // problematic
4    this.operation()
5    val buffer = new ListBuffer[String]
6    def report(msg: String) = buffer.add(msg)
7    def operation() = rp.report("log)
```

In the code above, we assume that initially the field `rp` of the class `MyReporter` is transitively initialized — thus the object may be used freely. However, at line 3 we reassign the field with **`this`**, which is not fully initialized. The assignment is valid in the context of strong monotonicity, because strong monotonicity only cares about the initialization state of objects rather than fields. However, the assignment makes the initialization state of the field `rp` go backward. Now usage of the field `rp` may potentially reach the uninitialized field `buffer`.

The initialization state of an object not only includes the set of initialized fields, but also the initialization states of the objects that the fields point to. Perfect monotonicity enforces that the initialization state of a field, i.e. the initialization state of the object that it points to, is monotone across mutations.

The type-and-effect system of Liu et al. [2020] implements perfect monotonicity with a simple rule: *only a transitively initialized value may be assigned to a field.* Note that the system distinguishes field assignment from field initialization in syntax, as is the case in Scala. In field initialization, a field may be initialized with non-initialized values. This rule is simple for programmers and easy to implement, so we also adopt this design in our analysis.

3 A Core Language

We first introduce a core language on which we will develop our analysis (Section 4).

3.1 Syntax

Our language resembles a subset of Scala having only top-level classes, mutable fields and methods. Our language is essentially the same as the language in [Liu et al. 2020].

$$
\begin{array}{rcl}
\mathcal{P} \in \text{Program} & ::= & (\overline{\mathcal{C}}, \mathcal{D}) \\
\mathcal{C} \in \text{Class} & ::= & \mathit{class}\ C(\overline{\hat{f}{:}T})\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \} \\
\mathcal{F} \in \text{Field} & ::= & \mathit{var}\ f{:}T = e \\
e \in \text{Exp} & ::= & x \mid \mathit{this} \mid e.f \mid e.m(\overline{e}) \mid \\
 & & \mathit{new}\ C(\overline{e}) \mid e.f = e; e \\
\mathcal{M} \in \text{Method} & ::= & \mathit{def}\ m(\overline{x{:}T}) : T = e \\
S, T, U \in \text{Type} & ::= & C, D
\end{array}
$$

A program $\mathcal{P}$ is composed of a list of class definitions and an entry class. The entry class must have the form $\mathit{class}\ D\ \{\ \mathit{def}\ \ \mathit{main}()\ :\ T\ =\ e\ \}$. The program runs by executing e.

A class definition contains class parameters ($\overline{\hat{f}{:}T}$), field definitions ($\mathit{var}\ f{:}T = e$) and method definitions ($\mathit{def}\ m(\overline{x{:}T}) : T = e$). Class parameters are also fields of the class. All class fields are mutable. As a convention, we use f to range over all fields and $\hat{f}$ to only range over class parameters.

An expression (e) can be a variable (x), a self reference (this), a field access ($e.f$), a method call ($e.m(\overline{e})$), a class instantiation ($\mathit{new}\ D(\overline{e})$), or a block expression ($e.f = e; e$). The block expression is used to avoid introducing the syntactic category of statements in the presence of assignments, which simplifies the presentation and meta-theory.

A method definition is standard. The body of a method is an expression, which could be a block expression to express a sequence of computations.

For the simplicity of presentation, we intentionally make the language simple so that it captures the essence of initialization problems while the rules fit in one page. We do have in mind straightforward extensions with **`if`**-expressions, Boolean and numeric values, as well as logic and arithmetic operations.

Sequencing and let-bindings can be encoded with method calls. For example, sequencing can be encoded with a two-parameter method:

```
def seq(b: B, c: C): C = c
seq(fooB(), barC())
```

We will use these extensions freely in the code examples and make sure that the theory carries over to the extensions trivially. We will discuss how to scale our technique to Scala in Section 5.3.

3.2 Semantics

The following constructs are used in defining the semantics:

$$\begin{array}{rcl} \Xi \in \text{ClassTable} & = & \text{ClassName} \rightharpoonup \text{Class} \\ \sigma \in \text{Store} & = & \text{Loc} \rightharpoonup \text{Obj} \\ \rho \in \text{Env} & = & \text{Variable} \rightharpoonup \text{Value} \\ o \in \text{Obj} & = & \text{ClassName} \times (\text{FieldName} \rightharpoonup \text{Value}) \\ l, \psi \in \text{Value} & = & \text{Loc} \end{array}$$

We use ψ to denote the value of *this*, σ to denote the heap, and ρ to denote the local variable environment of the current stack frame.

The big-step semantics for expressions have the form $[\![e]\!](\sigma, \rho, \psi) = (l, \sigma')$, which means that given the heap σ, environment ρ and value ψ for *this*, the expression e evaluates to the value l with the updated heap σ'. The semantics of programs have the form $[\![\mathcal{P}]\!]$, which simply evaluate the body of the entry method with a trivial initial setting for the heap, environment and the value for *this*.

The big-step semantics is standard, thus we omit the details due to space restriction. The only note is that non-initialized fields are represented by missing keys in the object, instead of a *null* value. Newly created objects have no fields, and new fields are gradually inserted during initialization until all fields defined by the class have been assigned.

The big-step semantics only cover terminating programs. It is straightforward to instrument it with a *fuel* to cover all programs [Amin and Rompf 2017]. A concrete step-indexed big-step semantics for the language can be found in Appendix A of the thesis of Liu [2020].

3.3 Type System

The language is equipped with a simple type system, to ensure that fields hold values of the right type, method calls and field access are allowed by the type of the value. The type system is simple and straightforward, thus we omit the detailed rules due to space restriction.

We will write $\vdash \mathcal{P}$ or $\vdash (C, \mathcal{D})$ to mean that the program is well typed, which is used as a pre-condition for the analysis (Rule A-Prog in Figure 1).

Note that the type system defends against simple ill-formed programs but does not guarantee initialization safety. It is the task of the analysis in Section 4 to ensure initialization safety of the language.

4 The Analysis

In this section, we detail the design of the abstract interpreter based on the core language.

4.1 Introduction

Our analysis takes the form of abstract definitional interpreters [Darais et al. 2017]. While it is very different in form from the type-and-effect system by Liu et al. [2020], it inherits the key design principles from the latter as discussed in Section 2.

The new analysis adopts the same design restriction that *method arguments must be transitively initialized*, while constructor arguments may take uninitialized objects. This design choice makes the analysis receiver-sensitive but insensitive to method arguments. We believe it achieves a good balance between expressiveness and performance.

The analysis is modular at the level of classes. It means that each class is checked independently. The analysis takes class constructors as entry points, and conducts the check of each class separately.

4.2 Abstract Domain

The abstract domain is based on the work of Liu [2020], which identifies the following basic abstractions:

- **Cold**: A cold object *may* have uninitialized fields.
- **Warm**: A warm object has all its fields initialized.
- **Hot**: A hot object has all its fields initialized and only reaches hot objects.

Hot objects are *transitively* initialized. Note that a *warm* object is not *transitively* initialized, because it may reach a cold object.

More formally, our analysis uses the following abstract domain:

$$\begin{array}{rcl} \hat{l} \in \text{ALoc} & ::= & \text{This}(C, \overline{\hat{v}}, \Omega) \mid \text{Warm}(C, \overline{\hat{v}}) \\ \hat{v} \in \text{AValue} & ::= & \text{Hot} \mid \text{Cold} \mid \hat{l} \\ \Omega & ::= & \{ f_1, f_2, \ldots \} \end{array}$$

The abstract values form a lattice:

$$\text{Hot} \sqsubset \hat{l} \sqsubset \text{Cold}$$

An abstract value $\hat{v}$ can be either `Hot`, `Cold` or an abstract address $\hat{l}$. An abstract address $\hat{l}$ can be either $\text{This}(C, \overline{\hat{v}}, \Omega)$ or $\text{Warm}(C, \overline{\hat{v}})$. They are called abstract addresses or abstract locations because conceptually they point to abstract objects. In the formal analysis, we do not need the concept of abstract objects nor abstract heap. However, the usage of abstract heap and objects will result in engineering benefits in the actual implementation (Section 5).

The abstract address $\text{This}(C, \overline{\hat{v}}, \Omega)$ denotes objects of the class `C` that are in the process of initialization in its constructor, where the class parameters take abstract values $\overline{\hat{v}}$ and the fields in Ω are initialized. As our analysis is modular at the level of classes, there is exactly one such address in checking a class. The object referred to by $\mathrm{This}(C, \overline{\hat{v}}, \Omega)$ begins with all fields being empty. As the initialization proceeds, more fields become initialized. Monotonicity ensures that the initial abstract values of the fields, provided by the mandatory field initializer, are always an over-approximation of the actual values stored in the fields. The abstract values for fields do not change during the initialization process until the object becomes fully initialized, i.e., *hot*. The fields of hot values are always hot.

The abstract address $\mathrm{Warm}(C, \overline{\hat{v}})$ denotes objects of the class `C`, where the class parameters take abstract values $\overline{\hat{v}}$. A warm object has all its fields initialized. However, it is not fully initialized, because it may reach non-hot objects. The abstract values of the fields of warm objects are determined by the abstract values of class parameters, thus they do not change with respect to a given warm value. In this sense, warm values serve as indices to summaries of abstract objects, which can be cached and reused during the analysis (Section 5.1).

4.3 The Rules

The essence of the analysis is presented in Figure 1. It assumes two helper functions, `initializerFor(C, f)` to get the initializer for the field `f` of the class `C`, and $lookup(\hat{l}, m)$ to look up the method definition of `m` associated with the abstract value $\hat{l}$. The two functions are easy to implement with the help of the class table Ξ, thus we omit their definitions.

At the high level, the analysis checks each class independently (rule A-Prog). In checking each class, the analysis checks each field one by one with updated initialized field set Ω (rule A-Class). For a field, it checks its initializer with the given abstract value for **`this`** (rule A-Field).

The main body of the rules are related to expression check. Expression check rules have the form $\Xi; \hat{l} \Vdash e \longrightarrow \hat{v}$, which means that given the abstract value $\hat{l}$ for **`this`**, the expression e takes the abstract value $\hat{v}$.

Variables are always hot (rule A-Var), as the analysis enforces that method arguments are hot. The expression **`this`** simply takes the given abstract value $\hat{l}$ (rule A-This).

There are several rules for field selection. If the receiver is hot, then the field selection is also hot (A-Sel1). If the selection selects a class parameter, then it simply returns the corresponding abstract value of that class parameter (rule A-Sel2 and A-Sel3).

Note that if the expression e in $e.f$ is *cold*, there are no corresponding rules. It means it is forbidden to select fields on cold objects.

If the field selection $e.f$ selects a field in the class body rather than a class parameter, there are two cases. (1) If e evaluates to $\mathrm{This}(C, \overline{\hat{v}}, \Omega)$, we need to check that f is in the initialized set Ω. If that is the case, then we evaluate the initializer of the field f as the value of the selection (rule A-Sel4). (2) If e evaluates $\mathrm{Warm}(C, \overline{\hat{v}})$, we evaluate the initializer of the field f as the value of the selection (rule A-Sel5).

The re-evaluation of a field initializer is sound because the initializer is always an over-approximation of values stored in the field, thanks to perfect monotonicity (rule A-Assign). Re-evaluation is expensive and might involve duplicate computation. In the implementation (Section 5.1), we will detail how to avoid duplicate computation with caching.

The rule A-Call1 capitalizes on *local reasoning*: if the receiver and method arguments are hot, the result value must be hot. If the receiver e_0 is not hot, it evaluates the body of the method m and returns it as the resulting value (rule A-Call2).

The rule A-New1 also exploits local reasoning: if the arguments to the constructor are hot, the result is also hot. Otherwise, a new expression evaluates to a warm value, with the class parameters taking the abstract values of the constructor arguments (rule A-New2). To ensure that the abstract domain is finite, we widen the abstract values of constructor arguments. Basically, we restrict that constructor arguments may only be either `Hot` or `Cold`. In addition, we need to also check the class again assuming that the class parameters take the widened abstract values.

The rule A-Assign enforces *perfect monotonicity*: only hot values may be assigned to a field. Without this restriction, the rules for field selection will lead to unsoundness.

To extend our language with **`if`**-expressions, we define the final value to be the join of the two branches:

$$\frac{\Xi; \hat{l} \Vdash e \longrightarrow Hot \quad \Xi; \hat{l} \Vdash e_1 \longrightarrow \hat{v}_1 \quad \Xi; \hat{l} \Vdash e_2 \longrightarrow \hat{v}_2}{\Xi; \hat{l} \Vdash if\ e\ then\ e_1\ else\ e_2 \longrightarrow \hat{v}_1 \sqcup \hat{v}_2} \text{ (A-If)}$$

4.4 Co-induction

The rules are mutually recursive: the class check depends on expression check, and expression check depends on class check (rule A-New2).

Meanwhile, the rules should be read co-inductively. It implies that the rules are *declarative* specifications instead of purely *algorithmic* definitions. We show how to make them algorithmic in Section 5.2.

We motivate the co-inductive reading with two examples. Given the following program:

```
class C:
  var f: C = foo()
  def foo(): C = foo()
```

In order to prove that the method call `foo()` is safe in line 2, the analysis will encounter exactly the same sub-goal to prove that the call `foo()` is safe. A coherent inductive interpretation of the expression check rules needs to assume that the call `foo()` returns a fixed-point value for the recursive call `foo()`. In this example, both `Cold` and `Hot` are fixed points for the expression `foo()`. We are only interested in least fixed points, which more precisely approximate the

Program check $\boxed{\Vdash (\overline{C}, \mathcal{D})}$

$$\frac{\vdash (C, \mathcal{D}) \qquad \Xi = \overline{C \to C} \qquad \overline{\Xi; \mathrm{This}(C, \overline{\mathrm{Hot}}, \emptyset) \Vdash C}}{\Vdash (\overline{C}, \mathcal{D})} \quad (\text{A-Prog})$$

Class check $\boxed{\Xi; \mathrm{This}(C, \overline{\hat{v}}, \Omega) \Vdash C}$

$$\frac{\overline{\Xi; \mathrm{This}(C, \overline{\hat{v}}, \Omega_i) \Vdash \mathcal{F}_i \qquad \Omega_{i+1} = \Omega_i \cup \{ f_i \}}}{\Xi; \mathrm{This}(C, \overline{\hat{v}}, \Omega_0) \Vdash \mathit{class}\ C(\overline{\hat{f}{:}T})\ \{ \overline{\mathcal{F}}\ \overline{\mathcal{M}} \}} \quad (\text{A-Class})$$

Field check $\boxed{\Xi; \mathrm{This}(C, \overline{\hat{v}}, \Omega) \Vdash \mathcal{F}}$

$$\frac{\Xi; \mathrm{This}(C, \overline{\hat{v}}, \Omega) \Vdash e \longrightarrow \hat{l}}{\Xi; \mathrm{This}(C, \overline{\hat{v}}, \Omega) \Vdash \mathit{var}\ f : D = e} \quad (\text{A-Field})$$

Expression check $\boxed{\Xi; \hat{l} \Vdash e \longrightarrow \hat{v}}$

$$\Xi; \hat{l} \Vdash x \longrightarrow \mathrm{Hot} \quad (\text{A-Var}) \qquad \Xi; \hat{l} \Vdash \mathit{this} \longrightarrow \hat{l} \quad (\text{A-This})$$

$$\frac{\Xi; \hat{l} \Vdash e \longrightarrow \mathrm{Hot}}{\Xi; \hat{l} \Vdash e.f \longrightarrow \mathrm{Hot}} \ (\text{A-Sel1}) \qquad \frac{\Xi; \hat{l} \Vdash e \longrightarrow \mathrm{This}(C, \overline{\hat{v}}, \Omega)}{\Xi; \hat{l} \Vdash e.\hat{f}_i \longrightarrow \hat{v}_i} \ (\text{A-Sel2}) \qquad \frac{\Xi; \hat{l} \Vdash e \longrightarrow \mathrm{Warm}(C, \overline{\hat{v}})}{\Xi; \hat{l} \Vdash e.\hat{f}_i \longrightarrow \hat{v}_i} \ (\text{A-Sel3})$$

$$\frac{\Xi; \hat{l} \Vdash e \longrightarrow \mathrm{This}(C, \overline{\hat{v}}, \Omega) \qquad f \in \Omega \qquad e' = \mathit{initializerFor}(C, f) \qquad \Xi; \mathrm{This}(C, \overline{\hat{v}}, \Omega) \Vdash e' \longrightarrow \hat{v}}{\Xi; \hat{l} \Vdash e.f \longrightarrow \hat{v}} \quad (\text{A-Sel4})$$

$$\frac{\Xi; \hat{l} \Vdash e \longrightarrow \mathrm{Warm}(C, \overline{\hat{v}}) \qquad e' = \mathit{initializerFor}(C, f) \qquad \Xi; \mathrm{Warm}(C, \overline{\hat{v}}) \Vdash e' \longrightarrow \hat{v}}{\Xi; \hat{l} \Vdash e.f \longrightarrow \hat{v}} \quad (\text{A-Sel5})$$

$$\frac{\Xi; \hat{l} \Vdash e_0 \longrightarrow \hat{l}_0 \qquad \overline{\Xi; \hat{l} \Vdash e \longrightarrow \mathrm{Hot}} \qquad \mathit{lookup}(\hat{l}_0, m) = \mathit{def}\ m(\overline{x{:}T}) : T = e_1 \qquad \Xi; \hat{l}_0 \Vdash e_1 \longrightarrow \hat{v}}{\Xi; \hat{l} \Vdash e_0.m(\overline{e}) \longrightarrow \hat{v}} \quad (\text{A-Call2})$$

$$\frac{\Xi; \hat{l} \Vdash e_0 \longrightarrow \mathrm{Hot} \qquad \overline{\Xi; \hat{l} \Vdash e \longrightarrow \mathrm{Hot}}}{\Xi; \hat{l} \Vdash e_0.m(\overline{e}) \longrightarrow \mathrm{Hot}} \ (\text{A-Call1}) \qquad \frac{\overline{\Xi; \hat{l} \Vdash e \longrightarrow \mathrm{Hot}}}{\Xi; \hat{l} \Vdash \mathit{new}\ C(\overline{e}) \longrightarrow \mathrm{Hot}} \ (\text{A-New1})$$

$$\frac{\overline{\Xi; \hat{l} \Vdash e \longrightarrow \hat{v}} \qquad \overline{\hat{v}' = \mathit{widen}(\hat{v})} \qquad \Xi; \mathrm{This}(C, \overline{\hat{v}'}, \emptyset) \Vdash \Xi(C)}{\Xi; \hat{l} \Vdash \mathit{new}\ C(\overline{e}) \longrightarrow \mathrm{Warm}(C, \overline{\hat{v}'})} \quad (\text{A-New2})$$

$$\frac{\Xi; \hat{l} \Vdash e_1 \longrightarrow \hat{v}_1 \qquad \Xi; \hat{l} \Vdash e_2 \longrightarrow \mathrm{Hot} \qquad \Xi; \hat{l} \Vdash e \longrightarrow \hat{v}}{\Xi; \hat{l} \Vdash e_1.f = e_2; e \longrightarrow \hat{v}} \quad (\text{A-Assign})$$

Helpers $\boxed{\mathit{widen}(\hat{v}) = \hat{v}}$

$$\begin{array}{rcll} \mathit{widen}(\hat{l}) & = & \mathrm{Cold} & \\ \mathit{widen}(\hat{v}) & = & \hat{v} & \text{otherwise} \end{array}$$

Figure 1. Co-inductive initialization check rules

runtime semantics of expressions, so more programs may be accepted. The greatest fixed point Cold is always safe, but it rejects too many programs, so it is not useful. We show how to compute the least fixed points efficiently in the implementation (Section 5.2).

The co-inductive interpretation is also needed for checking classes. Given the following example:

```
class A(b: B) { var b2 = new B(this) }
class B(a: A) { var a2 = new A(this) }
```

Checking the class A would encounter the sub-goal of checking the class A with the value $\mathrm{This}(A, Cold, \emptyset)$ for **this** recursively. The same holds for the class B with the value $\mathrm{This}(B, Cold, \emptyset)$. In this case, the co-inductive interpretation only needs to assume that the recursive sub-goal trivially holds.

We conjecture that a program that is well-formed according to the analysis does not get stuck at runtime.

Proposition 4.1 (Soundness)**.** $\Vdash \mathcal{P} \implies \forall k.\ [\![\mathcal{P}]\!](k) \neq Error$

In the above, k is the index to the step-indexed big-step semantics. For step-indexed semantics, there are three possible outcomes: (1) time out; (2) error; (3) a resulting value and an updated heap. Initialization safety is implied by soundness, as initialization errors will cause the program to fail at runtime.

We leave the soundness proof for future work.

5 Implementation

In this section, we discuss the implementation of the analysis: (1) How to enable caching and avoid duplicate computation? (2) How to compute the least fixed point of an expression? (3) How to scale the analysis to complex language features?

5.1 Caching

While the presentation in the previous section shows the essence of the analysis, it does not show how to perform caching to avoid duplicate computation when retrieving field values of warm objects.

For the purpose of caching, we perform some engineering operations of the abstract domain as follows:

$$\begin{array}{rcl} \hat{l} \in \mathrm{ALoc} & ::= & \mathrm{This}(C) \mid \mathrm{Warm}(C, \overline{\hat{v}}) \\ \hat{v} \in \mathrm{AValue} & ::= & \mathrm{Hot} \mid \mathrm{Cold} \mid \hat{l} \\ \hat{\sigma} \in \mathrm{AStore} & = & \mathrm{ALoc} \rightharpoonup \mathrm{AObj} \\ \hat{o} \in \mathrm{AObj} & = & \mathrm{ClassName} \times (\mathrm{FieldName} \rightharpoonup \mathrm{AValue}) \end{array}$$

As can be seen above, we introduced abstract object $\hat{o}$ and abstract heap $\hat{\sigma}$. We removed Ω from $\mathrm{This}(C)$, as it is implied by the present fields in the abstract object pointed to by $\mathrm{This}(C)$. Now for field selection, we can just retrieve the corresponding field value from the abstract object.

In checking a field definition, we put the field value in the abstract object that corresponds to **this**. Thanks to monotonicity, this cached value is only determined by the value of **this**, and we maintain an invariant in the implementation to ensure that it is only set once. Therefore, the abstract heap and abstract objects only serve as a cache of summaries. They do not play any essential role in the analysis, despite their engineering benefits, such as uniform field access.

Another implementation change we made is to remove the class parameter values from $\mathrm{This}(C)$, assuming they are always hot. In the rule A-New2, we use the value $\mathrm{Warm}(C, \overline{\hat{v}})$ for **this** to check the class C instead of $\mathrm{This}(C, \overline{\hat{v}})$. This change is motivated by two practical concerns: (1) using the warm value will enable immediate caching of the field values; (2) it avoids duplicate error reports in case the class suffers from initialization errors. Doing so is safe because all errors that can be detected in checking $\mathrm{This}(C, \overline{\hat{v}})$ can also be detected by checking $\mathrm{This}(C, \overline{Hot})$ and $\mathrm{Warm}(C, \overline{\hat{v}})$, thanks to *stackability* and mandatory field initializers.

One subtlety is how to construct mutually recursive warm objects, as the following program shows:

```
1  class A(b: B):
2    val b2 = new B(this)
3    val c = b2.a2
4
5  class B(a: A):
6    val a2 = new A(this)
7    val c = a2.b2
```

The object $\mathrm{Warm}(A, Cold)$ uses the object $\mathrm{Warm}(B, Cold)$ in line 3, and $\mathrm{Warm}(B, Cold)$ uses $\mathrm{Warm}(A, Cold)$ in line 7. Obviously there is no way to construct the two abstract warm objects in sequence. The solution is that when accessing a field of a warm value which is missing in the corresponding object, we simply evaluate the initializer of the field to get its value — this corresponds to the rule A-Sel5 in Figure 1. In contrast, for $\mathrm{This}(C)$, a missing key in the corresponding object means that field is not yet initialized, thus an error should be reported.

5.2 Fixed-point computation

The co-inductive rules in the previous section are declarative but algorithmic, as a valid derivation based on the rules will depend on oracle values which are the fixed point values of expressions in the presence of recursion. Here we show how to compute fixed points for the abstract values of expressions.

As mentioned before, we are only interested in least fixed points, as they admit more valid programs. The standard technique is to introduce an inductive cache in evaluating an expression ζ [Darais et al. 2017]:

$$\hat{\zeta} \in \mathrm{ACache} \quad = \quad (\mathrm{Exp} \times \mathrm{ALoc}) \rightharpoonup \mathrm{AValue}$$

The key of the cache ζ consists of the expression to be evaluated and the value for **this**. Thanks to monotonicity, we do not need to include the abstract store $\hat{\sigma}$ as part of the key to the cache. Thanks to the restriction that all method parameters are fully initialized, we can safely ignore the environment ρ of the concrete domain.

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

The iterative algorithm works on the granularity of classes. For each iteration, it checks whether ζ_{in} and ζ_{out} are the same. If not, it will use ζ_{out} as the new ζ_{in}, reset ζ_{out} to empty, revert heap changes in the last iteration, and check the class again until a fixed point is reached. The fixed point always exists as the checking function is monotone with respect to the abstract cache.

For most real-world programs, the fixed point is reached after one iteration. The following example shows where more than one iteration is needed:

```
class C:
  val self = foo(5)
  def foo(x: Int): C =
    if (x < 5) then this else foo(x - 1).self
```

The code above, when run, will access the uninitialized field `self`. However, if we run the iteration once, the recursive call `foo(x - 1)` will simply take the value `Hot` from the co-inductive cache. The error can only be detected in the second iteration, where the recursive call retrieves the updated cache value $\text{This}(C, \emptyset, \emptyset)$. The field `self` is not yet in the initialized set, thus accessing the field is an error.

5.3 Scalability

Scala has many features far beyond the core language, e.g. inheritance, nested classes, traits, lazy fields, functions.

Following Liu et al. [2020], the analysis performs full-construction analysis, i.e., it takes constructors of concrete classes as entry points and handles super-constructor calls as if they are inlined. This way, the precise class of **this** and all warm objects is known, so virtual method calls on these receivers can be statically resolved. Any argument that leaks to a virtual method that cannot be statically resolved is required to be fully initialized, so the analysis does not need to analyze bodies of such methods. Therefore, inheritance does not create more challenges in the implementation. This approach does raise some concerns about modularity, though it does not assume a closed world. The design trade-off is discussed further in Liu et al. [2020].

The analysis assumes erasure semantics for parametric polymorphism, therefore complex type-level features, such as F-bounded polymorphism, recursive types, refinement types and higher-kinded types do not pose a challenge in the implementation.

The traits are initialized following a scheme called *linearization* [Odersky 2019]. The implementation follows the linearization semantics in initialization as well as in the resolution of virtual method calls.

To handle nested classes, we augment warm values with a field `outer`, which represents the abstract value for the immediate outer reference of the class `klass`:

```
case class ThisRef(klass: ClassSymbol)
case class Warm(
    klass: ClassSymbol, outer: Value,
    ctor: Symbol, args: List[Value])
```

Note that in `ThisRef` (which corresponds to `This` in the paper), an `outer` field is not needed because we assume that outer references and class parameters of `ThisRef` are all hot.

Each object may have a matrix of outer references: each class in the inheritance chain has a list of outer references going outward. Why does it suffice to only store one outer reference in `Warm`? The insight here is that all outer references are determined by the immediate outer reference of the class at the bottom of the inheritance chain. While we do cache all outer references in the abstract warm object for fast access, in the warm value, which serves as a key to the abstract object, it suffices to store just the determining outer reference.

Local classes are handled as if they were inner classes located in the closest enclosing classes. This approach is safe because in the system, all method parameters are required to be *hot*. The only possible initialization effects that could be observed in a local class are the initialization effects of its enclosing class.

To handle functions, we introduce the value `Fun`:

```
case class Fun(
    expr: Tree, thisV: Addr,
    klass: ClassSymbol, env: Env)
```

For function application, we enforce the same restriction as method calls: the arguments must be hot. In most cases, the `env` is empty, as we require method arguments to be hot. Functions inside secondary constructors may contain a non-hot `env` if the arguments to the secondary constructor are not hot.

Lazy fields are treated as method calls. Field accesses in Scala are actually method calls, which could be overridden. The analysis follows the semantics closely.

5.4 Early Promotion

The implementation introduces an optimization called *early promotion*. Semantically, in the process of initialization, the initialization state of an object can be promoted naturally to *hot* when all objects reachable from it are initialized. The natural promotion happens when we use local reasoning for

new expressions (Rule A-New1 in Figure 1). This is called a *commitment point* in Summers and Müller [2011].

However, there are two possible improvements to natural promotion, which we dub *early promotion*. First, if all fields of an object are initialized, we may promote the object to hot before the commitment point, as the following code shows:

```
1 class C:
2   val a = 10
3   this.foo()
```

In the above, after line 2, we may promote **this** to hot, thus skip checking the body of the method `foo`, which improves performance of the checker. Of course, such a promotion is safe only if all the class parameters and outers are hot.

Second, a *warm* object can be thought of as promoted as long as it is impossible to access an uninitialized object from it even if it reaches an uninitialized object in the heap. This can be illustrated by the following example:

```
1 class C:
2   case class Data(value: Int)
3   val a = Data(3)
4   val ref = foo(a) // safe to leak a
```

In the code above, the field `a` at line 3 is initialized with a warm value when checking the class `C`. The value is not hot because the anonymous class instance implicitly captures the outer reference `C.this`, which is not hot. However, in normal code (without reflection), it is impossible to get hold of the uninitialized object pointed to by `C.this` from the value `a`, thus it is safe to leak the value `a` at line 4.

However, the following example shows that blind early promotion is unsound:

```
1 def qux[T](e: E[T]) = e.foo
2 abstract class E[T] { def foo: T }
3 class C:
4   val a: E[C] = new E { def foo = C.this.ref }
5   val ref: C = qux(a) // error
```

When this code runs, it will access the uninitialized field `ref`. Therefore, the checker should report an error at line 5.

Early promotion for warm objects works by checking that (1) each method can be safely called and its result value can be safely promoted; and (2) each field value can be safely promoted to `Hot`.

6 Evaluation

We implement the analysis in the Scala 3 compiler and run the analysis on several projects. The result is presented in Figure 2. As the projects are widely used in the industry, we expect all the warnings to be false positives [3].

Compared to the previous implementation based on the type-and-effect system, we can see the number of warnings has decreased for most projects. The reduction comes from three improvements: (1) allowing non-hot arguments to constructors; (2) early-promotion; and (3) allowing non-hot values to be assigned to local variables.

[3]The work by Liu et al. [2020] does report a few true positives. Our analysis is able to detect the same true positives.

The reduction of warnings in the project stdLib213 mainly benefits from the support of allowing non-hot arguments to constructors. The early promotion improvement helps suppress numerous warnings in the project Dotty and the project intent.

However, in some projects, such as Scalacheck and Scalap, the number of warnings has increased significantly. For the Scalap project, all of the new warnings are due to changes in how warnings are reported. More concretely, it comes from the change in reporting as shown in the following example:

```
1 class TestSuite:
2   test(this)
3   test(this)
4   // ...
```

The previous checker only reports one warning at line 2, while the new checker reports two warnings.

For the ScalaCheck project, the original 6 warnings disappear in the new checker, thanks to early promotion. However, we expect the 22 new warnings to be reported by the previous checker as well, but it does not. This looks like to be a bug of the previous checker.

There is another change in reporting incurred by the support of non-hot arguments to constructors. This can be demonstrated by the following program:

```
1 class A { var b = new B(this) }
2 class B(a: A):
3   println(a.b)
4   println(a.b)
```

Without the support of leaking **this** to constructors, the previous checker only reports one warning at line 1, not allowing the constructor to be called at all. The new checker does allow the constructor call, but then reports two warnings within the constructor body at lines 3 and 4.

7 Related Work

Our work takes inspiration from several milestone papers on the problem of initialization.

Typestate Inference for initialization. Our analysis is inspired by the type-and-effect system proposed by Liu et al. [2020]. Despite being different in form, our analysis inherits several design principles, such as local reasoning and perfect monotonicity, as well as the design restriction of requiring arguments to methods being fully initialized.

Our formal system as presented in Section 4 is as expressive as that of Liu et al. [2020]. Meanwhile, it supports leaking non-hot values to constructors without explicit annotations of initialization states on class parameters, as the following code shows:

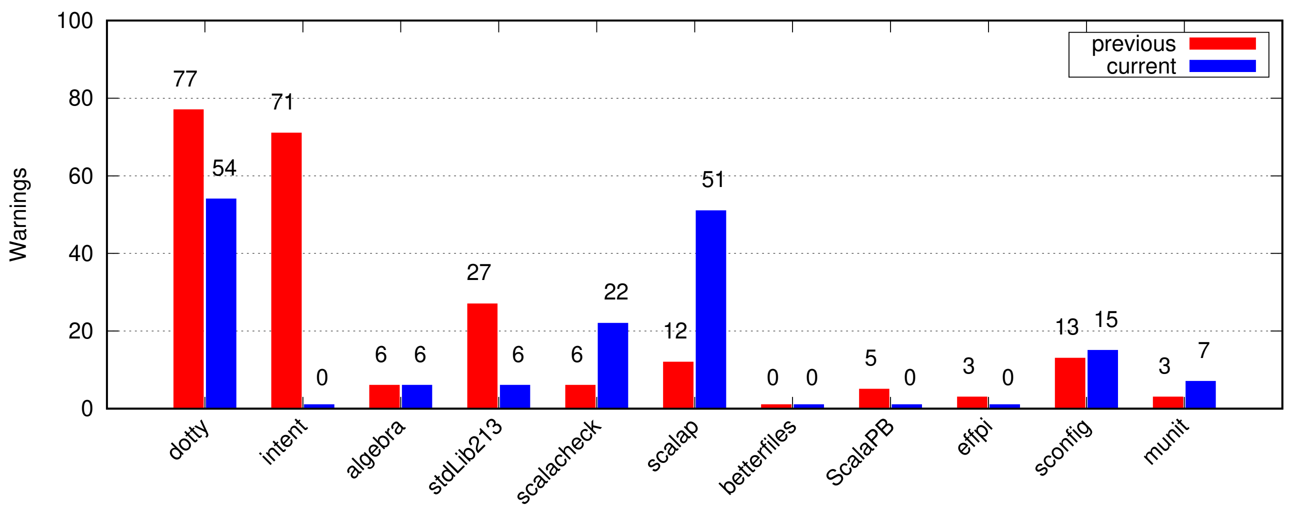

Figure 2. False positive warnings on Scala 3 community projects.

```
class A { val b = new B(this) }
class B(val a: A @cold) { val name = "B" }
```

To support the example above, the formal system in Liu et al. [2020] requires the annotation `@cold` on the class parameter `a`. However, the implementation of that system in the Scala 3 compiler does not support annotations [4] , thus it is less expressive than the implementation of our system in the Scala 3 compiler.

We claim our analysis to be simpler, easier to extend and more regular. It is conceptually simpler as it does not depend on concepts such as proxy effects and proxy potentials. The algorithm does not have a separate summarization phase. Meanwhile, the abstract interpreter follows the structure of concrete interpreters, so it is easier to maintain.

Our analysis also removes the *length restriction* of the aforementioned type-and-effect system. The length restriction ensures that the abstract domain is finite. For example, it is needed for the analysis to terminate for the following program:

```
class A:
  var a: A = this.g
  def g: A = this.g.g
```

However, the restriction looks artificial and makes the analysis irregular. Our analysis completely eliminates the restriction and does not have any fixed length limit on the abstraction.

The compiler for X10 [Zibin et al. 2012] employs an interprocedural analysis to ensure safe initialization, which removes the annotation burden required when calling final or private methods on *this*. However, the analysis algorithm is not presented in the paper. To call virtual methods on *this*, annotations are required on method definitions.

[4]The introduction of annotations requires changes to the standard library, which is difficult to convince for an experimental feature and it takes a long language improvement process.

Type systems for safe initialization. Fähndrich and Leino [2003] introduce raw types of the form $T^{\text{raw}(S)}$. A value of such a type is possibly under initialization, and all fields up to the superclass S are initialized. Class fields may not hold raw values; thus the system does not support creating cyclic data structures. *Delayed types* [Fähndrich and Xia 2007] overcome this limitation by ensuring that the initialization of objects forms stacked time regions.

Qi and Myers [2009] introduce a flow-sensitive type-and-effect system for initialization based on masked types. The system is expressive, but it leaves open the problems of typestate polymorphism and type-and-effect inference.

Summers and Müller [2011] show that initialization of cyclic data structures can be supported in a light-weight, flow-insensitive type system. The system cleverly uses subtyping to achieve typestate polymorphism. However, it leaves open the design of a dataflow analysis that enables the usage of already initialized fields.

The Billion-Dollar Fix [Servetto et al. 2013] introduces a new linguistic construct *placeholders* and *placeholder types* to support initialization of circular data structures. The work is orthogonal to the current work, in that we are constrained from introducing new language constructs and semantics.

Acknowledgments

We thank the anonymous reviewers of Scala 2021 for their constructive comments. We thank the Programming Methods Laboratory (LAMP) at EFPL for sponsoring the first author for part of this work. This research was supported by the Natural Sciences and Engineering Research Council of Canada and by the Waterloo-Huawei Joint Innovation Lab.

References

Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 666–679. http://dl.acm.org/citation.cfm?id=3009866

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

Joe Duffy. 2010. On partially-constructed objects. http://joeduffyblog.com/2010/06/27/on-partiallyconstructed-objects/.

Manuel Fähndrich and K. Rustan M. Leino. 2003. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, Ron Crocker and Guy L. Steele Jr. (Eds.). ACM, 302–312. https://doi.org/10.1145/949305.949332

Manuel Fähndrich and K Rustan M Leino. 2003. Heap monotonic typestates. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*.

Manuel Fähndrich and Songtao Xia. 2007. Establishing object invariants with delayed types. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 337–350. https://doi.org/10.1145/1297027.1297052

Fengyun Liu. 2020. *Safe initialization of objects.* Ph.D. Dissertation. EPFL.

Fengyun Liu, Ondřej Lhoták, Aggelos Biboudis, Paolo G. Giarrusso, and Martin Odersky. 2020. A type-and-effect system for object initialization. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 175:1–175:28. https://doi.org/10.1145/3428243

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

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

Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. 2013. The Billion-Dollar Fix - Safe Modular Circular Initialisation with Placeholders and Placeholder Types. In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7920)*, Giuseppe Castagna (Ed.). Springer, 205–229. https://doi.org/10.1007/978-3-642-39038-8_9

Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.* 12, 1 (1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

Alexander J. Summers and Peter Müller. 2011. Freedom before commitment: a lightweight type system for object initialisation. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 1013–1032. https://doi.org/10.1145/2048066.2048142

Yoav Zibin, David Cunningham, Igor Peshansky, and Vijay A. Saraswat. 2012. Object Initialization in X10. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7313)*, James Noble (Ed.). Springer, 207–231. https://doi.org/10.1007/978-3-642-31057-7_10