A Type-and-Effect System for Object Initialization

Fengyun Liu
EPFL

Aggelos Biboudis
EPFL

Ondřej Lhoták
University of Waterloo

Martin Odersky
EPFL

Abstract
Every newly created object goes through several initialization states: starting from a state where all fields are uninitialized until all of them are assigned. Any operation on the object during its initialization process, which usually happens in the constructor via this, has to observe the initialization states of the object for correctness, i.e. only initialized fields may be used. Checking safe usage of this statically, without manual annotation of initialization states in source code, is a challenge, due to aliasing and virtual method calls on this.

Mainstream languages either do not check initialization errors, like Java, C++, Scala, or they defend against them by not supporting useful initialization patterns, like Swift. In parallel, past research has shown that safe initialization can be achieved for varying degrees of expressiveness but by sacrificing syntactic simplicity.

We approach the problem by upholding two fundamental principles for reasoning about the initialization of objects: \textit{stackability} and \textit{monotonicity}. On this basis, we propose a novel type-and-effect system that can effectively ensure initialization safety while allowing flexible initialization patterns with almost zero annotation burden. The experiments on several real-world projects show that our system advances the state-of-the-art.

Keywords Object initialization, Type-and-effect system

1 Introduction
Object-oriented programming is unsafe if objects cannot be initialized safely. Safe initialization of objects is becoming a challenge as the code in constructors is getting more complex. From past research \cite{5,12,14,15}, two problems are identified and commonly recognized.

The Problem of Premature Usage. By \textit{premature usage}, we mean the usage of fields or calling methods on \textit{this} before the underlying object is fully initialized. The usage of already initialized fields in the constructor is safe and supported by almost all languages. Premature usage becomes a problem when an uninitialized field is accidentally used, directly or indirectly. Based on an extensive study of over sixty thousand classes \cite{5}, Gil et al report that over 8% constructors include method calls on \textit{this}. Such methods could be potentially overridden in a subclass, which makes it difficult to statically check whether it is safe to call such a method. Moreover, aliasing complicates the problem --- if a field aliases \textit{this}, we cannot assume the object pointed to by the field is fully initialized.

The Problem of Cyclic Data Structures. It is challenging to safely initialize cyclic data structures without resorting to \texttt{null}. For example, the following Scala code shows the initialization of two mutually dependent objects:

```scala
class Parent { val child: Child = new Child(this) }
```

The objective is to allow cyclic data structures while avoiding explicit use of \texttt{null} and preventing accidental premature usage of aliased objects. Accessing fields or calling methods on those aliased objects under initialization is an orthogonal concern, of which the importance is open to debate.

Designing a safe initialization system for a practical programming language is an art that strikes a balance among safety, usability and expressiveness.

Safety. A safe initialization checker should soundly over-approximate program semantics. While safety is a noble goal in theoretical work, in practice it may be weakened with unsafe switches, such as \texttt{@unchecked}, to support rare code patterns. Otherwise, the system may become unnecessarily complex for common usage, which harms usability or even makes the system impractical.

Usability. A user-friendly initialization system should not incur much syntactic overhead. The overhead manifests usually in the form of annotations. The rules imposed by the system should be easy to learn and reason about. The error messages should be easy to understand and facilitate bug fixes. A complex type system will have little chance of adoption due to the cognitive overhead and syntactic verbosity.

Expressiveness. An expressive analysis should support common and reasonable initialization patterns. In particular, usage of already initialized fields, calling methods on \textit{this}, and aliasing of \textit{this} to create cyclic data structures should be supported.

Existing programming languages sit at two extremes. On one extreme, we find languages like Java, C++, Scala, where programmers may use \textit{this} as if it is fully initialized, devoid of any safety guarantee. On the other extreme, we find languages like Swift, which ensures safe initialization, but is overly restrictive. The initialization of cyclic data structures is not supported, calling methods on \textit{this} is forbidden, even
the usage of already initialized fields is limited. For example, in the following Swift code, while the usage of x to initialize y is allowed, the usage of y to initialize f is illegal, which is a surprise:

```swift
class Position {
    var x, y: Int
    var f: () -> Int
    init() {
        x = 4
        y = x * x // OK
        f = { () -> Int in self.y } // error
    }
}
```

To solve the problem, we identify and uphold two golden principles in reasoning about initialization, which are scattered and obscured in past research (Section 2.1):

- **Monotonicity**: initialization state may not be reversed
- **Stackability**: objects are initialized in stack order

We argue that if we adhere to the two principles, then traditional class constructors should be replaced by class parameters, and fields should be initialized on declaration. This would impact how future class-based languages will be designed and used (Section 2.2).

Based on the two principles, we propose a novel type-and-effect system that can effectively ensure initialization safety while allowing flexible initialization patterns with almost zero annotation burden. Given the following Scala program:

```scala
abstract class AbstractFile {
    def name: String
}

class RemoteDoc(url: String) extends AbstractFile {
    val localFile: String = url.hashCode // error
    val extension: String = name.substring(4)
    def name: String = localFile
    init() {
        f = { () -> Int in self.y } // error
    }
}
```

Our system will report that the field `localFile` is used before initialization with the following trace:

```scala
→ val extension: String = name.substring(4)
→ def name: String = localFile
```

In summary, our contributions are the following:

- We advocate two fundamental principles in reasoning about initialization: **monotonicity** and **stackability** (Section 2). Based on the two principles, we propose that class-based programming languages should adopt class parameters and mandatory field initializers. As far as we know, this is the first work that bases the syntactic design of constructors on initialization principles.
- We propose a novel type-and-effect system to ensure safe initialization of objects, and prove its soundness. Our system improves the state-of-the-art [15] in terms of expressiveness and usability. Meanwhile, to our knowledge, we are the first to demonstrate the technique of controlling aliasing and leaking in a type-and-effect system for class-based languages.

## 2 Principled Design of Constructors

What are the fundamental principles for initialization? We reflect on the principles of initialization, and the reflections shed light on better design of class constructors.

### 2.1 Principled Initialization

We uphold two golden principles for initialization, namely **monotonicity** and **stackability**, which can be found in the past research but are scattered and obscured.

Roughly, monotonicity means that the initialization state of an object not only includes the fields that are assigned, but also the reachable objects. We give an informal definition below, which we make precise formally in the technical report included as supplemental material.

**Principle 1: Monotonicity.** An object is monotone with respect to initialization if during the lifetime of the object:

- The assigned fields continue to be assigned
- The objects pointed to by assigned fields are monotone

One obvious violation of monotonicity is to assign `null` to a field which is already initialized with a non-null value. To fix the billion-dollar mistake [7], `null` can be removed from the language in favor of the type `Option[T]`.

However, explicit usage of `null` is not the only way to violate monotonicity. Assignment of non-null values may also break monotonicity, as the following code shows:

```scala
trait Reporter { def report(msg: String): Unit }
class FileReporter(ctx: Context) extends Reporter {
    // ctx now reaches an uninitialized object
    val file: File = new File("report.txt")
    def report(msg: String) = file.write(msg)
}
```

In the code above, suppose `ctx` is a fully initialized value. Now the assignment at line 4 makes `this`, which is not fully initialized, reachable from `ctx`. This makes any operation on `ctx` dangerous, as it may indirectly reach uninitialized fields of the current object.

The principle of monotonicity can be found in raw types [3] and masked types [12], but violated in the freedom model [15]. As a result, the freedom model has to restore monotonicity to support the usage of already initialized fields in the implementation.
The second principle we uphold is stackability. We give an informal definition below, which we make precise formally in the technical report.

**Principle 2: Stackability** All fields of a class should be assigned at the end of the class constructor.

The principle stipulates that all fields of a class should be assigned at the end of the class constructor. Consequently, we know that all fields of an object of a new expression can be safely accessed, though the field may point to uninitialized objects.

If we push an object in a stack when it comes into existence, and remove it from the stack when all its fields are assigned, we will find that the object to be removed is always at the top of the stack. That is why we call the principle stackability. This is illustrated in Figure 1.

Without this principle, it will be difficult to reason about when an object becomes fully initialized. For example, in masked types [12], due to lack of stackability, the authors introduced conditionally masked types to type check the following program:

```java
class Knot {
    var self: Knot\self[this.self] = this
}
```

The conditional mask `Knot\self[this.self]` describes that the field `self` referencing a partially initialized object, which will become fully initialized when the field `this.self` is initialized.

With stackability, the system guarantees that an object `A` must be fully initialized if another object `B` whose initialization span covers that of `A` is fully initialized. For example, in the following code, the object pointed to by the field `this.child` must be fully initialized if the object pointed to by `this` is fully initialized.

```java
class Parent {
    var child: Child = new Child(this)
}
```

The principle of stackability dates back to delayed types [4], and followed in the freedom model [15]. However, it is not enforced in masked types [12], and the type system compensates with verbose type annotations and special typing rules.

### 2.2 Design of Constructors

The design of constructors should facilitate the enforcement of the principles of initialization. In the light of the two golden principles, we find that traditional class constructors (like in Java) are flawed in several aspects.

In traditional class constructors, there is no distinction between field initialization and field reassignment in syntax. The fields of a class are usually declared in the class body, and then initialized by assignment. From the syntax alone, it is not easy to distinguish field initialization from field reassignment. This causes several problems.

First, it makes enforcement of monotonicity more difficult. As we learned above, monotonicity can only be broken with reassignment, thus special rules should be enforced for field reassignment. For example, while a field may be initialized with an object under initialization, it is generally unsafe to do so for reassignment. The indistinguishability in syntax makes it harder to give different rules to field reassignment and field initialization.

Second, it complicates the check whether all fields of a class are assigned at the end of the constructor. This is because assignment of a field may happen in an if branch or in a method, thus non-trivial analysis has to be employed to enforce the rule. In the freedom model [15], definite assignment analysis [6] is used to ensure that all fields of a class are assigned at the end of the constructor.

Third, it complicates the check that an immutable field is not reassigned. The system of masked types [12] introduces the concept of must-mask to support initialization of immutable fields, e.g., `C\d!` means that the field `d` is definitely not assigned, while a normal mask `C\d` means the field `d` may not be assigned.

The culprit of the separation of field declaration and field initialization is class constructors, as the arguments for object initialization are only available as constructor parameters. However, that is not the only possibility for the design of object initialization, e.g., with Scala-like class parameters, there is no need for explicit constructors.

We advocate Scala-like class parameters and mandatory field initialization on declaration. We have seen several classes defined this way in the previous sections, and we will adopt the design in our formal language (Section 4.1). This design has many benefits: (1) we may distinguish field initialization and reassignment in syntax, which makes it possible to give different rules to them to enforce monotonicity; (2) it enforces the principle of stackability in syntax, as fields are initialized when they are declared; (3) enforcing no reassignment of immutable fields becomes as simple as a syntactic check.
In object-oriented programming languages, sometimes programmers create convenience constructors to call the main constructor (constructor chaining). For example, Scala has the concept of secondary constructors which are supposed to eventually call the implicit primary constructor [11]. In Swift, there are designated initializers and convenience initializers, the latter are supposed to call the former [1]. Our critique of constructors does not extend to convenience constructors or secondary constructors, as they have no impact on the reasoning principles of initialization.

3 Type-and-Effect System, Informally

In this section, we discuss the main ideas behind the type-and-effect system using examples.

3.1 Potentials and Effects

Consider the following erroneous program, which accesses the field y before it is assigned:

```scala
class Point {
    var x: Int = this.m() // Point.this.m!
    var y: Int = 10
}
```

A natural idea to ensure safe initialization is to analyze the fields that are accessed at each step of initialization, and check that only initialized fields are accessed. This leads to the fundamental effect in initialization:

- **field access effect**, e.g. C.this.f!

For a particular section of code, this effect indicates that a particular field may be read in that code. However, fields may also be accessed indirectly through method calls, as the following code shows:

```scala
class Point {
    var x: Int = this.m() // Point.this.m!
    var y: Int = 10
    def m(): Int = this.y // Point.this.y!
}
```

A natural idea is to introduce method calls as effects, which act as placeholders for the actual effects that happen in the method:

- **method call effects**, e.g. C.this.m!

If we first analyze effects of the method m and map the effect Point.this.m! to the set of effects {Point.this.y!}, then we may effectively check the initialization error in the code above.

The effects approach seems promising, however, we need to handle an anti-pattern in initialization, i.e. the escape of this in the constructor, as the following code shows:

```scala
class Point {
    var x: Int = this.m()
    def m(): Int = leak(this)
}
```

In the code above, the current object escapes the constructor indirectly via the method call leak(this). Such escape is dangerous, as uninitialized fields of the object may be accessed. To check against such anti-patterns using effects, we need another kind of effect:

- **leaking effects**, e.g. C.this↑

With the three kinds of effects, it seems we arrive at an effect system to ensure safe initialization objects. Unfortunately, such a system is unsound, because it is unable to handle aliasing, as the following code shows:

```scala
class Fact {
    var b = this.a // access this.a , but no leak
}
```

In the code above, the field x is used via the alias self before it is initialized. To check such errors, we need a way to represent the aliasing information in the effect system. That leads us to the concept of potentials. Potentials represent aliasing of objects (possibly under initialization), which can lead to initialization errors. A potential encodes aliasing information in the form of paths such as C.this or C.this.f. Operations on objects pointed to by potentials may give rise to initialization errors. In the code example above, the field self takes the potential of its initializer, which is Knot.this. Now an initialization checker may take advantage of the aliasing information and report an error for the code self.x.

From this perspective, effects are triggered from potentials. They fall into three categories as discussed before:

- **field access effects**, e.g. C.this.f!
- **method call effects**, e.g. C.this.m!
- **leaking effects**, e.g. C.this↑ and C.this↑

The leaking effect has a different interpretation under this new perspective. Semantically, potentials keep track of objects currently under initialization in order to maintain a directed segregation of initialized objects and objects under initialization: the latter may point to the former, but not vice versa. A leaking effect means that the object pointed to by the potential ascends to the initialized world, and the system gives up on tracking it.

Note that field access C.this.a! and field leaking C.this.a↑ are different effects, because field access does not necessarily leak the field, as demonstrated by the following example:

```scala
class Knot {
    var a = this
    var b = this.a // access this.a , but no leak
}
```

Aliasing and leaking may also happen through methods, as the following example shows:

```scala
class Fact {
    var value = leak(this.m())
    def m() = this
}
```
A similar distinction is drawn on methods: (1) the method invocation effect $C.this.m!$ means that the method $m$ is called with the receiver this; (2) the method leaking effect $C.this.m↑$ means that the return value of the call this.$m$ is leaked.

### 3.2 Two-Phase Checking

A common issue in analysis is how to deal with recursive methods. We tackle the problem with two phase checking. In the first phase, the system computes an effect summary for methods and fields. In the second phase, the system checks that no fields are used before they are initialized. During the checking, it uses the effect summary from the first phase. For example, assume the following program:

```scala
class C {
  def g(): Int = h()
  def h(): Int = g()
}
```

In the first phase, the computed summary for the methods $h$ and $g$ is as follows:

<table>
<thead>
<tr>
<th>method</th>
<th>effects</th>
<th>potentials</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h$</td>
<td>$Foo.this.g!$</td>
<td>$Foo.this.g$</td>
</tr>
<tr>
<td>$g$</td>
<td>$Foo.this.h!$</td>
<td>$Foo.this.h$</td>
</tr>
</tbody>
</table>

In the second phase, while checking the method call $h()$, the analysis propagates the effects associated with the method $h$ until it reaches the fixed point \{ $Foo.this.g!$, $Foo.this.h!$ \}. As the set does not contain accesses to any uninitialized fields of this nor leaking of this, the program passes the check. Note that the fixed point exists because the domain of effects and potentials is finite for a given program.

### 3.3 Full-Construction Analysis

Another common issue in initialization is how to handle inheritance. The approach we take is full-construction analysis: we treat the constructors of concrete classes as entry points, and check all super constructors as if they were inlined. The analysis spans the full duration of object construction. This way, all virtual method calls on this can be resolved statically. From our experience, full-construction analysis greatly improves user experience, as no annotations are required for the interaction between sub-classes and super-classes (Section 5).

In contrast, abstract masks were introduced in the work of [12] to support the interaction between subclasses and superclasses. As can be seen from the following code example, the syntactic verbosity and cognitive overhead form an obstacle for its practicality:

```scala
class C {
  T f; mask M += f;
  void initM() effect M -> () { this.f = ...; super.initM(); }
}
```

```
class D extends C {
  T g; mask M += g;
}
```

### 3.4 Cyclic Data Structures

It is a common practice to create cyclic data structures by aliasing this. This is supported with an annotation @cold, as the following example demonstrates:

```scala
class Parent { val child: Child = new Child(this) }
class Child(parent: Parent @cold) { }
class Friend(parent: Parent @cold) { val tag = 10 }
```

The annotation @cold corresponds to cold class parameters in the formal language (Section 4.1), which is the only annotation needed in our system. The effect system will ensure that the field parent is not used directly or indirectly inside the initializer of Child. However, aliasing the field to another cold class parameter is fine, thus the code `new Friend(this.parent)` at line 2 is accepted by the system. This allows programmers to create complex aliasing structure during initialization.

The @cold annotation dates back to a long established dichotomy between initialized objects and uninitialized objects as free/committed [15] or raw/cooked [3]. Semantically, a cold object means that the object may not be initialized. Method call or field access on cold objects will trigger the effect cold↑, which means the object should be fully initialized.

But when is it safe to use parent in the class Child? The answer is: when Child.this becomes transitively initialized. Note that, at line 1, the object created by new Child(this) is not transitively initialized due to the aliasing of this, despite the fact that all fields of the object are assigned; child becomes fully initialized along with Parent.this at the end of the class body of Parent. This is what is called commitment point in the work of [15].

Our system tracks the potentials of new Child(this) as warm[Child]. All fields of a warm value are assigned, but they may hold values that are not fully initialized. The following effects are possible on warm[C]:

- `warm[C].f!`: access field $f$ of a warm object
- `warm[C].m!`: call the method $m$ on a warm object
- `warm[C]↑`: leaking of a warm object

Programmers never need to write the effects explicitly, they are inferred and checked by the system. Just like C.this↑, the effects `warm[C]↑` are illegal in the constructor, as leaking may result in initialization errors. In the following code, the method call foo() is illegal, because the receiver has the potential `warm[Child]`, and the effect `warm[Child].foo!` results in the forbidden effect cold↑:

```scala
class Parent {
  var name: String = (new Child(this)).foo()
}
```
4.1 Syntax

Our language resembles a subset of Scala having only top-level classes, mutable fields and methods.

\[ P \in \text{Program} ::= (C, e) \]
\[ C \in \text{Class} ::= \text{class } C(f:D) \{ \bar{F} \ M \} \]
\[ F \in \text{Field} ::= \text{var } f:C = e \]
\[ e \in \text{Exp} ::= x \mid \text{this} \mid e.f \mid e.m(\bar{e}) \]
\[ M \in \text{Method} ::= \text{def } m(x:C) : D = e \]
\[ x, y, z \in \text{Variable} \]
\[ f, \hat{f}, \check{f} \in \text{FieldName} \]
\[ m \in \text{MethodName} \]
\[ C, D, E \in \text{ClassName} \]

The language has two salient features compared to conventional class-based programming languages:

- there are no class constructors;
- in field declaration \((F)\), field initializer is mandatory.

With this syntax, we get stackability for free, as all fields must be assigned at the end of the class body. Meanwhile, it enables us to give a different typing rule to field reassignment from field initialization to enforce monotonicity of initialization.

A program \(P\) is a pair of a list of class definitions and a term representing the execution entry point. A class is composed of class parameters \((\hat{f} : D)\), body fields \((\text{var } f : C = e)\) and methods \((\text{def } m(x : C) : D = e)\). A class parameter \(\hat{f}\) is also a field of its defining class. By default, we use \(f\) to range over all fields, and \(\hat{f}\) to range only over class parameters. The symbol \(\check{f}\) refers to class parameters that may accept a value that is not transitively initialized. These class parameters are called cold class parameters. The tilde annotation on \(\hat{f}\) is only used in the type-and-effect system, it does not have runtime semantics. That is the only annotation that is required in source code in our system.

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

4.2 Effects and Potentials

As seen from Figure 2, the definition of potentials \((\pi)\) and effects \((\phi)\) depends on roots \((\beta)\). Roots are the shortest extendable path that represents an alias of a value that may not be transitively initialized. There are two roots in the system:

- \(C.this\) represents aliasing of \(\text{this}\) inside class \(C\).
- \(\text{warm}[C]\) represents aliasing of a value of class \(C\), all fields of which are assigned, but it may not be transitively initialized.

Potentials \((\pi)\) represent aliasing information. They extend roots with field aliasing \(\beta.f\) and method aliasing \(\beta.m\). Field aliasing \(\beta.f\) represents aliasing of the field \(f\) of \(\beta\), while method aliasing \(\beta.m\) represents aliasing of the return value of method \(m\) with the receiver \(\beta\). The potential \(\text{cold}\) refers to a value that may not be initialized, which is used to represent the potential of cold class parameters.

Effects \((\phi)\) include field accesses, method calls and leakings of possibly uninitialized values. A leaking effect is represented with \(\pi\uparrow\), which means the leaking of the potential \(\pi\). The field access effect \(\beta.f!\) means that the field \(f\) is accessed on \(\beta\). The method call effect \(\beta.m!\) means the method \(m\) is called on \(\beta\).

To simplify our presentation, we use the syntax \(\Pi \uparrow\) to denote the set \(\{ \pi \uparrow \mid \pi \in \Pi \}\).

4.3 Expression Typing

Expression typing (Figure 3) has the form \(\Xi; \Gamma; C \vdash e : T\). \((\Phi; \Pi)\), it means that the expression \(e\) in class \(C\) under the environment \(\Gamma\), can be typed as \(T\), it produces effects \(\Phi\) and has the potential \(\Pi\). Generally, when typing an expression, the effects of sub-expressions will accumulate, while potentials may be refined (via selection), leaked (used as arguments to methods) or absorbed (when used as arguments to cold class parameters).

The definitions assume helper methods \(\text{fieldType}(\Xi, C, f)\), \(\text{methodType}(\Xi, C, m)\) and \(\text{constructorType}(\Xi, C)\) to look up in class table \(\Xi\) the type, respectively, of field \(C.f\), of method \(C.m\) and of the constructor of \(C\).

In the typing rule \(T\text{-Var}\), the effects are empty as accessing a variable cannot cause any runtime error. The potential is empty because the design of the system ensures that variables are transitively initialized, thus they do not need to be tracked in the system.

In the typing rule \(T\text{-This}\), the effect is empty as expected, and the potential is \(C.this\), as it aliases \(\text{this}\) in class \(C\).
The helper method is defined in Figure 2. There are several cases:

- selection of cold class parameter $\tilde{f}$ on $\beta$
- selection of non-cold class parameter $\hat{f}$ on $\beta$
- selection of body field $f$ on $\beta$
- selection of a field on $\pi$ where $\pi.f$ is too long

In the first case, the field $\tilde{f}$ may hold a value that is not transitively initialized, thus the potential is represented as cold. The effect is empty, as class parameters are always initialized before the class body is executed.

For the same reason, in the second case, the effects are empty. The potentials are empty because a non-cold class parameter may only hold a value that is transitively initialized, thus we do not need to track it in the system.

In the third case, selecting a body field $f$ produces the effect $\beta.f!$ due to field access, and the potential $\beta.f$ due to the fact that the field $f$ may hold a value which is not transitively initialized.

In the last case, if the length of $\pi.f$ exceeds the maximum length of potentials, the system just leaks the potential $\pi$, which is equivalent to say that $\pi$ is a transitively initialized value, thus the potential of the selection is empty. The system restricts the length of potentials to make the domain finite. In our formalization, we set the length to 2. In implementation, the maximum length of potentials may be parameterized.

The typing rule T-Call, first checks the receiver $e_0$ and the arguments $e_i$. Then it calls the helper function $call(T_0, m, \Pi)$. The definition of $call$ (defined in Figure 2) distinguishes two cases:

- the receiver is $\beta$
- the receiver is $\pi$ where $\pi.m$ is too long

In the first case, it produces the effect $\beta.m!$ and potential $\beta.m - \text{remember } \beta.m$! is a placeholder to say all effects associated with the method $m$, and $\beta.m$ to all potentials associated with the return value of the method $m$.

In the second case, the system just leaks the potential $\pi$, just as the case of selection. The resulting potential is empty, because both the receiver and arguments are fully initialized. The semantic justification for this rule is based on a property called scoped reachability, which we explain and formalize in the technical report.

Note that in the current system, method arguments must be transitively initialized, this fact is expressed in the method $call$ by leaking all potentials of the arguments as effects.

To type check new-expressions, the typing rule T-New first type checks all arguments, then it calls the helper method $init(C, \tilde{f}_i = \Pi_i)$. The helper method (defined in Figure 2) distinguishes two cases:

- the class parameters of $C$ accept values under initialization and there exists at least one corresponding argument whose potential is non-empty.
- either class $C$ does not accept values under initialization or all potentials for cold class parameters $\tilde{f}$ are empty.

In the first case, it leaks all potentials that do not correspond to cold class parameters, which is equivalent to say that these arguments are transitively initialized. The potentials corresponding to cold class parameters $\tilde{f}$ are absorbed by the fields $\tilde{f}$ in the resulting potential $\text{warm}[C]$.

In the second case, it leaks all potentials of the arguments to ensure that they are transitively initialized. The result
Expression Typing

\[ x : T \in \Gamma \]
\[ \Xi; \Gamma; C \vdash x : T ! (\emptyset, \emptyset) \]  

T-VAR

\[ \Xi; \Gamma; C \vdash \text{this} : C ! (\emptyset, \{C.\text{this}\}) \]  

T-THIS

\[ \Xi; \Gamma; C \vdash e : D ! (\Phi, \Pi) \quad (\Phi', \Pi') = \text{select}(\Pi, f) \quad E = \text{fieldType}(\Xi, D, f) \]
\[ \Xi; \Gamma; C \vdash e.f : E ! (\Phi \cup \Phi', \Pi') \]  

T-SEL

\[ (x_i : T_i, D) = \text{methodType}(\Xi, T_0, m) \quad (\Phi', \Pi') = \text{call}(T_0, m, \Pi) \]
\[ \Xi; \Gamma; C \vdash e_0, m(\Xi) : D ! (\Phi \cup \Phi_{\text{in}} \cup \Pi_{\text{in}} \cup \Phi', \Pi') \]  

T-CALL

\[ \bar{f_i} : T_i = \text{constrType}(\Xi, C) \quad \Xi; \Gamma; C \vdash e_1 : T_1! (\Phi_i, \Pi_i) \quad (\Phi', \Pi') = \text{init}(C, \bar{f_i} = \Pi_i) \]  

T-NEW

\[ \Xi; \Gamma; C \vdash e_0 : T_0! (\Phi_0, \Pi_0) \quad T_1 = \text{fieldType}(\Xi, T_0, f) \]
\[ \Xi; \Gamma; C \vdash e_1 : T_1! (\Phi_1, \Pi_1) \quad \Xi; \Gamma; C \vdash e : T_2! (\Phi_2, \Pi_2) \]  

T-BLOCK

\[ \Xi; \Gamma; C \vdash \text{null} : T ! (\Phi, \Pi) \]  

Figure 3. Expression Typing

Finally, to type check a block expression \(e_0.f = e_1; e\), the typing rule T-Block first type checks \(e_0\), \(e_1\) and \(e\) separately. Then in the final effect, it leaks the potentials \(\Pi_i\) for \(e_1\), which ensures that the value of \(e_1\) is transitively initialized. This is how monotonicity of initialization is enforced in the system.

4.4 Definition Typing

Definition typing (Figure 4) defines how programs, classes, fields and methods are checked. The check happens in two phases:

1. **first phase**: conventional type checking is performed and effect summaries are computed;
2. **second phase**: effect checking is performed to ensure initialization safety.

The two-phase checking is reflected in the typing rule T-Prog. When type checking a program \((\mathcal{C}, e)\), first each class is checked separately for well-typing and the effect summary for fields \(\Delta_{\mathcal{C}}\) and methods \(S_{\mathcal{C}}\) is computed. Then effect checking is performed modularly on each class with the help of the effect table \(E\).

The typing rule T-Prog also checks that the top-level expression \(e\) is well-typed with the empty environment and the type Null for this. The effect signature means that the expression may not use this, otherwise the potentials and effects of \(e\) cannot be both empty. The usage of Null for the type of this for the top-level expression \(e\) corresponds to the semantic trick to use a dummy value of the type Null for this at the top-level. It unifies the semantics and typing rules for top-level expressions and expressions inside classes, which simplifies the meta-theory.

When type checking a class, the rule T-Class checks that the body fields and methods are well-typed, and the associated effects and potentials are computed. The effects and potentials associated with a field are the effects and fields of the right-hand-side expression. The effects and potentials associated with a method are the effects and fields of the body expression of the method. The effect summaries are used during the second phase in T-Check, where it checks that given the already initialized fields, the effects on the right-hand-side of each field initialization are allowed.

The typing rule T-Field checks the right-hand-side expression \(e\) in an empty typing environment, as there are no variables in a class body (class parameters are fields of their defining class). In the typing rule T-Method, the method parameters \((\bar{x} : T)\) are used as a typing environment to check the method body. They correspond to the semantics for field initialization and method calls respectively.

potential is empty, as it must be a transitively initialized value.

When type checking a class, the rule T-Class checks that the body fields and methods are well-typed, and the associated effects and potentials are computed. The effects and potentials associated with a field are the effects and fields of the right-hand-side expression. The effects and potentials associated with a method are the effects and fields of the body expression of the method. The effect summaries are used during the second phase in T-Check, where it checks that given the already initialized fields, the effects on the right-hand-side of each field initialization are allowed.
4.5 Effect Checking

The effect checking judgment $E; \Omega; C \vdash \Phi$ (Figure 5) means that the effects $\Phi$ are permitted inside class $C$ when the fields in $\Omega$ are initialized. In the checking, it first computes the fixed-point of $\Phi$ with the helper function $\text{fix}$. Then it checks that there is no leaking of this, warm values, or cold values which may not be fully initialized. Finally, it checks that each accessed field is in the set $\Omega$, i.e., only initialized fields are used.

The fixed-point computation is relatively simple: it just propagates the effects recursively until a fixed-point is reached. The fixed-point always exists as the domain of effects $\Phi$ is finite. In Figure 5, we only show fixed-point computation for effects, the fixed-point computation for potentials looks similar, it suffices to replace $\Phi$s with $\Pi$s. For simplicity, we use the notation $E \vdash \Phi \leadsto \Phi'$ to mean for each $\phi \in \Phi$, perform the propagation and then union all the results for each $\phi$.

The main step in fixed-point computation is the propagation of effects and potentials. In effect propagation $E \vdash \phi \leadsto \Phi$, field access $\beta.f!$ is an atomic effect, thus it propagates to the empty set. For leaking effect $\pi \vdash$, we first propagate the potential $\pi$ to a set of potentials $\Pi$, and then leak each potential in $\Pi$. For a method call effect $C.this.m!$, it looks up the effects associated with the method from the effect table. For a method call effect $\text{warm}[C].m!$, it first looks up the effects $\Phi$ associated with the method $m$, and then replace $C.this$ with $\text{warm}[C]$ in $\Phi$.

In potentials propagation $E \vdash \pi \leadsto \Pi$, $\beta$ and cold propagate to empty, as they do not contain proxy aliasing information in the effect table. For a field potential like $C.this.f$, it just looks up the potentials associated with the field $f$ from the effect table. For a method potential $C.this.m$, it looks up the potentials associated with the method $m$ from the effect table. For potentials $\text{warm}[C].f$ and $\text{warm}[C].m$, first it looks up the potential associated with $f$ and $m$ respectively, then it replaces $C.this$ in the potentials with $\text{warm}[C]$. 

---

**Figure 4. Definition Typing**

\[
\begin{align*}
\text{Program Typing} & \quad \vdash P \\
\Xi = C \mapsto C & \quad \Xi(Null) = \text{class Null} & \quad \Xi; \emptyset; Null \vdash e : D \! \! (\emptyset, \emptyset) \\
\Xi \vdash C ! (\Delta_c, \Sigma_c) & \quad E = C \mapsto (\Delta_c, \Sigma_c) & \quad \Xi; E \vdash C \\
\end{align*}
\]

\[\begin{align*}
\text{Class Checking} & \quad \Xi; E \vdash C \\
(\Delta_\_\_) = E(C) & \quad E; (f_1, \cdots, f_{i-1}); C \vdash \Delta(f_i).f_i.st & \quad \Xi; E \vdash \text{class } C(f_i.D) \{ F \ M \} \\
\end{align*}\]

\[\begin{align*}
\text{Class Typing} & \quad \Xi \vdash C ! (\Delta, S) \\
\Xi; C \vdash F_i ! (\Phi_i, \Pi_i) & \quad \Delta = f_i \mapsto (\Phi_i, \Pi_i) & \quad \Xi; C \vdash M_i ! (\Phi_i, \Pi_i) & \quad S = m_i \mapsto (\Phi_i, \Pi_i) & \quad \Xi \vdash \text{class } C(f_i.D) \{ F \ M \} ! (\Delta, S) \\
\end{align*}\]

\[\begin{align*}
\text{Field Typing} & \quad \Xi; C \vdash F ! (\Phi, \Pi) \\
\Xi; \emptyset; C \vdash e : D ! (\Phi, \Pi) & \quad \Xi; C \vdash \text{var } x : D = e ! (\Phi, \Pi) \\
\end{align*}\]

\[\begin{align*}
\text{Method Typing} & \quad \Xi; C \vdash M ! (\Phi, \Pi) \\
\Xi; x:\ll T; C \vdash e : E ! (\Phi, \Pi) & \quad \Xi; C \vdash \text{def } m(x:\ll T) : E = e ! (\Phi, \Pi) \\
\end{align*}\]
4.6 Extensions
The value of a formal system depends crucially on its extensibility. We describe two extensions that are implemented in the prototype (Section 5).

**Functions.** Nowadays most languages combine object-oriented programming with functional programming, such as Java, Scala, Swift. To support functions, we add a new potential `Fun(Φ, Π)`, where Φ is the set of effects to be triggered when the function is called, while Π is the set of potentials for the result of the function call. The effect domain is still finite, as the set of function potentials is constrained by the number of function literals in a given program.

The addition improves expressiveness. For example, it enables the following code, which is rejected in Swift:

```java
class Rec {
    val even = (n: Int) => n == 0 || odd(n - 1)
    val odd = (n: Int) => n == 1 || even(n - 1)
    val flag: Boolean = odd(6)
}
```

In functional programming, the recursive binding construct `letrec` may introduce similar initialization patterns as the code above. With the latest checker [13], OCaml still does not support the code above in the same `let rec`.

**Dependent Potentials.** Inner classes [8] introduce more complexity. A reasonable initialization pattern found in Scala code is the interaction between inner and outer classes [9], as illustrated with the example from the Dotty compiler:

```java
class Trees {
    private var counter: Int = 0
    class ValDef (outer: Trees) { outer.counter += 1 }
    val theEmptyValDef = new ValDef(this)
}
class ValDef { counter += 1 }
```

The code above is semantically equivalent to the following after lowering:

```java
class Trees {
    var counter: Int = 0
    val theEmptyValDef = new ValDef(this)
}
```

In the formal system, the code above has to be rejected even if we automatically mark `outer` as `@cold`, because we cannot access the field `outer.counter` of the cold object `outer`. To support the code, we would like to record the aliasing information `outer = Trees.this` in the system. That is the idea of dependent potentials.
The dependent potential $C[f_i = \Pi_i]$ denotes a warm object of the type $C$, where the cold class parameters $f_i$ are bound to the potentials $\Pi_i$. We also need to introduce object construction effects $C[f_i = \Pi_i].init!$ to check the effects that may happen on the actual arguments to cold class parameters. For example, in the code above, summarizing the class ValDef will associate the effect ValDef.this.outer.counter with the constructor. Now in checking the class Trees, the effect expands as follows:

$$ValDef[outer = Trees.this].init! \
\Rightarrow ValDef[outer = Trees.this].outer.counter! \
\Rightarrow Trees.this.counter!$$

As the field counter is already initialized at the point, the code above will be accepted by the checker.

Naive addition of dependent potentials will make the analysis non-terminating, as the following example shows:

```scala
class C(c: C @cold) { val c2 = new C(this) }
```

The non-termination can be seen from the expansion:

$$C[c = C.this].init! \
\Rightarrow C[c = C[c = C.this]].init! \
\Rightarrow C[c = C[c = C[c = C.this]]].init!$$

We resort to a standard technique in abstract interpretation, widening [2]. As a dependent potential is always warm, it suffices to widen a dependent potential to warm if it exceeds some size limit, e.g.:

$$C[c = C.this].init! \
\Rightarrow C[c = C[c = C.this]].init! \
\Rightarrow C[c = warm[c]].init!$$

This guarantees that the expansion of effects always reaches a fixed point.

## 5 Evaluation

We implement a prototype of the type-and-effect system as a compiler plugin for the Scala 3 compiler, Dotty [10]. We evaluate the prototype on a significant number of real-world projects. To contrast, we also implement a prototype of the freedom model [15]. We collect the warnings that each system reports, without any modification of the source code.

The results are shown in Figure 6, where we can see that our system reports far fewer false positives on the projects. For the ScalaTest project, our system reports 8 true positives and we filed a bug report. In the following, we discuss several initialization examples we encountered in the experiment.

### 5.1 Supported Examples

The following code is a common pattern that our analysis can easy handle, while the freedom model falls short:

```scala
abstract class TokensCommon {
  private val map = mutable.Map.empty
  def enter(k: Int, v: String) = map(k) = v
}
object Tokens extends TokensCommon {
  final val CASE = 28; enter(CASE, "case")
  final val WITH = 26; enter(WITH, "with")
}
class A { val a = "Bonjour"; val b: Int = a.size }
class B extends A { override val a = "Hi" }

The code above will throw a null-pointer exception at runtime when initializing the field A.b, as the code.a.size has to access the field A.a, which is not yet initialized.

Meanwhile, in Scala the fields of a trait are implemented in the class that extends the trait, and initialized by a method call on this. Our system can handle such semantic subtlety easily without any annotations.

Here is another common pattern that motivates us to check the full construction duration of an object:

```scala
class Base { def g(): String = "hello" }
class Foo extends Base { val a = this.g() }
class Bar extends Base {
  val b: String = "b"
  override def g(): String = this.b
}
```

This program is correct. However, in the freedom model, in order to call g() in Foo, the method Base.g has to be annotated @free. For soundness, the overriding method Bar.g has to be annotated @free too: but now it may not access the field this.b in the body of the method Bar.g.

### 5.2 Challenging Examples

One design goal of our system is to keep the type system intact. Consequently, we require that all arguments to methods are fully initialized, which is in line with good initialization practices. Otherwise, new types must be introduced in the compiler to handle safe method overriding. Problems related to type inference and polymorphism will ensue.

However, during the experiment we encountered some reasonable code patterns that our system cannot handle, which we find interesting to discuss. The following code about LazyList construction is one such example:

```scala
trait LazyList[A] { ... }
implicit def helper[A](l: LazyList[A]): Helper[A] = new Helper(l)
class Helper[A](l: => LazyList[A]) {
  def #:: [B :>: A] (elem: => B): LazyList[B] = ...
}
class Test {
```

1We will submit an artifact for reproduction of the experimental results.
2https://github.com/scalatest/scalatest/issues/1481
Figure 6. Experiments on Dotty community projects.

In the code above, inside the class `Test`, we use `b` (before it is initialized) as a by-name argument to initialize the field `a`. Similar code patterns also appear in by-name implicits 3 and parser combinators.

Fortunately, in Scala, the example above can be supported with the language feature `inlining` 4. It suffices to add the modifier `inline` to the method `helper` and annotate the constructor parameter of the class `Helper` to accept values under initialization. However, whether the inlining trick works in general is still a question.

The following code is a common pattern in the Dotty compiler to create complex types, where we have to resort to `@unchecked`:

```scala
class RecType(parentExp: RecType => Type) {
  val parent = parentExp(this)
}
```

As compilers are performance-sensitive, we cannot just add the modifier `lazy` to the field `parent` to silence the warning about the leaking of `this`.

The lazy trick is a panacea when the initialization checker falls short, with the slight danger of turning actual initialization errors into non-termination.

6 Related Work

Our work takes inspiration from several milestone papers on the problem of initialization. Due to space limit, we only mention the most related work here.

Fähndrich et al [3] introduce raw types like $T_{\text{raw}(S)}$ — a value of such a type is possibly under initialization, and all fields up to the superclass `S` are initialized. To call a method on `this`, the target method has to be annotated with the modifier `Raw`. Class fields may not hold raw values, thus it is impossible to create cyclic data structures without resorting to `null`. To overcome the limitation, they introduce `delayed` types [4]. The system ensures that the initialization of objects forms stacked time regions.

Qi and Myers [12] introduce an expressive type system for initialization based on masked types. The principle of monotonicity is upheld in the system. However, the principle of stackability is not enforced. In the system, methods and constructors have effects, which are essentially the initialization status of `this` before and after the call. Their system does not include concepts similar to our `potentials`.

Summers and Müller [15] show that initialization of cyclic data structures can be supported in a light-weight type system. The original system violates monotonicity. In the implementation, they extend the system with `committed-only fields` and enforce monotonicity for those fields to support the usage of initialized fields.

The initialization in X10 [16] employs an inter-procedural 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 detail. To call virtual methods on `this`, annotations are required on method definitions.

The Billion-Dollar Fix [14] 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.

3https://docs.scala-lang.org/sips/byname-implicits.html
4https://dotty.epfl.ch/docs/reference/metaprogramming/inline.html
7 Conclusion
In this paper, we propose a syntactic redesign of constructors based on two principles: stackability and monotonicity. Furthermore, we develop a type-and-effect system that effectively ensures initialization safety with almost zero syntactic overhead. We show how our system supports common initialization patterns, such as the use of already initialized fields, calling methods on this and aliasing of this to create cyclic data structures.

Acknowledgments
We thank the anonymous reviewers of POPL2020 for their constructive comments. We gratefully acknowledge funding by the Swiss National Science Foundation under Grant 200021_166154 (Effects as Implicit Capabilities).

References