Initialization Patterns in Dotty

Fengyun Liu
Aggelos Biboudis
Martin Odersky
EFPL, Lausanne, Switzerland
first.last@epfl.ch

Abstract

Safe object initialization is important to avoid a category of runtime errors in programming languages. In this paper, we provide a case study of the initialization patterns on the Dotty compiler. In particular, we find that calling dynamic-dispatching methods, the usage of closures and instantiating nested classes are important for initialization of Scala objects. Based on the study, we conclude that existing proposals for safe initialization are inadequate for Scala.

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

Keywords Object initilization, Scala

ACM Reference Format:
Fengyun Liu, Aggelos Biboudis, and Martin Odersky. 2018. Initialization Patterns in Dotty. In *Proceedings of the 9th ACM SIGPLAN International Scala Symposium (Scala '18), September 28, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3241653.3241662

1 Introduction

Errors in object initialization, such as the usage of a field before its initialization, can be difficult to spot. This can be illustrated by the following code example:

```
trait Zen {
  val name: String
  val message = "hello, " + name
}
class Tao extends Zen {
  val name = "Tao"
}
```

The code above may look correct, but the result is surprising after running it:

```
println((new Tao).message)       // hello, null
```

We would expect the code to print "hello, Tao" instead! The problem is that when the field `message` is getting initialized, the field `name` in `Tao` is not yet initialized itself.

However, the following class, `Nirvana`, has no problem, as the fields declared in the primary constructor are initialized before the parent (i.e. `Zen`) is initialized.

```
class Nirvana(val name: String) extends Zen
println(new Nirvana("Nirvana")) // hello, Nirvana
```

There are several proposals for checking initialization in object-oriented languages [Summers and Müller 2011; Zibin et al. 2012], but none exists for Scala. Scala is a multi-paradigm programming language that offers a wide range of features, like traits and path-dependent types. Our ultimate goal is to augment the Scala language specification to ensure initialization safety.

For this purpose, we carry out a case study on Dotty, the prospective Scala compiler [Odersky et al. 2013]. Dotty has about 68KLOC. Given the complexity of Dotty, we assume it covers most important initialization patterns found in Scala programs.

We classify discovered initialization patterns into four different groups, based on the information required to reason about safe initialization:

- Category A: Intra-Class interaction
- Category B: Child-Parent interaction
- Category C: Inter-Class interaction
- Category D: Outer-Inner interaction

We find that calling dynamic-dispatching methods, the usage of closures and instantiating nested classes are important for initialization of Scala objects.

2 Category A: Intra-Class Interaction

In this category, we classify all initializations that can be reasoned locally, that is without the need to consider class hierachy or other external classes.

2.1 A1: Access Local Fields

This is a common pattern, where an initialized field is used to initialize other fields, as the following code shows [1]:

```
object Flags {
```

[1] All the code comes from the Dotty compiler, however, they are adapted for better presentation.

Scala '18, September 28, 2018, St. Louis, MO, USA

ACM ISBN 978-1-4503-5836-1/18/09…$15.00
https://doi.org/10.1145/3241653.3241662

```
  private final val TERMindex = 0
  private final val TERMS = 1 << TERMindex
}
```

The compiler should check that the field `TERMindex` is initialized before use.

2.2 A2: Call Local Methods

In this pattern, a local method is called during initialization. In the following code, the local method `nxId` is called to assign a unique ID for each instance of the `Tree`:

```
abstract class Tree[-T >: Untyped] {
  private def nxId = {
      Trees.nextId += 1
      Trees.nextId
  }
  private var myUniqueId: Int = nxId
}
```

For safety, the compiler should check that calling the method `nxId` will not accessing any non-initialized fields.

2.3 A3: Use Closures

Usage of closures during initialization is common. The usage results not only from direct code but also from syntactic desugaring, `for`-comprehensions are translated into method calls with closures as parameters. The following code shows that during initialization of objects of the class `Lifter`, a closure which references the field `thisPhase` escapes via the method call `ctx.atPhase`.

```
class Lifter(thisPhase: MiniPhase)(...ctx) {
  ctx.atPhase(thisPhase) { implicit ctx =>
    // ...
    generateProxies()(ctx.withPhase(thisPhase.next))
    liftLocals()(ctx.withPhase(thisPhase.next))
  }
}
```

The compiler should be able to check whether a closure accesses any non-initialized fields when it escapes to external methods.

3 Category B: Child-Parent Interaction

In this category, we classify all initializations that need to take inheritance and class hierachy into consideration.

3.1 B1: Call Deferred Methods

In this pattern, an abstract method may be called during initialization, as the following code shows:

```
abstract class AbstractFile {
  def name: String
  val extension: String = Path.extension(name)
}
```

The compiler has to ensure that subclasses of `AbstractFile` that implement the method `name` should not use, directly or indirectly, any possibly non-initialized fields of the object (declared in parent or child class).

3.2 B2: Access Deferred Fields

In this pattern, a deferred field is used during initialization of a parent, as the following code shows:

```
abstract class NamedType {
  val prefix: Type
  assert(
    prefix.isValueType || (prefix eq NoPrefix)
    s"invalid prefix $prefix"
  )
}
```

In this case, the compiler has to make sure that the child class initializes the field `prefix` before initializing the parent `NamedType`.

3.3 B3: Call Overriden Methods

In this pattern, a parent may call a method that is overriden in a child class, as the following code demonstrates:

```
class DottyUnpickler {
  val a = treeSectionUnpickler(...)

  def treeSectionUnpickler(...) = {
    new TreeSectionUnpickler(...)
  }
}
class TastyUnpickler extends DottyUnpickler {
  override def treeSectionUnpickler(...) =
    new QuotedTreeSectionUnpickler(...)
}
```

The compiler has to ensure that all overriding methods of `treeSectionUnpickler` do not directly or indirectly use any possibly non-initialized fields of the object (declared in parent or child class).

3.4 B4: Access Parent Fields

In the following code example, the child class `TreeBuffer` accesses the parent field `bytes` in `TastyBuffer` during initialization:

```
class TastyBuffer(initialSize: Int) {
  var bytes = new Array[Byte](initialSize)
}
class TreeBuffer extends TastyBuffer(50000) {
  val initialOffsetSize = bytes.length / ...
}
```

While in most cases we may assume that parent fields are fully initialized, for lazy fields and fields that refer to local closures or objects of nested classes it is not the same case. Thus, the compiler should be able to distinguish parent fields that can be safely accessed from children from fields that may cause initialization problems if accessed from children.

3.5 B5: Call Parent Methods

In the following code, the initialization of `JavaTokens` calls the method `enter` defined in the parent class `TokensCommon`:

```
abstract class TokensCommon {
  val tokenString = new Array[String](maxToken + 1)
  def enter(token: Int, str: String) = {
    tokenString(token) = str
  }
}
object JavaTokens extends TokensCommon {
  final val INSTANCEOF = 101
  enter(INSTANCEOF, "instanceof")
}
```

Generally, accessing parent methods is unsafe, as they may call methods defined in some child class through dynamic dispatch. The latter may access fields in child classes that are not yet initialized. The compiler should be able to differentiate methods that can be safely called while initializing a child, from those that cannot be called.

4 Category C: Inter-Class Interaction

In this category, we classify all initializations that need to take external classes into consideration.

4.1 C1: Escape of `this` to `new`

In the following code, `this` escapes to a new instance of `initialCtx`:

```
class ContextBase {
  val ictx = new InitialContext(this, ...)
}
class InitialContext(val base: ContextBase, ...)
```

As `this` is partially initialized, we must treat `ictx` as partially initialized as well. It implies it is unsafe to access any members of `ictx` that may indirectly access uninitialized fields of the current object referred by `this`, both in the current class or from child classes.

In the class `InitialContext`, the compiler must ensure that during initialization it doesn't directly or indirectly access any fields on the parameter `base`.

4.2 C2: Call Methods on Escaped `this`

In the following example, `this` of `TastyPickler` escapes to the class `TreePickler`, which in turn calls the method `newSection`:

```
class TastyPickler(val rootCls: ClassSymbol) {
  val sections = new mutable.ArrayBuffer[...]
  def newSection(name: String, buf: TastyBuffer) =
    sections += ...

  val treePkl = new TreePickler(this)
}
class TreePickler(pickler: TastyPickler) {
  val buf = new TreeBuffer
  pickler.newSection("ASTs", buf)
}
```

To ensure safety, the compiler has to tell which methods are safe to call on `pickler` during the initialization of a `TreePickler` object.

4.3 C3: Escape of `this` to Methods

These are hard cases of initialization. A typical example is the initialization of `RecType`, in which `this` escapes via the function `parentExp`:

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

In Dotty, the types `MethodType`, `HKTypeLambda` and `PolyType` also use the same pattern to decouple the initialization of 3-way cyclic data structures.

The following code is another example where `this` escapes to a method:

```
class Run(...ictx: Context) {
  def rootContext(implicit ctx: Context) = {
    ...
    start.setRun(this)
  }
  var myCtx = rootContext(ictx)
}
```

To ensure safe initialization in the aforementioned example, the compiler has to ensure that no uninitialized fields of `this` are accessed directly or indirectly:

1. during the method call `setRun`
2. via usage of `start` during the initialization of `this`
3. via aliases to `start` (and `this` if it's aliased in `setRun`) during the initialization of `this`

The checking requirement is daunting. Given the infrequency and ad-hoc nature of this initialization pattern, we think it is fine to make this pattern an exceptional case through some escape like `@unchecked`.

4.4 C4: Escape of Fully Initialized `this`

In the following code example, `this` safely escapes as all fields of the object are initialized:

```
final class RealProfiler(reporter: ProfileReporter)
{
  ...
  reporter.header(this) // fields initialized above
}
```

Note it is important that the class `RealProfiler` is effectively final, otherwise it is considered an unsafe escape, since calling methods on `this` may reach child fields that are not yet initialized.

5 Category D: Outer-Inner Interaction

In this category, we classify all initializations related to nested classes, which could be defined in parent class or current class.

5.1 D1: Instantiation of Nested Class

The following code shows that an instance of the nested class `EmptyValDef` is created during initialization:

```
object Trees {
  class ValDef {
    def setMods(x: Int) = ...
  }
  class EmptyValDef extends ValDef {
    setMods(5)
  }
  val theEmptyValDef = new EmptyValDef
}
```

The compiler should check that the method `setMods` can be safely called from `EmptyValDef`. It is not sufficient to check that it only accesses fields that are initialized in `ValDef`. In addition, the compiler has to ensure that `setMods` does not directly or indirectly access non-initialized fields of the object `Trees`.

5.2 D2: Instantiation of Nested Class in Parent

In the following example, an instance of the nested class `Info` is created during the initialization of an object of the class `ClassifiedNameKind`:

```
abstract class NameKind(...) {
  class Info extends NameInfo { ... }
}
class ClassifiedNameKind extends NameKind(...) {
  val info = new Info
}
```

There are several safety concerns here:

1. The initializer of `Info` may access methods of the class `NameKind`, which may indirectly access non-initialized fields of `ClassifiedNameKind` through dynamic dispatch.
2. Calling methods on `info` in `ClassifiedNameKind` may indirectly reach non-initialized fields.
3. Calling methods on `info` from child classes of the class `ClassifiedNameKind` may indirectly reach non-initialized fields of the child classes.
4. `info` should not be passed as a value of the type `Info` to external methods, unless it is completely safe to call any methods and access any fields on it.

5.3 D3: Call Methods on Instance of Nested Class

In the following example, we see that an instance of the nested class `FlagSet` is created and assigned to `JavaStatic`, then the method `toTermFlags` is called on `JavaStatic`:

```
object Flags {
  case class FlagSet(val bits: Long) {
    def toTermFlags =
      if (bits == 0) this
      else FlagSet(bits & ~KINDFLAGS | TERMS)
  }
  final val JavaStatic = FlagSet(31)
  final val JavaStaticTerm = JavaStatic.toTermFlags
}
```

For safety, the compiler has to ensure that calling the method `toTermFlags` on `JavaStatic` will not reach any non-initialized fields of the object `Flags`.

6 Related Work

Masked types [Qi and Myers 2009] are expressive, but verbose to use since it requires an involved manual annotation process from the programmer. The verbosity motivated more practical approaches [Summers and Müller 2011; Zibin et al. 2012].

X10 [Zibin et al. 2012] forbids leaking `this` out of the constructor during object initialization [Gil and Shragai 2009]. Therefore, it is impossible to safely initialize cyclic data structures with such a design. Dynamic dispatching is only allowed when `this` is not accessed with the annotation `@NoThisAccess`. X10 also prehibits creating an inner instance when the outer `this` is under initialization.

On the contrary, Summers *et al.* [Summers and Müller 2011] supports leaking `this` during initialization, thus it supports safe initialization of cyclic data structures. The design is simple, modular, sound and expressive. Types are used for *inter-class interactions*, while *intra-class interactions* are subject to data-flow analysis. However, dynamic dispatching is not addressed sufficiently in the paper, for which we conjecture something like `@NoThisAccess` in X10 will be needed. Inner classes is not discussed in the paper.

7 Future Work

We are prototyping a checker for Scala [2], which combines ideas from [Zibin et al. 2012] and [Summers and Müller 2011]. The design includes a) a data-flow analysis for checking intra-class and inner-outer interactions and b) type-based checking for parent-child and inter-class interactions. We are going to evaluate our prototype on real-world projects and see how much changes are required to migrate existing code.

Acknowledgments

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

[2] https://github.com/lampepfl/dotty/pull/4543

References

Joseph Yossi Gil and Tali Shragai. 2009. Are We Ready for a Safer Construction Environment?. In *European Conference on Object-Oriented Programming*. Springer, 495–519.

Martin Odersky et al. 2013. Dotty Compiler: A Next Generation Compiler for Scala. https://web.archive.org/web/20170325001401/http://dotty.epfl.ch/.

Xin Qi and Andrew C. Myers. 2009. Masked Types for Sound Object Initialization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, 53–65.

Alexander J Summers and Peter Müller. 2011. Freedom before commitment: a lightweight type system for object initialisation. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 1013–1032.

Yoav Zibin, David Cunningham, Igor Peshansky, and Vijay Saraswat. 2012. Object initialization in X10. In *European Conference on Object-Oriented Programming*. Springer, 207–231.