Safe Initialization

Scala 3 implements experimental safe initialization check, which can be enabled by the compiler option `-Ysafe-init`.

A Quick Glance

To get a feel of how it works, we first show several examples below.

Parent-Child Interaction

Given the following code snippet:

```
abstract class AbstractFile:
  def name: String
  val extension: String = name.substring(4)

class RemoteFile(url: String) extends AbstractFile:
  val localFile: String = s"${url.##}.tmp"  // error: usage of `localFile`
before it's initialized
  def name: String = localFile
```

The checker will report:

```
-- Warning: tests/init/neg/AbstractFile.scala:7:4 ----------------------------
7 |     val localFile: String = s"${url.##}.tmp"  // error: usage of
`localFile` before it's initialized
  |         ^
  |    Access non-initialized field value localFile. Calling trace:
  |     -> val extension: String = name.substring(4)    [ AbstractFile.scala:3
  |      -> def name: String = localFile                [ AbstractFile.scala:8
```

Inner-Outer Interaction

Given the code below:

```
object Trees:
  class ValDef { counter += 1 }
  class EmptyValDef extends ValDef
  val theEmptyValDef = new EmptyValDef
  private var counter = 0  // error
```

The checker will report:

```
-- Warning: tests/init/neg/trees.scala:5:14 ----------------------------------
5 |  private var counter = 0  // error
  |              ^
  |             Access non-initialized field variable counter. Calling trace:
  |              -> val theEmptyValDef = new EmptyValDef    [ trees.scala:4 ]
  |               -> class EmptyValDef extends ValDef       [ trees.scala:3 ]
  |                -> class ValDef { counter += 1 }          [ trees.scala:2 ]
```

Functions

Given the code below:

```
abstract class Parent:
  val f: () => String = () => this.message
  def message: String

class Child extends Parent:
  val a = f()
  val b = "hello"           // error
  def message: String = b
```

The checker reports:

```
-- Warning: tests/init/neg/features-high-order.scala:7:6 ---------------------
7 |  val b = "hello"           // error
  |      ^
  |Access non-initialized field value b. Calling trace:
  | -> val a = f()                                      [ features-high-order.s
  |   -> val f: () => String = () => this.message       [ features-high-order.s
  |    -> def message: String = b                       [ features-high-order.s
```

Design Goals

We establish the following design goals:

- Sound: checking always terminates, and is sound for common and reasonable usage (over-approximation)

- Expressive: support common and reasonable initialization patterns
- Friendly: simple rules, minimal syntactic overhead, informative error messages
- Modular: modular checking, no analysis beyond project boundary
- Fast: instant feedback
- Simple: no changes to core type system, explainable by a simple theory

By *reasonable usage*, we include the following use cases (but not restricted to them):

- Access fields on `this` and outer `this` during initialization
- Call methods on `this` and outer `this` during initialization
- Instantiate inner class and call methods on such instances during initialization
- Capture fields in functions

Principles

To achieve the goals, we uphold three fundamental principles: *stackability*, *monotonicity* and *scopability*.

Stackability means that all fields of a class are initialized at the end of the class body. Scala enforces this property in syntax by demanding that all fields are initialized at the end of the primary constructor, except for the language feature below:

```
var x: T = _
```

Control effects such as exceptions may break this property, as the following example shows:

```
class MyException(val b: B) extends Exception("")
class A:
  val b = try { new B } catch { case myEx: MyException => myEx.b }
  println(b.a)

class B:
  throw new MyException(this)
  val a: Int = 1
```

In the code above, the control effect teleport the uninitialized value wrapped in an exception. In the implementation, we avoid the problem by ensuring that the values that are thrown must be transitively initialized.

Monotonicity means that the initialization status of an object should not go backward: initialized fields continue to be initialized, a field points to an initialized

object may not later point to an object under initialization. As an example, the following code will be rejected:

```
trait Reporter:
  def report(msg: String): Unit

class FileReporter(ctx: Context) extends Reporter:
  ctx.typer.reporter = this                // 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` points to a transitively initialized object. Now the assignment at line 3 makes `this`, which is not fully initialized, reachable from `ctx`. This makes field usage dangerous, as it may indirectly reach uninitialized fields.

Monotonicity is based on a well-known technique called *heap monotonic typestate* to ensure soundness in the presence of aliasing [1]. Roughly speaking, it means initialization state should not go backwards.

Scopability means that there are no side channels to access to partially constructed objects. Control effects like coroutines, delimited control, resumable exceptions may break the property, as they can transport a value upper in the stack (not in scope) to be reachable from the current scope. Static fields can also serve as a teleport thus breaks this property. In the implementation, we need to enforce that teleported values are transitively initialized.

The principles enable *local reasoning* of initialization, which means:

> An initialized environment can only produce initialized values.

For example, if the arguments to an `new`-expression are transitively initialized, so is the result. If the receiver and arguments in a method call are transitively initialized, so is the result.

Rules

With the established principles and design goals, following rules are imposed:

1. In an assignment `o.x = e`, the expression `e` may only point to transitively initialized objects.

   This is how monotonicity is enforced in the system. Note that in an initialization `val f: T = e`, the expression `e` may point to an object under initialization.

This requires a distinction between mutation and initialization in order to enforce different rules. Scala has different syntax for them, it thus is not an issue.

2. Objects under initialization may not be passed as arguments to method calls.

   Escape of `this` in the constructor is commonly regarded as an anti-pattern. However, escape of `this` as constructor arguments are allowed, to support creation of cyclic data structures. The checker will ensure that the escaped non-initialized object is not used, i.e. calling methods or accessing fields on the escaped object is not allowed.

Modularity

The analysis takes the primary constructor of concrete classes as entry points. It follows the constructors of super classes, which might be defined in another project. The analysis takes advantage of TASTy for analyzing super classes defined in another project.

The crossing of project boundary raises a concern about modularity. It is well-known in object-oriented programming that superclass and subclass are tightly coupled. For example, adding a method in the superclass requires recompiling the child class for checking safe overriding.

Initialization is no exception in this respect. The initialization of an object essentially invovles close interaction between subclass and superclass. If the superclass is defined in another project, the crossing of project boundary cannot be avoided for soundness of the analysis.

Meanwhile, inheritance across project boundary has been under scrutiny and the introduction of open classes mitigate the concern here. For example, the initialization check could enforce that the constructors of open classes may not contain method calls on `this` or introduce annotations as a contract.

The feedback from the community on the topic is welcome.

Back Doors

Occasionally you may want to suppress warnings reported by the checker. You can either write `e: @unchecked` to tell the checker to skip checking for the expression `e`, or you may use the old trick: mark some fields as lazy.

Caveats

- The system cannot provide safety guarantee when extending Java or Scala 2 classes.
- Safe initialization of global objects is only partially checked.

References

1. Fähndrich, M. and Leino, K.R.M., 2003, July. *Heap monotonic typestates*. In International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO).
2. Fengyun Liu, Ondřej Lhoták, Aggelos Biboudis, Paolo G. Giarrusso, and Martin Odersky. 2020. *A type-and-effect system for object initialization*. OOPSLA, 2020.

‹ Option... TypeTest ›