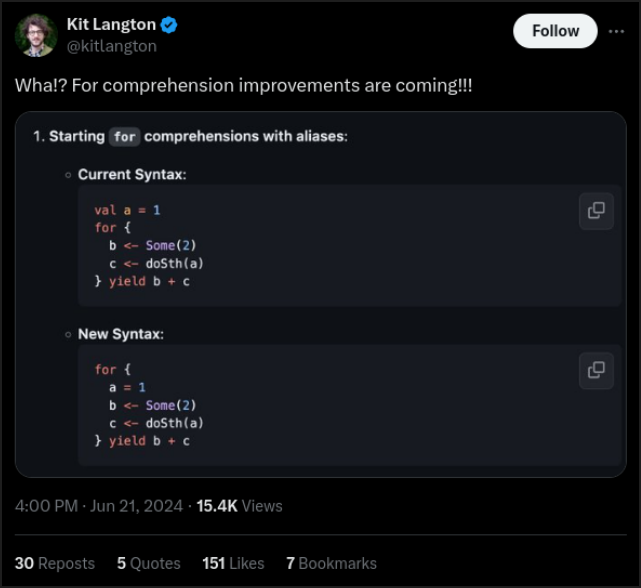
Scala 3 features you probably haven't used (yet)

Kacper Korban (VirtusLab)

July 3, 2024

Why?



Kit Langton
@kitlangton

Follow

Wha!? For comprehension improvements are coming!!!

1. Starting `for` comprehensions with aliases:
   - **Current Syntax:**

```scala
val a = 1
for {
  b <- Some(2)
  c <- doSth(a)
} yield b + c
```

   - **New Syntax:**

```scala
for {
  a = 1
  b <- Some(2)
  c <- doSth(a)
} yield b + c
```

4:00 PM · Jun 21, 2024 · **15.4K** Views

30 Reposts **5** Quotes **151** Likes **7** Bookmarks

What is this talk about?

- Overview of experimental Scala 3 features
- Motivation for adding them
- Overview of some incoming experimental features
- Plus a bonus

A solution to survive boring parts: A Jumping Frog

- You are standing in front of a pond.
- There is a path with some final number stone tiles going to the middle of the pond in a single line.
- There's also an invisible frog that is standing on a random stone tile.
- After each unit of time the frog jumps forward or backwards by exactly one tile. The frog cannot exit the path.
- Your task is to catch the frog, by standing on the same tile as the frog at some point in time.
- You can move by at most one tile in any direction on the path.
- You always move at the same time as the frog.

Come up with an algorithm that will ensure you can catch the frog and prove its correctness.

About Me

- Scala Developer at VirtusLab (formerly at Scala 3 team)
- GitHub/X: @KacperFKorban

OSS projects I have meaningful contributions to:

- scala/scala3 (compiler + scaladoc)
- softwaremill/quicklens
- softwaremill/magnolia
- VirtusLab/besom

OSS Projects I created:

- VirtusLab/Inkuire — Hoogle for Scala
- VirtusLab/avocADO — parallel for-comprehensions
- KacperFKorban/GUInep — automatic UI forms for functions

Experimental Features

namedTypeArguments

- **Feature Name:** namedTypeArguments
- **Import:** `import scala.language.experimental.namedTypeArguments`
- **TLDR:** Allows you to specify type arguments of functions by name.

namedTypeArguments — Example

This language feature allows us to write the following code:

```scala
def foo[A, B, C](a: A, b: B, c: C): Unit = ???

foo[A = Int, B = String, C = Boolean](1, "2", true)
```

An obvious benefit is improved readability.

namedTypeArguments — Motivation

Consider the following code with a potential inference error:

```
def foo[A, B, C, D, E](a: A, b: B, c: C, d: D, e: E): Unit
    = ???

foo(a1, b1, c1, d1, e1) // error can't infer type C
```

namedTypeArguments — Motivation

Consider the following code with a potential inference error:

```scala
def foo[A, B, C, D, E](a: A, b: B, c: C, d: D, e: E): Unit
    = ???

foo(a1, b1, c1, d1, e1) // error can't infer type C
```

To fix this, we would need to instantiate all type parameters:

```scala
foo[A1, B1, C1, D1, E1](a1, b1, c1, d1, e1)
```

namedTypeArguments — Motivation

Consider the following code with a potential inference error:

```scala
def foo[A, B, C, D, E](a: A, b: B, c: C, d: D, e: E): Unit
    = ???

foo(a1, b1, c1, d1, e1) // error can't infer type C
```

To fix this, we would need to instantiate all type parameters:

```scala
foo[A1, B1, C1, D1, E1](a1, b1, c1, d1, e1)
```

With named type arguments, we can write:

```scala
foo[C = C1](a1, b1, c1, d1, e1)
```

genericNumberLiterals

- **Feature Name:** genericNumberLiterals
- **Import:** `import scala.language.experimental.genericNumberLiterals`
- **TLDR:** Allows for using number literals for custom number types.

genericNumberLiterals — Usage

With this language feature, we can define a custom number type like this:

```scala
case class MyInt(value: Int):
  override def toString = value.toString()

object MyInt:
  def apply(digits: String): MyInt =
    MyInt(digits.toInt)
  given FromDigits[MyInt] with
    def fromDigits(digits: String) = MyInt(digits)

val myInt: MyInt = 123
```

genericNumberLiterals — Usage

There are 4 alternative typeclasses we can define:

```
// for parsing integers (e.g. 123)
FromDigits[T]
// for parsing decimal numbers (e.g. 123.45)
FromDigits.Decimal[T]
// for parsing integers with a radix (e.g. 0x123)
FromDigits.WithRadix[T]
// for parsing floating numbers with exponent
// (e.g. 1.23e-4)
FromDigits.Floating[T]
```

erasedDefinitions

- **Feature Name:** erasedDefinitions
- **Import:** `import scala.language.experimental.erasedDefinitions`
- **TLDR:** Allows for defining erased definitions.

erasedDefinitions — Motivation

Let's consider the following code:

```
sealed trait State
final class On extends State
final class Off extends State

class IsOff[S <: State]
object IsOff:
  given isOff: IsOff[Off] = new IsOff[Off]

class Machine[S <: State]:
  def turnedOn(using IsOff[S]): Machine[On] = new
      Machine[On]

val m = new Machine[Off]
m.turnedOn.turnedOn // error
```

The `IsOff[T]` parameter isn't used in runtime.

erasedDefinitions — Motivation

We can add the erased keyword to the IsOff implicit parameter:

```scala
sealed trait State
final class On extends State
final class Off extends State

class IsOff[S <: State]
given isOff: IsOff[Off] = new IsOff[Off]

class Machine[S <: State]:
  def turnedOn(using erased IsOff[S]): Machine[On] = new
     Machine[On]
  // def turnedOn(): Machine = new Machine()

val m = new Machine[Off]
m.turnedOn.turnedOn // error
```

This way, the IsOff[T] parameter is not present in the generated bytecode. The compiler also checks that erased parameters aren't used in computations.

erasedDefinitions — Motivation

We can also mark classes as erased. The following will achieve the same effect:

```
sealed trait State
final class On extends State
final class Off extends State

erased class IsOff[S <: State]
given isOff: IsOff[Off] = new IsOff[Off]

class Machine[S <: State]:
  def turnedOn(using IsOff[S]): Machine[On] = new
      Machine[On]
  // def turnedOn(): Machine = new Machine()

val m = new Machine[Off]
m.turnedOn.turnedOn // error
```

All usages of erased classes are marked as erased.

clauseInterleaving

- **Feature Name:** clauseInterleaving
- **Import:** `import scala.language.experimental.clauseInterleaving`
- **TLDR:** Allows for interleaving type parameter clauses and term clauses.

clauseInterleaving — Motivation

Let's consider the following code that defines a key-value store:

```
trait Key:
  type Value

class Store:
  def getOrElse(key: Key)(default: => key.Value): key.
      Value = ...
```

We would also like to allow for the default value to be passed a supertype of `key.Value`.
Like in e.g. `Option`.

```
trait Option[+A]:
  final def getOrElse[B >: A](default: => B): B
```

clauseInterleaving — Motivation

We might want to attempt to write the following code that contains a forward reference:

```
class Store:
  def getOrElse[V >: key.Value](key: Key)(default: => V):
      V = ...
```

clauseInterleaving — Motivation

We might want to attempt to write the following code that contains a forward reference:

```
class Store:
  def getOrElse[V >: key.Value](key: Key)(default: => V):
      V = ...
```

Clause interleaving allows us to write the following code:

```
class Store:
  def getOrElse(key: Key)[V >: key.Value](default: => V):
      V = ...
```

clauseInterleaving — Restrictions

There are some restrictions to clause interleaving:

- No type currying — the following is not allowed:
  ```scala
  def foo[T][U](t: T, u: U): Unit
  ```
- No clause interleaving in class definitions. (but `apply` methods are OK)
- No clause interleaving in lhs of extension methods.

Bigger experimental features

into — Background (implicitConversions)

Ols style implicit conversions (`implicit def`) give feature warnings in Scala 3 on declaration. There are plans to deprecate them in the far future.

```
case class Msg(msg: String)

implicit def stringToMsg(i: String): Msg = Msg(i) //
    feature warning
```

We should enable implicit conversions explicitly:

```
import scala.language.implicitConversions
```

into — Background (implicitConversions)

New style implicit conversions give feature warnings in Scala 3 on use.

```scala
case class Msg1(msg: String)

given Conversion[String, Msg1] with
  def apply(i: String): Msg1 = Msg1(i)

val str = "hello"
val msg1: Msg1 = str // feature warning
println(msg1)
```

We should again enable implicit conversions explicitly again:

```scala
import scala.language.implicitConversions
```

into

- **Feature Name:** into
- **Import:** `import scala.language.experimental.into`
- **TLDR:** Allows implicit conversions to be used on a specified parameter.

into — Motivation

The following code shows a common pattern in Scala:

```scala
enum MyList[+A]:
  case MyNil
  case MyCons(head: A, tail: MyList[A])

  def toList: List[A] = this match
    case MyNil => Nil
    case MyCons(h, t) => h +: t.toList

given [A]: Conversion[MyList[A], List[A]] =
  (x: MyList[A]) => x.toList

def loopOver[A](it: List[A])(f: A => Unit): Unit =
  it.iterator.foreach(f)

val myLst = MyList.MyCons(1, MyList.MyNil)
loopOver(myLst)(x => println(x)) // feature warning
```

into — Motivation

into allows us to write the following code, that will not give a feature warning at use site.

```
enum MyList[+A]:
  case MyNil
  case MyCons(head: A, tail: MyList[A])

def toList: List[A] = this match
  case MyNil => Nil
  case MyCons(h, t) => h +: t.toList

given [A]: Conversion[MyList[A], List[A]] =
  (x: MyList[A]) => x.toList

def loopOver[A](it: into List[A])(f: A => Unit): Unit =
  it.iterator.foreach(f)

val myLst = MyList.MyCons(1, MyList.MyNil)
loopOver(myLst)(x => println(x))
```

Note: into on varargs allows conversions on all varargs arguments.

namedTuples

- **Feature Name:** namedTuples
- **Import:** `import scala.language.experimental.namedTuples`
- **TLDR:** Allows you to name tuples.

namedTuples — Motivation

Consider the following code:

```scala
def analyzeString(input: String): (Int, Int) = {
  val it = input.iterator
  var l = 0
  var count = 0
  while (it.hasNext) {
    val c = it.next()
    l += 1
    if ("AEIOUaeiou".contains(c)) count += 1
  }
  (l, count)
}
```

namedTuples — Motivation

We can rewrite the code using named tuples, to make the code more readable:

```
def analyzeString(
  input: String
): (length: Int, vowelsCount: Int) = {
  val it = input.iterator
  var l = 0
  var count = 0
  while (it.hasNext) {
    val c = it.next()
    l += 1
    if ("AEIOUaeiou".contains(c)) count += 1
  }
  (length = l, vowelsCount = count)
}
```

namedTuples — Motivation

Another good use case is database operations (specifically results of joins):

```
def joinTables(
  t1: Table1,
  t2: Table2
): List[(id: Int, name: String, age: Int, address: String)
   ] =
  for
    ...
  yield
    (
      id = person.id,
      name = person.name,
      age = person.age,
      address = address.address
    )
```

Slightly different library that might benefit from this is: VirtusLab/iskra

namedTuples — Motivation

Somewhat unrelated addition of this feature: named pattern matching not only for classes.

```scala
case class City(name: String, population: Int)

def getCityInfo(city: City) =
  city match
    case c @ City(name = "Paris") =>
      "Paris is the capital of France"
    case City(population = 0) =>
      "This city is uninhabited"
    case City(name = n, population = p) =>
      s"$n has $p inhabitants"
```

modularity

- **Feature Name:** modularity
- **Import:** `import scala.language.experimental.modularity`
- **TLDR:** Allows for dependent class parameters and new type class style.

modularity — Motivation

Consider the following code:

```
class C:
  type T

def f(x: C): x.T = ...

val y: C { type T = Int }

val _: Int = f(y)
```

```
class F(val x: C):
  val result: x.T = ...

val _: Int = F(y).result // type error
```

Currently Scala is dependently typed for functions but not for classes.

modularity — Motivation

Consider the following OCaml style type class module definition:

```scala
trait Ordering:
  type T
  def compare(t1:T, t2: T): Int

class OrderedSet(val ord: Ordering):
  type Set = List[ord.T]

  def empty: Set = Nil
  extension (s: Set)
    def add(x: ord.T): Set = ...

object intOrdering extends Ordering:
  type T = Int
  def compare(t1: T, t2: T): Int = t1 - t2

val IntSet = new OrderedSet(intOrdering)

val set = IntSet.empty.add(6).add(8).add(23) // type error
```

modularity — Motivation

We can fix the code by making the ord parameter tracked:

```
trait Ordering:
  type T
  def compare(t1:T, t2: T): Int

class OrderedSet(tracked val ord: Ordering):
  type Set = List[ord.T]

  def empty: Set = Nil
  extension (s: Set)
    def add(x: ord.T): Set = ...

object intOrdering extends Ordering:
  type T = Int
  def compare(t1: T, t2: T): Int = t1 - t2

val IntSet = new OrderedSet(intOrdering)

val set = IntSet.empty.add(6).add(8).add(23)
```

modularity — Underlying intuition

The intuition behind tracked class parameters is thinking of the constructor as returning the declared class with a refinement to the class paremeter. e.g.

```
class OrderedSet(tracked val ord: Ordering) // :
    OrderedSet { val ord: ord.type }
```

modularity type class improvements — Motivation

Currently we define type classes like this:

```
trait ShowOld[T]:
  extension (x: T) def show: String

trait SemiGroupOld[T]:
  extension (x: T) def combine(y: T): T

trait MonoidOld[T] extends SemiGroupOld[T]:
  def unit: T
```

modularity type class improvements — Motivation

We create instances of type classes and use them like this:

```
given ShowOld[Int] with
  extension (x: Int) def show = x.toString

given [T: ShowOld] => ShowOld[List[T]] with
  extension (xs: List[T]) def show = xs.map(_.show).
      mkString("[", ", ", "]")

def combineAllOld[T: MonoidOld](xs: List[T]): T =
  // requires summon
  xs.foldLeft(summon[MonoidOld].unit)(_.combine(_))

def combineAllOld1[T](
  xs: List[T]
)(using m: MonoidOld[T]): T = // explicit using
  xs.foldLeft(m.unit)(_.combine(_))
```

modularity type class improvements — Motivation

The new modularity feature allows us to define type classes like this:

```scala
trait Show:
  type Self // Self is the type, we are defining the type
      class for
  extension (x: Self) def show: String

trait SemiGroup:
  type Self
  extension (x: Self) def combine(y: Self): Self

trait Monoid extends SemiGroup:
  def unit: Self
```

modularity type class improvements — Motivation

We define instances and use them like this:

```
given Int is Show: // `Int is Show` is equivalent for Show
    { type Self = Int }
  extension (x: Int) def show = x.toString

given [T: Show] => List[T] is Show:
  extension (xs: List[T]) def show = xs.map(_.show).
      mkString("[", ", ", "]")

def combineAll[T: Monoid as m](xs: List[T]): T = // `as`
    creates an alias
  xs.foldLeft(m.unit)(_.combine(_))

def combineAll1[T: Monoid](xs: List[T]): T = // default
    alias `T`
  xs.foldLeft(T.unit)(_.combine(_))
```

modularity type class improvements — Motivation

We also have new syntax for aggregate type class instances:

```
def combineAllAndPrint[T: {Monoid, Show}](xs: List[T]): T
  =
  xs.foldLeft(T.unit)(_.combine(_)).tap(x => println(T.
    show(x)))

def combineAllAndPrint1[T: {Monoid as m, Show as s}](xs:
  List[T]): T =
  xs.foldLeft(m.unit)(_.combine(_)).tap(x => println(s.
    show(x)))
```

Bonus

Bonus — setters in Scala

Let's consider the rules for defining field setters in Scala:

```
class ClassWithAGetterAndASetter:
  private var _value: Int = 0
  def value: Int = _value
  def value_=(newValue: Int): Unit = _value = newValue

val c = ClassWithAGetterAndASetter()

println(c.value)

c.value = 2 // c.value_=(2)

println(c.value) // 2
```

Bonus — unary operators in Scala

Next, let's take a look at the rules for defining unary operators in Scala:

```scala
class ClassWithUnaryOp:
  private var i: Int = 1
  def unary_! : Int = -i

val cwo = ClassWithUnaryOp()

println(!cwo) // -1
```

Bonus — putting it together

Can we combine them together?

```
class AClass(private var i: Int):
  def unary_! : Int = i
  def unary_!_=(i: Int): Unit = this.i = i // parsing
      error
```

Bonus — putting it together

What if we use backticks?

```
class AClass(private var i: Int):
  def unary_! : Int = i
  def `unary_!_=`(i: Int): Unit = this.i = i

val ac = AClass(1)

println(!ac)

!ac = 2 // ac.`unary_!_=`(2)

println(!ac)
```

Works!

Incoming experimental features

betterFors

- **Feature Name:** betterFors
- **Import:** `import scala.language.experimental.betterFors`
- **TLDR:** Simplifies desugaring of for-comprehensions and allows them to start with an alias.

betterFors — Motivation

Currently, we are often forced to write code like this:

```scala
val a = 1
for {
  b <- Some(2)
  c <- doSth(a)
} yield b + c
```

betterFors — Motivation

Currently, we are often forced to write code like this:

```scala
val a = 1
for {
  b <- Some(2)
  c <- doSth(a)
} yield b + c
```

With the new changes, we will be able to write:

```scala
for {
  a = 1
  b <- Some(2)
  c <- doSth(a)
} yield b + c
```

betterFors — Motivation

Let's consider the following code:

```scala
for {
  a <- doSth(arg)
  b = a
} yield a + b
```

Currently desugars to:

```scala
doSth(arg).map { a =>
  val b = a
  (a, b)
}.map { case (a, b) => a + b }
```

betterFors — Motivation

Let's consider the following code:

```
for {
  a <- doSth(arg)
  b = a
} yield a + b
```

Currently desugars to:

```
doSth(arg).map { a =>
  val b = a
  (a, b)
}.map { case (a, b) => a + b }
```

New desugaring:

```
doSth(arg).map { a =>
  val b = a
  a + b
}
```

betterFors — Motivation

Let's consider the following code:

```
for {
  a <- List(1, 2, 3)
} yield a
```

Currently desugars to:

```
List(1, 2, 3).map(a => a)
```

betterFors — Motivation

Let's consider the following code:

```
for {
  a <- List(1, 2, 3)
} yield a
```

Currently desugars to:

```
List(1, 2, 3).map(a => a)
```

New desugaring:

```
List(1, 2, 3)
```

alternativeBindPatterns

- **Feature Name:** alternativeBindPatterns
- **Import:** –
- **TLDR:** Allows binds in alternative pattern matches.

alternativeBindPatterns — Motivation

Consider the following code:

```
enum Command:
  case Get, North, Go, Pick, Up
  case Item(name: String)

import Command.*
```

alternativeBindPatterns — Motivation

Consider the following code:

```
enum Command:
  case Get, North, Go, Pick, Up
  case Item(name: String)

import Command.*
```

We might want write the following code:

```
def loop(cmds: List[Command]): Unit =
  cmds match
    case Pick :: Up :: Item(name) :: Nil =>
      ???
    case Get :: Item(name) :: Nil =>
      ???
```

alternativeBindPatterns — Motivation

The following code will be more expressive:

```
def loop(cmds: List[Command]): Unit =
  cmds match
    case Pick :: Up :: Item(name) :: Nil |
         Get :: Item(name) :: Nil =>
      ???
```

multipleAssignments

- **Feature Name:** multipleAssignments
- **Import:** –
- **TLDR:** Allows multiple assignments in a single statement.

multipleAssignments — Motivation

The following code implements a Fibonacci sequence iterator:

```scala
class FibonacciIterator() extends Iterator[Int]:
  private var a: Int = 0
  private var b: Int = 1

  def hasNext = true
  def next() =
    val r = a
    val n = a + b
    a = b
    b = n
    r
```

The semantics aren't clear, because of the temporary variable `n`.

multipleAssignments — Motivation

With the new proposal, it can be rewritten as:

```scala
class FibonacciIterator() extends Iterator[Int]:
  private var a: Int = 0
  private var b: Int = 1

  def hasNext = true
  def next() =
    val r = a
    (a, b) = (b, a + b)
    r
```

multipleAssignments — Motivation

With the new proposal, it can be rewritten as:

```
class FibonacciIterator() extends Iterator[Int]:
  private var a: Int = 0
  private var b: Int = 1

  def hasNext = true
  def next() =
    val r = a
    (a, b) = (b, a + b)
    r
```

Acceptance criteria:

- single level tuple on the left
- any expression that returns a tuple of the correct type on the right

Thank you!

Questions?