Scala Typesystem II

Mikhail Mutcianko, Alexey Shcherbakov

СПБгУ, СП

25 февраля 2021

Syntax recap: Advaned Features

Parameter lists

```scala
def foldLeft[B](z: B)(op: (B, A) => B): B
```

- logical grouping
- partial application
- passing implicit arguments
- assisting the typechecker

Type inference of multi-paren

Types are inferred sequentially per each complete parenthesis

```
def foo[A](x: A, f: A => Int) = ???


object Demo {
  foo(1, i => i * i) // <<< error: missing parameter type

}
```

Type inference of multi-paren

Types are inferred sequentially per each complete parenthesis

```scala
def foo[A](x: A, f: A => Int) = ???
def bar[A](x: A)(f: A => Int) = ???

object Demo {
  foo(1, i => i * i) // <<< error: missing parameter type

}
```

Type inference of multi-paren

Types are inferred sequentially per each complete parenthesis

```
def foo[A](x: A, f: A => Int) = ???
def bar[A](x: A)(f: A => Int) = ???

object Demo {
  foo(1, i => i * i) // <<< error: missing parameter type
  bar(2)(i => i * i)
}
```

Named parameters

- arguments could be labelled with their parameter names
- order of named arguments can be rearranged
- the unnamed arguments must come first

Named parameters

- arguments could be labelled with their parameter names
- order of named arguments can be rearranged
- the unnamed arguments must come first

```scala
def printName(first: String, last: String) = ???

printName("John", "Smith")                   // Prints "John Smith"
printName(first = "John", last = "Smith") // Prints "John Smith"
printName(last = "Smith", first = "John") // Prints "John Smith"
printName(last = "Smith", "john")           // <<< error: positional after named argument
```

Call by-name parameters

By-name parameters are only evaluated when used. They are in contrast to by-value parameters. To make a parameter called by-name, simply prepend => to its type.

- evaluated at each use within the function
- not the same thing as function-typed parameter
- can be used to pass blocks of code

Call by-name parameters

Example

```
p: () => Boolean  // a function input parameter
p: => Boolean     // a by-name parameter
```

Call by-name parameters

Example

```
p: () => Boolean  // a function input parameter
p: => Boolean     // a by-name parameter

myAssert(() => 5 > 3)

```

Call by-name parameters

Example

```
p: () => Boolean  // a function input parameter
p: => Boolean     // a by-name parameter

myAssert(() => 5 > 3)
byNameAssert(5 > 3)
```

Call by-name parameters

Example

```
p: () => Boolean   // a function input parameter
p: => Boolean      // a by-name parameter

myAssert(() => 5 > 3)
byNameAssert(5 > 3)
```

```
def whileLoop(condition: => Boolean)(body: => Unit): Unit =
  if (condition) {
    body
    whileLoop(condition)(body)
  }

whileLoop (i > 0) {
  println(i)
  i -= 1
}
```

Type aliases

A type alias creates a new named type for a specific, existing type

```
type Word = List[Char]
type Sentence = List[Word]
type Paragraph = List[Sentence]
```

Type aliases

A type alias creates a new named type for a specific, existing type

```
type Word = List[Char]
type Sentence = List[Word]
type Paragraph = List[Sentence]
```

Poor practice example:

```
type IntMaker = () => Int
IntMaker
```

Advanced OOP

Final keyword

In Scala `final` keyword is used to restrict inheritance

- final val/var/def - prohibits overriding
- final class - prohibits inheritance

Linearization

Scala `linearization`[1] is a deterministic process that puts all traits in a linear inheritance hierarchy

Linearization

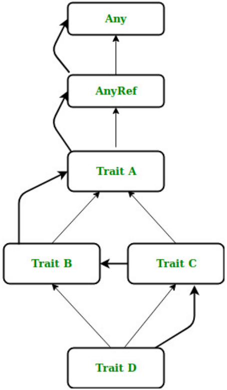
Scala `linearization`[1] is a deterministic process that puts all traits in a linear inheritance hierarchy

1. start at the first extended class or trait and write that complete hierarchy down
2. take the next trait and write this hierarchy down
   - remove all classes/traits from this hierarchy which are already in the linearized hierarchy
   - add the remaining traits to the bottom of the linearized hierarchy to create the new linearized hierarchy
3. repeat step 2 for every trait
4. place the class itself as the last type extending the linearized hierarchy

Linearization

```scala
trait A
trait B extends A
trait C extends A
class D extends B with C
```

Linearization

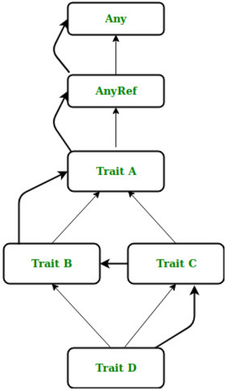



```scala
trait A
trait B extends A
trait C extends A
class D extends B with C
```

bold — `Linearization`

light — `Inheritance`

Linearization




```
trait A
trait B extends A
trait C extends A
class D extends B with C
```

Actual hierarchy:

$\mathtt{D} \rightarrow \mathtt{C} \rightarrow \mathtt{B} \rightarrow \mathtt{A} \rightarrow \mathtt{AnyRef} \rightarrow \mathtt{Any}$

bold — `Linearization`

light — `Inheritance`

Abstract overrides

Scala allows invoking an abstract method of a superclass

- the meaning of super is not known at compile time in a single trait
- `super` calls in a trait are dynamically bound via linearization
- trait methods that need to call `super` must be annotated with `abstract override`

Abstract overrides

Example

```
trait IntPrinter
  { def print(a: Int) }
class PrinterImpl extends IntPrinter
  { def print(a: Int) = println(a) }
trait DoublingPrinter extends IntPrinter
  { abstract override def print(a: Int) = super.print(a*2) }

|\pause|(new PrinterImpl).print(3)                      // prints 3
|\pause|(new PrinterImpl with DoublingPrinter).print(3) // prints 6
```

Self type

Self-types are a way to declare that a trait must be mixed into another trait, even though it doesn't directly extend it. That makes the members of the dependency available without imports

Self type

Self-types are a way to declare that a trait must be mixed into another trait, even though it doesn't directly extend it. That makes the members of the dependency available without imports

```
trait Defns { this: Aggregate => ...}
trait Utils { this: Aggregate => ...}

object Aggregate extends Defns with Utils with ...
```

Path dependent types

In Scala, a nested type is bound to a specific instance of the outer type, not to the outer type itself

Path dependent types

In Scala, a nested type is bound to a specific instance of the outer type, not to the outer type itself

```scala
class Foo {
  class Bar
  var bar: Bar = new Bar
}
val a = new Foo
val b = new Foo
a.bar = b.bar //error: type mismatch; found: b.Bar; required: a.Bar
```

Path dependent types

In Scala, a nested type is bound to a specific instance of the outer type, not to the outer type itself

```scala
class Foo {
  class Bar
  var bar: Foo#Bar = new Bar
}
val a = new Foo
val b = new Foo
a.bar = b.bar // OK
```

Path dependent types

Muli-paren

```scala
case class Animal(foodName: String) {
  case class Food(name: String)
  val food = Food(foodName)
}

def feed(animal: Animal)(food: animal.Food) = ???

val cat = Animal("fish")
val fish = Animal("seaweed")
feed(cat)(fish.food) // error: found: fish.Food, required: cat.Food
```

Compound types

- express that the type of an object is a subtype of several other types
- resulting type is an intersections of object types
- the general form is: `A with B with C ... { refinement }`

Compound types

```scala
trait Str { def str: String }
trait Count { def count: Int }

def repeat(cd: Str with Count): String =
  Iterator.fill(cd.count)(cd.str).mkString

repeat(new Str with Count {
  val str = "test"
  val count = 3
}) // "testtesttest"
```

Polymorphic Types

Subtype polymorphism

Subclasses of a class can define their own unique behaviors while providing a common access interface. Instances of a subclass can be passed to a base class

```scala
trait Animal { def speak }
class Human extends Animal { def speak = println("foo") }

val animal: Animal = new Human
animal.speak
```

Type parameters

Allow abstracting over types in a class or a method

- class parameters are bound on construction
- method parameters are bound on invocation

Type parameters

Allow abstracting over types in a class or a method

- class parameters are bound on construction
- method parameters are bound on invocation

```
class A[T] {
  def foo(t: T) = ???
}

val a = new Foo[Int]
a.foo(123)    // OK
a.foo("123")  // compile error: found: String, required: Int
```

Type parameters

Allow abstracting over types in a class or a method

- class parameters are bound on construction
- method parameters are bound on invocation

```
class A {
  def foo[T](t: T) = ???
}

val a = new Foo[Int]
a.foo(123)    // OK
a.foo("123")  // OK
```

Type erasure

Scalac warning:

```
non-variable type argument Int in type pattern Seq[Int] (the
underlying of Seq[Int]) is unchecked since it is eliminated by erasure
```

Type erasure

Scalac warning:

```
non-variable type argument Int in type pattern Seq[Int] (the
underlying of Seq[Int]) is unchecked since it is eliminated by erasure
```

Type parameters do not affect evaluation in Scala

We can assume that all type parameters and type arguments are removed before evaluating the program

Type erasure

```scala
def process(thing: SomeThing[_]) = thing match {
  case _: Int         => "an int"
  case _: Seq[Int]    => "some ints"
  case _: Seq[String] => "some strings"
}

process(123) == "an int"
process(Seq(1,2,3)) == "some ints"
process(Seq("1", "2", "3")) == "some ints"
```

Type erasure

```scala
def process(thing: SomeThing[_]) = thing match {
  case _: Int         => "an int"
  case _: Seq[Int]    => "some ints"
  case _: Seq[String] => "some strings"
}

process(123) == "an int"
process(Seq(1,2,3)) == "some ints"
process(Seq("1", "2", "3")) == "some ints"
```

Type bounds

Type bounds limit the concrete values of the type variables and possibly reveal more information about the members of such types

- `A <: B` means: `A` is a subtype of `B`
- `A >: B` means: `A` is a supertype of `B`, or `B` is a subtype of `A`

Type bounds

Upper bounds

```scala
trait Animal                    { def fitness: Int }
trait Reptile extends Animal
trait Mammal  extends Animal
trait Zebra   extends Mammal { def zebraCount: Int }
trait Giraffe extends Mammal

def selection[A <: Animal](a1: A, a2: A): A =
  if (a1.fitness > a2.fitness) a1 else a2
```

Type bounds

Lower and mixed bounds

```
def reptilize[A >: Reptile](stuff: A): A = ???
```

- the type parameter `A` that can range only over supertypes of `Reptile`
- `A` could be one of `Reptile`, `Animal`, `AnyRef` or `Any`

Type bounds

Lower and mixed bounds

```scala
def reptilize[A >: Reptile](stuff: A): A = ???
```

- the type parameter `A` that can range only over supertypes of `Reptile`
- `A` could be one of `Reptile`, `Animal`, `AnyRef` or `Any`

```scala
def hide[A >: Zebra <: Animal](stuff: A): A = ???
```

- restrict `A` any type on the interval between `Zebra` and `Animal`

Variance

Variance is the correlation of subtyping relationships of complex types and the subtyping relationships of their component types

Scala supports variance annotations of type parameters of generic classes

```scala
class Foo[+A] // A covariant class
class Bar[-A] // A contravariant class
class Baz[A]  // An invariant class
```

Covariance

Let `C[T]` be a parameterized type and `A`, `B` are types such that `A <: B` and if `C[A] <: C[B]` then `C` is covariant

```
class Foo[A] { def f(x: Foo[A]) = ??? }

val foo = new Foo[Number]
foo.f(new Foo[Int]) // error: type mismatch found: Foo[Int], required: Foo[Number]
```

Covariance

Let `C[T]` be a parameterized type and `A`, `B` are types such that `A <: B` and if `C[A] <: C[B]` then `C` is covariant

```
class Foo[+A] { def f[T <: A](x: Foo[T]) = ??? }

val foo = new Foo[Number]
foo.f(new Foo[Int]) // OK
```

Java array variace

Early version of Java had no generics but polymorphic array algorithms were still necessary:

```
boolean equalArrays (Object[] a1, Object[] a2);
void shuffleArray(Object[] a);
```

Java array variace

Early version of Java had no generics but polymorphic array algorithms were still necessary:

```java
boolean equalArrays (Object[] a1, Object[] a2);
void shuffleArray(Object[] a);
```

```java
Zebra[] zebras = new Zebra[]{ new Zebra() }  // Array containing 1 `Zebra`
Mammal[] mammals = zebras  // Allowed because arrays are covariant in Java
mammals[0] = new Giraffe() // Allowed because a `Giraffe` is a subtype of `Mammal`
Zebra zebra = zebras[0]    // Get the first `Zebra` ... which is actually a `Giraffe`!
```

To mitigate invalid array type storage issue Java has introduced `ArrayStoreException` [4]

```
Object x[] = new String[3];
x[0] = new Integer(0); // ArrayStoreException is thrown
```

var overriding

Question

Why can't you override a var?

var overriding

Question

Why can't you override a var?

```scala
class Foo {          var animal: Animal }
class Bar { override var animal: Cat    } extends Foo

val bar: Bar      = new Bar
val barAsFoo: Foo = bar
barAsFoo.animal   = new Dog
bar.animal // <<< ???
```

Scala collections variance

- `immutable` Scala collections are covariant
- `mutable` Scala collections are invariant
- `Array` in Scala is invariant

Scala collections variance

- `immutable` Scala collections are covariant
- `mutable` Scala collections are invariant
- `Array` in Scala is invariant

```scala
val zebras: Array[Zebra] = Array(new Zebra)
val mammals: Array[Mammal] = zebras //error: found: Array[Zebra], required: Array[Mammal]
mammals(0) = new Giraffe
val zebra: Zebra = zebras(0)
```

Scala collections variance

- `immutable` Scala collections are covariant
- `mutable` Scala collections are invariant
- `Array` in Scala is invariant

```scala
val zebras: List[Zebra] = List(new Zebra)
val mammals: List[Mammal] = zebras
val something = mammals :+ new Giraffe
val zebra: Zebra = zebras.tail.head // OK
```

Scala function variance

If `A2 <: A1` and `B1 <: B2` then `A1 => B1 <: A2 => B2`

So functions are contravariant in argument type(s) and covariant in result type

```scala
trait Function1[-T, +U] {
  def apply(x: T): U
}
```

Scala function variance

If `A2 <: A1` and `B1 <: B2` then `A1 => B1 <: A2 => B2`

So functions are contravariant in argument type(s) and covariant in result type

```
trait Function1[-T, +U] {
  def apply(x: T): U
}
```

```
class Foo[+A] { def f(x: Foo[A]) = ??? }
//                     ^
// error: covariant type A occurs in contravariant position in type Foo[A] of value x
```

Variance checks

The Scala compiler will check that there are no problematic combinations when compiling a class with variance annotations. Roughly:

- covariant type parameters can only appear in method results
- contravariant type parameters can only appear in method parameters
- invariant type parameters can appear anywhere

Variance checks

The Scala compiler will check that there are no problematic combinations when compiling a class with variance annotations. Roughly:

- covariant type parameters can only appear in method results
- contravariant type parameters can only appear in method parameters
- invariant type parameters can appear anywhere
- covariant type parameters may appear in lower bounds of method type parameters
- contravariant type parameters may appear in upper bounds of method

Type members

Abstract types, such as traits and abstract classes, can in turn have abstract type members. This means that the concrete implementations define the actual types.

```scala
trait Buffer {
  type T
  val element: T
  def copyFrom(t: T)
}
abstract class SeqBuffer extends Buffer {
  type U
  type T <: Seq[U]
  def length = element.length
}
```

Type members

Abstract types, such as traits and abstract classes, can in turn have abstract type members. This means that the concrete implementations define the actual types.

```
trait Buffer {
  type T
  val element: T
  def copyFrom(t: T)
}
class IntSeqBuffer extends Buffer {
  override type T = Seq[Int]
  def length = element.length
  def copyFrom(t: Seq[Int]) = ???
}
```

[1] https://www.scala-lang.org/files/archive/spec/2.11/05-classes-and-objects.html#class-linearization

[2] http://lampwww.epfl.ch/~amin/dot/fpdt.pdf

[3] https://stackoverflow.com/questions/12935731/any-reason-why-scala-does-not-explicitly-support-dependent-types/12937819#12937819

[4] https://docs.oracle.com/javase/7/docs/api/java/lang/ArrayStoreException.html