Introduction To Scala

Hassan Chafi, Arvind Sujeeth, Kevin Brown, Sungpack Hong, Zach Devito and Kunle Olukotun
Oracle Labs
Stanford University
Why Use Scala for DSLs?

- Flexible and overloadable syntax
  - Methods can be infixed, no need for new syntax for each DSLs
  - First-class functions
- Expressive type system
  - Implicit conversion
  - Abstract Types
- Modular development of languages
  - Traits
  - Mixin composition
def qsort[T <% Ordered[T]](list : List[T]) : List[T] = {
  list match {
    case Nil => Nil
    case x::xs =>
      val (before,after) = xs partition (_ < x)
      qsort(before) ++ (x :: qsort(after))
  }
}
def qsort[T <% Ordered[T]](list : List[T]) : List[T] = {
    list match {
      case Nil => Nil
      case x::xs =>
        val (before,after) = xs partition (_ < x)
        qsort(before) ++ (x :: qsort(after))
    }
  }
Parameterized Types

- Similar to Generics in Java

```scala
val a: List[Int]
val b: List[Float]
```

- Can be constrained by type bounds

```scala
class Vector[T <: Arith[T]] { .. }
```
def qsort[T <% Ordered[T]](list : List[T]) : List[T] = {
  list match {
    case Nil => Nil
    case x :: xs =>
      val (before,after) = xs partition (_ < x)
      qsort(before) ++ (x :: qsort(after))
  }
}

Intro to Scala: QuickSort
PATTERN MATCHING
Case Classes

abstract class Term

case class Var(name: String) extends Term

case class Fun(arg: String, body: Term) extends Term

case class App(f: Term, v: Term) extends Term

val x1 = Fun("x", Var("y"))
val x2 = Fun("x", Var("y"))
val y1 = Var("y")
println("" + x1 + " == " + x2 + " => " + (x1 == x2))
println("" + x1 + " == " + y1 + " => " + (x1 == y1))

Fun(x,Var(y)) == Fun(x,Var(y)) => true
Fun(x,Var(y)) == Var(y) => false

def isIdentityFun(term: Term): Boolean = term match {
  case Fun(x, Var(y)) if x == y => true
  case _ => false
}
val id = Fun("x", Var("x"))
println(isIdentityFun(id))
abstract class Expr

case class Var(name: String) extends Expr

case class Number(num: Double) extends Expr

case class UnOp(operator: String, arg: Expr) extends Expr

case class BinOp(operator: String,
                 left: Expr, right: Expr) extends Expr

def simplifyTop(expr: Expr): Expr = expr match {
  case UnOp("-", UnOp("-", e)) => e  // Double negation
  case BinOp("+", e, Number(0)) => e  // Adding zero
  case BinOp("*", e, Number(1)) => e  // Multiplying by one
  case _ => expr
}
expr match {
  case BinOp(op, left, right) =>
    println(expr + " is a binary operation")
  case _ =>
}

expr match {
  case BinOp(_, _, _) => println(expr + " is a binary operation")
  case _ => println("It's something else")
}
def describe(x: Any) = x match {
  case 5 => "five"
  case true => "truth"
  case "hello" => "hi!"
  case Nil => "the empty list"
  case _ => "something else"
}
Variable Patterns

expr match {
    case 0 => "zero"
    case somethingElse => "not zero: " + somethingElse
}

scala> import math.{E, Pi}
import math.{E, Pi}

scala> E match {
    case Pi => "strange math? Pi = " + Pi
    case _  => "OK"
}
res11: java.lang.String = OK
Variable Patterns

scala> E match {
        case pi => "strange math? Pi = " + pi
        case _  => "OK"
    }
<console>:9: error: unreachable code
    case _  => "OK"
    ^
Constructor Patterns

```expr match {
    case BinOp("+", e, Number(0)) => println("a deep match")
    case _ =>
}
```
expr match {
    case List(0, _, _) => println("found it")
    case _ =>
}

expr match {
    case List(0, _*) => println("found it")
    case _ =>
}
Typed Pattern

```scala
def generalSize(x: Any) = x match {
  case s: String => s.length
  case m: Map[_, _] => m.size
  case _ => -1
}
```
Variable Binding

expr match {
    case UnOp("abs", e @ UnOp("abs", _)) => e
    case _ =>
}

Pattern Guards

scala> def simplifyAdd(e: Expr) = e match {
   | case BinOp("+", x, y) if x == y =>
   |     BinOp("\times", x, Number(2))
   | case _ => e
   |

simplifyAdd: (e: Expr)Expr
def qsort[T <% Ordered[T]](list : List[T]) : List[T] = {
  list match {
    case Nil => Nil
    case x::xs =>
      val (before,after) = xs partition (_ < x)
      qsort(before) ++ (x :: qsort(after))
  }
}
Patterns Everywhere

scala> val myTuple = (123, "abc")
myTuple: (Int, java.lang.String) = (123,abc)

scala> val (number, string) = myTuple
number: Int = 123
string: java.lang.String = abc

scala> val exp = new BinOp("*", Number(5), Number(1))
exp: BinOp = BinOp(*,Number(5.0),Number(1.0))

scala> val BinOp(op, left, right) = exp
op: String = *
left: Expr = Number(5.0)
right: Expr = Number(1.0)
def qsort[T <% Ordered[T]](list : List[T]) : List[T] = {
  list match {
    case Nil => Nil
    case x::xs =>
      val (before,after) = xs partition (_ < x)
      qsort(before) ++ (x :: qsort(after))
  }
}
Defining Operators

```scala
class Rational(n: Int, d: Int) {
  ...
  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
  def * (that: Rational): Rational =
    new Rational(numer * that.numer, denom * that.denom)
  ...
}
```

- Now can write \( x+y \) and so forth
Simple Extendable Syntax

- Methods can be written like infix operators
- Equivalent to `matrix1.*(matrix2).+(matrix3)`

```
val sigma = matrix1 * matrix2 + matrix3
```

```
  a  b  c  d  e
```

- Equivalent to `a.b(c).d(e)`
def qsort[T <% Ordered[T]](list : List[T]) : List[T] = {
   list match {
      case Nil => Nil
      case x::xs => 
         val (before,after) = xs partition(_ < x)
         qsort(before) ++ (x :: qsort(after))
   }
}
First Class Functions

```scala
scala> var increase = (x: Int) => x + 1
increase: (Int) => Int = <function1>

scala> increase(10)
res0: Int = 11

scala> increase = (x: Int) => x + 9999
increase: (Int) => Int = <function1>

scala> increase(10)
res1: Int = 10009
```

- Can write functions as unnamed literals and call them
Can also pass them as values, as with any other object, these are then called function values.
First Class Functions

```scala
scala> val someNumbers = List(-11, -10, -5, 0, 5, 10)
someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> someNumbers.foreach((x: Int) => println(x))
-11
-10
-5
0
5
10

scala> someNumbers.filter((x: Int) => x > 0)
res4: List[Int] = List(5, 10)
```

- Can also pass them as values, as with any other object, these are then called function values
Short Forms for Function Literals

scala> someNumbers.filter((x) => x > 0)
res5: List[Int] = List(5, 10)

scala> someNumbers.filter(x => x > 0)
res6: List[Int] = List(5, 10)

scala> someNumbers.filter(_ > 0)
res7: List[Int] = List(5, 10)

scala> val f = _ + _
<console>:4: error: missing parameter type for expanded function ((x$1, x$2) => x$1.$plus(x$2))

scala> val f = (_: Int) + (_: Int)
f: (Int, Int) => Int = <function2>

- Can leave off argument types when they can be inferred
- Can leave off parenthesis, when type is inferred and only one argument
- Use underscores to be even more concise, can only use argument once
- Add type information if compiler complains
def qsort[T <% Ordered[T]](list : List[T]) : List[T] = {
  list match {
    case Nil => Nil
    case x::xs =>
      val (before,after) = xs partition (_ < x)
      qsort(before) ++ (x :: qsort(after))
  }
}
Implicit Conversions

```scala
val range = 0 until 10
for (i <- range) {
  //
}
```

- `until` is not a keyword, and 0 is an Int, Int doesn’t define a method until
  - Type error
  - Fix it up with implicits

```scala
implicit def intWrapper(x: Int) = new runtime.RichInt(x)
```

- Compiler tries to fix type errors by inserting calls to implicit methods
  - Need to be in scope, and unambiguous
Implicits are useful for DSLs

- Can add functionality to existing types by implicitly converting them to an enriched type
- In our framework, implicits are used extensively to lift concrete types into our IR

\[
\text{Expr} + 5 \Rightarrow \text{Expr} + \text{Const}(5) \Rightarrow \text{Op}(+,\text{Expr, Const}(5))
\]
Implicit Parameters

```scala
object Greeter {
    def greet(name: String)(implicit prompt: PreferredPrompt) {
        println("Welcome, "+ name +". The system is ready.")
        println(prompt.preference)
    }
}

scala> val bobsPrompt = new PreferredPrompt("relax> ")
bobsPrompt: PreferredPrompt = PreferredPrompt@74a138

scala> Greeter.greet("Bob")(bobsPrompt)
Welcome, Bob. The system is ready.
relax>
```
Implicit Parameters

```
object JoesPrefs {
    implicit val prompt = new PreferredPrompt("Yes, master> ")
}
```

```
scala> Greeter.greet("Joe")
<console>:10: error: could not find implicit value for parameter prompt: PreferredPrompt
    Greeter.greet("Joe")
    ^

scala> import JoesPrefs._
import JoesPrefs._

scala> Greeter.greet("Joe")
Welcome, Joe. The system is ready.
Yes, master>
```
def Sum[T:Numeric](selector: TSource => T): T = {
    val n = implicitly[Numeric[T]]
    import n._
    var sum = n.zero
    for(e <- source) {
        sum += selector(e)
    }
    sum
}
Automatically Constructed Implicit Parameters

- Compiler can construct some requested information at compile time and pass it via an implicit parameter at runtime
- Manifests provide static type information at runtime

```scala
def name[T](implicit m: scala.reflect.Manifest[T]) = m.toString

name[Int => Int] // returns "scala.Function1[int, int]"
```

- SourceContext provides source location information (useful for debugging)
View Bounds

- View Bounds are just syntactic sugar for requesting an implicit conversion

```scala
def f[T <% Ordered[T]](a: T) = a.OrderedMethod
def f[T](a: T)(implicit ev: T => Ordered[T]) = a. OrderedMethod
```
def qsort[T <% Ordered[T]](list : List[T]) : List[T] = {
  list match {
    case Nil => Nil
    case x::xs =>
      val (before, after) = xs partition (_ < x)
      qsort(before) ++ (x :: qsort(after))
  }
}
Writing new Control Structures

```scala
scala> println("Hello, world!")
Hello, world!

scala> println { "Hello, world!" }
Hello, world!

def withPrintWriter(file: File)(op: PrintWriter => Unit) {
    val writer = new PrintWriter(file)
    try { op(writer) } finally { writer.close()}
}

val file = new File("date.txt")
withPrintWriter(file) { writer =>
    writer.println(new java.util.Date)
}
```
For Expressions

```scala
for (i <- 1 to 4)
  println("Iteration "+ i)
Iteration 1
Iteration 2
Iteration 3
Iteration 4

for (i <- 1 until 4)
  println("Iteration "+ i)
```

// Not common in Scala...

```
for (i <- 0 to filesHere.length - 1)
  println(filesHere(i))
```

- In Scala you usually iterate over collections directly
For Expressions

```scala
val filesHere = (new java.io.File(".")).listFiles
for (file <- filesHere)
  println(file)

Can also filter in for

for (file <- filesHere if file.getName.endsWith(".scala"))
  println(file)

for (
  file <- filesHere
  if file.isFile
  if file.getName.endsWith(".scala")
) println(file)
```
Nested Iteration

```scala
def fileLines(file: java.io.File) = 
  scala.io.Source.fromFile(file).getLines().toList

def grep(pattern: String) = 
  for ( 
    file <- filesHere 
    if file.getName.endsWith(".scala"); 
    line <- fileLines(file) 
    if line.trim.matches(pattern) 
  ) println(file +": "+ line.trim)

grep(".*gcd.*")
```

- Notice semicolon
  - Use curly braces if you want to get rid of it
Producing a new Collection

```scala
def scalaFiles =
  for {
    file <- filesHere
    if file.getName.endsWith(".scala")
  } yield file
```

- Don’t need to return the same type (think map, not filter)
For-expressions’ Flexibility

```javascript
for (x <- expr₁) yield expr₂
expr₁.map(x => expr₂)

for (x <- expr₁ if expr₂) yield expr₃
expr₁.withFilter(x => expr₂).map(x => expr₃)

for (x <- expr₁) body
expr₁.foreach(x => body)

for (x <- expr₁; if expr₂; y <- expr₃) body
expr₁.withFilter(x => expr₂).foreach(x =>
  expr₃.foreach(y => body))
```
MOREADVANCEDFEATURES
# Abstract Types

```scala
abstract class Buffer {
  type T
  val element: T
}
```

```scala
abstract class SeqBuffer extends Buffer {
  type U
  type T <: Seq[U]
  def length = element.length
}
```

```scala
abstract class IntSeqBuffer extends SeqBuffer {
  type U = Int
}
```

```scala
val buff = new IntSeqBuffer {
  type T = List[U]
  val element = List(5, 19)
}
```
TRAITS:
Sane Multiple Inheritance
The Basics

trait Philosophical {
  def philosophize() {
    println("I consume memory, therefore I am!")
  }
}

class Frog extends Philosophical {
  override def toString = "green"
}

scala> val frog = new Frog
frog: Frog = green

scala> frog.philosophize()
I consume memory, therefore I am!

- Traits can have fields, methods both abstract and concrete but:
- No constructor
- Super calls are dynamically bound
Traits are also Types

scala> val phil: Philosophical = frog
phil: Philosophical = green

scala> phil.philosophize()
I consume memory, therefore I am!
Can Mix-In Multiple Traits

class Animal

class Frog extends Animal with Philosophical {
    override def toString = "green"
}

class Animal

trait HasLegs

class Frog extends Animal with Philosophical with HasLegs {
    override def toString = "green"
}
Can Override Trait Values

class Animal
class Frog extends Animal with Philosophical {
    override def toString = "green"
    override def philosophize() {
        println("It ain't easy being "+ toString +"!")
    }
}

scala> val phrog: Philosophical = new Frog
phrog: Philosophical = green

scala> phrog.philosophize()
It ain't easy being green!
class Rational(n: Int, d: Int) {
    // ...
    def < (that: Rational) =
        this.numer * that.denom > that.numer * this.denom
    def > (that: Rational) = that < this
    def <= (that: Rational) = (this < that) || (this == that)
    def >= (that: Rational) = (this > that) || (this == that)
}

class Rational(n: Int, d: Int) extends Ordered[Rational] {
    // ...
    def compare(that: Rational) =
        (this.numer * that.denom) - (that.numer * this.denom)
}
Mix-in Composition

abstract class AbsIterator {
  type T
  def hasNext: Boolean
  def next: T
}

trait RichIterator extends AbsIterator {
  def foreach(f: T => Unit) { while (hasNext) f(next) }
}

class StringIterator(s: String) extends AbsIterator {
  type T = Char
  private var i = 0
  def hasNext = i < s.length()
  def next = { val ch = s charAt i; i += 1; ch }
}

val iter = new StringIterator("DSLs are Cool") with RichIterator
Mix-in composition

- Allows you to structure your DSL implementation in a modular fashion
- Each related set of language features can be in their own traits
- Can now construct new DSLs starting out with the parts of existing ones
Traits as Stackable Modifications

```scala
trait BasicIntQueue {
  private val buf = new ArrayBuffer[Int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}

trait Doubling extends BasicIntQueue {
  override def put(x: Int) { super.put(2 * x) }
}

trait Incrementing extends BasicIntQueue {
  override def put(x: Int) { super.put(x + 1) }
}

trait IncrementingDoublingQueue extends BasicIntQueue with Doubling with Incrementing
trait DoublingIncrementingQueue extends BasicIntQueue with Incrementing with Doubling
```
Summary

- Scala has some good features for developing DSLs and DSL infrastructure
  - Flexible language constructs
  - Powerful type system
  - Features for modular development of components
Reducing Code Duplication

```scala
object FileMatcher {
    private def filesHere = (new java.io.File(".")).listFiles

    def filesEnding(query: String) =
        for (file <- filesHere; if file.getName.endsWith(query)) yield file

    def filesContaining(query: String) =
        for (file <- filesHere; if file.getName.contains(query)) yield file

    def filesRegex(query: String) =
        for (file <- filesHere; if file.getName.matches(query)) yield file
}
```

Want something like:

```scala
def filesMatching(query: String, method) =
    for (file <- filesHere; if file.getName.method(query)) yield file
```

- Note, all functions have shared logic
Reducing Code Duplication

```scala
def filesMatching(query: String, matcher: (String, String) => Boolean) = {
  for (file <- filesHere; if matcher(file.getName, query)) yield file
}

def filesEnding(query: String) = filesMatching(query, _.endsWith(_))
def filesContaining(query: String) = filesMatching(query, _.contains(_))
def filesRegex(query: String) = filesMatching(query, _.matches(_))
```

Underscore placeholders are filled in by function parameters in order
Look for other opportunities to simplify your code
Check Scala’s collections for other opportunities to use higher order functions and avoid code duplication