A Scalable Language

Martin Odersky
FOSDEM 2009

The software landscape today …

… resembles a tower of Babel with many little (or not so little) languages playing together.

E.g.

> *JavaScript* on the client
> *Perl/Python/Ruby/Groovy* for server side scripting
> *JavaFX* for the UI
> *Java* for the business logic
> *SQL* for database access

all cobbled together with a generous helping of XML.

This is both good and bad

Good: Every language can concentrate on what it's best at.

Bad: Cross language communication:
complicated, fragile, source of misunderstandings.

Problematic: Cross language communication is controlled by a common type system (neither static nor dynamic).

It's based on low-level representations such as XML trees or (worse) strings (as in JDBC database queries).

Alternative: Scalable languages

A language is *scalable* if it is suitable for very small as well as very large programs.

A single language for extension scripts and the heavy lifting.

Application-specific needs are handled through libraries and embedded DSL's instead of external languages.

Scala shows that this is possible.

Scala is a scripting language

It has an interactive read-eval-print-loop (REPL).
Types can be inferred.
Boilerplate is scrapped.

```
scala> var capital = Map("US" → "Washington", "France" → "Paris")
capital: Map[String, String] = Map(US → Washington, France → Paris)
scala> capital += ("Japan" → "Tokio")
scala> capital("France")
res7: String = Paris
```


Scala is the Java of the future

It has basically everything Java has now.
(sometimes in different form)

It has closures.
(proposed for Java 7, but rejected)

It has traits and pattern matching.
(I would not be surprised to see them in Java 8, 9 or 10)

It compiles to .class files, is completely interoperable and runs about as fast as Java

```
object App {
  def main(args: Array[String]) {
    if (args exists (_.toLowerCase == "-help"))
      printUsage()
    else
      process(args)
  }
}
```

7

Interoperability

Scala fits seamlessly into a Java environment

Can call Java methods, select Java fields, inherit Java classes, implement Java interfaces, etc.

None of this requires glue code or interface descriptions

Java code can also easily call into Scala code

Scala code resembling Java is translated into virtually the same bytecodes.

⇒ Performance is usually on a par with Java

Scala | Martin Odersky, FOSDEM 2009 | EPFL

8

Scala is a composition language

New approach to module systems:

component = class or trait

composition via mixins

Abstraction through

- parameters,
- abstract members (both types and values),
- self types

gives *dependency injection* for free

```
trait Analyzer { this: Backend =>
  …
}
trait Backend extends Analyzer
              with Optimization
              with Generation {

  val global: Main
  import global._

  type OutputMedium <: Writable
}
```

Scala | Martin Odersky, FOSDEM 2009 | EPFL

9

Is Scala a “kitchen-sink language”?

Not at all. In terms of feature count, Scala is roughly comparable to today’s Java and smaller than C# or C++.

But Scala is *deep*, where other languages are *broad*.

Two principles:

1. *Focus on abstraction and composition, so that users can implement their own specialized features as needed.*

2. Have the same sort of constructs work for very small as well as very large programs.

Scala | Martin Odersky, FOSDEM 2009 | EPFL

10

Scala compared to Java

Scala adds	Scala removes
+ a pure object system	- static members
+ operator overloading	- primitive types
+ closures	- break, continue
+ mixin composition with traits	- special treatment of interfaces
+ existential types	- wildcards
+ abstract types	- raw types
+ pattern matching	- enums

Modeled in libraries:
assert, enums, properties, events, actors, using, queries, …

Scala | Martin Odersky, FOSDEM 2009 | EPFL

11

Scala cheat sheet (1): Definitions

Scala method definitions:

```
def fun(x: Int): Int = {
  result
}

def fun = result
```

Scala variable definitions:

```
var x: Int = expression
val x: String = expression
```

Java method definition:

```
int fun(int x) {
  return result
}
```

(no parameterless methods)

Java variable definitions:

```
int x = expression
final String x = expression
```

Scala | Martin Odersky, FOSDEM 2009 | EPFL

12

Scala cheat sheet (2): Expressions

Scala method calls:

```
obj.meth(arg)
obj meth arg
```

Scala choice expressions:

```
if (cond) expr1 else expr2

expr match {
   case pat1 => expr1
   ....
   case patn => exprn
}
```

Java method call:

```
obj.meth(arg)
```

(no operator overloading)

Java choice expressions, stmts:

```
cond ? expr1 : expr2
if (cond) return expr1;
else return expr2;

switch (expr) {
    case pat1 : return expr1;
    ...
    case patn : return exprn ;
}  // statement only
```

Scala | Martin Odersky, FOSDEM 2009 | EPFL

Scala cheat sheet (3): Objects and Classes

13

Scala Class and Object

```
class Sample(x: Int, val p: Int) {
  def instMeth(y: Int) = x + y
}

object Sample {
  def staticMeth(x: Int, y: Int) =
    x * y
}
```

Java Class with statics

```
class Sample {
  private final int x;
  public final int p;
  Sample(int x, int p) {
    this.x = x;
    this.p = p;
  }
  int instMeth(int y) {
    return x + y;
  }
  static int staticMeth(int x, int y) {
    return x * y;
  }
}
```

Scala | Martin Odersky, FOSDEM 2009 | EPFL

Scala cheat sheet (4): Traits

14

Scala Trait

```
trait T {
  def abstractMth(x: String): Int
  def concreteMth(x: String) =
    x + field
  var field = "!"
}
```

Scala mixin composition:

```
class C extends Super with T
```

Java Interface

```
interface T {
  int abstractMth(String x)
}
```

(no concrete methods)

(no fields)

Java extension + implementation:

```
class C extends Super implements T
```

Scala | Martin Odersky, FOSDEM 2009 | EPFL

Spring Cleaning

15

Scala's syntax is lightweight and concise.

Due to:

- semicolon inference,
- type inference,
- lightweight classes,
- extensible API's,
- closures as control abstractions.

```
var capital = Map("US" -> "Washington",
                  "Canada" -> "ottawa")
capital += ("Japan" -> "Tokyo")
for (c <- capital.keys)
  capital(c) = capital(c).capitalize
assert(capital("Canada") == "Ottawa")
```

Average reduction in LOC: ≥ 2

due to concise syntax and better abstraction capabilities

➔ Scala feels like a cleaned up Java …

… with one major difference

16

It's `x: Int` instead of `int x`

Why the change?

Works better with type inference:

`var x = 0` instead of `x = 0 // that's not a definition!`

Works better for large type expressions:

```
val x: HashMap[String, (String, List[Char])] = …
```

instead of

```
public final HashMap<String, Pair<String, List<Char>>> x = …
```

Scalability demands extensibility

17

Take numeric data types

Today's languages support int, long, float, double.

Should they also support BigInt, BigDecimal, Complex, Rational, Interval, Polynomial?

There are good reasons for each of these types

But a language combining them all would be too complex.

Better alternative: Let users *grow* their language according to their needs.

Scala | Martin Odersky, FOSDEM 2009 | EPFL

Adding new datatypes - seamlessly

18

For instance type `BigInt`:

```
def factorial(x: BigInt): BigInt =
  if (x == 0) 1 else x * factorial(x - 1)
```

Compare with using Java's class:

```
import java.math.BigInteger
def factorial(x: BigInteger): BigInteger =
  if (x == BigInteger.ZERO)
    BigInteger.ONE
  else
    x.multiply(factorial(x.subtract(BigInteger.ONE)))
}
```

Scala | Martin Odersky, FOSDEM 2009 | EPFL

19

Implementing new datatypes - seamlessly

Here's how BigInt is implemented

+ is an identifier; can be used as a method name

Infix operations are method calls:
a + b is the same as a.+(b)
a add b is the same as a.add(b)

```
im    java.math.BigInteger
clas   igInt(val bigInteger: BigIn    r)
exten s java.lang.Number {
  def + (that: BigInt) =
    new BigInt(this.bigInteger add that.bigInteger)
  def - (that: BigInt) =
    new BigInt(this.bigInteger subtract that.bigInteger)
  …   // other methods implemented analogously
}
```

Scala | Martin Odersky, FOSDEM 2009 | EPFL

20

Adding new control structures

For instance `using` for resource control (proposed for Java 7)

```
using (new BufferedReader(new FileReader(path))) {
  f => println(f.readLine())
}
```

Instead of:

```
val f = new BufferedReader(new FileReader(path))
try {
  println(f.readLine())
} finally {
  if (f != null) f.close()
}
```

Scala | Martin Odersky, FOSDEM 2009 | EPFL

21

Implementing new control structures:

Here's how one would go about implementing `using`:

T is a type parameter...

… supporting a close method

```
def using[T <: { def close() }]
         (resource: T)
         (block: T => Unit) {
  try {
    block(resource)
  } finally {
    if (resource != null) resource.close()
  }
}
```

A closure that takes a T parameter

Scala | Martin Odersky, FOSDEM 2009 | EPFL

22

Break and continue

Scala does not have them. Why?

> They are a bit imperative; better use many smaller functions.
> Issues how to interact with closures.
> They are not needed!

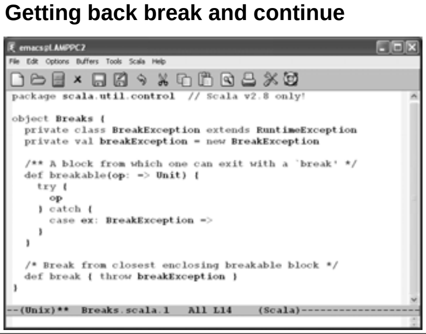
We can support them purely in the libraries.

```
import scala.util.control.Breaks._
breakable {
  for (x <- elems) {
    println(x * 2)
    if (x > 0) break
  }
}
```

Scala | Martin Odersky, FOSDEM 2009 | EPFL

23

Getting back break and continue



Scala | Martin Odersky, FOSDEM 2009 | EPFL

24

What makes Scala scalable?

Many factors: strong typing, inference, little boilerplate,…

But mainly, its tight integration of functional and object-oriented programming

Functional programming:

Makes it easy to build interesting things from simple parts, using

- higher-order functions,
- algebraic types and pattern matching,
- parametric polymorphism.

Object-oriented programming:

Makes it easy to adapt and extend complex systems, using

- subtyping and inheritance,
- dynamic configurations,
- classes as partial abstractions.

Scala | Martin Odersky, FOSDEM 2009 | EPFL

Scala is object-oriented

25

Every value is an object

Every operation is a method call

Exceptions to these rules in Java (such as primitive types, statics) are eliminated.

```
scala> (1).hashCode
res8: Int = 1

scala> (1).+(2)
res10: Int = 3
```

Scala | Martin Odersky, FOSDEM 2009 | EPFL

Scala is functional

26

Scala is a functional language, in the sense that every function is a value.

Functions can be anonymous, curried, nested.

Many useful higher-order functions are implemented as methods of Scala classes. E.g:

```
scala> val matrix = Array(Array(1, 0, 0),
     |                    Array(0, 1, 0),
     |                    Array(0, 0, 1))

matrix: Array[Array[Int]] = Array([I@164da25,…

scala> matrix.exists(row => row.forall(0 ==))
res13: Boolean = false
```

Scala | Martin Odersky, FOSDEM 2009 | EPFL

Functions *are* objects

27

If functions are values, and values are objects, it follows that functions themselves are objects.

The function type S => T is equivalent to scala.Function1[S, T], where Function1 is defined as follows:

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

So functions are interpreted as objects with apply methods.

For example, the *anonymous successor* function

(x: Int) => x + 1

is expanded to:

```
new Function1[Int, Int] {
  def apply(x: Int) =
    x + 1
}
```

Scala | Martin Odersky, FOSDEM 2009 | EPFL

Why should I care?

28

Since (=>) is a class, it can be subclassed.

So one can *specialize* the concept of a function.

An obvious use is for arrays, which are mutable functions over integer ranges.

A bit of syntactic sugaring lets one write:

a(i) = a(i) + 2 for

a.update(i, a.apply(i) + 2)

```
class Array [T] (l: Int)
extends (Int => T) {

def length: Int = l

def apply(i: Int): T = …

def update(i: Int, x: T):Unit

def elements: Iterator[T]

def exists(p: T => Boolean)

…

}
```

Scala | Martin Odersky, FOSDEM 2009 | EPFL

Partial functions

29

Another useful abstraction are partial functions.

These are functions that are defined only in some part of their domain.

What's more, one can inquire with the isDefinedAt method whether a partial function is defined for a given value.

```
trait PartialFunction[-A, +B]
extends (A => B) {
 def isDefinedAt(x: A):Boolean
}
```

Scala treats blocks of pattern matching cases as instances of partial functions.

This lets one write control structures that are not easily expressible otherwise.

Scala | Martin Odersky, FOSDEM 2009 | EPFL

Developing new paradigms

30

Scala's flexibility makes it possible for users to grow the language into completely new paradigms.

Case in point: concurrent programming

Since Scala is interoperable, Java threads and concurrent libraries are available.

But it's also possible to explore completely new paradigms.

Scala | Martin Odersky, FOSDEM 2009 | EPFL

Erlang-style actors

31

Two principal constructs (adopted from Erlang):

Send (`!`) is asynchronous; messages are buffered in an actor's mailbox.

`receive` picks the first message in the mailbox which matches any of the patterns $msgpat_i$.

If no pattern matches, the actor suspends.

```
// asynchronous message send
actor ! message
// message receive
receive {
  case msgpat1 => action1
  …
  case msgpatn => actionn
}
```

A pattern matching block of type PartialFunction[*MessageType*, *ActionType*]

Scala — Martin Odersky, FOSDEM 2009 — EPFL

A simple actor

32

```
case class Data(bytes: Array[Byte])
case class Sum(receiver: Actor)
val checkSumCalculator =
  actor {
    var sum = 0
    loop {
      receive {
        case Data(bs) => sum += hash(bs)
        case Sum(receiver) => receiver ! sum
      }
    }
  }
}
```

Spawn a new actor

repeatedly receive messages

Scala — Martin Odersky, FOSDEM 2009 — EPFL

Implementing receive

33

Using partial functions, it is straightforward to implement receive:

```
def receive [T] (f: PartialFunction[Message, T]): T = {
  self.mailBox.extractFirst(f.isDefinedAt)
  match {
    case Some(msg) =>
      f(msg)
    case None =>
      self.wait(messageSent)
  }}
```

Here,

`self` designates the currently executing actor,

`mailBox` is its queue of pending messages, and

`extractFirst` extracts first queue element matching given predicate.

Scala — Martin Odersky, FOSDEM 2009 — EPFL

Other Approaches to Scalability

34

C++

> Hard to scale down.

> Scaling up is possible for expert users.

.NET

> Many languages with common interoperability.

> Hard to do something that's really different.

Java

> *Lingua franca* makes it easy to understand other people's code.

> Not easy to scale down or up → pressure to add new languages.

Scala — Martin Odersky, FOSDEM 2009 — EPFL

Where are we now?

35

Scala

> Easy to scale down and up.

> Works well with a mix of expert users (for the framework) and non-experts (for the application code).

Scala solves the expressiveness challenge for doing this.

But does it also solve the safety issues?

> Problem: How to ensure that domain-specific code stays within its domain-specific library/language?

> For instance: How to ensure that a query formulated in Scala is non-recursive?

Addressed by ongoing project: *Pluggable type systems*

Scala — Martin Odersky, FOSDEM 2009 — EPFL

The Scala community

36

50000 downloads in 2008

300+ trak contributors

20+ messages/day on the mailing lists

Industrial adoption has started, among others at:

Twitter, Sony Pictures, Nature.com, Reaktor, Mimesis Republic, EDF Trading, …

Scala LiftOff conference, May 2008.

Scala talks in many conferences; next two at QCon, London, March 10-12.

Scala — Martin Odersky, FOSDEM 2009 — EPFL

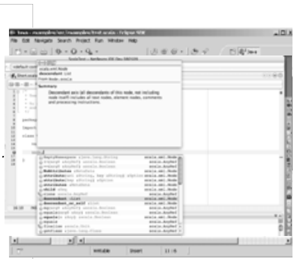
37

Tool support

> Standalone compiler: *scalac*
> Fast background compiler: *fsc*
> Interactive interpreter shell and script runner: *scala*
> Web framework: *lift*
> Testing frameworks: *Specs, ScalaCheck, ScalaTest, SUnit, …*

IDE plugins for:

> *Eclipse* (supported by EDF)
> *IntelliJ* (supported by JetBrains)
> *Netbeans* (supported by Sun)



Scala | Martin Odersky, FOSDEM 2009 | EPFL ECOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

38

Tool support

> Standalone compiler: *scalac*
> Fast background compiler: *fsc*
> Interactive interpreter shell and script runner: *scala*
> Web framework: *lift*
> Testing frameworks: *Specs, ScalaCheck, ScalaTest, SUnit, …*

IDE plugins for:

> *Eclipse* (supported by EDF)
> *IntelliJ* (supported by JetBrains)
> *Netbeans* (supported by Sun)

Scala | Martin Odersky, FOS

39

Who's using it?

Open source projects:
lift
wicket
NetLogo
SPDE: Scala branch for Processing
Isabelle: GUI and code extractor

Companies:
Twitter: infrastructure
Sony Pictures: middleware
Nature.com: infrastructure
SAP community: ESME company messaging
Reaktor: many different projects
Mimesis Republic: multiplayer games
EDF: trading, …

Scala | Martin Odersky, FOSDEM 2009 | EPFL ECOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

40

Learning Scala

To get started:
First steps in Scala, by Bill Venners published in Scalazine at www.artima.com

Scala for Java Refugees by Daniel Spiewack (great blog series)

To continue:
Programming in Scala, by *Odersky, Spoon, Venners, published by Artima,com*

Other books are in the pipeline.

Scala | Martin Odersky, FOSDEM 2009

Thank You

To try it out:

scala-lang.org

Thanks also to the (past and present) members of the Scala team:

Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sebastian Hack, Philipp Haller, Sean McDirmid, Ingo Meier, Adriaan Moors, Stéphane Micheloud, Nikolay Mihaylov, Anders Nielssen, Tiark Rompf, Lukas Rytz, Michel Schinz, Lex Spoon, Erik Stenman, Geoffrey Alan Washburn, Matthias Zenger.

Scala | Martin Odersky, FOSDEM 2009 | EPFL ECOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

42

Relationship between Scala and other languages

Main influences on the Scala design: Java, C# for their syntax, basic types, and class libraries,
Smalltalk for its uniform object model,
Eiffel for its uniform access principle,
Beta for systematic nesting,
ML, Haskell for many of the functional aspects.
OCaml, OHaskel, PLT-Scheme, as other (less tightly integrated) combinations of FP and OOP.
Pizza, Multi Java, Nice as other extensions of the Java platform with functional ideas.
(Too many influences in details to list them all)
Scala also seems to influence other new language designs, see for instance the closures and comprehensions in LINQ/C# 3.0.

Scala | Martin Odersky, FOSDEM 2009 | EPFL ECOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE