Programming languages - state of the art

The last 15 years have seen rapid progress in programming languages.

In 1996,

- garbage collection was considered a risky bet,
- strong typing was considered impractical by many,
- generic types were only found in ``academic'' languages

Java has changed all of this.

Nevertheless there remain many things to improve.
Today, it is still hard to:

- reason about correctness of programs,
- define and integrate domain specific languages,
- define truly reusable components.
How to advance?

The work on Scala was motivated by two hypotheses:

**Hypothesis 1:** A general-purpose language needs to be *scalable*; the same concepts should describe small as well as large parts.

**Hypothesis 2:** Scalability can be achieved by unifying and generalizing *functional* and *object-oriented* programming concepts.
Why unify FP and OOP?

Both have complementary strengths for composition:

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

- Scala is an object-oriented and functional language which is completely interoperable with Java. (the .NET version is currently under reconstruction.)
- It removes some of the more arcane constructs of these environments and adds instead:
  (1) a uniform object model,
  (2) pattern matching and higher-order functions,
  (3) novel ways to abstract and compose programs.
- An open-source distribution of Scala has been available since Jan 2004.
- Currently: ≥ 2000 downloads per month.
Scala is interoperable

Scala programs interoperate seamlessly with Java class libraries:

- Method calls
- Field accesses
- Class inheritance
- Interface implementation

all work as in Java.

Scala programs compile to JVM bytecodes.

Scala’s syntax resembles Java’s, but there are also some differences.

```scala
object Example1 {
  def main(args: Array[String]) {
    val b = new StringBuilder()
    for (i ← 0 until args.length) {
      if (i > 0) b.append(" ")
      b.append(args(i).toUpperCase)
    }
    Console.println(b.toString)
  }
}
```
Scala programs interoperate seamlessly with Java class libraries:

- Method calls
- Field accesses
- Class inheritance
- Interface implementation

all work as in Java.

Scala programs compile to JVM bytecodes.

Scala’s syntax resembles Java, but there are also some differences.

Scala’s version of the extended for loop
(use <- as an alias for ←)

Array[String] instead of String[

Arrays are indexed
args(i) instead of args[i]
Scala is functional

The last program can also be written in a completely different style:

- Treat arrays as instances of general sequence abstractions.
- Use higher-order functions instead of loops.

```scala
object Example2 {
  def main(args: Array[String]) {
    println(args.map(_.toUpperCase).mkString " ")
  }
}
```
Scala is functional

The last program can also be written in a completely different style:

- Treat arrays as instances of general sequence abstractions.
- Use higher-order functions instead of loops.

```scala
object Example2 {
  def main(args: Array[String]) {
    println(args
      map (_.toUpperCase) mkString
      )
  }
}
```

`map` is a method of `Array` which applies the function on its right to each array element.

A closure which applies the `toUpperCase` method to its `String` argument `map (_.toUpperCase)`.

`mkString` is a method of `Array` which forms a string of all elements with a given separator between them.
Scala is concise

Scala’s syntax is lightweight and concise.

Contributors:
- semicolon inference,
- type inference,
- lightweight classes,
- extensible API’s,
- closures as control abstractions.

Average reduction in LOC wrt Java: ≥ 2

due to concise syntax and better abstraction capabilities

```
var capital = Map( "US" → "Washington",
                 "France" → "paris",
                 "Japan" → "tokyo" )

capital += ( "Russia" → "Moskow" )

for ( (country, city) ← capital )
capital += ( country → city.capitalize )

assert ( capital("Japan") == "Tokyo" )
```
Scala is precise

All code on the previous slide used library abstractions, not special syntax.

Advantage: Libraries are extensible and give fine-grained control.

Elaborate static type system catches many errors early.

```scala
import scala.collection.mutable._
val capital =
  new HashMap[String, String]
with SynchronizedMap[String, String] {
  override def default(key: String) = "?"
}
capital += ("US" → "Washington",
            "France" → "Paris",
            "Japan" → "Tokyo")
assert( capital("Russia") == "?" )
```
Scala is precise

All code on the previous slide used library abstractions, not special syntax.

Advantage: Libraries are extensible and give fine-grained control.

Elaborate static type system catches many errors early.

```scala
import scala.collection.mutable._
val capital =
  new HashMap[String, String]
    with SynchronizedMap[String, String] {
    override def default(key: String) = "?"
  }

capital += ("US" -> "Washington",
            "France" -> "Paris",
            "Japan" -> "Tokyo")

assert( capital("Russia") == "?" )
```
Big or small?

Every language design faces the tension whether it should be big or small:
- Big is good: expressive, easy to use.
- Small is good: elegant, easy to learn.

Can a language be both big and small?

Scala’s approach: concentrate on abstraction and composition capabilities instead of basic language constructs.

<table>
<thead>
<tr>
<th>Scala adds</th>
<th>Scala removes</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ a pure object system</td>
<td>- static members</td>
</tr>
<tr>
<td>+ operator overloading</td>
<td>- special treatment of primitive types</td>
</tr>
<tr>
<td>+ closures as control abstractions</td>
<td>- break, continue</td>
</tr>
<tr>
<td>+ mixin composition with traits</td>
<td>- special treatment of interfaces</td>
</tr>
<tr>
<td>+ abstract type members</td>
<td>- wildcards</td>
</tr>
<tr>
<td>+ pattern matching</td>
<td></td>
</tr>
</tbody>
</table>
Scala is extensible

Guy Steele has formulated a benchmark for measuring language extensibility [Growing a Language, OOPSLA 98]:

Can you add a type of complex numbers to the library and make it work as if it was a native number type?

Similar problems: Adding type BigInt, Decimal, Intervals, Polynomials...

```scala
import Complex._
scala> val x = 1 + 1 * i
x: Complex = 1.0+1.0*i
scala> val y = x * i
y: Complex = -1.0+1.0*i
scala> val z = y + 1
z: Complex = 0.0+1.0*i
```
Implementing complex numbers

```scala
object Complex {
  val i = new Complex(0, 1)
  implicit def double2complex(x: double): Complex = new Complex(x, 0)
...
}

class Complex(val re: double, val im: double) {
  def +(that: Complex): Complex = new Complex(this.re + that.re, this.im + that.im)
  def -(that: Complex): Complex = new Complex(this.re - that.re, this.im - that.im)
  def *(that: Complex): Complex = new Complex(this.re * that.re - this.im * that.im,
                                            this.re * that.im + this.im * that.re)
  def /(that: Complex): Complex = { 
    val denom = that.re * that.re + that.im * that.im 
    new Complex((this.re * that.re + this.im * that.im) / denom,
                 (this.im * that.re - this.re * that.im) / denom)
  }
  override def toString = re+(if (im < 0) "-"+(-im) else "+"+im)+"I"
...
}
```

The Scala Experience, OOPSLA 2007 Tutorial
Implementing complex numbers

```
object Complex {
  val i = new Complex(0, 1)
  implicit def double2complex(x: double): Complex = new Complex(x, 0)
  ...
}

class Complex(val re: double, val im: double) {
  def +(that: Complex): Complex = new Complex(this.re + that.re, this.im + that.im)
  def -(that: Complex): Complex = new Complex(this.re - that.re, this.im - that.im)
  def *(that: Complex): Complex = new Complex(this.re * that.re - this.im * that.im, this.re * that.im + this.im * that.re)
  def / (that: Complex): Complex = {
    val denom = that.re * that.re + that.im * that.im
    new Complex((this.re * that.re + this.im * that.im) / denom, (this.im * that.re - this.re * that.im) / denom)
  }
  override def toString = re+(if (im < 0) ":"+-(-im) else "+"+im)+"*I"
  ...
}```

- Implicit conversions for mixed arithmetic
- Objects replace static class members
- Infix operations are method calls: a + b is the same as a.+(b)
- + is an identifier; can be used as a method name
- Class parameters instead of fields + explicit constructor
Implicits are Poor Man’s Type Classes

/** A “type class” */
class Ord[T] { def < (x: T): Boolean }

/** An “instance definition” */
implicit def intAsOrd(x: Int) =
    new Ord { def < (y: T) = x < y }

/** Another instance definition */
implicit def listAsOrd[T](xs: List[T])(implicit tAsOrd: T => Ord[T]) =
    new Ord {
        def < (ys: List[T]) = (xs, ys) match {
            case (__, Nil) => false
            case (Nil, __) => true
            case (x :: xs, y :: ts) => x < y && xs < ys
        }
    }

The Scala Experience, OOPSLA 2007 Tutorial
Tool support

Scala tool support is extensive and improving rapidly:

- Standalone compiler: `scalac`
- Fast background compiler: `fsc`
- Interactive interpreter shell and script runner: `scala`
- Testing framework: `SUnit`
- Eclipse plugin
- IntelliJ plugin (written by JetBrains)
The Scala compiler at work

Step 1:
Replace infix operators by method calls.

Replace == by equals.

```scala
capital += ( "Russia" → "Moskow" )
capital += ( country → city.capitalize )
assert ( capital("Japan") == "Tokyo" )
```
The Scala compiler at work

Step 1:

Replace infix operators by method calls.

Replace `==` by `equals`.

```
var capital = Map("US".→("Washington"),
    "France".→("paris"),
    "Japan".→("tokyo" ) )

capital = capital.("Russia".→("Moskow" ) )

for ( (country, city) ← capital )
capital = capital.+ (country.→ (city.capitalize))

assert (capital("Japan").equals("Tokyo" ) )
```
The Scala compiler at work

Step 2:

Expand for loop to foreach + closure.

Add empty parameter list () to parameterless methods.

```scala
var capital = Map("US"→("Washington"),
                  "France"→("paris"),
                  "Japan"→("tokyo") )
capital = capital.+("Russia"→("Moskow"))
for ((country, city) ← capital )
capital = capital.+(country→(city.capitalize))
assert (capital("Japan").equals("Tokyo"))
```
The Scala compiler at work

Step 2:

Expand for loop to foreach + closure.

Add empty parameter list () to parameterless methods.

```scala
val capital = Map("US".→("Washington"),
    "France".→("paris"),
    "Japan".→("tokyo") )
capital = capital.+("Russia".→("Moskow"))
capital.foreach {
    case (country, city) =>
        capital = capital.+(country.→(city.capitalize()))
}
assert (capital("Japan").equals("Tokyo"))
```
The Scala compiler at work

Step 3:

Expand closures to instances of anonymous inner classes.

Expand object application to apply methods.

```scala
... capital.foreach {
  case (country, city) =>
    capital = capital.+(country.→(city.capitalize))
}
assert (capital("Japan").equals("Tokyo"))
```
The Scala compiler at work

Step 3:
Expand closures to instances of anonymous inner classes.
Expand object application to apply methods.

```scala
... 
private class anonfun$0()
extends Function1[String, String] {
  def apply(cc: (String, String)) = {
    val country = cc._1
    val city = cc._2
    capital = capital.+(country.→(city.capitalize()))
  }
}
capital.foreach( new anonfun$0() )
assert (capital.apply("Japan").equals("Tokyo"))
```
The Scala compiler at work

Step 4:

Expand pairs to objects of class Tuple2

Add implicit conversions.

Expand imports.

Expand fancy names.

```scala
...
private class anonfun$0()
extends Function1[String, String] {
  def apply(cc: (String, String)) = {
    val country = cc._1
    val city = cc._2
    capital = capital.+(country.→(city.capitalize()))
  }
}
}
capital.foreach( new anonfun$0() )
assert (capital.apply("Japan").equals("Tokyo"))
```
The Scala compiler at work

Step 4:

Expand pairs to objects of class Tuple2

Add implicit conversions.

Expand imports.

Expand fancy names.

private class anonfun$0 extends Function1[String, String] {
  def apply(cc: Tuple2[String, String]) = {
    val country = cc._1
    val city = cc._2
    capital = capital.$plus
    (Predef.any2arrowAssoc(country).$minus$greater
     (Predef.stringWrapper(city).capitalize()))
  }
}
capital.foreach( new anonfun$0() )
Predef.assert (capital.apply("Japan").equals("Tokyo"))
The Scala compiler at work

Step 5
Convert to Java
(In reality, the compiler generates bytecodes, not source)

```
... 

private class anonfun$0()
extends Function1<String, String> {
  void apply(Tuple2<String, String> cc) {
    final String country = cc._1;
    final String city = cc._2;
    capital = capital.$plus
      (Predef.any2arrowAssoc(country).$minus$greater
       (Predef.stringWrapper(city).capitalize()));
  }
}

capital.foreach( new anonfun$0() );
Predef.assert(capital.apply("Japan").equals("Tokyo"));
```
Performance

- How large is the overhead introduced by the Scala to Java generation?
- At first sight there’s a lot of boilerplate added:
  - forwarding method calls,
  - ancillary objects,
  - inner anonymous classes.
- Fortunately, modern JIT compilers are good at removing the boilerplate.
- So average execution times are comparable with Java’s.
- Startup times are somewhat longer, because of the number of classfiles generated (we are working on reducing this).
# Shootout data

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ratio</td>
<td></td>
</tr>
<tr>
<td>C++ g++</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>C gcc</td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>Digital Mars</td>
<td>1.2</td>
<td></td>
</tr>
<tr>
<td>Eiffel SmartEiffel</td>
<td>1.4</td>
<td></td>
</tr>
<tr>
<td>Clean</td>
<td>1.4</td>
<td></td>
</tr>
<tr>
<td>Pascal Free Pascal</td>
<td>1.4</td>
<td></td>
</tr>
<tr>
<td>Haskell GHC</td>
<td>1.6</td>
<td></td>
</tr>
<tr>
<td>OCaml</td>
<td>1.7</td>
<td></td>
</tr>
<tr>
<td>Ada 95 GNAT</td>
<td>1.7</td>
<td></td>
</tr>
<tr>
<td>Lisp SBCL</td>
<td>1.7</td>
<td></td>
</tr>
<tr>
<td>SML MLton</td>
<td>1.8</td>
<td></td>
</tr>
<tr>
<td>Scala</td>
<td>1.8</td>
<td></td>
</tr>
<tr>
<td>Java JDK -server</td>
<td>1.9</td>
<td></td>
</tr>
<tr>
<td>BASIC FreeBASIC</td>
<td>1.9</td>
<td></td>
</tr>
<tr>
<td>Oberon-2 OO2C</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>Forth bigForth</td>
<td>2.3</td>
<td></td>
</tr>
<tr>
<td>Nice</td>
<td>2.3</td>
<td></td>
</tr>
<tr>
<td>C# Mono</td>
<td>2.6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>score</th>
<th>×</th>
</tr>
</thead>
<tbody>
<tr>
<td>75.4</td>
<td>1</td>
</tr>
<tr>
<td>71.1</td>
<td>1</td>
</tr>
<tr>
<td>65.4</td>
<td>1</td>
</tr>
<tr>
<td>52.9</td>
<td>2</td>
</tr>
<tr>
<td>52.2</td>
<td>3</td>
</tr>
<tr>
<td>52.2</td>
<td>2</td>
</tr>
<tr>
<td>48.4</td>
<td>1</td>
</tr>
<tr>
<td>45.1</td>
<td>2</td>
</tr>
<tr>
<td>43.8</td>
<td>2</td>
</tr>
<tr>
<td>43.3</td>
<td>3</td>
</tr>
<tr>
<td>41.8</td>
<td>2</td>
</tr>
<tr>
<td>41.4</td>
<td>1</td>
</tr>
<tr>
<td>40.7</td>
<td>1</td>
</tr>
<tr>
<td>40.5</td>
<td>2</td>
</tr>
<tr>
<td>37.0</td>
<td>7</td>
</tr>
<tr>
<td>33.4</td>
<td>1</td>
</tr>
<tr>
<td>33.3</td>
<td>4</td>
</tr>
<tr>
<td>28.9</td>
<td>2</td>
</tr>
</tbody>
</table>

**Caveat:**

*These data should not be overinterpreted – they are a snapshot, that's all!*
The JVM as a compilation target

The JVM has turned out to be a good platform for Scala.

Important aspects:

• High-performance memory system with automatic garbage collection.

• Aggressive JIT optimizations of function stacks.

If I had two wishes free for a future version of the JVM, I would pick

1. Support for tail-calls.
2. Extend the class-file format with true support for inner classes.
The Scala design

Scala strives for the tightest possible integration of OOP and FP in a statically typed language.

This continues to have unexpected consequences.

Scala unifies

- algebraic data types with class hierarchies,
- functions with objects

This gives a nice & rather efficient formulation of Erlang style actors
ADTs are class hierarchies

Many functional languages have algebraic data types and pattern matching.

⇒ Concise and canonical manipulation of data structures.

Object-oriented programmers object:

• ADTs are not extensible,
• ADTs violate the purity of the OO data model,
• Pattern matching breaks encapsulation,
• and it violates representation independence!
Pattern matching in Scala

Here's a set of definitions describing binary trees:

```scala
abstract class Tree[T]
case object Empty extends Tree
case class Binary(elem: T, left: Tree[T], right: Tree[T]) extends Tree
```

And here's an inorder traversal of binary trees:

```scala
def inOrder [T] ( t: Tree[T] ): List[T] = t match {
case Empty => List()
case Binary(e, l, r) => inOrder(l) ::: List(e) ::: inOrder(r)
}
```

This design keeps

- **purity**: all cases are classes or objects.
- **extensibility**: you can define more cases elsewhere.
- **encapsulation**: only parameters of case classes are revealed.
- **representation independence** using extractors [ECOOP 07].
Extractors

... are objects with unapply methods. unapply is called implicitly for pattern matching

```scala
object Twice {
  def apply(x: Int) = x*2
  def unapply(z: Int) = if (z%2==0) Some(z/2) else None
}
val x = Twice(21)
x match {
  case Twice(y) => println(x+" is two times "+y)
  case _ => println("x is odd")
}
```
Functions are objects

Scala is a functional language, in the sense that every function is a value.
If functions are values, and values are objects, it follows that functions themselves are objects.
The function type \( S \Rightarrow T \) is equivalent to `scala.Function1[S, T]` where `Function1` is defined as follows:

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

So functions are interpreted as objects with apply methods.
For example, the `anonymous successor` function
\( (x: \text{Int}) \Rightarrow x + 1 \) is expanded to

``` scala
new Function1[Int, Int] {
  def apply(x: Int): Int =
    x + 1
}
```
Why should I care?

• 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.
• Another bit of syntactic sugaring lets one write:

```scala
val a = new Array[Int](10)

a(i) = a(i) + 2  for
a.update(i, a.apply(i) + 2)
```

```scala
class Array[T] ( length: Int )
  extends (Int => T)  {
    def length: Int = ...
    def apply(i: Int): A = ...
    def update(i: Int, x: A): unit = ...
    def elements: Iterator[A] = ...
    def exists(p: A => Boolean):Boolean
      = ...
  }
```
Partial functions

- 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.

```scala
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.
Example: Erlang-style actors

- 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 \( mspat_i \).
- If no pattern matches, the actor suspends.

A partial function of type

\[
\text{PartialFunction}[\text{MessageType}, \text{ActionType}]
\]
A simple actor

```scala
case class Elem(n: Int)
case class Sum(receiver: Actor)
val summer =
  actor {
    var sum = 0
    loop {
      receive {
        case Elem(n) => sum += n
        case Sum(receiver) => receiver ! sum
      }
    }
  }
```
Implementing receive

- Using partial functions, it is straightforward to implement receive:

- Here, `self` designates the currently executing actor, `mailBox` is its queue of pending messages, and `extractFirst` extracts first queue element matching given predicate.

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

- A possible objection to Scala's library-based approach is:
  Why define actors in a library when they exist already in purer, more optimized form in Erlang?
- First reason: interoperability
- Another reason: libraries are much easier to extend and adapt than languages.

Experience:
Initial versions of actors used one thread per actor
⇒ lack of speed and scalability
Later versions added a non-returning `receive` called react which makes actors event-based.
This gave great improvements in scalability.
An application: *lift* Web Framework

*lift* is a Web framework similar to Rails and SeaSide, which uses many features of Scala

- **Actors** – for AJAX/Comet ready apps
- **Closures** – for HTML form elements
- **Traits/Mixins** – for persistence, data binding, query building using POJO’s (or POSO’s?)
- **Pattern Matching** – for extensible URL matching
- **Flexible Syntax** – for embedded DSL’s

Written by David Pollak at Circleshare

Use case: *Skittr*, a *Twittr* clone.

Excellent scalability: $10^6$ concurrent actors on a two processor system.
Summing Up

- Scala blends functional and object-oriented programming.
- This has worked well in the past: for instance in Smalltalk, Python, or Ruby.
- However, Scala is goes farthest in unifying FP and OOP in a statically typed language.
- This leads to pleasant and concise programs.
- Scala feels similar to a modern scripting language, but without giving up static typing.
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,
- 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.
Lessons Learned

1. Don’t start from scratch
2. Don’t be overly afraid to be different
3. Pick your battles
4. Think of a “killer-app”, but expect that in the end it may well turn out to be something else.
5. Provide a path from here to there.
Thank You

To try it out:

scala-lang.org

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