The Scala Experience
Safe Programming Can Be Fun!

Martin Odersky

EPFL
Lausanne, Switzerland

TS-2844

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
- The Java™ programming language 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 the Java programming language; (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 January 2004
- Currently: ≥ *2000* downloads per month

Scala Is Interoperal

- Scala programs interoperate seamlessly with Java class libraries:
  - Method calls
  - Field accesses
  - Class inheritance
  - Interface implementation
- All work as in the Java programming language
- Scala programs compile to J Virtual Machine (JVM™) bytecodes

```
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  ing)
  }
```

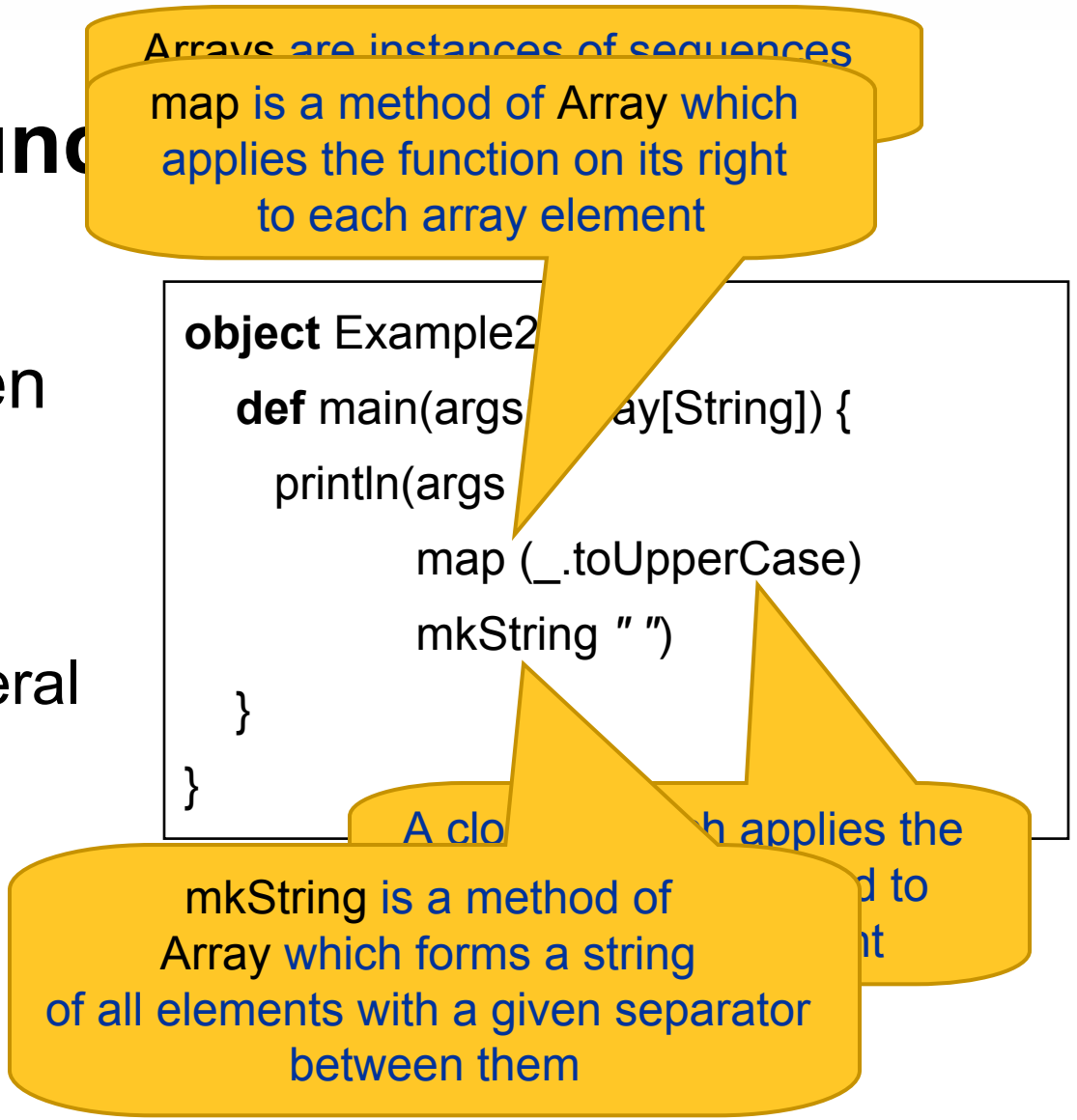
Array[String] instead of String[]

Scala's version of the extended (use <- as an alias for ←

Arrays are indexed args(i) instead of args[i]

Scala Is Func

- 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 Is Concise

- Scala's syntax is lightweight and concise
- Contributors
  - Semicolon inference
  - Type inference
  - Lightweight classes
  - Extensible APIs
  - Closures as control abstractions

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

capital += ( "Russia" → "Moskow" )

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

assert ( capital("Japan") == "Tokyo" )
```

- Average reduction in LOC wrt Java programming language: ≥ 2
- Due to concise syntax and better abstraction capabilities

Scala Is Pr

- All code on the previous slide used
- Advantage: Libraries are extensible and give fine-grained control
- Elaborate static type system catches many errors early

Specify map implementation: HashMap Specify map type: String to String

Mixin trait SynchronizedMap to make capital map thread-safe

Provide a default value: "?"

```
import scala.c      tion.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

Scala Adds	Scala Removes
+ a pure object system	- static members
+ operator overloading	- special treatment of primitive types
+ closures as control abstractions	- break, continue
+ mixin composition with traits	- special treatment of interfaces
+ abstract type members	- wildcards
+ pattern matching	

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._
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 C

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

Objects r

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

Class parameters
instead of fields+ explicit constructor

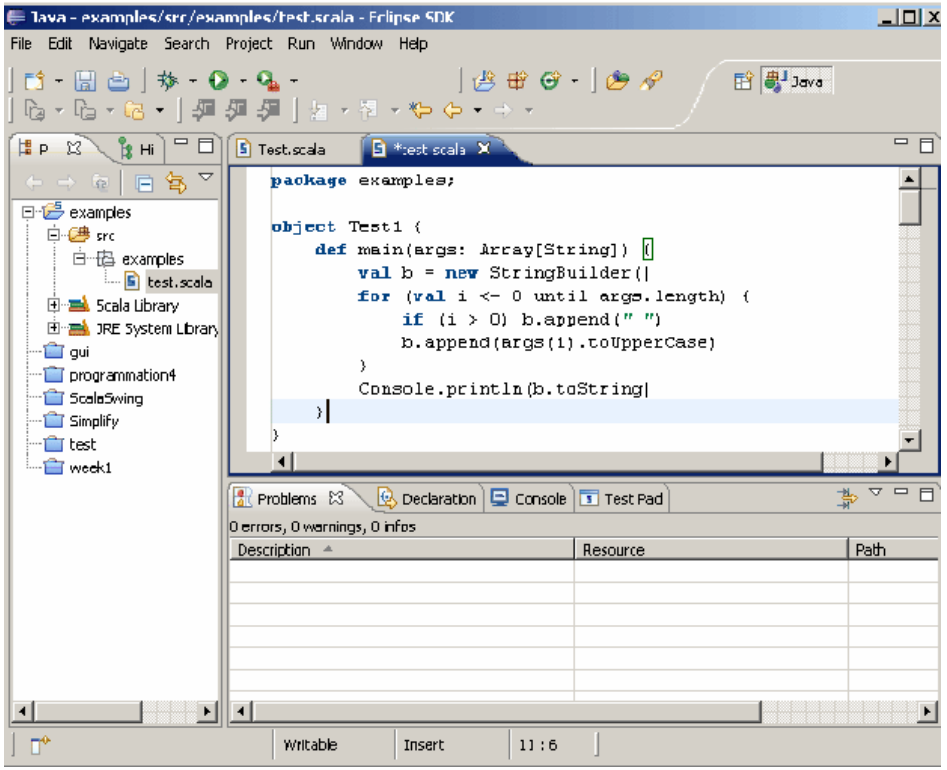
```
ew Complex(0, 1)
t def double2complex(x: double): Com          x, 0)

}
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

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 plug-in
  - IntelliJ plug-in (written by JetBrains)



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 += ("Russia" → "Moskow")

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

assert ( capital("Japan") == "Tokyo" )
```

```
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

```
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" ))
```

```
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

```
...
capital.foreach {
    (country, city) =>
        capital = capital.+(country
→(city.capitalize))
}

assert (capital("Japan").equals("Tokyo" ))
→(city.capitalize()))
```

```
private class anonfun$0()
extends Function1[String, String] {
    def apply(cc: String, String)) = {
        val country = cc._1
        val city = cc._2
        capital = capital.+(country
    }
}

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: (String, String)) = {
        val country = cc._1
        val city = cc._2
        capital = capital.updated(country, city.capitalize())
    }
}
capital.foreach( new anonfun$0() )
assert (capital.apply("Japan").equals("Tokyo" ))
```

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

The Scala Compiler at Work

Step 5:

- Convert to Java platform
- (In reality, the compiler generates bytecodes, not source)

```
...

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"
));
```

Performance

- How large is the overhead introduced by the Scala to Java platform 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 platform's
- Startup times are somewhat longer, because of the number of classfiles generated (we are working on reducing this)

Shootout Data

Gentoo :
Intel Pentium 4
Computer
Language
Shootout

31 Mar 2007

Caveat:
These data should not be overinterpreted—they are a snapshot, that's all!

ratio	language	score	×
	best possible	**100.0**	
1.0	C++ **g++**	75.4	
1.1	C **gcc**	71.1	1
1.2	D **Digital Mars**	65.4	
1.4	Eiffel **SmartEiffel**	52.9	2
1.4	**Clean**	52.2	3
1.4	Pascal **Free Pascal**	52.2	2
1.6	Haskell **GHC**	48.4	
1.7	**OCaml**	45.1	2
1.7	Ada 95 **GNAT**	43.8	2
1.7	Lisp **SBCL**	43.3	3
1.8	SML **MLton**	41.8	2
1.8	**Scala**	41.4	1
1.9	Java **JDK -server**	40.7	
1.9	BASIC **FreeBASIC**	40.5	2
2.0	Oberon-2 **OO2C**	37.0	7
2.3	Forth **bigForth**	33.4	1
2.3	**Nice**	33.3	4
2.6	C# **Mono**	28.9	2

The Java Virtual Machine as a Compilation Target

- The JVM machine 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 machine, I would pick:
  - Support for tail-calls
  - Extend the class-file format with true support for inner classes

The terms “Java Virtual Machine” and “JVM” mean a Virtual Machine for the Java™ platform.

The Scala Design

- Scala strives for the tightest possible integration of OOP and FP in a statically typed language
- In the following, I present three examples where:
  - Formerly separate concepts in FP and OOP are identified
  - The fusion leads to something new and interesting
- Scala unifies:
  - Algebraic data types with class hierarchies
  - Functions with objects
  - Modules with objects

1st Unification: 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
  - Violates representation independence!

Pattern Ma

The case modifier of an object or class means you can pattern match on it

- Here's a a set of definitions describing binary trees:
- And here's an in order traversal of binary trees:
- 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]

```
abst  class Tree[T]

case object Empty extends Tree

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

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

2nd Unification: Functions Are Objects

- Scala is a functional language, in the sense that every function is a value
- Functions can be anonymous, curried, nested
- Familiar higher-order functions are implemented as methods of Scala classes

```
matrix.exists( row => row.forall(0 ==)) )
```

- Here, matrix is assumed to be of type Array[Array[int]]

Function Classes

- If functions are values, and values are objects, it follows that functions themselves are objects
- In fact, the function type S => T is equivalent to scala.Function1[S, T] where Function1 is defined as follows in the standard Scala library:

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

- Analogous conventions exist for functions with more than one argument
- Hence, 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): 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:
  - a(i) = a(i) + 2 for
  - a.update(i, a.apply(i) + 2)

```
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

```
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 mspati
- If no pattern matches, the actor suspends

```
// asynchronous message send
actor ! message

// message receive
receive {
    case msgpat1 => action1
    ...
    case msgpatn => actionn
}
```

A partial function of type PartialFunction[MessageType, ActionType]

Example: Orders and Cancellations

```
val orderManager =
  actor {
    loop {
      receive {
        case Order(item) =
          val o = handleO der(sender, it n); sender ! Ack(o)
        case Cancel(o: Order) =>
          if (o.pending) { cancelOrder(o); sender ! Ack(o) }
          else sende
        case x =>
          junk += x
      }}}
val customer = actor {
  orderManager ! myOrder
  ordermanager receive { case Ack => ... }
}
```

repeatedly receive messages

otherwise, if a

Spawn

Order was received...

... otherwise file message in junk

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

```
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?

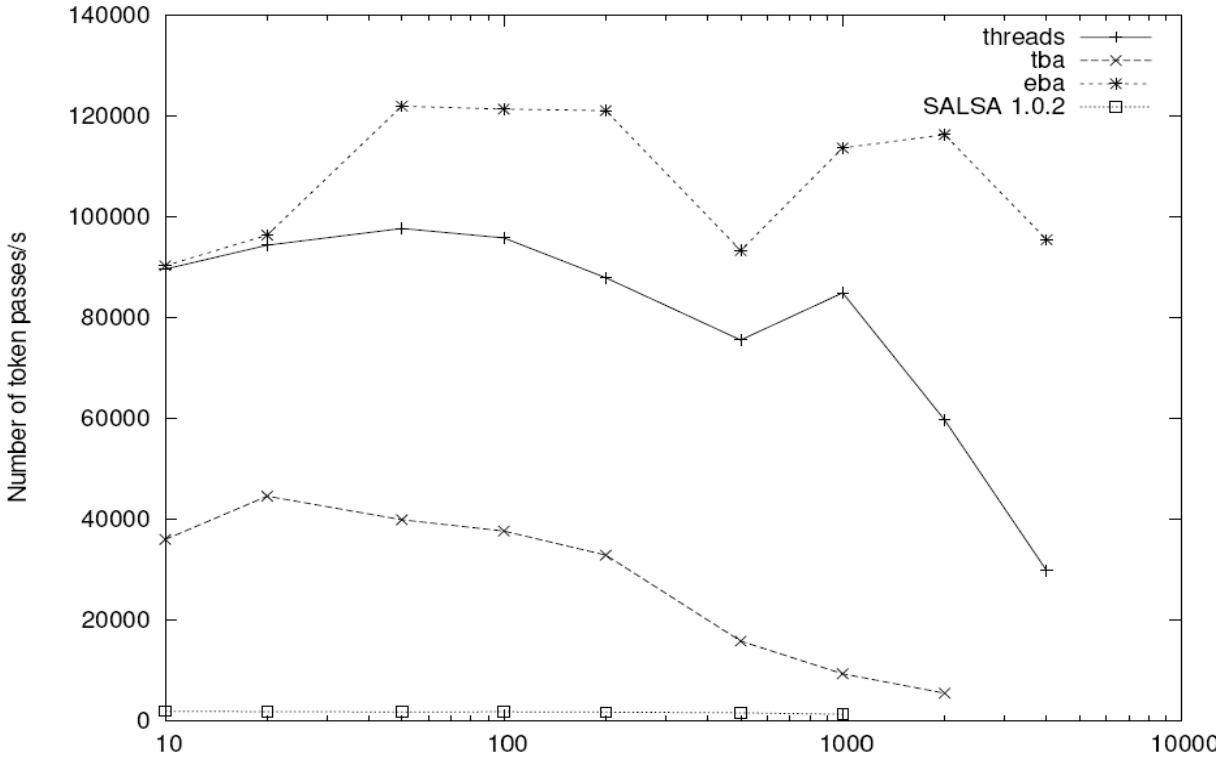
- A possible objection to Scala's library-based approach is:
  - **Why define actors in a library when they exist already in purer, more optimized form in Erlang?**
- One good reason: libraries are much easier to extend and adapt than languages

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

Performance: React vs. Receive

- Number of token passes per second in a ring of processes



3rd Unification: Modules Are Objects

- Scala has a powerful type system which enables new ways of **abstracting** and **composing** components
- Main innovation: components in Scala can define **required services** as well as **provided services**
- This is supported with a type system where types can be members of classes (abstract or concrete)
- **In Scala:**
  - Component ≅ Class
  - Interface ≅ Trait
  - Required Component ≅ Abstract Type Member or
  - Composition ≅ Modular Mixin Composition
- For more info: [Scalable Component Abstractions, OOPSLA 05]

Conclusion

- Scala blends functional and object-oriented programming
- This has worked well in the past: for instance, in Smalltalk, Python, or Ruby
- However, Scala is the first to unify 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

Try it out: scala-lang.org

- Thanks to the (past and present) members of the Scala team:
  - Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Philipp Haller, Sean McDermid, Adriaan Moors, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, Matthias Zenger

Relationship Between Scala and Other Languages

- Main influences on the Scala design: Java programming language, 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 programming language, Nice as other extensions of 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

The Scala Experience
Safe Programming Can Be Fun!

Martin Odersky

EPFL
Lausanne, Switzerland

TS-2844