

# Tackling Concurrency – Language or Library?

Martin Odersky

EPFL

# Concurrency is indispensable

---

Two converging trends:

- Concurrent Programs
  - ▷ Distributed Software
  - ▷ Webservices
  - ▷ Rich user-interactions
  - ▷ Reactive systems
- Concurrent Hardware
  - ▷ Multi-cores
  - ▷ Hyperthreading
  - ▷ GRID computing

Both require concurrency made accessible for “ordinary programmers”.

# Concurrency is hard

---

- State-space explosion due to interleaving
  - ... makes it much harder to test programs exhaustively.
- Races. vs Deadlock
  - ... “overengineering” often leads from one to the other.
- Nondeterminism
  - ... Hard to reproduce results, including faulty ones.

# Remedies?

---

No wonder, then, that a large number of solutions to the concurrency problems have been promoted ...

- (Various kinds of) locks and monitors
- STM's
- Pi-calculus and CCS
- CSP, Occam
- Asynchronous Pi-calculus, Pict
- Join calculus, functional nets, JoCaml, polyphonic C#.
- Concurrent logic programming, Oz
- Actors, Erlang
- CML synchronous events
- Ada rendezvous

... just to list some examples.

# What to do?

---

- At present, it seems to be too hard to predict which language constructs (or combinations thereof) will be instrumental in solving the concurrency problem.
- More experimentation is needed.
- But to gain useful experience, we need to apply many different concepts to large, real-life problems.
- If (concept  $\Leftrightarrow$  programming language), this is expensive and slow.
- What's more, the results might be inconclusive!

## A library-based approach

---

As long as we can model concurrency concepts by libraries, things are easier.

- Less investment to try out something new.
- Easier to modify initial designs.
- Easier to mix and match different approaches.

But...

- Can we get acceptable usability and efficiency that way?

This will depend crucially on the abstraction capabilities of the underlying programming language.

Our claim: Scala's blend of FP and OOP provides a good foundation for concurrency libraries.

## The rest of this talk

---

- Gives an introduction to Scala,
- Shows how unifying constructs in OOP and FP gives new design possibilities.
- Presents an implementation of Erlang's actor model in Scala.  
*(Erlang's actors are one of very few success stories for concurrent programming on a large scale).*
- Shows how this model can be made much more efficient on the JVM by trading off *thread-based* and *event-based* implementations.

# 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) higher-order functions,
- (3) pattern matching, and
- (4) novel ways to abstract and compose programs.

An open-source distribution of Scala has been available since Jan 2004.

Currently:  $\approx$  1000 downloads per month.

# A unified object model

---

In Scala, every value is an object and every operation is a method invocation.

**Example:** A class for natural numbers

```
abstract class Nat {  
  def isZero: boolean  
  def pred: Nat  
  def succ: Nat = new Succ(this)  
  def + (x: Nat): Nat = if (x.isZero) this else succ + x.pred  
  def - (x: Nat): Nat = if (x.isZero) this else pred - x.pred  
}
```

Here are the two canonical implementations of Nat:

```
class Succ(n: Nat) extends Nat {  
  def isZero: boolean = false;  
  def pred: Nat = n  
}
```

```
object Zero extends Nat {  
  def isZero: boolean = true;  
  def pred: Nat = error("Zero.pred")  
}
```

# Higher-order functions

---

- 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. E.g.:  
`matrix exists (row => row forall (0 ==))`
- Here, `matrix` is assumed to be of type `Array[Array[int]]`, using Scala's `Array` class (explained below)

# Functions in an object-oriented world

---

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

In fact, the function type  $S \Rightarrow T$  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  $x: \text{int} \Rightarrow x + 1$  is expanded as follows.

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

## Why should I care?

---

Since  $\Rightarrow$  is a class, it can be subclassed.

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

An obvious use is for arrays – mutable functions over integer ranges.

```
class Array[A](length: int) extends (int  $\Rightarrow$  A) {  
  def length: int = ...  
  def apply(i: int): A = ...  
  def update(i: int, x: A) { ... }  
  def elements: Iterator[A] = ...  
  def exists(p: A  $\Rightarrow$  boolean): boolean = ...  
}
```

Another bit of syntactic sugaring lets one write:

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

# 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 express control structures that are not easily expressible otherwise (see below).

# Pattern matching

---

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!”

# Pattern matching in Scala

---

Scala sees algebraic data types as special cases of class hierarchies.

- No separate algebraic data types – every type is a class.
- Can pattern match directly over classes.
- A pattern can access the constructor parameters of a (case) class.

# Example (1): A Class Hierarchy of Terms

---

```
class Term[T]
case class Lit    (x: int)           extends Term[int]
case class Succ   (t: Term[int])     extends Term[int]
case class IsZero (t: Term[int])     extends Term[boolean]
case class If[T]  (c: Term[boolean],
                  t1: Term[T],
                  t2: Term[T])      extends Term[T]
```

- The `case` modifier in front of a class means you can pattern match on it.
- Note that some subclasses *instantiate* the type parameter `T`.
- One cannot describe this hierarchy using a plain old ADT, but a GADT would do.

## Example (2): A Typed Evaluator

---

```
class Term[T]
case class Lit    (x: int)           extends Term[int]
case class Succ   (t: Term[int])     extends Term[int]
case class IsZero (t: Term[int])     extends Term[boolean]
case class If[T]  (c: Term[boolean],
                  t1: Term[T],
                  t2: Term[T])       extends Term[T]

def eval[T](t: Term[T]): T = t match {
  case Lit(n)           => n // T = int
  case Succ(u)          => eval(u) + 1 // T = int
  case IsZero(u)        => eval(u) == 0 // T = boolean
  case If(c, t1, t2)    => eval(if (eval(c)) t1 else t2)
}
```

Note that `eval` instantiates a differently for each case.

# Actors

---

An actor is a process that communicates with other actors via message passing.

Two principal constructs (adopted from Erlang).

```
actor ! message           // asynchronous message send
receive {                 // message receive
  case msgpat1 ⇒ action1
  ...
  case msgpatn ⇒ actionn
}
```

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 `msgpati`.

If no pattern matches, the actor suspends.

# Actors and threads

---

On the JVM, actors are executed by threads.

`actor { body }` creates a new actor, which runs the body *body*.

The `self` method returns the currently executing actor.

Both methods are defined by object `Actor`:

```
trait Actor { ... }  
object Actor {  
  def self: Actor ...  
  def actor { body: => unit }: Actor ...  
  ...  
}
```

A set of worker threads is used to execute all runnable actors.

Every Java thread is also an actor, so even the main thread can execute `receive`.

# Example: orders and cancellations

---

```
import scala.actors.Actor._

val orderManager = actor {
  loop {
    receive {
      case Order(sender, item) =>
        val o = handleOrder(sender, item); sender ! Ack(o)
      case Cancel(o : Order) =>
        if (o.pending) { cancelOrder(o); sender ! Ack(o) }
        else sender ! NoAck
      case x =>
        junk += x
    }}}

val customer = actor {
  orderManager ! myOrder
  ordermanager receive { case Ack => ... }}
```

# Implementing receive

---

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

```
object Actor {  
  ...  
  def receive[A](f: PartialFunction[Message, A]): A = {  
    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.

# Sender ID

---

Messages in the actors library carry the identity of the sender with them.

This enables the following operations:

```
sender          // The actor that sent the message that was last
                // received by self.
reply(msg)     // reply with 'msg' to sender
a !? msg      // send 'msg' to 'a', return reply.
a forward msg  // send 'msg' to 'a', with current 'sender' as sender-id
```

With these additions, the order manager can be written as follows.

```
val orderManager = actor {
  loop {
    receive {
      case Order(item) =>
        val o = handleOrder(sender, item); reply(Ack(o))
      case Cancel(o: Order) =>
        if (o.pending) { cancelOrder(o); reply(Ack(o)) }
        else reply(NoAck)
      case x =>
        junk += x
    }
  }
}
val customer = actor {
  orderManager !? myOrder match { case Ack => ... }
}
```

## Example: producers

---

Let's write an abstraction of *producers*, which act like iterators on the outside, but which use a `produce` method to generate values.

A typical use case:

```
class InOrder(n: IntTree) extends Producer[int] {  
  def produceValues = traverse(n)  
  def traverse(n: IntTree) {  
    if (n != null) {  
      traverse(n.left)  
      produce(n.elem)  
      traverse(n.right)  
    }  
  }  
}
```

- Producers implement method `produceValues` in class `Producer`.
- They call `produce`, which is inherited from `Producer`.

# Implementing producers

---

Producers are implemented in terms of two actors:

A *producer* actor which runs `produceValues` ...

```
class Producer[T] extends Iterator[T] {  
  protected def produceValues  
  def produce(x: T) { coordinator !? Some(x) }  
  private val producer = actor {  
    produceValues  
    coordinator ! None  
  }  
}
```

... and a *coordinator* actor which synchronizes

- (1) requests from clients (messages `Next` and `HasNext`),
- (2) values coming from the producer (messages `Some` and `None`).

# Coordinators

---

Several strategies are possible for the `coordinator` actor.

The following yields maximum parallelism through an internal queue:

```
private val coordinator = actor {  
  val q = new Queue[Option[Any]]  
  loop {  
    receive {  
      case HasNext if !q.isEmpty =>  
        reply(q.front != None)  
      case Next if !q.isEmpty =>  
        q.dequeue match { case Some(x) => reply(x) }  
      case x: Option[_] =>  
        q += x; reply()  
    }  
  }  
}
```

Why did we not we use the coordinator's mailbox to buffer produced values?

## 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 is that libraries are much easier to extend and adapt than languages.

For instance, the first version of the Scala library attached one thread to each Actor (just like Erlang does).

This is a problem for Java, where threads are expensive.

Erlang is *much* better at handling many threads, but even it can be overwhelmed by huge numbers of actors.

# Event-based actors

---

An alternative are event-based actors.

Normally, this means inversion of control, with a global rewrite of the program.

But if actors are implemented as a library, it is easy to implement a variation of `receive` (call it `react`) which liberates the running thread when it blocks for a message.

The only restriction is that `react` should never return normally:

```
def react(f: PartialFunction[Message, unit]): Nothing = ...
```

Client-code is virtually unchanged between the multi-threaded and event-based versions of the library.

## Coordinators again

---

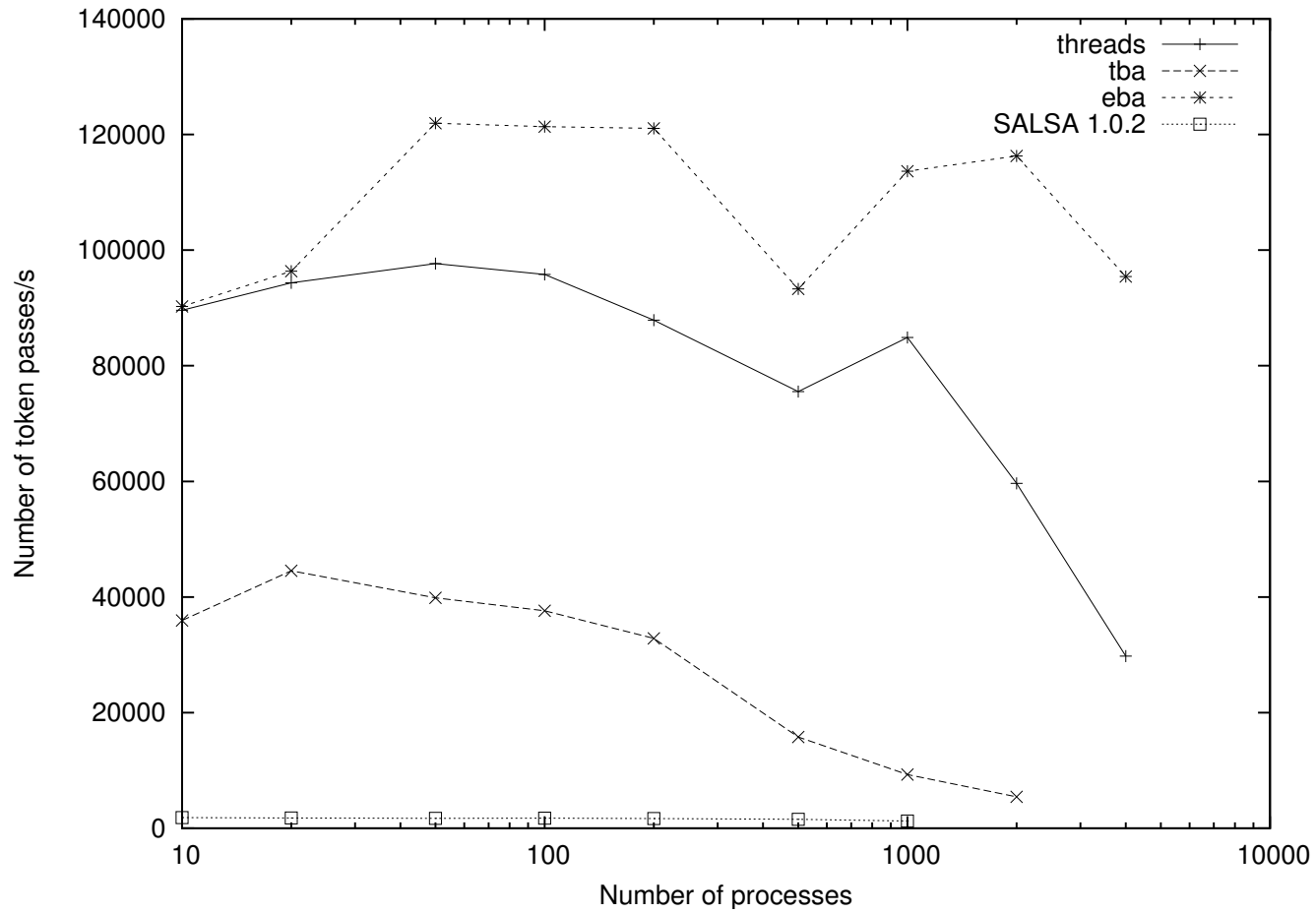
For instance, it is possible to economize one thread in `Producer` by changing the `receive` in the coordinator process to a `react`:

```
private val coordinator = actor {  
  val q = new Queue[Option[Any]]  
  loop {  
    react {  
      case HasNext if !q.isEmpty =>  
        reply(q.front != None)  
      case Next if !q.isEmpty =>  
        q.dequeue match { case Some(x) => reply(x) }  
      case x: Option[_] =>  
        q += x; reply()  
    }  
  }  
}
```

Note: This depends on the way `loop` is implemented (see below).

# Performance: react vs. receive

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



# Composing Actors

---

Actors can be composed for *chaining* and *selective communication*.

```
a orElse b      // runs first a. If a tries to suspend,  
                // actor b is run instead.  
  
a andThen b     // runs a followed by b.  
                // This works even if a ends in a react.
```

The `loop` combinator in `Actor` is implemented in terms of `andThen`:

```
def loop(body: => unit) = body andThen loop(body)
```

Hence, the body of `loop` can end in a `react` invocation.

# Actors and Channels

---

Message-based concurrency can be expressed in two forms:

**Actors:** messages are sent directly to an actor process

**Processes and channels:** messages are sent via channels.

Advantages of the actor model:

- + Simplicity, one less indirection.
- + Lax/flexible typing, types recovered through pattern matching.
- + Locality – receives are restricted to one thread.

Advantages of the process/channel model:

- + Generality – actors are a special case.
- + Strong typing.
- + Private communications.

# Best of Both Worlds

---

- In the Scala API, actors are special cases of channels.

```
trait OutputChannel[-Msg] {  
  def !(msg: Msg)  
  def forward(msg: Msg)  
}  
class Channel[Msg] extends InputChannel[Msg] with OutputChannel[Msg]  
trait Actor extends OutputChannel[Any] { ... }
```

- This allows the creation of channels separate from actors, for, e.g. better type discrimination, or private communication.
- `orElse` provides for selective communication over several channels.
- Restriction to ensure locality:

Only the actor which created a channel can receive messages from that channel.

# Summary

---

- Actors are a nice structuring method for concurrent systems.
  - + High-level communication through messages and pattern matching.
  - + Races are avoided by design.
- The safety of actors rests on a policy that actors communicate only through mailboxes, not through other shared memory.
- In Erlang, this is enforced by the language.
- In the Scala API, we leave this to the programmer.
  - + more flexibility, potentially better performance,
  - higher risk.

Can we design a type system to control memory locality?

# Conclusion

---

- Despite 10+ years of research, there are still interesting things to be discovered at the intersection of functional and object-oriented programming.
- Much previous research concentrated on simulating *some of X* in *Y*, where  $X, Y \in \{\text{FP}, \text{OOP}\}$ .
- More things remain to be discovered if we look at symmetric combinations.
- Scala is one attempt to do so.

Try it out: [scala.epfl.ch](http://scala.epfl.ch)

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, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, Matthias Zenger.