A scalable language works for very small and very large programs. Cross-languages communication becomes unnecessary.

If a domain-specific concern is embedded in a program, it is possible to type-check it w.r.t. the rest of the program. Scala embeds concerns such as relational queries or grammars as libraries in a type-safe way.

Scalable Languages

A language is scalable if it is suitable for very small as well as very large programs. This means that a single language can be used both for extension scripts and for the heavy lifting.

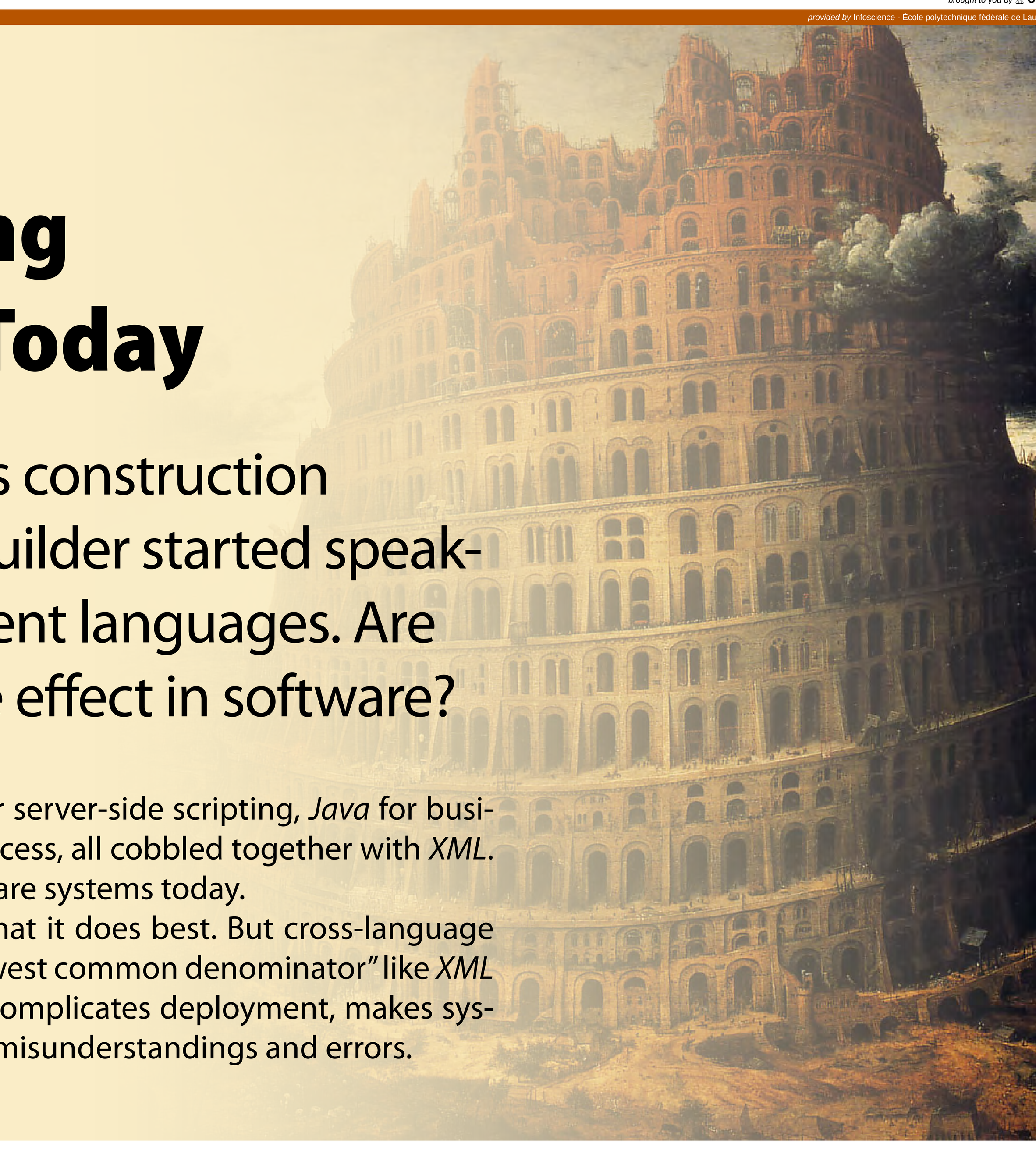
Domain-specific needs are provided for by libraries and embedded languages, instead of external tools.

Scala shows that such languages can exist. It is equally suitable for financial applications, massive multiplayer online games, web frameworks such as *Lift* or compilers. Unsurprisingly, the Scala compiler itself is written in Scala.

Scripting Script writers are primarily concerned with conciseness. Scala's type inference, efficient syntax and boiler-plate-scrapping features all fit this requirement perfectly. An interactive shell is available.

Composition There is no dedicated module system in Scala. Instead, classes and traits can be composed via mixins. Module abstraction is obtained through type parameters, type members and self types.

Programming Languages Today



The Tower of Babel's construction stopped when its builder started speaking too many different languages. Are we seeing the same effect in software?

JavaScript on the client, *Python* for server-side scripting, *Java* for business logic and *SQL* for database access, all cobbled together with *XML*. This is quite typical for large software systems today.

Each language may be used at what it does best. But cross-language communication must rely on a "lowest common denominator" like *XML* or worse, strings (as in *SQL*). This complicates deployment, makes systems fragile and is a big source of misunderstandings and errors.

Research Today, Scala is a language that scales down and up easily. It also works well with its mixed community of expert (designing the framework) and nonexpert users. The original goal of designing a more expressive language is all but reached: small and large problems can be solved, experts and beginners can use the language at their own level.

However, Scala's capability to guarantee safety is still limited, despite its rich type system. Domain-specific safety properties cannot always be encoded in the type system. Various research projects at LAMP are looking at means to provide *pluggable type-systems* and type annotations.

Also, our claim that "everything can be implemented as a library" still remains contentious, and LAMP is researching means to implement modern language features, such as transactions or embedded query languages, as libraries.

This poster was conceived by Gilles Dubochet

scala

Martin Odersky with the current and past LAMP team: Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sebastian Hack, Philipp Haller, Sean McDirmid, Ingo Maier, Adriaan Moors, Stéphane Micheloud, Nikolay Mihaylov, Lukas Rytz, Michel Schinz, Alexander Spoon, Erik Stenman, Geoffrey Washburn and Matthias Zenger.

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Scala vs. Java

Except for type annotations, Scala's syntax is very similar to Java's. On the other hand, features such as semicolons and type inference and lightweight classes and functions mean Scala's syntax feels a lot lighter.

In Scala	In Java
Method definitions	
`def mth(x: Int): Int = {` `  result }`	`int mth(int x) {` `  return result; }`
Variable definitions	
`var x: Int = …`	`int x = …;`
`val s: String = …`	`final String s = …;`
Method calls	
`obj.mth(arg)`	`obj.mth(arg);`
`obj mth arg`	*no operator overloading*
Choice expressions	
`if (cond) exp1 else exp2`	`cond ? exp1 : exp2;`

`expr match {`
`  case pat1 => exp1`
`  case patn => expn`
`}`
	`switch (expr) {`
	`  case pat1 : return exp1;`
	`  case patn : return expn;`
	`}`

Classes	

`class Sample (x: Int,`
`              val p: Int)`
`{`
`  def mth1(y: Int) = …`
`}`
	`class Sample {`
	`  private final int x;`
	`  public final int p;`
	`  Sample(int x, int p) {`
	`    this.x = x;`
	`    this.p = p; }`
	`  int mth1(int y) {`
	`    return …; }`
	`}`

Objects	

`object Sample {`
`  def mth2(x: Int) = …`
`}`
	`class Sample {`
	`  static int mth2(int x) {`
	`    return …; }`
	`}`

Traits and mixins	

`trait T {`
`  def mth1(x: String): Int`
`  def mth2(x: Int) = …`
`  var field = …`
`}`
	`interface T {`
	`  int mth1(String x);`
	`}`
	no concrete methods or fields

`class C extends Sup with T`	`class C extends Sup` `        implements T`

Object-Oriented

Every value in Scala is an object. Java's deviations from a pure object model — primitives, statics — have been removed.

Functions are objects too: their behaviour is implemented by their "apply" method. They can be specialised by extension.

Scala's object model is richer than that of Java, and permits a form of multiple inheritance.

Besides parametric class types, similar to those of Java, Scala provides structural, existential and path-dependant types.

Example A queue of integers is defined and implemented (ImpQ). Dbl modifies any queue's behaviour. A new instance q is created, inheriting both the standard implementation and the doubling behaviour.

```
abstract class IntQ {
 def get: Int
 def put(x: Int)
}
class ImpQ extends IntQ {
 private val buf =
  new ArrayBuffer[Int]
 def get = buf.remove(0)
 def put = buf += x
}
trait Dbl extends IntQ {
 abstract override
 def put(x: Int) =
  super.put(2*x)
}
val q = new ImpQ with DblQ
q.put(1); q.get == 2
```

Functional

Many algorithms are written concisely using functional idioms.

Scala is a functional language in the sense that every function is a value. Functions can be anonymous, curried or nested.

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

A function can be partial if it is not defined on all of its domain. It can be tested for whether it is defined on a given value.

Pattern matching blocks are partial functions, which allows complex control structures to be expressed easily.

Example The quick-sort algorithm is implemented on the right. The first element becomes the pivot (pvt) and quick sort is recursively called on all elements smaller, respectively larger than pvt.

```
def sort(list: List) =
 list match {
  case Nil => Nil
  case pvt :: rst =>
   sort(rst filter (_ < pvt))
   ::: List(pvt) :::
   sort(rst filter (_ > pvt))
 }
```

Interoperable

Seen at the byte-code level, *Scala is just another Java library*.

Scala fits seamlessly into a Java environment. All of Java's concepts map transparently to Scala. It is possible to call Java methods, select fields, inherit classes or implement interfaces from Scala code.

Java frameworks and tools, like *jdb*, *Wicket*, *Hibernate*, *Spring* or *Terracotta* also just work.

Performance is comparable with Java, and JVM improvements benefit directly to Scala.

Scala libraries can also be used from Java, albeit not as easily.

Example The Scala program on the right implements a minimal echo server using Java's NIO library. Java classes and methods, like ByteBuffer of write, are used transparently and have no performance overhead.

```
val s: ServerSocketChannel = …
while (s.isOpen) {
 val client = s.accept
 val bs =
  ByteBuffer.allocate(4096)
 client.read(bs)
 bs.flip
 client.write(bs) }
```

Concurrency using Actors and Messages

Actors encapsulate *state* and *behaviour* (like objects). They are also *active* and communicate through asynchronous message passing.

Actors implement the concurrent programming model of Erlang in Scala. Like objects, actors have state and behaviour. Unlike objects, they do not communicate through method calls but by sending asynchronous messages to other actors' *mailboxes*.

The treatment of incoming messages is done on an actor's thread so that all actors work concurrently. If an actor's mailbox is empty, the actor blocks until it receives something.

Concurrent actors can synchronize by waiting for messages. This is safer than lock-based synchronization.

Example The *ping-pong* program is a fascinating concurrent system where two players send each other "ping", respectively "pong" messages until they get bored.

Actors are defined using the actor method. The react loop of the actor is defined as a partial function on all messages it can receive at that point.

An actor will block until one message on which it reacts is received.

An actor sends messages using the send (!) operator and the implicit sender reference.

```
val player1 = actor {
  loop { react {
    case Ping =>
      sender ! Pong
    case Stop => exit
  } }
}
val player2 = actor {
  player1 ! Ping
  loop { react {
    case Pong =>
      if (gotBored) {
        sender ! Stop
        exit
      }
      else sender ! Ping
  } }
}
```

Performance It is not necessary to map each actor to a JVM thread. Instead, a thread pool is shared amongst actors so that blocked actors do not use scarce resources. Performance of actor-based concurrent application was found to be very high.

Inversion of control In publish-subscribe, a reaction to a message is in the publisher despite it conceptually being part of the subscriber's behaviour. In contrast, control in actors is not inverted.