State of Scala

Martin Odersky
Typesafe and EPFL
Trajectory

10 years ago
- The plan: Unify functional and object-oriented programming in a practical language.

8 years ago
- First Scala release (experimental, students as subjects)

5 years ago
- Open source system forms around Scala

3 years ago
- Industrial adoption starts, spreads rapidly through internet companies and financial sector.

1 year ago
- First Scala Days conference demonstrates strength of community, demand for industrial support.
#1 Concern: Concurrency and Parallelism

Two motivations:

- Make use of multicore/GPU processing power (scale-up)
- Distribute computations over many nodes (scala-out)

Two approaches:

- **Explicit**: Programmer defines and manages processes
- **Implicit**: Control of parallelism given to system

- Implicit is simpler,
- But explicit is often needed when nodes and connections can fail.
Scala’s Toolbox
Different Tools for Different Purposes

Implicit:

Collections

- Parallel Collections
- Distributed Collections

Parallel DSLs

Explicit:

Actors
- Software transactional memory
- Futures

Akka
Scala Collections: Building Interesting Things in Space

- De-emphasize destructive updates
- Focus on *transformers* that map collections to collections
- Have a complete range of *persistent* collections

```scala
scala> val ys = List(1, 2, 3)
ys: List[Int] = List(1, 2, 3)

scala> val xs: Seq[Int] = ys
xs: Seq[Int] = List(1, 2, 3)

scala> xs map (_ + 1)
res0: Seq[Int] = List(2, 3, 4)

scala> ys map (_ + 1)
res1: List[Int] = List(2, 3, 4)
```
Some General Scala Collections

Constructed automatically using \textit{decodify}. 
An Example

• Task: Phone keys have mnemonics assigned to them.

```scala
val mnemonics = Map(
  '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",
  '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")
```

• Assume you are given a dictionary `dict` as a list of words. Design a class Coder with a method `translate` such that

```scala
new Coder(dict).translate(phoneNumber)
```

produces all phrases of words in `dict` that can serve as mnemonics for the phone number.

• Example: The phone number “7225276257” should have the mnemonic

```
Scala rocks
```

as one element of the list of solution phrases.
Program Example: Phone Mnemonics

- This example was taken from:

- Tested with Tcl, Python, Perl, Rexx, Java, C++, C

- Code size medians:
  - 100 loc for scripting languages
  - 200-300 loc for the others
class Coder(words: List[String]) {

    private val mnemonics = Map(
        '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",
        '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")

    /** Invert the mnemonics map to give a map from chars 'A' ... 'Z' to '2' ... '9' */
    private val charCode: Map[Char, Char] = ??

    /** Maps a word to the digit string it can represent, e.g. “Java” -> “5282” */
    private def wordCode(word: String): String = ??

    /** A map from digit strings to the words that represent them, e.g. “5282” -> Set(“Java”, “Kata”, “Lava”, ...) */
    private val wordsForNum: Map[String, Set[String]] = ??

    /** Return all ways to encode a number as a list of words */
    def encode(number: String): Set[List[String]] = ??

    /** Maps a number to a list of all word phrases that can represent it */
    def translate(number: String): Set[String] = encode(number) map (x => x.mkString " ")
}

Outline of Class Coder
class Coder(words: List[String]) {

    private val mnemonics = Map(
        '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",
        '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")

    /** Invert the mnemonics map to give a map from chars 'A' ... 'Z' to '2' ... '9' */
    private val charCode: Map[Char, Char] =

    /** Maps a word to the digit string it can represent */
    private def wordCode(word: String): String = ??

    /** A map from digit strings to the words that represent them */
    private val wordsForNum: Map[String, List[String]] = ??

    /** Return all ways to encode a number as a list of words */
    def encode(number: String): Set[List[String]] = ??

    /** Maps a number to a list of all word phrases that can represent it */
    def translate(number: String): Set[String] = encode(number) map (_ mkString " ")
}
class Coder(words: List[String]) {

  private val mnemonics = Map(
    '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",
    '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")

  /** Invert the mnemonics map to give a map from chars 'A' ... 'Z' to '2' ... '9' */
  private val charCode: Map[Char, Char] =
    for ((digit, str) <- mnemonics; ltr <- str) yield (ltr -> digit)

  /** Maps a word to the digit string it can represent */
  private def wordCode(word: String): String = ??

  /** A map from digit strings to the words that represent them */
  private val wordsForNum: Map[String, List[String]] = ??

  /** Return all ways to encode a number as a list of words */
  def encode(number: String): Set[List[String]] = ??

  /** Maps a number to a list of all word phrases that can represent it */
  def translate(number: String): Set[String] = encode(number) map (_ mkString " ")
}
class Coder(words: List[String]) {

    private val mnemonics = Map(
        '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",
        '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")

    /** Invert the mnemonics map to give a map from chars 'A' ... 'Z' to '2' ... '9' */
    private val charCode: Map[Char, Char] =
        for ((digit, str) <- mnemonics; letter <- str) yield (letter -> digit)

    /** Maps a word to the digit string it can represent, e.g. “Java” -> “5282” */
    private def wordCode(word: String): String =

    /** A map from digit strings to the words that represent them */
    private val wordsForNum: Map[String, Set[String]] = ??

    /** Return all ways to encode a number as a list of words */
    def encode(number: String): Set[List[String]] = ??

    /** Maps a number to a list of all word phrases that can represent it */
    def translate(number: String): Set[String] = encode(number) map (_ mkString " ")
}
class Coder(words: List[String]) {

    private val mnemonics = Map(
        '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",
        '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")

    /** Invert the mnemonics map to give a map from chars 'A' ... 'Z' to '2' ... '9' */
    private val charCode: Map[Char, Char] =
        for ((digit, str) <- m; letter <- str) yield (letter -> digit)

    /** Maps a word to the digit string it can represent, e.g. “Java” -> “5282” */
    private def wordCode(word: String): String =
        word.toUpperCase map charCode

    /** A map from digit strings to the words that represent them */
    private val wordsForNum: Map[String, Set[String]] = ??

    /** Return all ways to encode a number as a list of words */
    def encode(number: String): Set[List[String]] = ??

    /** Maps a number to a list of all word phrases that can represent it */
    def translate(number: String): Set[String] = encode(number) map (_ mkString " ")
}

Class Coder (2)
class Coder(words: List[String]) {

    private val mnemonics = Map(
        '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",
        '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")

    /** Invert the mnemonics map to give a map from chars 'A' ... 'Z' to '2' ... '9' */
    private val charCode: Map[Char, Char] =
        for ((digit, str) <- m; letter <- str) yield (letter -> digit)

    /** Maps a word to the digit string it can represent */
    private def wordCode(word: String): String =
        word.toUpperCase map charCode

    /** A map from digit strings to the words that represent them,
     * e.g. "5282" -> Set("Java", "Kata", "Lava", ...) */
    private val wordsForNum: Map[String, List[String]] =

    /** Return all ways to encode a number as a list of words */
    def encode(number: String): Set[List[String]] = ??

    /** Maps a number to a list of all word phrases that can represent it */
    def translate(number: String): Set[String] = encode(number) map (_ mkString " ")
}

Class Coder (3)
class Coder(words: List[String]) {

  private val mnemonics = Map(
    '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",
    '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")

  /** Invert the mnemonics map to give a map from chars 'A' ... 'Z' to '2' ... '9' */
  private val charCode: Map[Char, Char] =
    for ((digit, str) <- mnemonics; letter <- str) yield (letter -> digit)

  /** Maps a word to the digit string it can represent */
  private def wordCode(word: String): String =
    word.toUpperCase map charCode

  /** A map from digit strings to the words that represent them,
  * e,g. "5282" -> Set("Java", "Kata", "Lava", ...) */
  private val wordsForNum: Map[String, List[String]] =
    (words groupBy wordCode) withDefaultValue List()

  /** Return all ways to encode a number as a list of words */
  def encode(number: String): Set[List[String]] = ??

  /** Maps a number to a list of all word phrases that can represent it */
  def translate(number: String): Set[String] = encode(number) map (_ mkString " ")
}
class Coder(words: List[String]) { ...

/** Return all ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] =

/** Maps a number to a list of all word phrases that can represent it */
def translate(number: String): Set[String] = encode(number) map (_.mkString " ")
}
class Coder(words: List[String]) { ... 
/** Return all ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] = 
  if (number.isEmpty)
    Set(List())
/** Maps a number to a list of all word phrases that can represent it */
def translate(number: String): Set[String] = encode(number) map (_ mkString " ")
}
```scala
class Coder(words: List[String]) { ...

/** Return all ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] = 
    if (number.isEmpty) 
        Set(List())
    else {
        for {
            splitPoint <- 1 to number.length
            Set(List())
        }
    }

/** Maps a number to a list of all word phrases that can represent it */
def translate(number: String): Set[String] = encode(number) map (_ mkString " ")
}
```
```scala
class Coder(words: List[String]) { ...  

  /** Return all ways to encode a number as a list of words */  
  def encode(number: String): Set[List[String]] =  
    if (number.isEmpty)  
      Set(List())  
    else {  
      for {  
        splitPoint <- 1 to number.length  
        word <- wordsForNum(number take splitPoint)  
      }  

  }  

  /** Maps a number to a list of all word phrases that can represent it */  
  def translate(number: String): Set[String] = encode(number) map (_ mkString " ")
```
class Coder(words: List[String]) { ... 

/** Return all ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] = 
  if (number.isEmpty)
    Set(List())
  else {
    for {
      splitPoint <- 1 to number.length
      word <- wordsForNum(number take splitPoint)
      rest <- encode(number drop splitPoint)
    } yield word :: rest
  }.toSet

/** Maps a number to a list of all word phrases that can represent it */
def translate(number: String): Set[String] = encode(number) map (_ mkString " ")
}
In Summary

• In the original experiment:
  – Scripting language programs were shorter and often even faster than Java/C/C++ because their developers tended to use standard collections.

• In Scala’s solution:
  – Obtained a further 5x reduction in size because of the systematic use of functional collections.

• Scala’s collection’s are also easily upgradable to parallel.

Think collection transformers, not CRUD!
A Solution in Java

By Josh Bloch, author of Java collections
Going Parallel

• In Scala 2.9, collections support parallel operations.
• Two new methods:
  
  c.par returns parallel version of collection c
  c.seq returns sequential version of collection c

• The right tool for addressing the PPP (popular parallel programming) challenge.
• I expect this to be the cornerstone for making use of multicores for the rest of us.
class Coder(words: ParVector[String]) {

    private val mnemonics = Map(
        '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",
        '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")

    /** Invert the mnemonics map to give a map from chars 'A' ... 'Z' to '2' ... '9' */
    private val charCode: Map[Char, Char] =
        for ((digit, str) <- m; letter <- str) yield (letter -> digit)

    /** Maps a word to the digit string it can represent */
    private def wordCode(word: String): String = word.toUpperCase map charCode

    /** A map from digit strings to the words that represent them */
    private val wordsForNum: Map[String, List[String]] = words groupBy wordCode

    /** Return all ways to encode a number as a list of words */
    def encode(number: String): Set[List[String]] =
        if (number.isEmpty) Set(List())
        else {
            for {
                splitPoint <- (1 to number.length).par
                word <- wordsForNum(number take splitPoint)
                rest <- encode(number drop splitPoint)
            } yield word :: rest
        }.toSet

    }
Parallel Collections

- Split work by number of Processors
- Each Thread has a work queue that is split exponentially. Largest on end of queue
- Granularity balance against scheduling overhead
- On completion threads "work steals" from end of other thread queues
General Collection Hierarchy

Traversable

← Iterable

← Seq

GenTraversable

↑ GenIterable

↑ GenSeq

ParIterable

↑ ParIterable

↑ ParSeq
Going Distributed

• Can we get the power of parallel collections to work on 10’000s of computers?
• Hot technologies: MapReduce (Google’s and Hadoop)
• But not everything is easy to fit into that mold
• Sometimes 100’s of map-reduce steps are needed.
• Distributed collections retain most operations, provide a powerful frontend for MapReduce computations.
• Scala’s uniform collection model is designed to also accommodate parallel and distributed.
The Future

Scala’s persistent collections are

- easy to use: *few steps to do the job*
- concise: *one word replaces a whole loop*
- safe: *type checker is really good at catching errors*
- fast: *collection ops are tuned, can be parallelized*
- scalable: *one vocabulary to work on all kinds of collections: sequential, parallel, or distributed.*

I see them play a rapidly increasing role in software development.
Akka: Event-driven Middleware in Scala

- Unified programming model for:
  - **scaling up** to many cores
  - **scaling out** to many nodes
  - fault-tolerance

- Actors:
  - lightweight (run millions on a single node)
  - send and receive messages async
  - location transparent

- Transparent and adaptive:
  - load-balancing and routing
  - replication and fail-over
  - elasticity: grow/shrink dynamically

- Decouples app logic from low-level mechanisms
Akka vs scala.actors

Akka:

- Best performance
- Scalable to clusters and clouds
- Best safety through encapsulation

Scala Actors:

- Simple to get started
- Good integration with threads
- Blocking as well as non-blocking receives
  - Erlang-inherited mailbox model can lead to inefficient code

Over the next releases we plan to merge the two libraries.
Other New Developments

• Scala-based reflection
• Type Dynamic
• Effects in an optional pluggable type system (?)

• Libraries:
  – I/O, including event-based
  – STM
  – Reactive programming, GUI

• Most effort will go in tools
  – Eclipse IDE
  – REPL
  – SBT (simple build tool)
  – Type debugger
Going further

But how do we keep a bunch of Fermi’s happy?
  – How to find and deal with 10000+ threads in an application?
  – Parallel collections and actors are necessary but not sufficient for this.

Our bet for the mid term future: parallel embedded DSLs.
  – Find parallelism in domains: physics simulation, machine learning, statistics, ...

Joint work with Kunle Olukuton, Pat Hanrahan @ Stanford.
EPFL side funded by ERC.
Example: Liszt - A DSL for Physics Simulation

- Mesh-based
- Numeric Simulation
- Huge domains
  - millions of cells
- Example: Unstructured Reynolds-averaged Navier Stokes (RANS) solver
val // calculating scalar convection (Liszt)

val Flux = new Field[Cell, Float]
val Phi = new Field[Cell, Float]
val cell_volume = new Field[Cell, Float]
val deltat = .001
...
until converged {
  for (f <- interior_faces) {
    val flux = calc_flux(f)
    Flux(inside(f)) -= flux
    Flux(outside(f)) += flux
  }
  for (f <- inlet_faces) {
    Flux(outside(f)) += calc_boundary_flux(f)
  }
  for (c <- cells(mesh)) {
    Phi(c) += deltat * Flux(c) / cell_volume(c)
  }
  for (f <- faces(mesh))
    Flux(f) = 0.f
}
Other Developments

Scala 2.9 also introduces two new “magic” traits:

DelayedInit and Dynamic

A lot of work has been done on the tools side:
Scala IDE for Eclipse 2.0,
SBT 0.10,
enhanced REPL,
Migration Manager Diagnostics (MiMaLib)
Planned Work

Over the next releases, we plan to introduce:

- **Scala STM**, a standard STM API,
- Enhanced support for **@specialized**,
- **Slick**, A common framework for connecting with databases and distributed collections.
- **Scala.reflect**, a Scala-centric reflection library.
- **Scala.react**, a reactive programming framework.
- An updated **Scala.NET**

We are also exploring:

- Adding **Scala.IO** to the standard distribution
- An **effect system** for Scala
- A **Scala to Javascript** translator
DelayedInit and App

Trait Application is convenient, but has hidden problems:

```scala
object First extends Application {
  println("hi there!")
}
```

Body is executed as part of the object initializer. 

`main` method inherited from `Application` is empty.

Body is neither optimized nor threading-friendly.

Solution: Replace `Application` with `App`.
DelayedInit and App

Trait App solves these problems:

```scala
object First extends App {
  println("hi there!")
}
```

Body is now stored in a buffer, executed by `main` method. Here’s an outline of this trait.

```scala
trait App extends DelayedInit {
  def main(args: Array[String]) =
    for (code <- inits) code()
}
```
Conclusion

Scala has the right “genes” to scale into parallel + distributed computing:

• Strong, practical foundation: JVM
• Functional programming reduces liabilities
• Library-centric approach makes for building tailored solutions.

Makes it possible to write:

• High-level, easy to use libraries (e.g., collections)
• Safer concurrency concepts (e.g., actors)
• Parallel DSLs (e.g., EPFL/Stanford project)
Thank You