An Overview of Scala

Philipp Haller, EPFL

(Lots of things taken from Martin Odersky's Scala talks)

The Scala Programming Language

- Unifies functional and object-oriented programming concepts
- Enables embedding rich domain-specific languages (DSLs)
- Supports high-level concurrent programming through library extensions that are efficient and expressive

A Scalable Language

- Language scalability: express small + large programs using the same constructs
- Unify and generalize object-oriented and functional programming concepts to achieve language scalability
- Interoperable with Java
  - .NET version under reconstruction
- Open-source distribution available since 2004
  - 5000 downloads/month (from EPFL)

From Java to Scala

object instead of **static** members

```
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)

}
```

Array[String] instead of **String[]**

Scala's version of the extended **for** loop

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

- S… ften the sa… Java's (… electi… ance)
- Scala compiles to JVM bytecodes

Functional Scala

Applies function on its right to each array element

```
object Example2 {
    def main(args: Ar    [String]) {
        println(args
                    .map(arg => arg.toUpperCase)
                    .mkString("  "))
    }
}
```

Closure that applies **toUpperCase** method to its String argument

Forms a string of all elements with a given separator between them

- Arrays are insta   es of
  abstractions
- ns instead of loops

Principles of Scala

Integrate OOP and FP as tightly as possible in a statically-typed language

(a) Unify algebraic data types (ADTs) and class hierarchies

(b) Unify functions and objects

ADTs and Pattern Matching

- FP: ADTs and pattern matching → concise and canonical manipulation of data structures
- OOP objects against ADTs:
  - Not extensible
  - Violate purity of OOP data model
- OOP objects against pattern matching:
  - Breaks encapsulation
  - Violates representation independence

Matching in Scala

case modifier enables pattern matching

```
abstract class Tree[+T]
case object Leaf extends Tree[Nothing]
case class  Fork(elem: T, left: Tree[T], right: Tree[T])
            extends Tree[T]
```

Binary trees

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

In-order traversal

- Purity: cases are objects or classes
- Extensibility: can define more cases elsewhere
- Encapsulation: only parameters revealed
- Representation independence: extractors [ECOOP'07]

Extractors

- Objects with unapply methods
- Pattern matcher implicitly calls unapply methods (if they exist)

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

Principles of Scala

Integrate OOP and FP as tightly as possible in a statically-typed language

✓ (a) Unify algebraic data types (ADTs) and class hierarchies

(b) Unify functions and objects

Functions in Scala

- Functions are first-class values
- Values are objects → functions are objects
- Function type `A => B` equivalent to type `Function1[A, B]`:

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

- Compilation of anonymous functions:

```
(x: Int) => x + 1
```

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

Subclassing Functions

- Arrays are mutable functions over integer ranges:

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

- Syntactic sugar:

```
a(i) = a(i) + 2
```

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

Partial Functions

- Defined only for subset of domain:

```
trait PartialFunction[-A, +B] extends (A => B) {
  def isDefinedAt(x: A): Boolean
}
```

- Anonymous partial functions:

```
{ case pat_1: A => body_1: B
  ...
  case pat_n: A => body_n: B }
```

```
new PartialFunction[A, B] {
  def isDefinedAt(x: A): Boolean = ...
  def apply(x: A): B = ... }
```

Principles of Scala

Integrate OOP and FP as tightly as possible in a statically-typed language

✓ (a) Unify algebraic data types (ADTs) and class hierarchies

✓ (b) Unify functions and objects

Library Extensions

- Functional objects enable rich embedded DSLs
- First-class partial functions enable definition of control structures in libraries
- Example: Scala Actors concurrency library

Scala Actors

- Two basic operations (adopted from Erlang)

```
actor ! message                   // message send

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

- Asynchronous send (`!`) buffers messages in receivers's mailbox
- Synchronous `receive` waits for message that matches any of the patterns $msgpat_i$

A Simple Actor

```
case class Data(bytes: Array[Byte])
case class Sum(receiver: Actor)

val checkSumCalculator: Actor =
  actor {
    var sum = 0
      loop {
        receive {
          case Data(bs)       => sum += hash(bs)
          case Sum(receiver) => receiver ! sum
        }
      }
  }
```

Implementing receive

```
def receive[R](f: PartialFunction[Message, R]): R =
  synchronized {
    mailbox.dequeueFirst(f.isDefinedAt) match {
      case Some(msg) =>
        f(msg)
      case None =>
        waitingFor = f.isDefinedAt
        suspendActor()

  }
}
```

Extracts first queue element matching given predicate

Queue of pending messages

Library vs. Language

- Libraries much easier to extend and adapt than languages
- Example: thread-based receive requires one VM thread per actor
  - Problem: high memory consumption and context switching overhead
  - Solution: second, non-returning receive operation called react that makes actors event-based
  - Haller, Odersky: *Actors that Unify Threads and Events*, Coordination'07

Extension: Joins for Actors

- Joins: high-level, declarative synchronization constructs (based on join calculus)
- Goal: enable join patterns alongside normal message patterns
- Example:

```
receive {
  case Put(x) & Get() => Get reply x
  case Some(other)    => ...
}
```

Implementing Joins

- Problem: outcome of matching depends on multiple message sends
  - When sending a `Get` message, the pattern `case Put(x) & Get()` matches *iff* there is also a `Put` message in the mailbox
- Idea: use extensible pattern matching to search mailbox

Matching Join Patterns

```
{ case &(Get(), Put(x)) => ... }
```

(gets compiled into)

```
new PartialFunction[?, Unit] {
  def isDefinedAt(y: ?) =
    &.unapply(y) match {
      case Some((Get(), Put(x))) => true
      case None => false }
```

Matching Join Patterns (cont'd)

```scala
{ case &(Get(), Put(x)) => ... }
```

(gets compiled into)

```scala
new PartialFunction[?, Unit] {
  def isDefinedAt(y: ?) =
    &.unapply(y) match {
      case Some((u, v)) =>
        Get.unapply(u) match {
          case true =>
            Put.unapply(v) match {
              case Some(x) => true
              case None    => false }
          case false => false }
      case None => false }
```

Scala Joins: Summary

- Novel implementation based on extensible pattern matching (Scala, F#)
  - New library-based solution
- More consistency checks
  - Re-use variable binding
  - Re-use guards
- More expressive
  - Nested patterns and guards
  - Dynamic join patterns

Scala Actors: Summary

- Scala library extension for high-level concurrent programming
  - Pair of message receive operations (receive/react) allows trade-off between efficiency and flexibility
- Message handlers as first-class partial functions
  - Enables extension of actor behavior
- Support for expressive join-style message patterns (Haller, Van Cutsem: *Implementing Joins using Extensible Pattern Matching*, Coordination'08)

Application: *lift* Web Framework

- Similar to Rails and Seaside, exercises many features of Scala
  - Actors: AJAX/Comet-style applications
  - Closures: HTML form elements
  - Traits/Mixins: persistence, data binding, query building
  - Pattern matching: extensible URL matching
- Use case: *Skittr*, a *Twittr* clone
- Excellent scalability: 10^6 actors on dual-core

Scala: Conclusion

- Integration of FP and OOP as tight as possible
- A scalable language: the same constructs express small and large programs
- Enables high-level concurrency libraries that are efficient and expressive
  - Example: Scala Actors
- Try it out: `http://www.scala-lang.org/`