Introduction to Scala

Computer Science Department, University of Crete

Parallel Programming

Based on slides by D. Malayeri, S.D. Vick, P. Haller, M. Madsen, J. Bonér
Introduction

- 1ο Μέρος: Εισαγωγή στη γλώσσα Scala
- 2ο Μέρος: Παράλληλος προγραμματισμός σε Scala
What is Scala?

- Scala is a statically typed language
  - Combines Object-Oriented Programming and Functional Programming
  - Developed in EPFL, lead by Martin Odersky
  - Influenced by Java, ML, Haskell, Erlang, and other languages
- Many high-level language abstractions
  - Uniform object model
  - Higher-order functions, pattern matching
  - Novel ways to compose and abstract expressions
- Managed language runtime
  - Runs on the Java Virtual Machine
  - Runs on the .NET Virtual Machine
Goals of Scala

- Create a language with better support for component software

Hypotheses:

- Programming language for component software should be scalable
  - The same concepts describe small and large parts
  - Rather than adding lots of primitives, focus on abstraction, composition, decomposition
- Language that unifies OOP and functional programming can provide scalable support for components
Why use Scala?

- Runs on the JVM
  - Can use any Java code in Scala
  - Almost as fast as Java
- Much shorter code
  - Odersky reports 50% reduction in most code
  - Local type inference
- Fewer errors
  - No NullPointer errors
- More flexibility
  - As many public classes per source file as you want
  - Operator overloading
- All of the above, for .NET too
Why learn Scala?

- Creating a trend in web service programming
  - LinkedIn
  - Twitter
  - Ebay
  - Foursquare
  - List is growing
Features of Scala (1)

- Both functional and object-oriented
  - Every value is an object
  - Every function is a value (including methods)
- Scala is statically typed
  -Includes local type inference system

Java 1.5

```java
Pair p = new Pair<Integer, String>(1, "Scala");
```

Scala

```scala
val p = new Pair(1, "Scala");
```
Features of Scala (2)

- Supports lightweight syntax for anonymous functions, higher-order functions, nested functions, currying
- ML-style pattern matching
- Integration with XML
  - Can write XML directly in Scala program
  - Can convert XML DTD into Scala class definitions
- Support for regular expression patterns
- Allows defining new control structures without using macros, and while maintaining static typing
- Any function can be used as an infix or postfix operator
- Can define methods named +, <= or ::
Features of Scala (3)

- Actor-based programming, distributed, concurrent
- Embedded DSLs, usable as scripting language
- Higher-kindred types, first class functions, closures
- Delimited continuations
- Abstract Types, Generics

Warning: Scala is the gateway drug to ML, Haskell, ...
Java

```java
public class Person {
    public final String name;
    public final int age;
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Scala

```scala
class Person(val name: String, val age: Int) {}
```
... and its use

Java

```java
import java.util.ArrayList;
...
Person[] people;
Person[] minors;
Person[] adults;
{ ArrayList<Person> minorsList = new ArrayList<Person>();
    ArrayList<Person> adultsList = new ArrayList<Person>();
    for (int i = 0; i < people.length; i++)
        (people[i].age<18 ? minorsList : adultsList).add(people[i]);
    minors=minorsList.toArray(people);
    adults=adultsList.toArray(people);
}
```

Scala

```scala
val people: Array[Person] = Array(
    new Person("Joe", 24),
    new Person("William", 23),
    new Person("Jack", 22),
    new Person("Averell", 21))
val (minors, adults) = people partition(_.age < 18)
```
Scala unifies class hierarchies and abstract data types (ADTs)
Introduces pattern matching for objects
Uses concise manipulation of immutable data structures
Example: Pattern matching

Class hierarchy for binary trees

abstract class Tree[T]
  case object Empty extends Tree[Nothing]
  case class Binary[T](elem: T, left: Tree[T], right: Tree[T]) extends Tree[T]

In-order traversal

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

- Extensibility
- Encapsulation: only constructor params exposed
- Representation independence
First-class functions make collections more powerful
Especially immutable ones

```scala
people.filter(_.age >= 18)
 .groupBy(_.surname)
 .values
 .count(_.length >= 2)
```
The Scala Object System

- Class-based
- Single Inheritance
- Can define singleton objects easily
- Subtyping is nominal: it is a subtype if declared to be a subtype
- Traits, compound types, views
  - Flexible abstractions
trait Nat;

object Zero extends Nat {
  def isZero: Boolean = true;
  def pred: Nat =
    throw new Error("Zero.pred");
}

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

- Similar to interfaces in Java
- They may have implementations of methods
- And can contain state!
- Can have multiple inheritance
trait Similarity {
    def isSimilar(x: Any): Boolean;
    def isNotSimilar(x: Any): Boolean = !isSimilar(x);
}

class Point(xc: Int, yc: Int) extends Similarity {
    var x: Int = xc;
    var y: Int = yc;
    def isSimilar(obj: Any) =
        obj.asInstanceOf[Point].x == x;
}
Mixin Class Composition (1)

- Mixin: “A class which contains a combination of methods from other classes.”
- Basic inheritance model is single inheritance
- But mixin classes allow more flexibility

```scala
class Point2D(xc: Int, yc: Int) {
    val x = xc;
    val y = yc;
    // methods for manipulating Point2Ds
}
class ColoredPoint2D(u: Int, v: Int, c: String) extends Point2D(u, v) {
    var color = c;
    def setColor(newCol: String): Unit = color = newCol;
}
class Point3D(xc: Int, yc: Int, zc: Int) extends Point2D(xc, yc) {
    val z = zc;
    // code for manipulating Point3Ds
}
class ColoredPoint3D(xc: Int, yc: Int, zc: Int, col: String)
    extends Point3D(xc, yc, zc) with ColoredPoint2D(xc, yc, col);

// ERROR: cannot mixin classes with classes, only traits
```
Mixing Class Composition (2)

- Fix: extract the code to be added into a trait
- Mixin the trait selectively into subclasses

```scala
class Point2D(xc: Int, yc: Int) {
    val x = xc;
    val y = yc;
    // methods for manipulating Point2Ds
}

trait Color {
    var color: String = null;
    def setColor(c: String): Unit = color = c;
}

class ColoredPoint2D(u: Int, v: Int, c: String) extends Point2D(u, v) with Color {
    color = c;
}

class Point3D(xc: Int, yc: Int, zc: Int) extends Point2D(xc, yc) {
    val z = zc;
    // code for manipulating Point3Ds
}

class ColoredPoint3D(xc: Int, yc: Int, zc: Int, col: String) extends Point3D(xc, yc, zc) with Color;
```
Mixin Class Composition (3)

- Mixin composition adds members explicitly defined in ColoredPoint2D (members that were not inherited)
- Mixing a class $C$ into another class $D$ is legal only as long as $D$’s superclass is a subclass of $C$’s superclass.
- *i.e.*, $D$ must inherit at least everything that $C$ inherited
- Why?

Remember that only members explicitly defined in ColoredPoint2D are mixin inherited. So, if those members refer to definitions that were inherited from Point2D, they had better exist in ColoredPoint3D. They do, since ColoredPoint3D extends Point3D which extends Point2D.
Mixin Class Composition (3)

- Mixin composition adds members explicitly defined in ColoredPoint2D (members that were not inherited)
- Mixing a class C into another class D is legal only as long as D’s superclass is a subclass of C’s superclass.
- *i.e.*, D must inherit at least everything that C inherited
- Why?
- Remember that only members explicitly defined in ColoredPoint2D are mixin inherited
- So, if those members refer to definitions that were inherited from Point2D, they had better exist in ColoredPoint3D
  - They do, since ColoredPoint3D extends Point3D which extends Point2D
Views (1)

- Defines an *implicit coercion* from one type to another
- Similar to conversion operators in C++ and C#

```scala
trait Set[T] {
  def extend(x: T): Set[T]
  def contains(x: T): Boolean
}

// ...
implicit def list2set[T](list: List[T]) : Set[T] = new Set[T] {
  def extend(x: T): Set[T] = list2set(x :: list)
  def contains(x: T): Boolean =
    !list.isEmpty && ((list.head == x) || (list.tail contains x))
}
```
Implicit views are inserted automatically by the Scala compiler.

If e is of type T then a view is applied to e if:
- Expected type of e is not T (or a supertype)
- A member selected from e is not a member of T

Compiler uses only views in scope.
Lazy Views

- Many containers have lazy views
- Do not compute until absolutely necessary
- Different meaning but same name with implicit views (!)

```
scala> (1 to 1000000000).filter(_%2 ==0).take(10).toList
java.lang.OutOfMemoryError: GC overhead limit exceeded
    at java.lang.Integer.valueOf(Integer.java:832)
    at scala.runtime.BoxesRunTime.boxToInteger(BoxesRunTime.java:69)
    at scala.collection.immutable.Range.foreach(Range.scala:166)
    at scala.collection.TraversableLike$class.filterImpl(TraversableLike.scala:258)
    at scala.collection.TraversableLike$class.filter(TraversableLike.scala:270)
    at scala.collection.AbstractTraversable.filter(Traversable.scala:104)
    ...
```

```
scala> (1 to 1000000000).view.filter(_%2 ==0).take(10).toList
res19: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
```
class Array[A] {
  def get(index: Int): A
  def set(index: Int, elem: A): Unit
}

- **Array[String]** is not a subtype of **Array[Any]**
- If it were, we could do the following:

```scala
val x = new Array[String](1);
val y : Array[Any] = x;
y.set(0, new FooBar());
// just stored a FooBar in a String array!
```
Variance Annotations (2)

- Covariance is OK with functional data structures
- ... because they are immutable

```scala
trait GenList[+T] {
  def isEmpty: Boolean;
  def head: T;
  def tail: GenList[T]
}
object Empty extends GenList[Any] {
  def isEmpty: Boolean = true;
  def head: Any = throw new Error("Empty.head");
  def tail: GenList[Any] = throw new Error("Empty.tail");
}
class Cons[+T](x: T, xs: GenList[T]) extends GenList[T] {
  def isEmpty: Boolean = false;
  def head: T = x;
  def tail: GenList[T] = xs
}
```
Variance Annotations (3)

- Can also have contravariant type parameters
  - Useful for an object that can only be written to
- Scala checks that variance annotations are sound
  - Covariant positions: Immutable field types, method results
  - Contravariant: method argument types
  - Type system ensures that covariant parameters are only used covariant positions
  - (similar for contravariant)
- If no variance specified, then *Invariant*
  - Neither superclass, nor subclass
Functions are Objects

- Every function is a value
  - Values are objects, so functions are also objects
- The function type $S \Rightarrow T$ is equivalent to the class type `scala.Function1[S, T]`

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

- For example, the anonymous successor function $(x: Int) \Rightarrow x + 1$ or in shorter code $(\_ + 1)$ expands to

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

- Arrays (mathematically): Mutable functions over integer ranges

**Syntactic Sugar**

\[ a(i) = a(i) + 2 \text{ for } a.update(i, a.apply(i) + 2) \]

**Example**

```scala
class Array[T](_length: Int) extends java.io.Serializable with java.lang.Cloneable {
  def length: Int = ...
  def apply(i: Int): T = ...
  def update(i: Int, x: T): Unit = ...
  override def clone: Array[T] = ...
}
```
Partial Functions

- Functions that are defined only for some objects
- Test using `isDefinedAt`

**Example**

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

- Blocks of pattern-matching cases are instances of partial functions
- This lets programmers write control structures that are not easy to express otherwise
Automatic Closure Construction

- Allows programmers to make their own control structures
- Can tag the parameters of methods with the modifier `=>`
- When method is called, the actual `=>` parameters are not evaluated and a no-argument function is passed
Example: Custom loop construct

```scala
object TargetTest1 {
  def loopWhile(cond: => Boolean)(body: => Unit): Unit = {
    if (cond) {
      body;
      loopWhile(cond)(body);
    }
  }

  def main(args: Array[String]) {
    var i = 10;
    loopWhile (i > 0) {
      Console.println(i);
      i = i - 1;
    }
  }
}
```
Types as Class Members

abstract class AbsCell {
  type T;
  val init: T;
  private var value: T = init;
  def get: T = value;
  def set(x: T): Unit = { value = x }
}

def createCell(): AbsCell =
  new AbsCell { type T = Int; val init = 1 }

- Clients of createCell cannot rely on the fact that \( T \) is \( \text{Int} \), since this information is hidden from them.
Scala Parallel Collections

val list = (1 to 10000).toList
list.map(_ + 42)

- Sequential map, addition
Scala Parallel Collections

```scala
val list = (1 to 10000).toList
list.par.map(_ + 42)
```

- Parallel list
- Many data structures available
  - ParArray
  - ParVector
  - mutable.ParHashMap
  - mutable.ParHashSet
  - immutable.ParHashMap
  - immutable.ParHashSet
  - ParRange
  - ParTrieMap
Examples: Operators

```scala
val lastNames = List(
    "Smith", "Jones", "Frankenstein", "Bach", "Jackson", "Rodin"
).par
lastNames.map(_.toUpperCase)

val parArray = (1 to 10000).toArray.par
parArray.fold(0)(_ + _)

val lastNames = List(
    "Smith", "Jones", "Frankenstein", "Bach", "Jackson", "Rodin"
).par
lastNames.filter(_.head >= 'J')
```
Examples: Create

```scala
click import scala.collection.parallel.immutable.ParVector
val pv1 = new ParVector[Int]
val pv2 = Vector(1,2,3,4,5,6,7,8,9).par
```
Parallel Collections

- Side-effecting operations can lead to non-determinism
  - side effects are reordered or concurrent
- Non-associative operations lead to non-determinism
  - order of operations changes
Example: Race!

```scala
var sum = 0
val list = (1 to 1000).toList.par
list.foreach(sum += _);
sum
// something

var sum = 0
list.foreach(sum += _);
sum
// something else
```
val list = (1 to 1000).toList.par
list.reduce(_-_)  // some result
list.reduce(_-_)  // some other result
list.reduce(_-_)  // yet another result, depending on what subtraction runs first
The Actor Model

- A model of concurrent computation
- Introduced in 1973 (Lisp, Simula)
- Main idea: *Everything is an Actor*
  - Similar to OO idea that *Everything is an Object*
- An actor can:
  - Send messages to other actors
  - Create new actors
  - React to messages it receives
- There is no constraint on order between these
  - Can occur in parallel across actors, also for any actor
  - Parallel computation and communication
Akers in Scala

- Initial built-in implementation
- Language primitives
- Built into the language
  - Obsolete now
- Integration with Akka library
  - Akka: library with distributed actors
  - Concurrency
  - Scalability
  - Fault-tolerance
  - Single unified programming model
  - Managed runtime (contained into the library)
  - Open Source
Actors in Akka

- Goal: Program at very high level of abstraction
- Do not think of shared state, threads, state visibility, locks, collections, etc.
- Only think how messages flow into the system
- Runtime system does the rest
  - High CPU utilization
  - Low latency
  - Scalability
  - Built-in support for error detection and recovery
Parallel and Distributed

- Akka actors are distributable by design
  - Designed to scale up (more threads) and scale out (more nodes)
  - Same program, different deployments
  - Perfect for cloud deployment
    - Elastic, dynamic
    - Fault-tolerant, self-healing
    - Adaptive load-balancing, migration
    - Loosely coupled, allows dynamic changes at runtime
What is an Actor

- Unit of code organization in Akka
- Actors help create concurrent, scalable and fault-tolerant applications
- Like Java-EE Servlets and session beans, Actors help organize code to keep “policy” and “business logic” separate
- Used in telecom systems with “9 nines” uptimes
- Abstraction intuitively: Virtual Machines in the Cloud (but faster)
  - Encapsulated, decoupled, black boxes
  - Manage their own memory and behavior
  - Communicate asynchronously, non-blocking messages
  - Can grow and shrink on demand, add new actors, stop some
  - Hot-deploy: change behavior at runtime, add new components, new code
  - Actors are the same, but for a single application
Actor uses

- May be alternative to:
  - Thread
  - Object instance, component
  - Callback Listener
  - Singleton, service
  - Load-balancer, router, thread pool
  - Jave EE Session Bean, Message-Driven Bean
  - Out-of-process service
  - FSM
Theoretical definition

- Fundamental unit of computation that embodies:
  - Processing
  - Storage
  - Communication

- 3 axioms - When an actor receives a message, it can:
  - Create new actors
  - Send messages to actors it knows
  - Designate how it should handle the next message received
Core Actor operations

- Define
- Create
- Send
- Become
- Supervise
Define an Actor

```scala
import akka.actor._

class Summer extends Actor {
  var sum = 0

  def receive = {
    case ints: Array[Int] =>
      sum += ints.reduceLeft((a, b) => (a + b) % 7)
    case "print" => println("Sum:" + sum)
  }
}
```
Create an Actor

- Create an instance of an Actor
- Very lightweight in Akka: 2.7 million actors per GB RAM
- Very strong encapsulation:
  - state
  - behavior
  - message queue
- State and behavior are indistinguishable
- Only way to observe state: send a message, see reaction
import akka.actor._
class Summer extends Actor {
  var sum = 0

  def receive = {
    case ints: Array[Int] =>
      sum += ints.reduceLeft((a, b) => (a+b) % 7)
    case "print" => println("Sum:" + sum)
  }
}

val system = ActorSystem("SummerSystem")
val summer = system.actorOf(Props[Summer], name = "summer")
Actors form Hierarchies

- The system is “guardian actor”
- Can create actors with `context.actorof()`, guarded by creating actor
- Hierarchies can be tall trees
- Name resolution works like a file system: `Actor /summer/someother`
Asynchronous and non-blocking: “Fire and Forget”

Everything happens Reactively

▶ An Actor is passive until a message is sent to it
▶ Messages are “kinetic energy” in Actor System
▶ But light messages may trigger heavy reactions

Everything is asynchronous and lockless

Lightweight: single machine can handle millions of messages per second
import akka.actor._
class Summer extends Actor {
  var sum = 0

  def receive = {
    case ints: Array[Int] =>
      sum += ints.reduceLeft((a, b) => (a + b) % 7)
    case "print" => println("Sum:" + sum)
  }
}

val system = ActorSystem("SummerSystem")
val summer = system.actorOf(Props[Summer], name = "summer")
summer tell (1 to 10).toArray
summer ! (1 to 20).toArray
import akka.actor._

class SomeActor extends Actor {
    def receive = {
        case User(name) =>
            sender tell ("Hi " + name)
    }
}
Remote Deployment

scala {
  actor {
    provider = akka.remote.RemoteActorRefProvider
    deployment {
      /Summer {
        remote = akka://SummerSystem@machine42:31337
      }
    }
  }
}
Actor Become

- Dynamically redefine actor behavior
- Triggered reactively by receiving a message
- Type system analogy: Object changes type
  - change interface, protocol, implementation
- Actor will now react differently to messages
- Behaviors are stacked, can be pushed and popped
Why?

- Let an actor with high contention become load-balancer, distribute work “behind”
- Implement FSM
- Graceful degradation
- Generic Worker easy spawn, becomes whatever is needed
  
- Very useful once you get used to it
context become {
    case NewMessage =>
    ...
}

Example: load balancing

```scala
val router = 
  system.actorOf(
    Props[SomeActor].withRouter(
      RoundRobinRouter(nrOfInstances = 5)
    )
  )
```

Pratikakis (CSD)
Scala
CS342, 2018 60/66
Example: load balancing++

```scala
val resizer = 
    DefaultResizer(lowerBound = 2, upperBound = 15)

val router = 
    system.actorOf(
        Props[SomeActor].withRouter(
            RoundRobinRouter(resizer = Some(resizer))
        )
    )
```
Failure Management, Traditionally

- Single thread of control
- If thread blows up, we’re $#@%@ed
- Must do explicit error handling within thread
- Errors do not propagate between threads
  - No way to find out if something broke
- Leads to defensive programming
  - if(printf()) ...
  - Error handling tangled with business logic
  - Error checking salted all over the code base
- Things shouldn’t be that bad
Manage another Actor’s failures
Error handling in actors by letting Actors monitor (supervise) each other for failure
If an Actor crashes, notification will be sent to supervisor
Clean separation of processing and error handling
Every actor has default supervisor strategy, usually sufficient
Example: Supervision

class Supervisor extends Actor {
  override val supervisorStrategy =
    OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 minute)
  {
    val worker = context.actorOf(Props[Worker])

    def receive = {
      case n: Int => worker.forward(n)
    }
  }
}
Example: Supervision

class Supervisor extends Actor {
    override val supervisorStrategy =
        AllForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 minute)
    {
        val worker = context.actorOf(Props[Worker])

        def receive = {
            case n: Int => worker forward n
        }
    }
}
class Worker extends Actor {
  ...
  override def preRestart(reason: Throwable, message: Option[Any]) {
    // Clean up before restart
  }

  override def postRestart(reason: Throwable) {
    // Initialize after restart
  }
}