The Scala Programming Language

Mooly Sagiv

Slides taken from
Martin Odersky (EPFL)
Donna Malayeri (CMU)
Hila Peleg (TAU)

Modern Functional Programming

- Higher order
- Modules
- Pattern matching
- Statically typed with type inference
- Two viable alternatives
  - Haskel
    - Pure lazy evaluation and higher order programming leads to Concise programming
    - Support for domain specific languages
    - I/O Monads
    - Type classes
  - OCaml
    - Encapsulated side-effects via references

Then Why aren’t FP adapted?

- Education
- Lack of OO support
  - Subtyping increases the complexity of type inference
- Programmers seeks control on the exact implementation
- Imperative programming is natural in certain situations

Why Scala?

(Coming from OCaml)

- Runs on the JVM/.NET
  - Can use any Java code in Scala
- Combines functional and imperative programming in a smooth way
- Effective library
- Inheritance
- General modularity mechanisms

The Java Programming Language

- Designed by Sun 1991-95
- Statically typed and type safe
- Clean and Powerful libraries
- Clean references and arrays
- Object Oriented with single inheritance
- Interfaces with multiple inhertitence
- Portable with JVM
- Effective JIT compilers
- Support for concurrency
- Useful for Internet

Java Critique

- Downcasting reduces the effectiveness of static type checking
  - Many of the interesting errors caught at runtime
    - Still better than C, C++
- Huge code blowouts
  - Hard to define domain specific knowledge
  - A lot of boilerplate code
  - Sometimes OO stands in our way
  - Generics only partially helps

Why Scala?
(Coming from Java/C++)

- Runs on the JVM/.NET
  - Can use any Java code in Scala
  - Almost as fast as Java (within 10%)
- Much shorter code
  - Odersky reports 50% reduction in most code over Java
  - Local type inference
- Fewer errors
  - No Null Pointer problems
- More flexibility
  - As many public classes per source file as you want
  - Operator overloading

Scala

- Designed and implemented by Martin Odersky [2001-]
- Motivated by towards “ordinary” programmers
- Scalable version of software
  - Focused on abstractions, composition, decomposition
- Unifies OOP and FP
  - Exploit FP on a mainstream platform
  - Higher order functions
  - Pattern matching
  - Lazy evaluation
- Interoperates with JVM and .NET
- Better support for component software
- Much smaller code

Scala

- Scala is an object-oriented and functional language which is completely interoperable with Java (.NET)
- Remove some of the more arcane constructs of these environments and adds instead:

  (1) a uniform object model,

  (2) pattern matching and higher-order functions,

  (3) novel ways to abstract and compose programs

Getting Started in Scala

- scala
  - Runs compiled scala code
  - Or without arguments, as an interpreter!
- scalac - compiles
- fsc - compiles faster! (uses a background server to minimize startup time)
- Go to scala-lang.org for downloads/documentation
- Read Scala: A Scalable Language
  (see http://www.artima.com/scalazine/articles/scalable-language.html)

Plan

- ✓ Motivation
- Scala vs. Java
- Modularity
- Discussion

Features of Scala

- Scala is both functional and object-oriented
  - every value is an object
  - every function is a value--including methods
- Scala is statically typed
  - includes a local type inference system:
  - **in Java 1.5:**

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

  - **in Scala:**

```
val p = new MyPair(1, "scala");
```

Basic Scala

- Use `var` to declare variables:

```
var x = 3;
x += 4;
```

```
let x = ref 3 in
x := !x  + 4
```

OCaml

- Use `val` to declare values (final vars)

```
val y = 3;
y += 4; // error
```

- Notice no types, but it is statically typed

```
var x = 3;
x = “hello world”; // error
```

- Type annotations:

```
var x : Int = 3;
```

Scala is interoperable

Scala programs interoperate seamlessly with Java class libraries:

- Method calls
- Field accesses
- Class inheritance
- Interface implementation

all work as in Java

Scala programs compile to JVM bytecodes

Scala's syntax resembles Java's, but there are also some differences

var: Type instead of Type var

```
object Example {
  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)
  }
}
```

Scala's version of the extended **for** loop
(use <- as an alias for ←)

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

Scala is functional

The last program can also be written in a completely different style:

- Treat arrays as instances of general sequence abstractions
- Use higher-order functions instead of loops

Arrays are instances of sequences with map and mkString methods

to each array el

```
object Example2 {
   def main(args: Array[String]) {
      println(args.
               map (_.toUpperCase) .
               mkString " ")
   }
}
```

A clo
h applies the
to its

mkString is a method of Array which forms a string of all elements with a given separator between them

mk_string map (fun x -> toUpperCase(x), args), " "

Functions, Mapping, Filtering

- Defining lambdas – nameless functions (types sometimes needed)

  val f = x :Int => x + 42;   ***f is now a mapping int-> int***

- Closures! ***A way to haul around state***

  ```
  var y = 3;
  val g = {x : Int => y += 1; x+y; }
  ```

- Maps (and a cool way to do some functions)

  ```
  List(1,2,3).map(_+10).foreach(println)
  ```

- Filtering (and ranges!)

  ```
  (1 to 100). filter (_ % 7 == 3). foreach (println)
  ```

  - (Feels a bit like doing unix pipes?)

Scala is concise

Scala's syntax is lightweight and concise

Contributors:

- type inference
- lightweight classes
- extensible API's
- closures as control abstractions

```
var capital = Map( "US" → "Washington",
                   "France" → "paris",
                   "Japan" → "tokyo" )

capital += ( "Russia" → "Moskow" )

for ( (country, city) ← capital )
    capital += ( country → city.capitalize )

assert ( capital("Japan") == "Tokyo" )
```

Average reduction in LOC wrt Java: ≥ 2

due to concise syntax and better abstraction capabilities

Big or small?

Every language design faces the tension whether it should be big or small:

- Big is good: expressive, easy to use
- Small is good: elegant, easy to learn

Can a language be both big and small?

Scala's approach: concentrate on abstraction and composition capabilities instead of basic language constructs

Scala adds	Scala removes
+ a pure object system	- static members
+ operator overloading	- special treatment of primitive types
+ closures as control abstractions	- break, continue
+ mixin composition with traits	- special treatment of interfaces
+ abstract type members	- wildcards
+ pattern matching	

The Scala design

Scala strives for the tightest possible integration of OOP and FP in a statically typed language

This continues to have unexpected consequences

Scala unifies

- algebraic data types with class hierarchies,
- functions with objects

Has some benefits with concurrency

ADTs are class hierarchies

Many functional languages have algebraic data types and pattern matching

$\Longrightarrow$

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*
- *and it violates representation independence!*

Pattern matching in Scala

The **case** modifier of an object or class means you can pattern match on it

Here's a a set of definitions describing binary trees:

```
a… class Tree[T]

case object Empty extends Tree

case class Binary(elem: T, left: Tree[T], right: Tree[T])
        extends Tree
```

And here's an inorder traversal of binary trees:

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

This design keeps

- purity: all cases are classes or objects
- extensibility: you can define more cases elsewhere
- encapsulation: only parameters of case classes are revealed
- representation independence using extractors [Beyond the scope of the course]

Pattern Scala vs. OCaml

```
abstract class Tree[T]

case object Empty extends Tree

case class Binary(elem: T, left: Tree[T], right: Tree[T])
        extends Tree
```

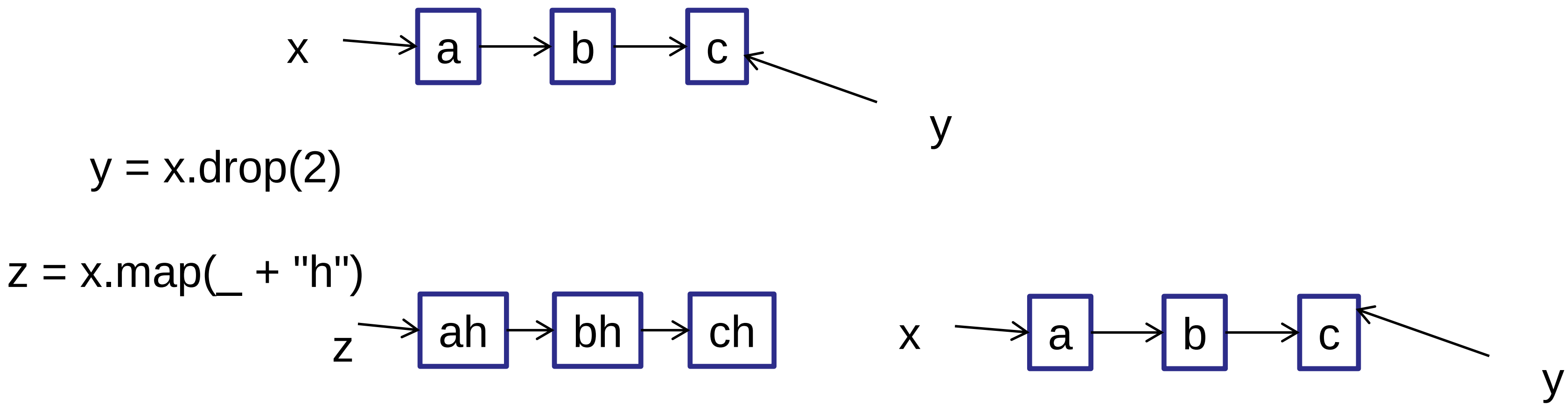
```
def inOrder [T] ( t: Tree[T] ): List[T] = t match {
 case Empty          => List()
 case Binary(e, l, r) => inOrder(l) ::: List(e) ::: inOrder(r)
}
```

```
type Tree = Empty | Binary of Element * Tree * Tree
```

```
let rec InOrder (t : tree)   =   match t with

  | Empty -> []

  | Binary (element, left, right) -> List.append(

              List.append(inOrder(left),  [element]),  InOrder(right))
```

Mutable vs. Immutable Data Structures

- Basic data structures in Scala are immutable
- Operations will copy (if they must)

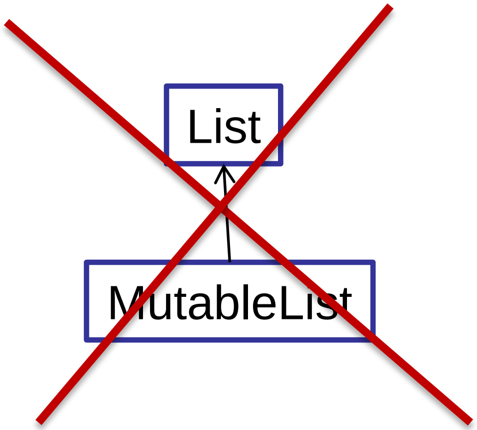


- Many positive consequences

Mutable vs. Immutable

- Mutable and immutable collections are not the same type hierarchy!
- Have to copy the collection to change back and forth, can't cast

x.toList



More features

- 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

Other features

- Allows defining new control structures without using macros, and while maintaining static typing
- Any function can be used as an infix or postfix operator
- Semicolon inference
- Can define methods named $+$, $<=$ or `::`

Automatic Closure Construction

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

While loop example

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

  var i = 10;
  loopWhile (i > 0) {
   Console.println(i);
    i = i – 1


  }
}
```

Define loopWhile method

Use it with nice syntax

Scala object system

- Class-based
- Single inheritance
- Can define singleton objects easily (no need for static which is not really OO)
- Traits, compound types, and views allow for more flexibility

Dependent Multiple Inheritance (C++)

```
class A {
     field a1;
     field a2;
     method m1();
     method m3();
};

class C extends A {
     field c1;
     field c2;
     method m1();
     method m2();
};

class D extends A {
     field d1;
     method m3();
     method m4();
};

class E extends C, D {
     field e1;
     method m2();
     method m4();
     method m5();
};
```

Traits

- Similar to interfaces in Java
- They may have implementations of methods
- But can’t contain state
- Can be multiply inherited from

Classes and Objects

```
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;
}
```

More on Traits

- Halfway between an interface and a class, called a *trait*
- A class can incorporate as multiple Traits like Java interfaces but unlike interfaces they can also contain behavior, like classes
- Also, like both classes and interfaces, traits can introduce new methods
- Unlike either, the definition of that behavior isn't checked until the trait is actually incorporated as part of a class

Another Example of traits

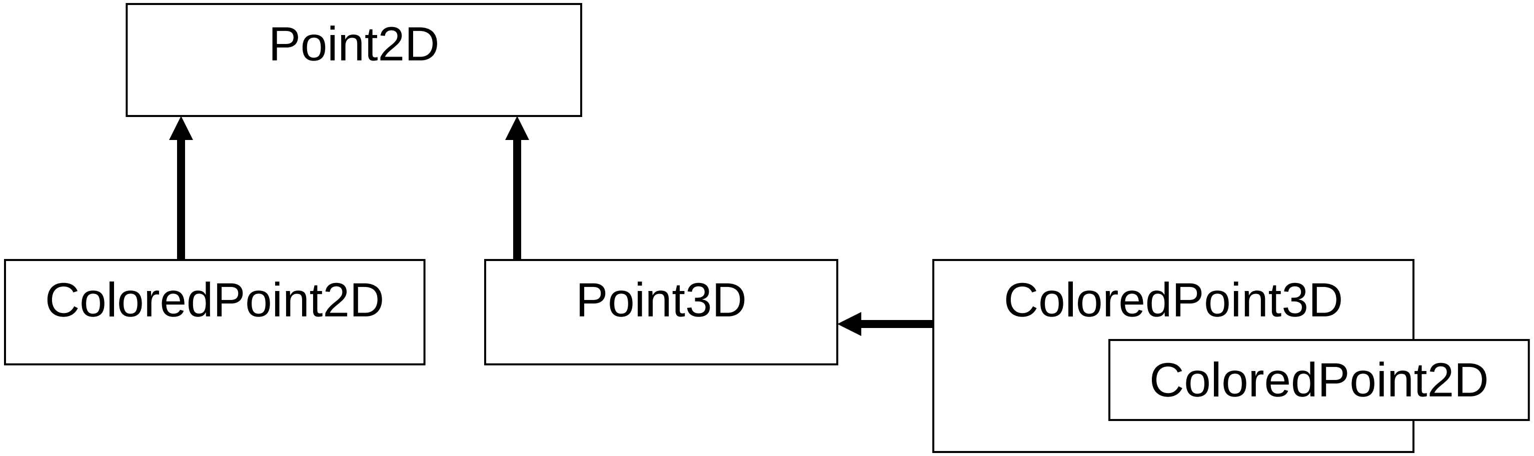
```
trait Similarity {
  def isSimilar(x: Any): Boolean;
  def isNotSimilar(x: Any): Boolean = !isSimilar(x);
}

class Point(xc: Int, yc: Int) with Similarity {
  var x: Int = xc;
  var y: Int = yc;
  def isSimilar(obj: Any) =
    obj.isInstanceOf[Point] &&
    obj.asInstanceOf[Point].x == x &&
    obj.asInstanceOf[Point].y == y ;
}
```

Mixin class composition

- Basic inheritance model is single inheritance
- But mixin classes allow more flexibility

```
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;
}
```

Mixin class composition example




```
class Point3D(xc: Int, yc: Int, zc: Int)
              extends Point2D(xc, yc) {
                val z = zc;
     // code for manipulating Point3Ds
class ColoredPoint3D(x Int, yc: Int, zc: Int, col: String)
              extends Point3D(xc, yc, zc)
              with ColoredPoint2D(xc, yc, col);
```

Mixin class composition

- Mixin composition adds members explicitly defined in ColoredPoint2D
  (members that weren't 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?

Mixin class composition

- 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

- Defines a *coercion* from one type to another
- Similar to conversion operators in C++/C#

```
trait Set {
  def include(x: int): Set;
  def contains(x: int): boolean
}

def view(list: List) : Set = new Set {
  def include(x: int): Set = x prepend list;
  def contains(x: int): boolean =
    !isEmpty &&
    (list.head == x || list.tail contains x)
}
```

Covariance vs. Contravariance

- Enforcing type safety in the presence of subtyping
- If a function expects a formal argument of type T1 → T2 and the actual argument has a type S1 → S2 then
  - what do have to require?
- If a function assumes a precondition T1 and ensures a postcondition T2
- If the caller satisfies a precondition S1 and requires that S2 holds after the call
  - What do we have to require?

Variance annotations

```
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 this:

```
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

- **Covariance is ok with immutable data structures**

```
trait GenList[+T] {
  def isEmpty: boolean;
  def head: T;
  def tail: GenList[T]
}
object Empty extends GenList[All] {
  def isEmpty: boolean = true;
  def head: All = throw new Error("Empty.head");
  def tail: List[All] = 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

- 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)

Missing

- Compound types
- Types as members
- Actors and concurrency
- Libraries

Resources

- The Scala programming language home page
  (see http://www.scala-lang.org/)
- The Scala mailing list
  (see http://listes.epfl.ch/cgi-bin/doc_en?liste=scala)
- The Scala wiki (see http://scala.sygneca.com/)
- A Scala plug-in for Eclipse
  (see http://www.scala-lang.org/downloads/eclipse/index.html)
- A Scala plug-in for IntelliJ
  (see http://plugins.intellij.net/plugin/?id=1347)

References

- The Scala Programming Language as presented by Donna Malayeri (see http://www.cs.cmu.edu/~aldrich/courses/819/slides/scala.ppt)
- The Scala Language Specification 2.7
- (seehttp://www.scala-lang.org/docu/files/ScalaReference.pdf)
- The busy Java developer's guide to Scala: Of traits and behaviorsUsing Scala's version of Java interfaces(see http://www.ibm.com/developerworks/java/library/j-scala04298.html)
- First Steps to Scala (in Scalazine) by Bill Venners, Martin Odersky, and Lex Spoon, May 9, 2007 (see http://www.artima.com/scalazine/articles/steps.html)

Summing Up [Odersky]

- Scala blends functional and object-oriented programming.
- This has worked well in the past: for instance in Smalltalk, Python, or Ruby
- However, Scala is goes farthest in unifying FP and OOP in a statically typed language
- This leads to pleasant and concise programs
- Scala feels similar to a modern scripting language, but without giving up static typing

Lessons Learned[Odersky]

1. Don’t start from scratch
2. Don’t be overly afraid to be different
3. Pick your battles
4. Think of a “killer-app”, but expect that in the end it may well turn out to be something else
5. Provide a path from here to there

Scala Adaptation

- Twitter
- Gilt
- Foursquare
- Coursera
- Guardian
- UBS
- Bitgold
- Linkin
- Verizen
- Yammer

Summary

- An integration of OO and FP
  - Also available in Ruby but with dynamic trying
- Static typing
- Concise
- Efficient
- Support for concurrency
- Already adapted
- But requires extensive knowledge

Languages

- Ocaml
- Javascript
- Scala

Concepts &Techniques

- Syntax
  - Context free grammar
  - Ambiguous grammars
  - Syntax vs. semantics
  - Predictive Parsing
- Static semantics
  - Scope rules
- Semantics
  - Small vs. big step
- Runtime management
- Functional programming
  - Lambda calculus
  - Recursion
  - Higher order programming
  - Lazy vs. Eager evaluation
  - Pattern matching
  - Continuation
- Types
  - Type safety
  - Static vs. dynamic
  - Type checking vs. type inference
  - Most general type
  - Polymorphism
  - Type inference algorithm