Extreme Computing

Distributed Data-Parallel Programming

THE UNIVERSITY of EDINBURGH

Amir Shaikhha, Fall 2023

Acknowledgements

The lecture slides draw on notes by several folks to which I am grateful, in particular:

- P. Bhatotia (formerly Univ. of Edinburgh, now TUM)
- M. Odersky (EPFL)
- C. Koch (EPFL)
- H. Miller (CMU)
- M. Zaharia (Berkeley & DataBricks)
- The many researchers whose work I will mention in the slides (I will give pointers to their research papers)

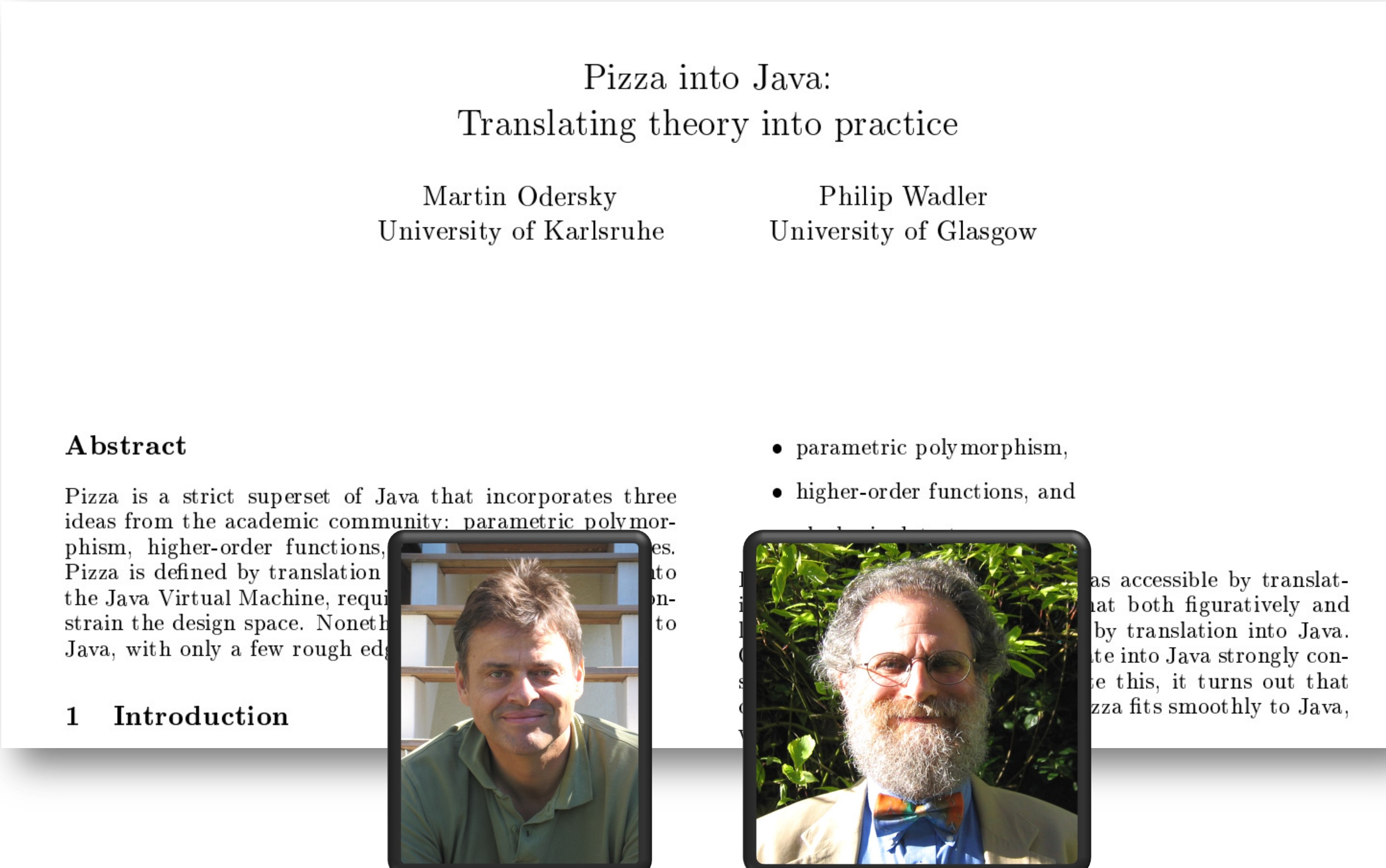
Part 2

Functional Collections in Scala

What is Scala?

- Statically typed
- OO & FP
- Originally running on the JVM
  - Fully interoperable with Java
  - As fast as Java
- JavaScript Backend
  - Interoperable with JavaScript
- LLVM Backend
  - Interoperable with native C code

Make Java Better


Pizza into Java:
Translating theory into practice

Martin Odersky
University of Karlsruhe

Philip Wadler
University of Glasgow

Abstract

Pizza is a strict superset of Java that incorporates three ideas from the academic community: parametric polymor- phism, higher-order functions, es. Pizza is defined by translation to the Java Virtual Machine, requi on- strain the design space. Noneth to Java, with only a few rough ed

1 Introduction

- parametric polymorphism,
- higher-order functions, and

as accessible by translat- at both figuratively and by translation into Java. te into Java strongly con- e this, it turns out that zza fits smoothly to Java,

Make a Better Java

- 2004: First release
- 2007: Adoption begins
- 2008: First Scala conference
- 2021: Scala 3 released

Philosophy

- Scalable Language
- Abstraction and Composition
- Growable Language

Java vs. Scala Example

Java:

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

Scala:

```
class Person(val name: String, val age: Int)
```

Java vs. Scala Example (cont.)

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:

```
val people: Array[Person]
val (minors, adults) =
  people.partition(_.age < 18) // a lambda
```

Basics

- Every value is an object
- Every operation is a method call
- Everything is an expression
  - No statements
  - No need for return and side-effects

Example: Expressions

```
val a: Int = 10 // type can also be inferred
val b = a + 10  // same as a.+(10)

def max(x: Int, y: Int) =
  if (x > y) x else y

val res = max(10, 5)

var x = 0
val t = {
  x = x + 10
  x - 1
}

val u = println("hello, world")
```

Classes & Traits

Scala Classes

- Will behave exactly like a Java class

Scala Traits

- Like Java interfaces
  - In addition allow concrete methods, fields, types
- Like Scala classes
  - Without constructor parameters
- Allow (a form of) multiple inheritence

Example: Complex Numbers

```
class Complex(val re: Int, val im: Int) {

  def +(that: Complex) =
    new Complex(this.re + that.re, this.im + that.im)

  // ...

  override def toString =
    "%d + %di".format(re, im)
}

val c1 = new Complex(1, 2)
val c2 = new Complex(2, 2)
c1 + c2
```

Example: Trait

```
trait Ordered[A] extends java.lang.Comparable[A] {
  def <  (that: A): Boolean = (this compareTo that) <  0
  def >  (that: A): Boolean = (this compareTo that) >  0
  def <= (that: A): Boolean = (this compareTo that) <= 0
  def >= (that: A): Boolean = (this compareTo that) >= 0
}

case class Person(val name: String, val age: Int)
     extends Ordered[Person] {

  def compareTo(that: Person): Int =
    if (name < that.name) -1
    else if (name > that.name) 1
    else age - that.age
}

val p1 = new Person("anton", 10)
val p2 = new Person("berta", 5)
val p3 = new Person("anton", 9)
val ps = List(p1, p2, p3)
ps.sorted
```

Functional Programming

- Use of functions
  - The mathematical sense
  - Referential transparency (no side effects)
- Immutable objects
- Functions are values

FP in Scala

- Immutable variables instead of mutable variables
  - Use **val** instead of **var**
- Immutable collections in the standard library
- Function literals
- Higher-order functions
  - Functions that take or return functions
  - Almost eliminate the need for loops over collections

FP in Scala (cont.)

- Function literals

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

- Equivalent forms

```
(x: Int) => x + 1
x => x + 1            // infer type
_ + 1                 // placeholder notation
```

- Higher-order functions

```
val xs = List(1, 2, 3, 4, 5)
xs.foreach(println)
xs.forall(_ < 10)
xs.map(_ * 2)
```

Everything is an object

- Functions are objects, too
- Instances of trait `Function1[A, B]`
  - Generated by the compiler

```
trait Function1[R, A] {
  def apply(x: A): R
}
```

Syntactic Sugar

- Why does this one work?

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

- `fun(args)` is desugared to `fun.apply(args)`
- You can define your own `apply` methods
- You can extend `FunctionN`

Scala Collections

- Generic
  - `List[T]`
  - `Seq[T]`
  - `Map[K, V]`
- Mutable and immutable implementations
  - Default is immutable

Example: Maps

```
val capitals = Map("France" -> "Paris",
                   "Switzerland" -> "Bern",
                   "Sweden" -> "Stockholm")

val someCity = capitals("France")

val resOfAdd = capitals + ("Romania" ->
"Bucharest")

val filtered = capitals.filter(_._2 == "Paris")
```

Function Subtypes

- Many collections are functions
  - `Seq[T]` is `Int => T`
  - `Set[T]` is `T => Boolean`
  - `Map[K,V]` is `K => V`

```
val even = Set(2, 4, 6, 8, 10)
val res1 = even(4)
val res2 = even(3)
```

For comprehensions

- More general than for-loops
- Syntactic sugar for
  - `flatMap`
  - `filter`
  - `map`

```
for (p <- persons; pr <- p.projects;
  if pr.overdue) yield p.name
```

Pattern Matching

- A powerful switch statement
  - Expression, really
- A way to match and deconstruct structured data

```
// Define a set of case classes for representing binary trees.
sealed abstract class Tree
case class Node(elem: Int, left: Tree, right: Tree) extends Tree
case object Leaf extends Tree

// Return the in-order traversal sequence of a given tree.
def inOrder(t: Tree): List[Int] = t match {
  case Node(e, l, r) => inOrder(l) ::: List(e) ::: inOrder(r)
  case Leaf          => List()
}
```

What to use for this course

- Version
  - Scala 2.12
- Testing
  - ScalaTest
- Build tool
  - sbt

Testing with ScalaTest

```
import collection.mutable.Stack
import org.scalatest._

class ExampleSpec extends FlatSpec with Matchers {

  "A Stack" should "pop values in last-in-first-out order" in {
    val stack = new Stack[Int]
    stack.push(1)
    stack.push(2)
    stack.pop() should be (2)
    stack.pop() should be (1)
  }

  it should "throw NoSuchElementException if an empty stack is popped" in {
    val emptyStack = new Stack[Int]
    a [NoSuchElementException] should be thrownBy {
      emptyStack.pop()
    }
  }
}
```

QUESTIONS?

DEMO TIME ☺