Lightning-Fast Standard Collections With ScalaBlitz

Dmitry Petrashko (@dark_dimius)

École Polytechnique Fédérale de Lausanne

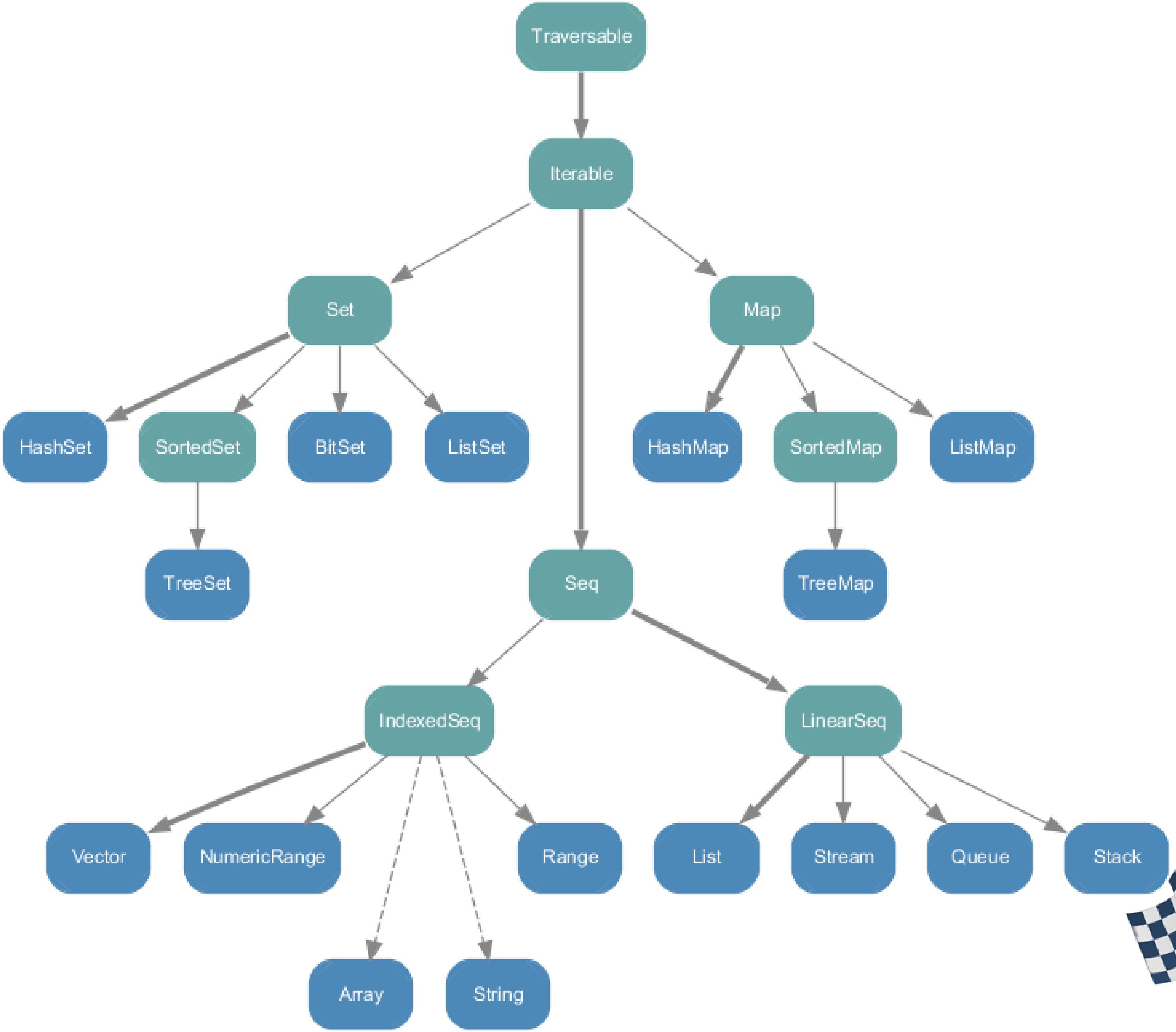
17 June 2014

Outline

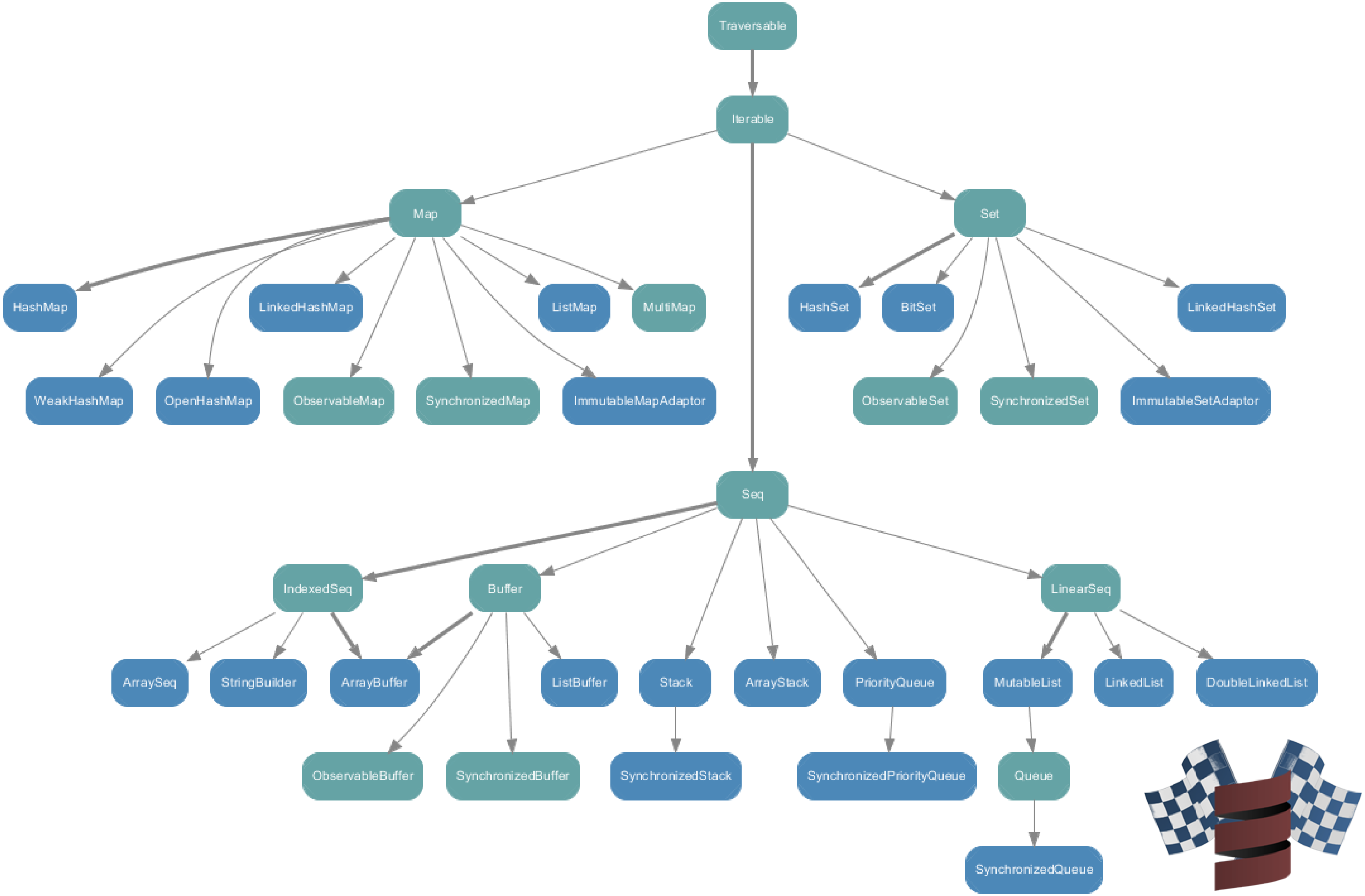
- Example: Scala collections vs Java collections
- What stops Scala collections from being fast on an example
- Observations:
  - Macro-based operations: huge bytecode?
  - Interop with specialization
- How to use optimize{}
- Supported collections & speedups
- Future work

Scala Collections:

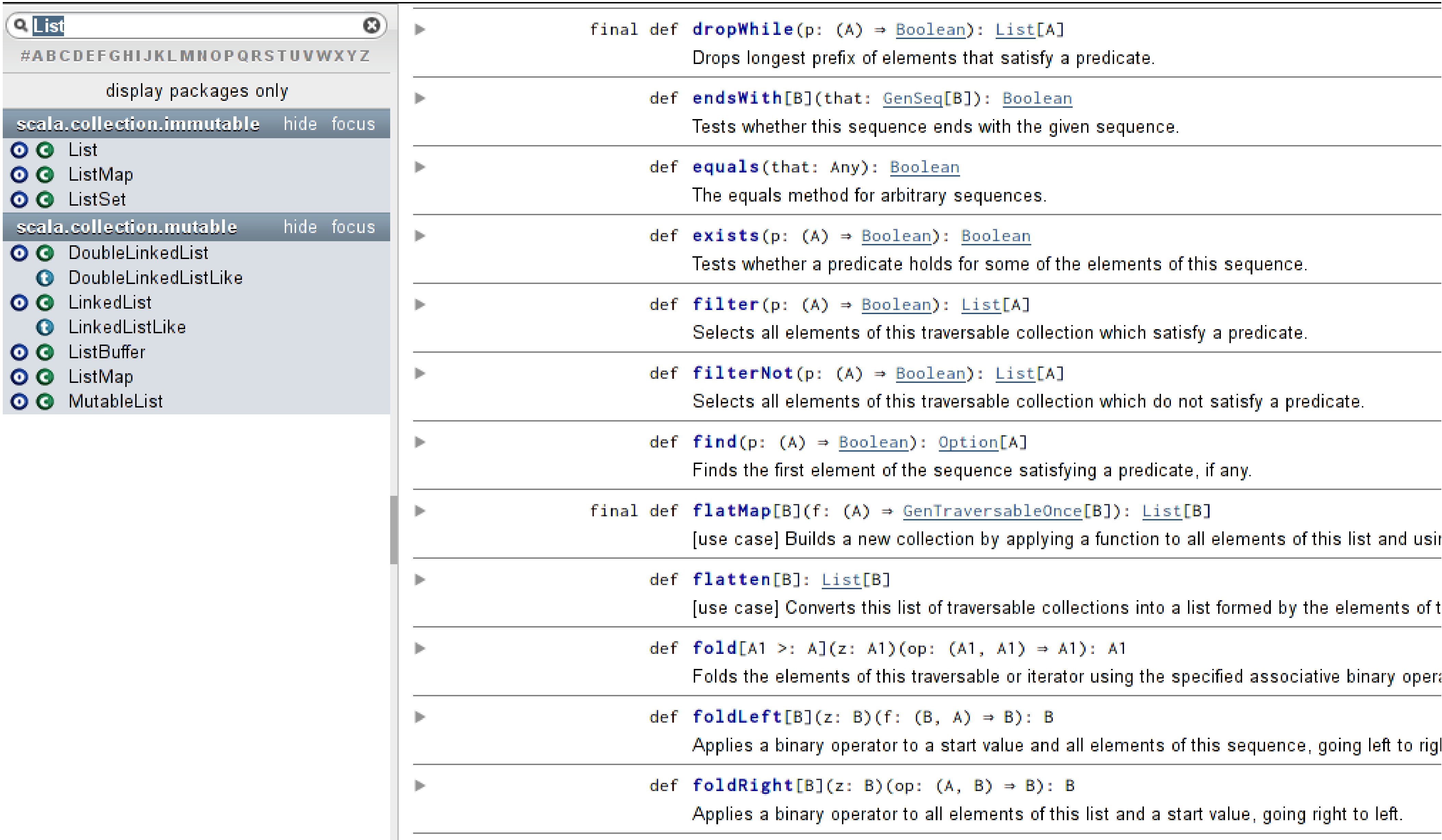
Scala Collections: Variety of flavors



Scala Collections: Variety of flavors



Scala Collections: API



```
▶ final def dropWhile(p: (A) ⇒ Boolean): List[A]
            Drops longest prefix of elements that satisfy a predicate.

▶       def endsWith[B](that: GenSeq[B]): Boolean
            Tests whether this sequence ends with the given sequence.

▶       def equals(that: Any): Boolean
            The equals method for arbitrary sequences.

▶       def exists(p: (A) ⇒ Boolean): Boolean
            Tests whether a predicate holds for some of the elements of this sequence.

▶       def filter(p: (A) ⇒ Boolean): List[A]
            Selects all elements of this traversable collection which satisfy a predicate.

▶       def filterNot(p: (A) ⇒ Boolean): List[A]
            Selects all elements of this traversable collection which do not satisfy a predicate.

▶       def find(p: (A) ⇒ Boolean): Option[A]
            Finds the first element of the sequence satisfying a predicate, if any.

▶ final def flatMap[B](f: (A) ⇒ GenTraversableOnce[B]): List[B]
            [use case] Builds a new collection by applying a function to all elements of this list and usi

▶       def flatten[B]: List[B]
            [use case] Converts this list of traversable collections into a list formed by the elements of t

▶       def fold[A1 >: A](z: A1)(op: (A1, A1) ⇒ A1): A1
            Folds the elements of this traversable or iterator using the specified associative binary opera

▶       def foldLeft[B](z: B)(f: (B, A) ⇒ B): B
            Applies a binary operator to a start value and all elements of this sequence, going left to rig

▶       def foldRight[B](z: B)(op: (A, B) ⇒ B): B
            Applies a binary operator to all elements of this list and a start value, going right to left.
```

Scala Collections: Performance

Java	Scala

public double average(int[] data) {
  int sum = 0;
  for(int i = 0; i < data.length; i++) {
    sum += data[i];
  }
  return sum * 1.0d / data.length
}
	def average(x: Array[Int]) =
	 x.reduce(_ + _) * 1.0 /x.size

Scala Collections: Performance

Java	Scala

public double average(int[] data) {
  int sum = 0;
  for(int i = 0; i < data.length; i++) {
    sum += data[i];
  }
  return sum * 1.0d / data.length
}
	def average(x: Array[Int]) =
	 x.reduce(_ + _) * 1.0 /x.size

20 msec

650 msec

But why?

But why?

Java

```
public double average(int[] data) {
  int sum = 0;
  for(int i = 0; i < data.length; i++) {
    sum += data[i];
  }
  return sum * 1.0d / data.length
}
```

Cycle body:

- Range check
- addition
- increment

But why?

Java

```java
public double average(int[] data) {
  int sum = 0;
  for(int i = 0; i < data.length; i++) {
    sum += data[i];
  }
  return sum * 1.0d / data.length
}
```

Scala

```scala
def average(x: Array[Int]) = {
 x.reduce(_ + _) * 1.0 /x.size
}
```

20 msec

650 msec

But why?

Scala

```
def average(x: Array[Int]) =
 x.reduce(_ + _) * 1.0 /x.size
```

99% of time is spent in `reduce`

```
def reduce
(op: Function2[Obj, Obj, Obj]): Obj = {
    var first = true
    var acc: B = 0.asInstanceOf[B]

    this.foreach{ e =>
      if (first) {
        acc = e
        first = false
      }
      else acc = op.apply(acc, e)
    }
    acc
  }
```

But why?

```
def reduce
(op: Function2[Obj, Obj, Obj]): Obj = {
    var first = true
    var acc: B = 0.asInstanceOf[B]

    this.foreach{ e =>
      if (first) {
        acc = e
        first = false
      }
      else acc = op.apply(acc, e)
    }
    acc
  }
```

```
def foreach(f: Funtion1[Obj, Obj]) {
    var i = 0
    val len = length
    while (i < len) {
        f.apply(this(i));
        i += 1
    }
  }
```

Scala cycle body:

But why?

```
def reduce
(op: Function2[Obj, Obj, Obj]): Obj = {
    var first = true
    var acc: B = 0.asInstanceOf[B]

    this.foreach{ e =>
      if (first) {
        acc = e
        first = false
      }
      else acc = op.apply(acc, e)
    }
    acc
  }
```

```
def foreach(f: Funtion1[Obj, Obj]) {
    var i = 0
    val len = length
    while (i < len) {
        f.apply(this(i));
        i += 1
    }
  }
```

Scala cycle body:

- range check

But why?

```
def reduce
(op: Function2[Obj, Obj, Obj]): Obj = {
    var first = true
    var acc: B = 0.asInstanceOf[B]

    this.foreach{ e =>
      if (first) {
        acc = e
        first = false
      }
      else acc = op.apply(acc, e)
    }
    acc
  }
```

```
def foreach(f: Funtion1[Obj, Obj]) {
    var i = 0
    val len = length
    while (i < len) {
        f.apply(this(i));
        i += 1
    }
  }
```

Scala cycle body:

- range check
- boxing of element

But why?

```
def reduce
(op: Function2[Obj, Obj, Obj]): Obj = {
    var first = true
    var acc: B = 0.asInstanceOf[B]

    this.foreach{ e =>
      if (first) {
        acc = e
        first = false
      }
      else acc = op.apply(acc, e)
    }
    acc
  }
```

```
def foreach(f: Funtion1[Obj, Obj]) {
    var i = 0
    val len = length
    while (i < len) {
        f.apply(this(i));
        i += 1
    }
  }
```

Scala cycle body:

- range check
- boxing of element
- dynamic dispatch(foreach arg)

But why?

```
def reduce
(op: Function2[Obj, Obj, Obj]): Obj = {
    var first = true
    var acc: B = 0.asInstanceOf[B]

    this.foreach{ e =>
      if (first) {
        acc = e
        first = false
      }
      else acc = op.apply(acc, e)
    }
    acc
  }
```

```
def foreach(f: Funtion1[Obj, Obj]) {
    var i = 0
    val len = length
    while (i < len) {
        f.apply(this(i));
        i += 1
    }
  }
```

Scala cycle body:

- range check
- boxing of element
- dynamic dispatch(foreach arg)
- predicate check(first?)

But why?

```
def reduce
(op: Function2[Obj, Obj, Obj]): Obj = {
    var first = true
    var acc: B = 0.asInstanceOf[B]

    this.foreach{ e =>
      if (first) {
        acc = e
        first = false
      }
      else acc = op.apply(acc, e)
    }
    acc
  }
```

```
def foreach(f: Funtion1[Obj, Obj]) {
    var i = 0
    val len = length
    while (i < len) {
        f.apply(this(i));
        i += 1
    }
  }
```

Scala cycle body:

- range check
- boxing of element
- dynamic dispatch(foreach arg)
- predicate check(first?)
- dynamic dispatch(reduce arg)

But why?

```
def reduce
(op: Function2[Obj, Obj, Obj]): Obj = {
    var first = true
    var acc: B = 0.asInstanceOf[B]

    this.foreach{ e =>
      if (first) {
        acc = e
        first = false
      }
      else acc = op.apply(acc, e)
    }
    acc
  }
```

```
def foreach(f: Funtion1[Obj, Obj]) {
    var i = 0
    val len = length
    while (i < len) {
        f.apply(this(i));
        i += 1
    }
  }
```

Scala cycle body:

- range check
- boxing of element
- dynamic dispatch(foreach arg)
- predicate check(first?)
- dynamic dispatch(reduce arg)
- addition

But why?

```
def reduce
(op: Function2[Obj, Obj, Obj]): Obj = {
    var first = true
    var acc: B = 0.asInstanceOf[B]

    this.foreach{ e =>
      if (first) {
        acc = e
        first = false
      }
      else acc = op.apply(acc, e)
    }
    acc
  }
```

```
def foreach(f: Funtion1[Obj, Obj]) {
    var i = 0
    val len = length
    while (i < len) {
        f.apply(this(i));
        i += 1
    }
  }
```

Scala cycle body:

- range check
- boxing of element
- dynamic dispatch(foreach arg)
- predicate check(first?)
- dynamic dispatch(reduce arg)
- addition
- boxing of result

But why?

```
def reduce
(op: Function2[Obj, Obj, Obj]): Obj = {
    var first = true
    var acc: B = 0.asInstanceOf[B]

    this.foreach{ e =>
      if (first) {
        acc = e
        first = false
      }
      else acc = op.apply(acc, e)
    }
    acc
  }
```

```
def foreach(f: Funtion1[Obj, Obj]) {
    var i = 0
    val len = length
    while (i < len) {
        f.apply(this(i));
        i += 1
    }
  }
```

Scala cycle body:

- range check
- boxing of element
- dynamic dispatch(foreach arg)
- predicate check(first?)
- dynamic dispatch(reduce arg)
- addition
- boxing of result
- increment

But why?

Java cycle body:

- range check
- addition
- increment

Scala cycle body:

- range check
- boxing of element
- dynamic dispatch(foreach arg)
- predicate check(first?)
- dynamic dispatch(reduce arg)
- addition
- boxing of result
- Increment

Have ~same cost:

- single boxing(allocation)
- 4 dynamic dispatches
- 15 additions

Scala Collections: Performance

Can we fix it?

Java	Scala

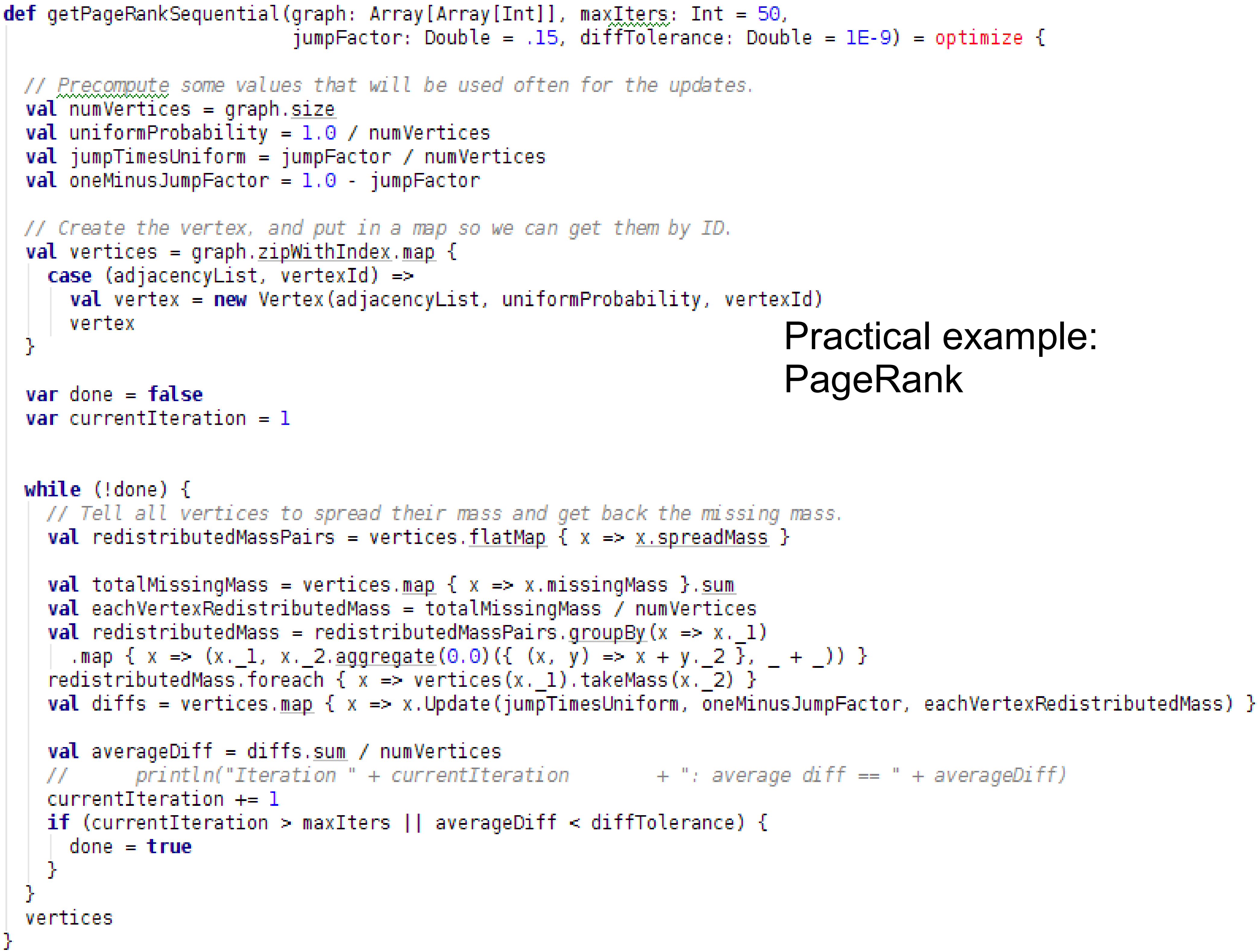
public double average(int[] data) {
  int sum = 0;
  for(int i = 0; i < data.length; i++) {
    sum += data[i];
  }
  return sum * 1.0d / data.length
}
	import scala.collection.optimizer._
	
	def average(x: Array[Int]) = optimize{
	 x.reduce(_ + _) * 1.0 /x.size
	}

20 msec

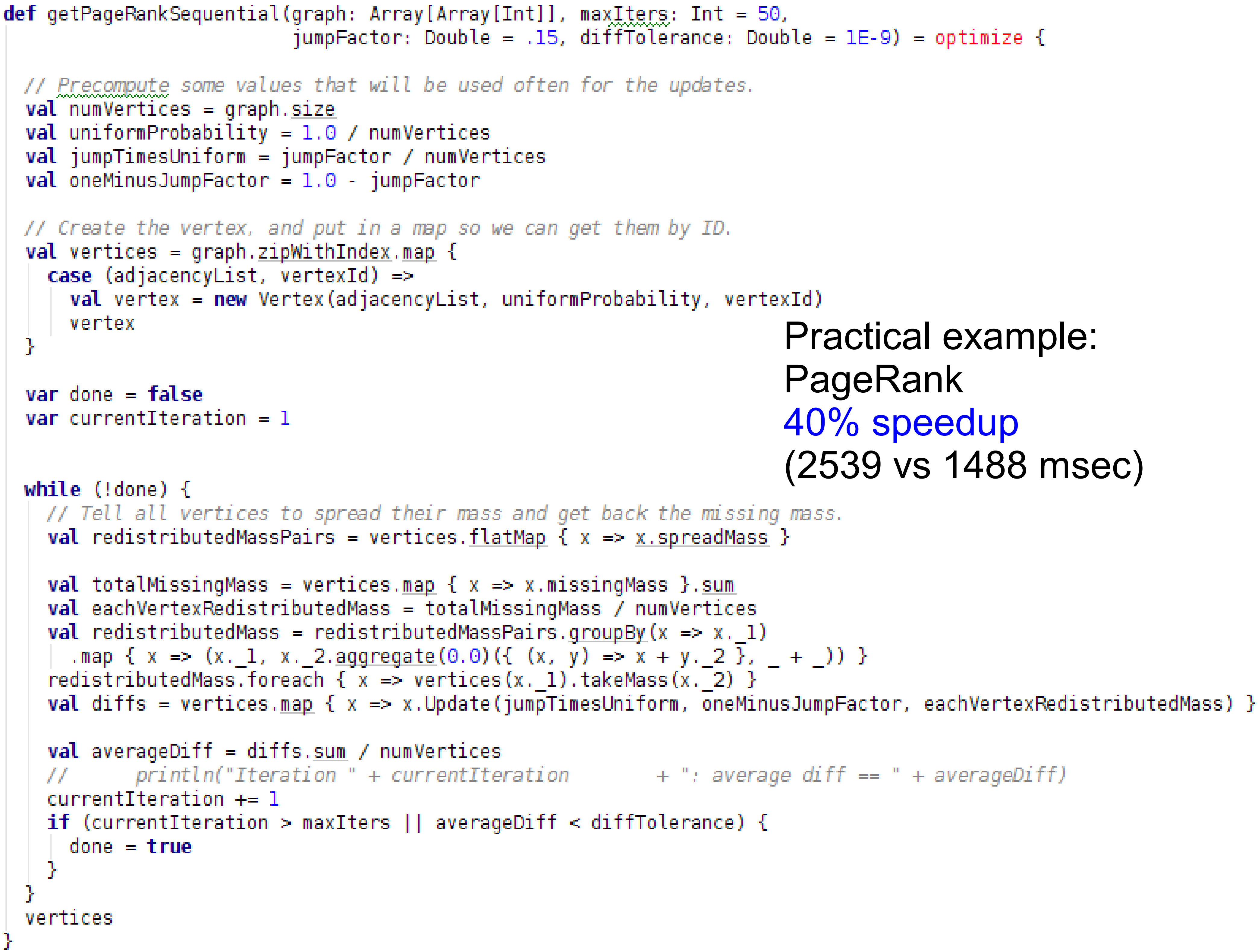
~~650 msec~~ 20 msec.

Is it that bad?

Practical example: PageRank



```
def getPageRankSequential(graph: Array[Array[Int]], maxIters: Int = 50,
                          jumpFactor: Double = .15, diffTolerance: Double = 1E-9) = optimize {

  // Precompute some values that will be used often for the updates.
  val numVertices = graph.size
  val uniformProbability = 1.0 / numVertices
  val jumpTimesUniform = jumpFactor / numVertices
  val oneMinusJumpFactor = 1.0 - jumpFactor

  // Create the vertex, and put in a map so we can get them by ID.
  val vertices = graph.zipWithIndex.map {
    case (adjacencyList, vertexId) =>
      val vertex = new Vertex(adjacencyList, uniformProbability, vertexId)
      vertex
  }

  var done = false
  var currentIteration = 1


  while (!done) {
    // Tell all vertices to spread their mass and get back the missing mass.
    val redistributedMassPairs = vertices.flatMap { x => x.spreadMass }

    val totalMissingMass = vertices.map { x => x.missingMass }.sum
    val eachVertexRedistributedMass = totalMissingMass / numVertices
    val redistributedMass = redistributedMassPairs.groupBy(x => x._1)
      .map { x => (x._1, x._2.aggregate(0.0)({ (x, y) => x + y._2 }, _ + _)) }
    redistributedMass.foreach { x => vertices(x._1).takeMass(x._2) }
    val diffs = vertices.map { x => x.Update(jumpTimesUniform, oneMinusJumpFactor, eachVertexRedistributedMass) }

    val averageDiff = diffs.sum / numVertices
    //      println("Iteration " + currentIteration        + ": average diff == " + averageDiff)
    currentIteration += 1
    if (currentIteration > maxIters || averageDiff < diffTolerance) {
      done = true
    }
  }
  vertices
}
```



```
def getPageRankSequential(graph: Array[Array[Int]], maxIters: Int = 50,
                          jumpFactor: Double = .15, diffTolerance: Double = 1E-9) = optimize {

  // Precompute some values that will be used often for the updates.
  val numVertices = graph.size
  val uniformProbability = 1.0 / numVertices
  val jumpTimesUniform = jumpFactor / numVertices
  val oneMinusJumpFactor = 1.0 - jumpFactor

  // Create the vertex, and put in a map so we can get them by ID.
  val vertices = graph.zipWithIndex.map {
    case (adjacencyList, vertexId) =>
      val vertex = new Vertex(adjacencyList, uniformProbability, vertexId)
      vertex
  }

  var done = false
  var currentIteration = 1


  while (!done) {
    // Tell all vertices to spread their mass and get back the missing mass.
    val redistributedMassPairs = vertices.flatMap { x => x.spreadMass }

    val totalMissingMass = vertices.map { x => x.missingMass }.sum
    val eachVertexRedistributedMass = totalMissingMass / numVertices
    val redistributedMass = redistributedMassPairs.groupBy(x => x._1)
      .map { x => (x._1, x._2.aggregate(0.0)({ (x, y) => x + y._2 }, _ + _)) }
    redistributedMass.foreach { x => vertices(x._1).takeMass(x._2) }
    val diffs = vertices.map { x => x.Update(jumpTimesUniform, oneMinusJumpFactor, eachVertexRedistributedMass) }

    val averageDiff = diffs.sum / numVertices
    //      println("Iteration " + currentIteration        + ": average diff == " + averageDiff)
    currentIteration += 1
    if (currentIteration > maxIters || averageDiff < diffTolerance) {
      done = true
    }
  }
  vertices
}
```

Practical example:
PageRank

40% speedup

(2539 vs 1488 msec)

Operation overhead

Scala cycle body:

- range check
- boxing of element
- dynamic dispatch(foreach arg)
- predicate check(first?)
- dynamic dispatch(reduce arg)
- addition
- boxing of result
- increment

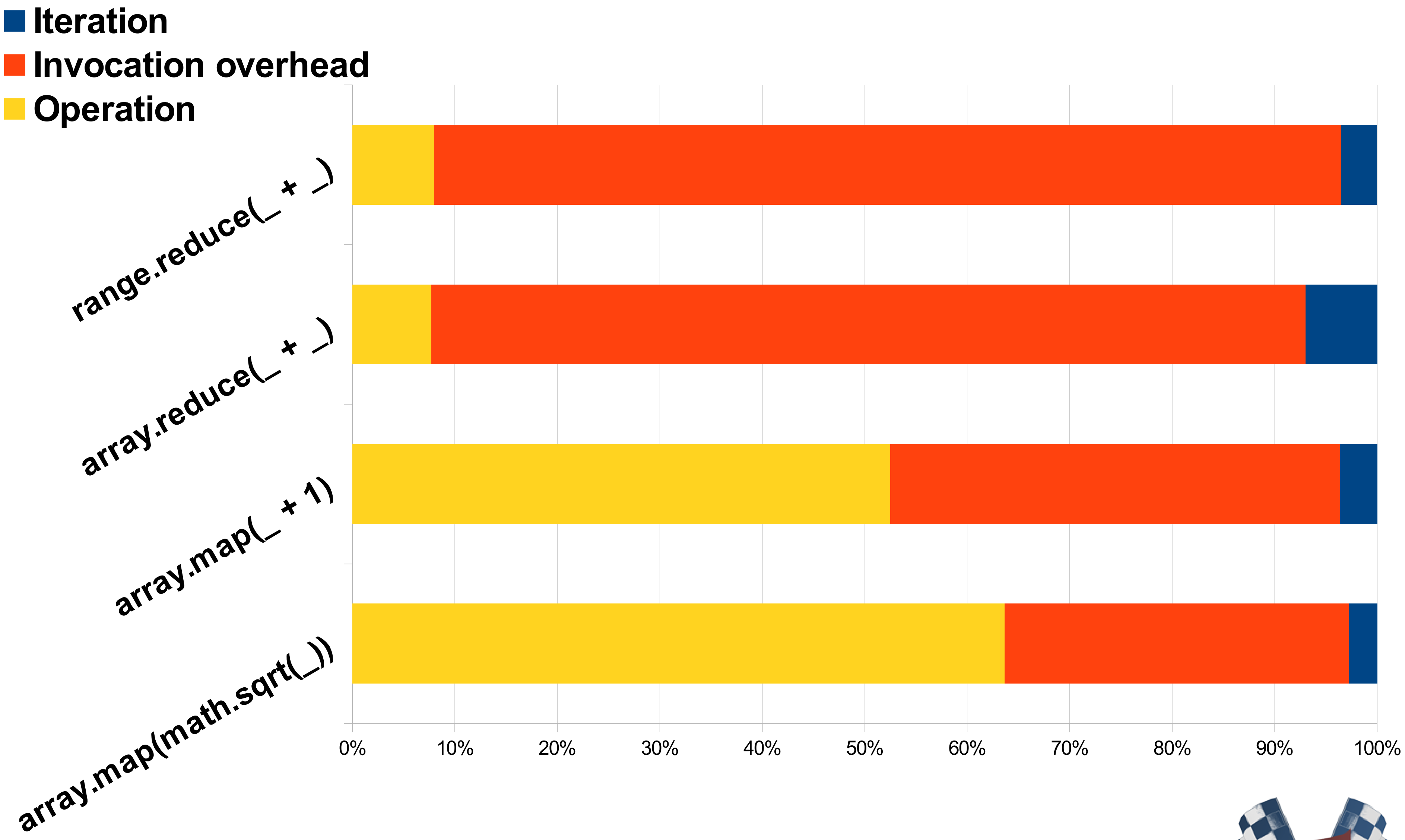
The faster is the operation you perform on elements, the more prone you are to this slowdown

Operation overhead*

	Operations time	Invocation overhead	Iteration time
range.reduce(_ + _)	4.5	50	2
array.reduce(_ + _)	4.5	50	4
array.map(_ + 1)	60	50	4
array.map(math.sqrt(_))	95	50	4

*Those values are very hard to measure and are approximate

Operation overhead



Operation overhead

Scala cycle body:

- range check
- boxing of element
- dynamic dispatch(foreach arg)
- predicate check(first?)
- dynamic dispatch(reduce arg)
- addition
- boxing of result
- increment

The faster is the operation you perform on elements, the more prone you are to this slowdown

ScalaBlitz history

ScalaBlitz 1 year ago:

- GSOC project developed in cooperation and under supervision of Alex Prokopec
- Aimed on shipping better parallel collections for Scala
  - Better API
  - Best performance

ScalaBlitz history

Scala parallel collections API: what's wrong with it?

```
list.par.scanLeft(0)(_ + _).foldRight(0)(_ + _)
```

ScalaBlitz history

Scala parallel collections API: what's wrong with it?

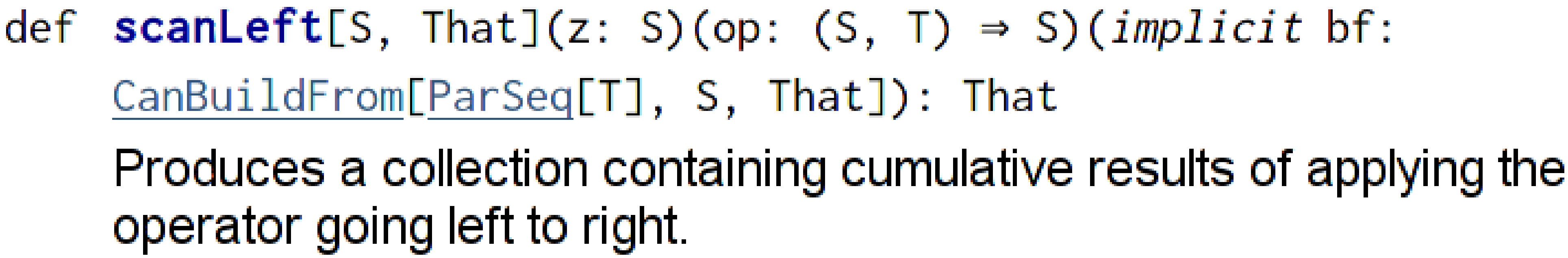
list.par

List(1, 2, 3, 4, 5) => ParVector(1, 2, 3, 4, 5)

ScalaBlitz history

Scala parallel collections API: what's wrong with it?

list.par.scanLeft(0)(_ + _)



```
def  scanLeft[S, That](z: S)(op: (S, T) ⇒ S)(implicit bf:
     CanBuildFrom[ParSeq[T], S, That]): That
```

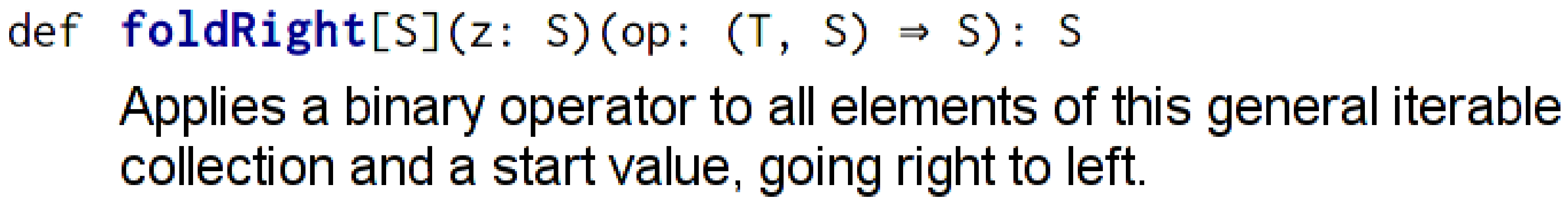
Produces a collection containing cumulative results of applying the operator going left to right.

ParVector(1, 2, 3, 4, 5) => ParVector(0, 1, 3, 6, 10, 15)

ScalaBlitz history

Scala parallel collections API: what's wrong with it?

list.par.scanLeft(0)(_ + _).foldRight(0)(_ + _)



```
def  foldRight[S](z: S)(op: (T, S) ⇒ S): S
```

Applies a binary operator to all elements of this general iterable collection and a start value, going right to left.

ParVector(0, 1, 3, 6, 10, 15) => 0 + 15 + 10 + 6 + 3 + 1 + 0 = 35

ScalaBlitz history

Scala parallel collections performance?
range.reduce(_ + _)

Range	ParRange (4 cores)	ScalaBlitz (1 core)	ScalaBlitz (2 cores)	ScalaBlitz (4 cores)
415 msec	8174 msec	20.4 msec	10.2 msec	5.3 msec

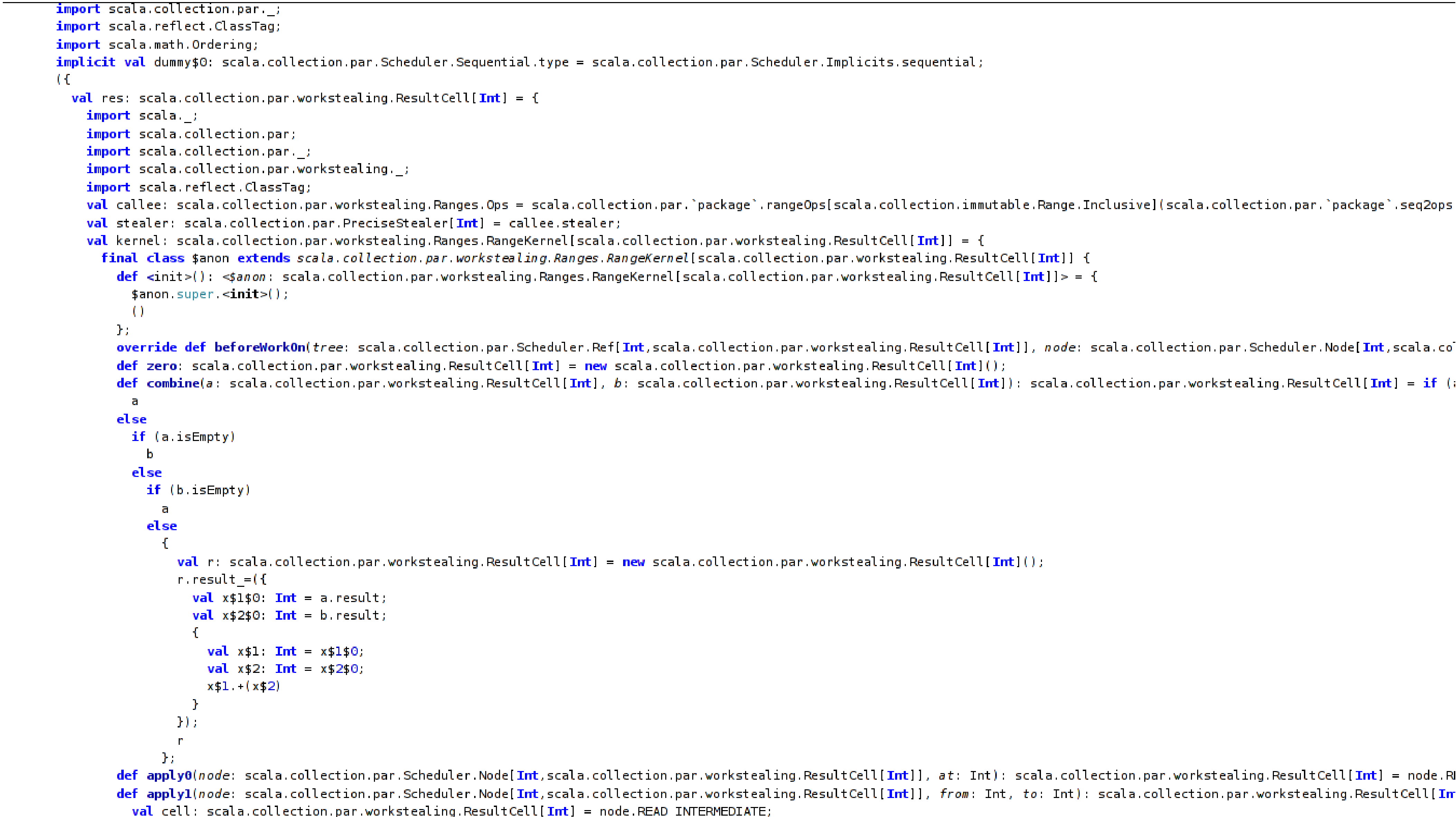
How?

Macro:

- Uses quasiquotes to analyze and generate code

Macro-based operations: huge bytecode?

optimize{ (1 to 10).reduce(_ + _)}



```
import scala.collection.par._;
import scala.reflect.ClassTag;
import scala.math.Ordering;
implicit val dummy$0: scala.collection.par.Scheduler.Sequential.type = scala.collection.par.Scheduler.Implicits.sequential;
({
  val res: scala.collection.par.workstealing.ResultCell[Int] = {
    import scala._;
    import scala.collection.par;
    import scala.collection.par._;
    import scala.collection.par.workstealing._;
    import scala.reflect.ClassTag;
    val callee: scala.collection.par.workstealing.Ranges.Ops = scala.collection.par.`package`.rangeOps[scala.collection.immutable.Range.Inclusive](scala.collection.par.`package`.seq2ops
    val stealer: scala.collection.par.PreciseStealer[Int] = callee.stealer;
    val kernel: scala.collection.par.workstealing.Ranges.RangeKernel[scala.collection.par.workstealing.ResultCell[Int]] = {
      final class $anon extends scala.collection.par.workstealing.Ranges.RangeKernel[scala.collection.par.workstealing.ResultCell[Int]] {
        def <init>(): <$anon: scala.collection.par.workstealing.Ranges.RangeKernel[scala.collection.par.workstealing.ResultCell[Int]]> = {
          $anon.super.<init>();
          ()
        };
        override def beforeWorkOn(tree: scala.collection.par.Scheduler.Ref[Int,scala.collection.par.workstealing.ResultCell[Int]], node: scala.collection.par.Scheduler.Node[Int,scala.co
        def zero: scala.collection.par.workstealing.ResultCell[Int] = new scala.collection.par.workstealing.ResultCell[Int]();
        def combine(a: scala.collection.par.workstealing.ResultCell[Int], b: scala.collection.par.workstealing.ResultCell[Int]): scala.collection.par.workstealing.ResultCell[Int] = if (
          a
        else
          if (a.isEmpty)
            b
          else
            if (b.isEmpty)
              a
            else
              {
                val r: scala.collection.par.workstealing.ResultCell[Int] = new scala.collection.par.workstealing.ResultCell[Int]();
                r.result_=({
                  val x$1$0: Int = a.result;
                  val x$2$0: Int = b.result;
                  {
                    val x$1: Int = x$1$0;
                    val x$2: Int = x$2$0;
                    x$1.+(x$2)
                  }
                });
                r
              };
        def apply0(node: scala.collection.par.Scheduler.Node[Int,scala.collection.par.workstealing.ResultCell[Int]], at: Int): scala.collection.par.workstealing.ResultCell[Int] = node.R
        def apply1(node: scala.collection.par.Scheduler.Node[Int,scala.collection.par.workstealing.ResultCell[Int]], from: Int, to: Int): scala.collection.par.workstealing.ResultCell[In
          val cell: scala.collection.par.workstealing.ResultCell[Int] = node.READ INTERMEDIATE;
```

Observations: bytecode size

In practice size is almost same or even decreased due to inlining of closures.

Original	ScalaBlitz
1964 +1693 = 3657 bytes	2488 bytes

Observations: collection specializations

Some operations cannot be optimized further without specializing the collection: Eg, count, filter, find

Observations: collection specializations

Some operations cannot be optimized further without specializing the collection: Eg, count, filter, find

```
def count(p: Funtion1[Object, Object]): Int = {
    var cnt = 0
    this.foreach{ x =>
      if (p(x)) cnt += 1
    }
    cnt
  }

def foreach[U](f: A => U) {
    var these = this
    while (!these.isEmpty) {
      f(these.head)
      these = these.tail
    }
```

Original cycle body:

Observations: collection specializations

Some operations cannot be optimized further without specializing the collection: Eg, count, filter, find

```
def count(p: Funtion1[Object, Object]): Int = {
    var cnt = 0
    this.foreach{ x =>
      if (p(x)) cnt += 1
    }
    cnt
  }

def foreach[U](f: A => U) {
    var these = this
    while (!these.isEmpty) {
      f(these.head)
      these = these.tail
    }
```

Original cycle body:

- range check

Observations: collection specializations

Some operations cannot be optimized further without specializing the collection: Eg, count, filter, find

```
def count(p: Funtion1[Object, Object]): Int = {
    var cnt = 0
    this.foreach{ x =>
      if (p(x)) cnt += 1
    }
    cnt
  }
```

Original cycle body:

- range check
- dynamic dispatch

```
def foreach[U](f: A => U) {
    var these = this
    while (!these.isEmpty) {
      f(these.head)
      these = these.tail
    }
```

Observations: collection specializations

Some operations cannot be optimized further without specializing the collection: Eg, count, filter, find

```
def count(p: Funtion1[Object, Object]): Int = {
    var cnt = 0
    this.foreach{ x =>
      if (p(x)) cnt += 1
    }
    cnt
  }

def foreach[U](f: A => U) {
    var these = this
    while (!these.isEmpty) {
      f(these.head)
      these = these.tail
    }
```

Original cycle body:

- range check
- dynamic dispatch
- unboxing of element

Observations: collection specializations

Some operations cannot be optimized further without specializing the collection: Eg, count, filter, find

```
def count(p: Funtion1[Object, Object]): Int = {
    var cnt = 0
    this.foreach{ x =>
      if (p(x)) cnt += 1
    }
    cnt
  }

def foreach[U](f: A => U) {
    var these = this
    while (!these.isEmpty) {
      f(these.head)
      these = these.tail
    }
```

Original cycle body:

- range check
- dynamic dispatch
- unboxing of element
- predicate check

Observations: collection specializations

Some operations cannot be optimized further without specializing the collection: Eg, count, filter, find

```
def count(p: Funtion1[Object, Object]): Int = {
    var cnt = 0
    this.foreach{ x =>
      if (p(x)) cnt += 1
    }
    cnt
  }

def foreach[U](f: A => U) {
    var these = this
    while (!these.isEmpty) {
      f(these.head)
      these = these.tail
    }
```

Original cycle body:

- range check
- dynamic dispatch
- unboxing of element
- predicate check
- increment

Observations: collection specializations

Some operations cannot be optimized further without specializing the collection: Eg, count, filter, find

```
def countSB(x: List[Int]) = {
    var head = x
    var count = 0
    while (!head.isEmpty) {
      if (x.head > 0) count += 1
      head = head.tail
    }
    count
}
```

ScalaBlitz cycle body:

Observations: collection specializations

Some operations cannot be optimized further without specializing the collection: Eg, count, filter, find

```
def countSB(x: List[Int]) = {
    var head = x
    var count = 0
    while (!head.isEmpty) {
      if (x.head > 0) count += 1
      head = head.tail
    }
    count
}
```

ScalaBlitz cycle body:

- range check

Observations: collection specializations

Some operations cannot be optimized further without specializing the collection: Eg, count, filter, find

```
def countSB(x: List[Int]) = {
    var head = x
    var count = 0
    while (!head.isEmpty) {
      if (x.head > 0) count += 1
      head = head.tail
    }
    count
}
```

ScalaBlitz cycle body:

- range check
- unboxing of element

Observations: collection specializations

Some operations cannot be optimized further without specializing the collection: Eg, count, filter, find

```
def countSB(x: List[Int]) = {
    var head = x
    var count = 0
    while (!head.isEmpty) {
      if (x.head > 0) count += 1
      head = head.tail
    }
    count
}
```

ScalaBlitz cycle body:

- range check
- unboxing of element
- predicate check

Observations: collection specializations

Some operations cannot be optimized further without specializing the collection: Eg, count, filter, find

```
def countSB(x: List[Int]) = {
    var head = x
    var count = 0
    while (!head.isEmpty) {
      if (x.head > 0) count += 1
      head = head.tail
    }
    count
}
```

ScalaBlitz cycle body:

- range check
- unboxing of element
- predicate check
- count increment

Observations: collection specializations

Some operations cannot be optimized further without specializing the collection: Eg, count, filter, find

Original cycle body:

- range check
- dynamic dispatch
- unboxing of element
- predicate check
- increment

ScalaBlitz cycle body:

- range check
- unboxing of element
- predicate check
- increment

Potential gain of combining with http://scala-miniboxing.org/
see “Miniboxing: Specialization on a Diet” talk by Vlad Ureche tomorrow.

Caveats

Generated code is harder to debug.
Looking forward to “Easy Metaprogramming For Everyone!”
by Eugene Burmako and Denys Shabalin

That isn't a big problem if we maintain same guarantees as Scala Collections

Hard to understand stack-traces and runtime profiles.

A bit slower for tiny collections(several elements)

No custom CanBuildFrom support(yet)

Supported collections& Speedups

	Range	Array	HashMap &HashSet	Immutable Map&Set	List
reduce(_ +_)	44x	33x	5.1x	1.1x	4.3x
sum	38x	29x	1.7x	1.1x	2.8x
product	27x	19x	1.6x	1.1x	1.6x
min & max	both constant	25x	1.7x	same	1.2x
map(_ + 1)	10x	10x	1.3x	1.5x	unsupported
flatmap(x => List(x, x))	1.1x	1.3x	1.3x	1.3x	unsupported
find(_ < 0) &friends	12x	10x	2.4x	same	unsupported
count(_ > 0)	3.8x	3.3x	1.3x	same	unsupported

What does unsupported collection mean?

What does unsupported
collection mean?

Nothing bad,
operation will simply be performed
by Scala collections

Future work: operation fusion

```
def minAvgMax(xs: List[Int]) = {
    val avg = xs.sum * 1.0 / xs.size
    (xs.min, avg, xs.max)
  }
```

Current status: 4 independent operations over collection:

- sum
- size (also linear time!)
- min
- max

Idea:
interleave operations,
use single iteration over collection to perform all 4.

Future work: deforestation

```
val minMaleAge = people.filter(_.isMale).map(_.age).min
```

Current status: 2 intermediate collections

- filter
- map

Idea:
use stream-like pipelining

Thanks for your attention!

Questions?

dark@d-d.me