Specialization in Scala 2.8

or Making your program special

Iulian Dragoș
Scala Days 2010

Parametric Polymorphism

```
class Vector[A](size: Int) {
  var arr: Array[A] = new Array[A](size)

  def apply(i: Int): A = arr(i)
  def update(i: Int, e: A) {
    arr(i) = e
  }
  def resize(newSize: Int) { .. }
}
```

Parametric Polymorphism

```
class Vector[A](size: Int) {
  var arr: Array[A] = new Array[A](size)

  def apply(i: Int): A = arr(i)
  def update(i: Int, e: A) {
    arr(i) = e
  }
  def resize(newSize: Int) { .. }
}
```

```
class Vector(size: Int) {
  val arr: Array[Object] =
    new Array[Object](size)
  def apply(i: Int): Object = arr(i)
  def update(i: Int, e: Object) {
    arr(i) = e
  }
  def resize(newSize: Int) { .. }
}
```

Parametric Polymorphism

```
class Vector[A](size: Int) {
  var arr: Array[A] = new Array[A](size)

  def apply(i: Int): A = arr(i)
  def update(i: Int, e: A) {
    arr(i) = e
  }
  def resize(newSize: Int) { .. }
}
```

```
class Vector(size: Int) {
  val arr: Array[Object] =
    new Array[Object](size)
  def apply(i: Int): Object = arr(i)
  def update(i: Int, e: Object) {
    arr(i) = e
  }
  def resize(newSize: Int) { .. }
}
```

Parametric Polymorphism

```
class Vector[A](size: Int) {
  var arr: Array[A] = new Array[A](size)

  def apply(i: Int): A = arr(i)
  def update(i: Int, e: A) {
    arr(i) = e
  }
  def resize(newSize: Int) { .. }
}
```

```
class Vector(size: Int) {
  val arr: Array[Object] =
    new Array[Object](size)
  def apply(i: Int): Object = arr(i)
  def update(i: Int, e: Object) {
    arr(i) = e
  }
  def resize(newSize: Int) { .. }
}
```

```
class StringVector(size: Int) {
class DoubleVector(size: Int) {
class IntVector(size: Int) {
  val arr: Array[Int] =
    new Array[Int](size)
  def apply(i: Int): Int = arr(i)
  def update(i: Int, e: Int) {
    arr(i) = e
  }
  def resize(newSize: Int) { .. }
}
```

Example

```
val v = new Vector[Int]
v(0) = v(0) + 1
```

```
val v = new Vector
v.update(0,
  Int.box(Int.unbox(v.apply(0)) + 1))
```

```
val v = new IntVector
v.update(0,
  v.apply(0) + 1)
```

Parametric Polymorphism

- Generics
  - compact code
  - single representation forces boxing/unboxing of primitive types
- Specialization
  - good performance
  - risk of code explosion
  - separate compilation is tricky

Idea

- separate compilation is a must
  - specializations are derived at definition site *for primitive types*
- reasonable code size
  - specialized and type-erased code coexist
- calls are rewritten to specialized versions opportunistically

On-demand specialization

User annotates type parameters, optionally indicating what types to specialize on.

```
trait Function1[@specialized(Int, Float, Double) -T1,
                @specialized +R] {
  def apply(x: T1): R
}
```

Example

```
class Matrix[@specialized A](val rows: Int, val cols: Int) {
  val arr: Array[Array[A]] = new Array[Array[A]](rows, cols)

  def apply(i: Int, j: Int): A = {
    if (i < 0 || i >= rows
     || j < 0 || j >= cols)
      throw new IndexOutOfBoundsException

    arr(i)(j)
  }

  def update(i: Int, j: Int, e: A) {
    arr(i)(j) = e
  }
}
```

Example

```
def mult(m: Matrix[Int], n: Matrix[Int]) {
  val p = new Matrix[Int](m.rows, n.cols)

  for (i <- 0 until m.rows)
    for (j <- 0 until n.cols) {
      var sum = 0
      for (k <- 0 until n.rows)
        sum += m(i, k) * n(k, j)
      p(i, j) = sum
    }
}
```

Example

```
def mult(m: Matrix[Int], n: Matrix[Int]) {
  val p = new IntMatrix(m.rows, n.cols)

  for (i <- 0 until m.rows)
    for (j <- 0 until n.cols) {
      var sum = 0
      for (k <- 0 until n.rows)
        sum += m.applyInt(i, k) * n.applyInt(k, j)
      p.updateInt(i, j, sum)
    }
}
```

Performance

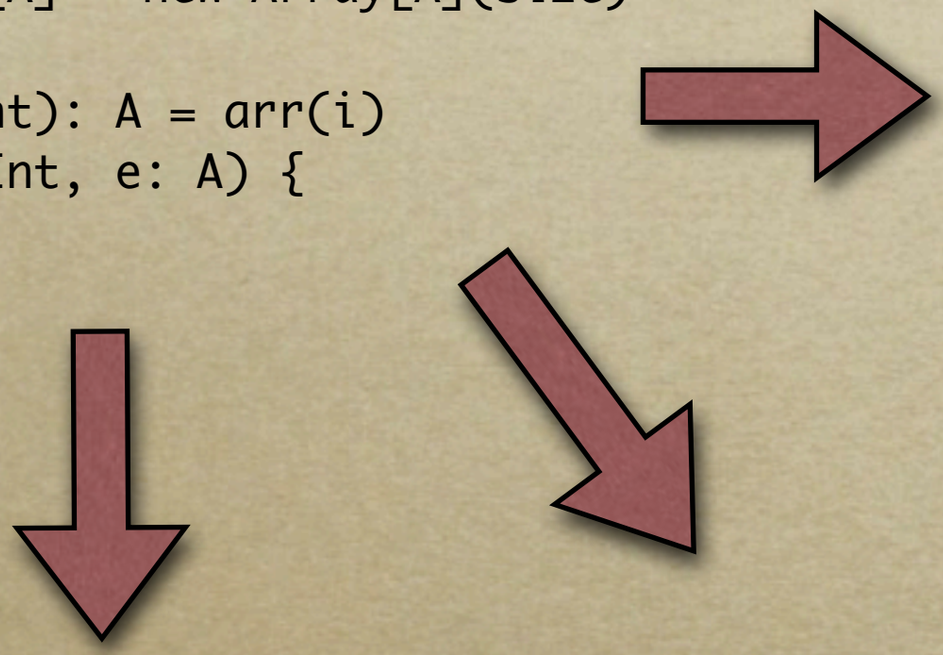
Multiplying two matrices of 200x100 is **2.5 times** faster with specialization

“Conditional compilation”

```
def test[@specialized A](x: A) = x match {
  case i: Int => "int"
  case d: Double => "double"
  case e: Float => "float"
  case _ => "other"
}
```

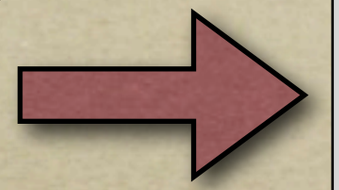
Implementation

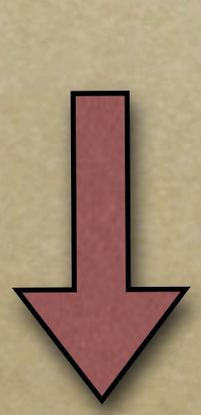
```
class Vector[@specialized A](size: Int) {
  val arr: Array[A] = new Array[A](size)

  def apply(i: Int): A = arr(i)
  def update(i: Int, e: A) {
    arr(i) = e
  }
}
```

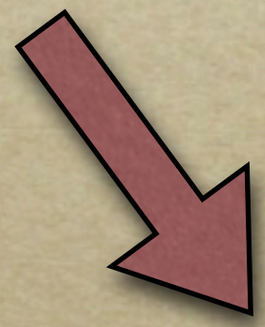


Implementation

```
class Vector[@specialized A](size: Int) {
  val arr: Array[A] = new Array[A](size)

  def apply(i: Int): A = arr(i)
  def update(i: Int, e: A) {
    arr(i) = e
  }
}
```







```
class Vector(size: Int) {
  val arr: Array[Object] =
    new Array[Object](size)
  def apply(i: Int): Object = arr(i)
  def update(i: Int, e: Object) {
    arr(i) = e
  }
  ...
}
```

Implementation

```
class Vector[@specialized A](size: Int) {
  val arr: Array[A] = new Array[A](size)

  def apply(i: Int): A = arr(i)
  def update(i: Int, e: A) {
    arr(i) = e
  }
}
```

```
class Vector(size: Int) {
  val arr: Array[Object] =
    new Array[Object](size)
  def apply(i: Int): Object = arr(i)
  def update(i: Int, e: Object) {
    arr(i) = e
  }
  ...
}
```

```
class IntVector(size: Int)
        extends Vector[Int] {
 ..
}
```

Implementation

```
class Vector[@specialized A](size: Int) {
  val arr: Array[A] = new Array[A](size)

  def apply(i: Int): A = arr(i)
  def update(i: Int, e: A) {
    arr(i) = e
  }
}
```

```
class Vector(size: Int) {
  val arr: Array[Object] =
    new Array[Object](size)
  def apply(i: Int): Object = arr(i)
  def update(i: Int, e: Object) {
    arr(i) = e
  }
  ...
}
```

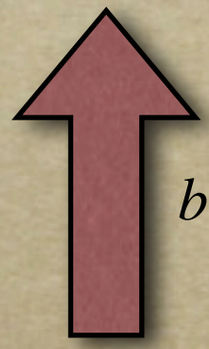
```
class IntVector(size: Int)
        extends Vector[Int] {
 ..
}
```

```
class DoubleVector(size: Int)
        extends Vector[Double] {
 ..
}
```

Specialized variants

Specialized variants are additional methods derived from a method using specialized type parameters.

```
class Vector(size: Int) {
  def update(i: Int, e: Object) { arr(i) = e }
```

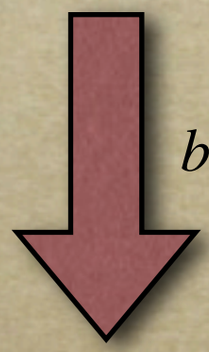



```
  def updateInt(i: Int, e: Int)
  def updateDouble(i: Int, e: Double)
  def updateFloat(i: Int, e: Floate)
  ...
}
```

Specialized variants

Specialized variants are additional methods derived from a method using specialized type parameters.

```
class IntVector(size: Int) extends Vector[Int] {
  def update(i: Int, e: Object)
```

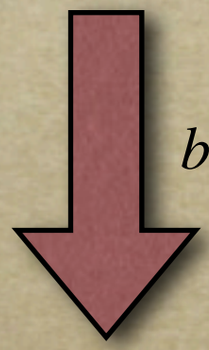



```
  def updateInt(i: Int, e: Int) = arr(i) = e

  ...
}
```

Specialized variants

Specialized variants are additional methods derived from a method using specialized type parameters.

```
class DoubleVector(size: Int) extends Vector[Double] {
  def update(i: Int, e: Object)
```




```
  def updateDouble(i: Int, e: Double) = arr(i) = e

  ...
}
```

Specialized overrides

Specialized overrides ensure variants and the generic version of a method have the same body.

```
class SyncDoubleVector(size: Int) extends Vector[Double] {
  override def update(i: Int, e: Object) = synchronized ..
```

Specialized overrides

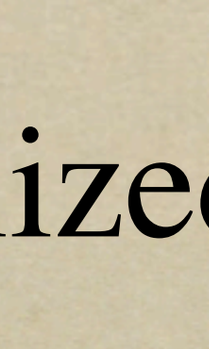
Specialized overrides ensure variants and the generic version of a method have the same body.

```
class SyncDoubleVector(size: Int) extends DoubleVector {
  override def update(i: Int, e: Object) =
```

Specialized overrides

Specialized overrides ensure variants and the generic version of a method have the same body.

```
class SyncDoubleVector(size: Int) extends DoubleVector {
  override def update(i: Int, e: Object) =
```




```
    override def updateDouble(i: Int, e: Double) = synchronized ..

  ...
}
```

Specialization

- Fields
  - only available through accessors
  - specialized subclasses add specialized fields
- Specialized polymorphic methods
  - expanded to definitions with no specialized type parameters

Method Expansion

```
class List[@specialized A] {
  def ::[@specialized B >: A](x: B): List[B] =
    new ::(x, this) ..
}
```

Method Expansion

```
class List[@specialized A] {
  def ::[@specialized B >: A](x: B): List[B] =
    new ::(x, this) ..

  def ::(x: Int): List[Int] = new ::(x, this) // ??
}
```

Method Expansion

```
class List[@specialized A] {
  def ::[@specialized B >: A](x: B): List[B] =
    new ::(x, this) ..

  def ::(x: Int): List[Int] = new ::(x, this) // ??
}
```

- Type bounds may be:
  - conflicting
  - satisfiable
  - valid

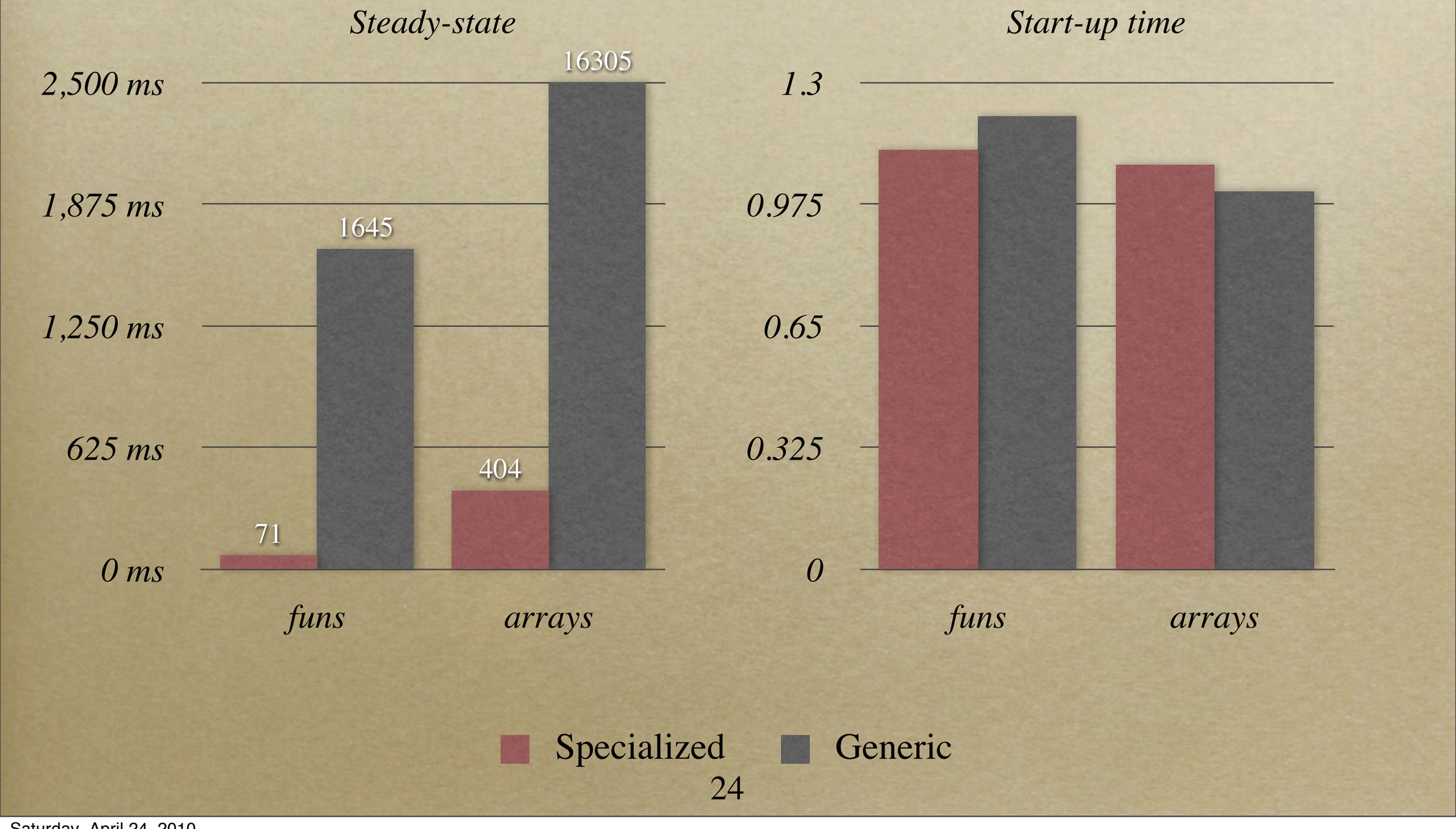
Specializing the standard library

- functions, tuples, products up to two parameters
  - on Int, Long, Double plus Unit and Boolean for result types
- Range.foreach (fast C-like fors)
- to do (not in this Release Candidate)
  - array-backed collections
  - the Numeric hierarchy
  - ...

Speedup

- *two benchmarks*
  - funs*: tests performance of function literals (maps a function on an array of 1000 integers)*
  - arrays*: test performance of array-backed collections (same operation, using a Scala collection, operation is inlined)*

Speedup


Steady-state
2,500 ms
1,875 ms
1,250 ms
625 ms
0 ms
16305
1645
404
71
funs
arrays
Start-up time
1.3
0.975
0.65
0.325
0
funs
arrays
Specialized
Generic

What next

- *more specialization in the library*
- *allow user-defined specialized classes*
  - *for instance, a user-defined bit set implementation of* Set[Boolean]