Miniboxing

Front-end Specialization on the JVM

JVMLS, 29th of July 2014

Vlad Ureche
twitter/github: VladUreche
email: vlad.ureche@epfl.ch

Research warning!

Not the industry-proof drop-in solution that you might expect!

```
def identity[T](x: T): T = x
```

```
def identity[T](x: T): T = x
```

Erased to j.l.Object => boxing

```
def identity[@specialized T](x: T): T = x
```

For reference types (e.g. String)

```
def identity[@specialized T](x: T): T = x
```

For reference types (e.g. String)

```
def identity[@specialized T](x: T): T = x

def identity_V(x: Unit): Unit = x
def identity_Z(x: Boolean): Boolean = x
def identity_B(x: Byte): Byte = x
def identity_C(x: Char): Char = x
def identity_S(x: Short): Short = x
def identity_I(x: Int): Int = x
def identity_L(x: Long): Long = x
def identity_F(x: Float): Float = x
def identity_D(x: Double): Double = x
```

For reference types (e.g. String)

```
def identity[@specialized T](x: T): T = x

def identity_V(x: Unit): Unit = x
def identity_Z(x: Boolean): Boolean = x
def identity_B(x: Byte): Byte = x
def identity_C(x: Char): Char = x
def identity_S(x: Short): Short = x
def identity_I(x: Int): Int = x
def identity_L(x: Long): Long = x
def identity_F(x: Float): Float = x
def identity_D(x: Double): Double = x
```

For primitive types

```
def tupled[@specialized T1,
           @specialized T2]
          (t1: T1, t1: T2): T = x
```

```
def tupled[@specialized T1,
           @specialized T2]
           (t1: T1, t1: T2): T = x
```

10^2 methods

```
def tupled[@specialized T1,
           @specialized T2]
          (t1: T1, t1: T2): T = x
```

10^2 methods

same problem for classes

```
def tupled[@specialized T1,
           @specialized T2]
           (t1: T1, t1: T2): T = x
```

10^2 methods

same problem for classes

upfront bytecode :(

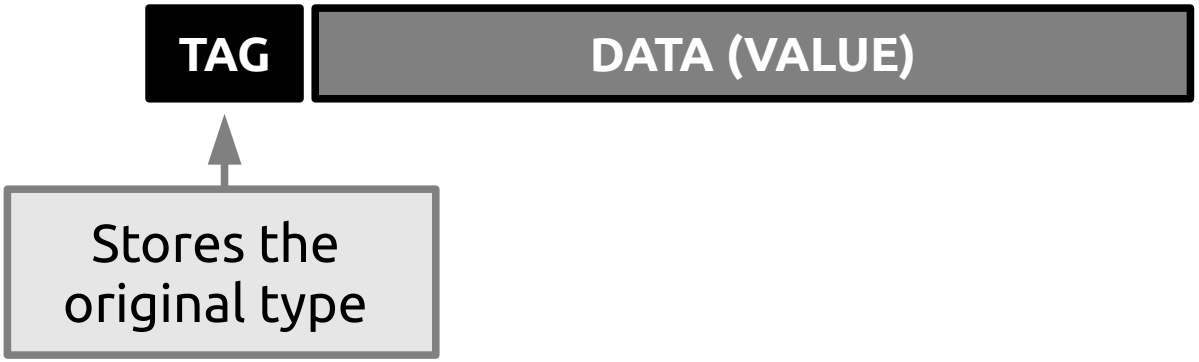
Miniboxing


Miniboxing

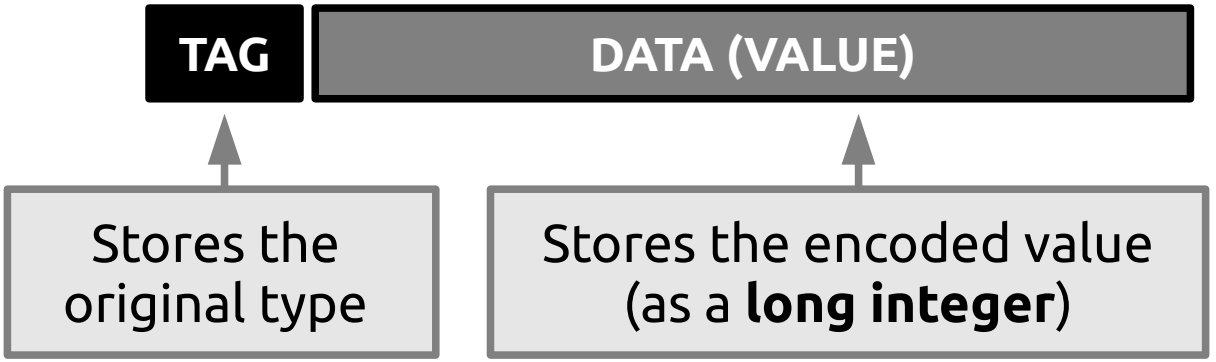
= tagged union

= tagged union

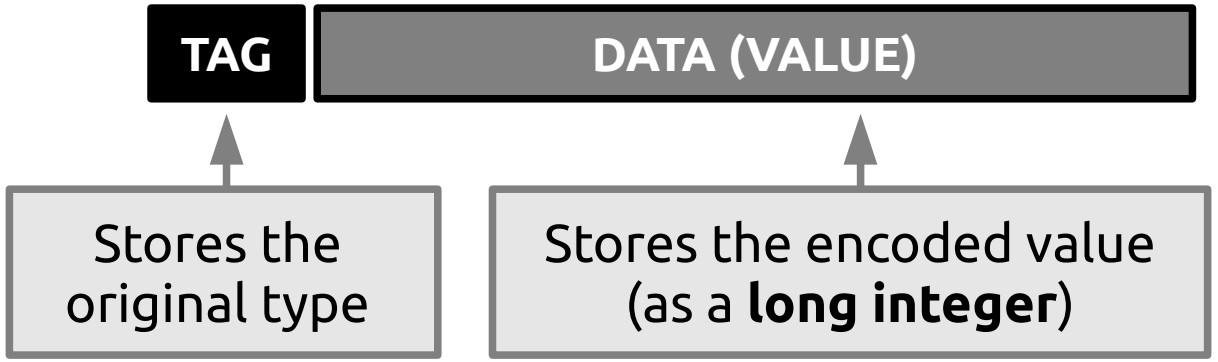
= tagged union



= tagged union

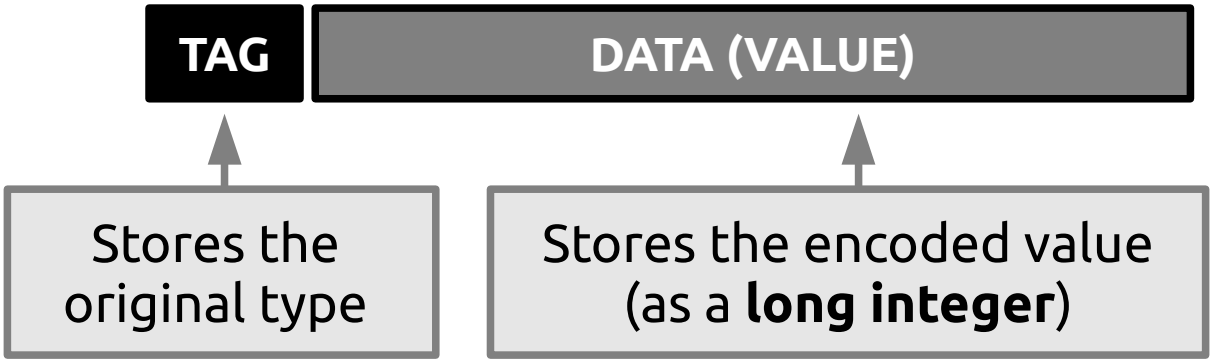


= tagged union



+ tag hoisting (light reified type)

= tagged union



+ tag hoisting (light reified type)

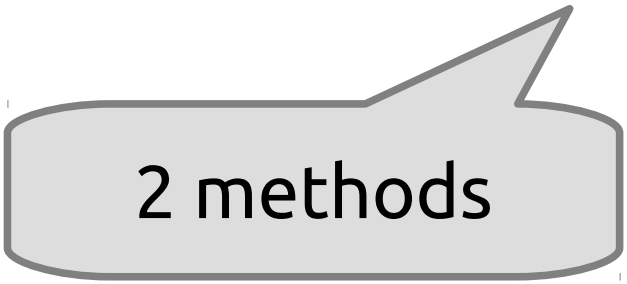
+ runtime specialization*

* poor man's **classdynamic**

```
def identity[@miniboxed T](x: T): T = x
```

```
def identity[@miniboxed T](x: T): T = x
def identity[T](T_Tag: Byte, x: Long): Long = x
```

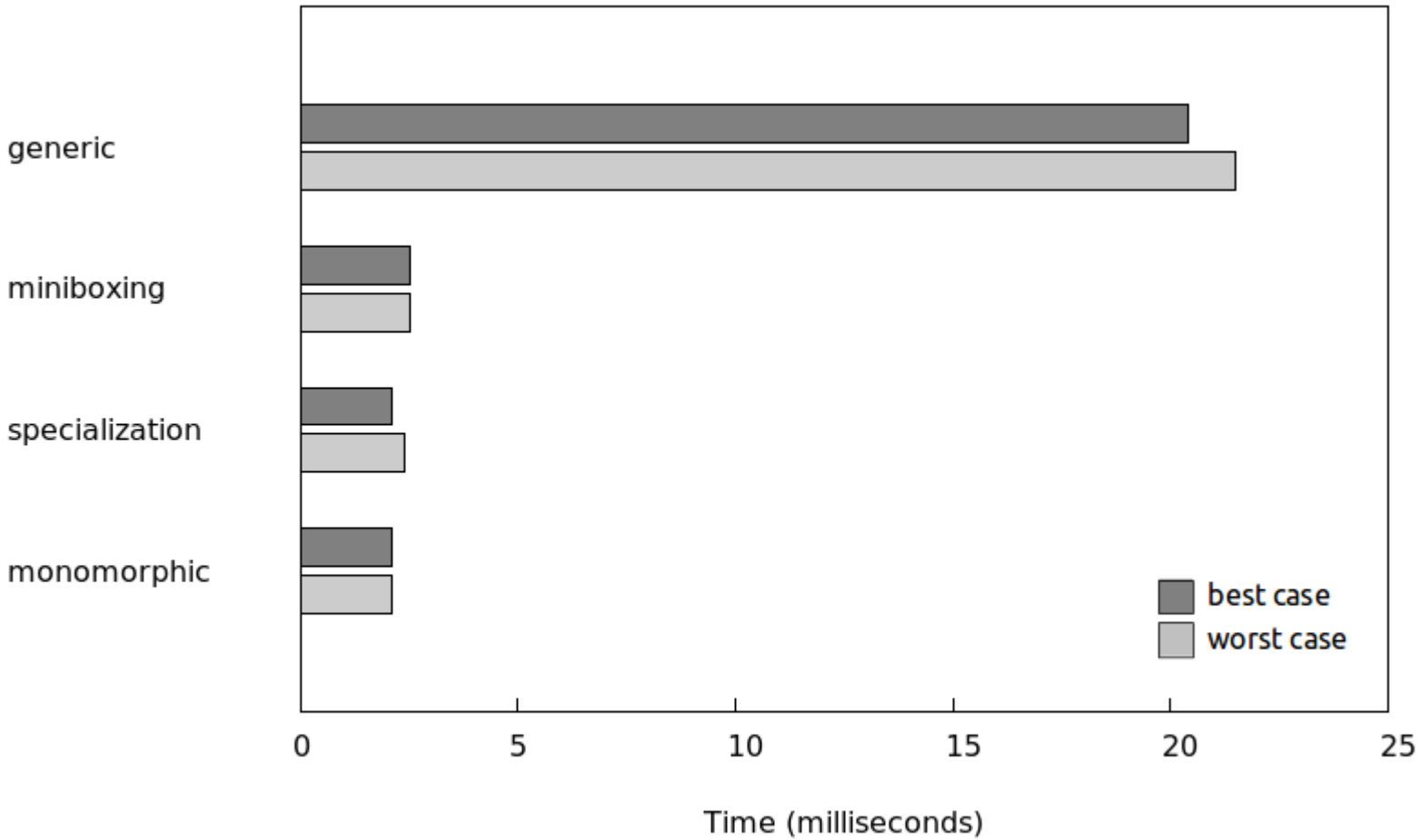
```
def identity[@miniboxed T](x: T): T = x
def identity[T](T_Tag: Byte, x: Long): Long = x
```



Benchmarks

Benchmarks

on ArrayBuffer.reverse



10x faster

Evaluation

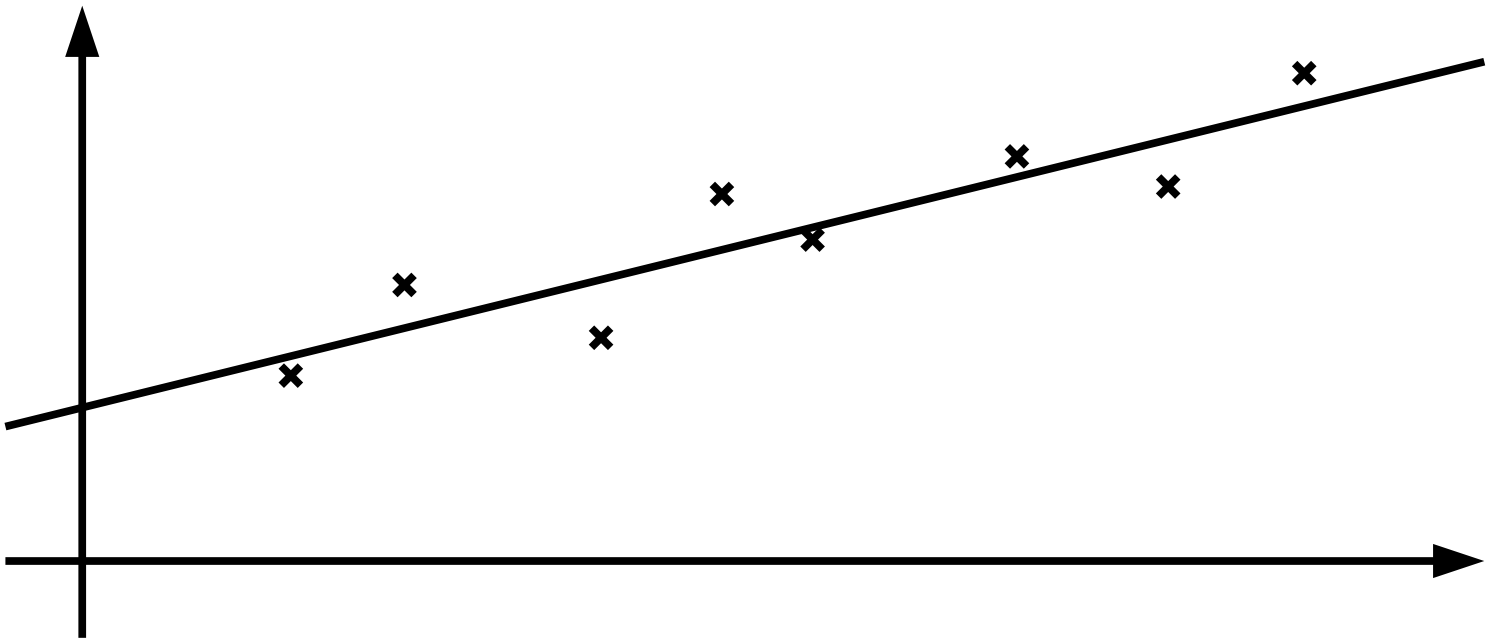
on the Scala linked list

- mock-up of Scala linked list
  - Function1 / Function2 / Tuple2
  - Traversable / TraversableLike
  - Iterator / Iterable / IterableLike
  - LinearSeqOptimized
  - Builder / CanBuildFrom

Benchmarks

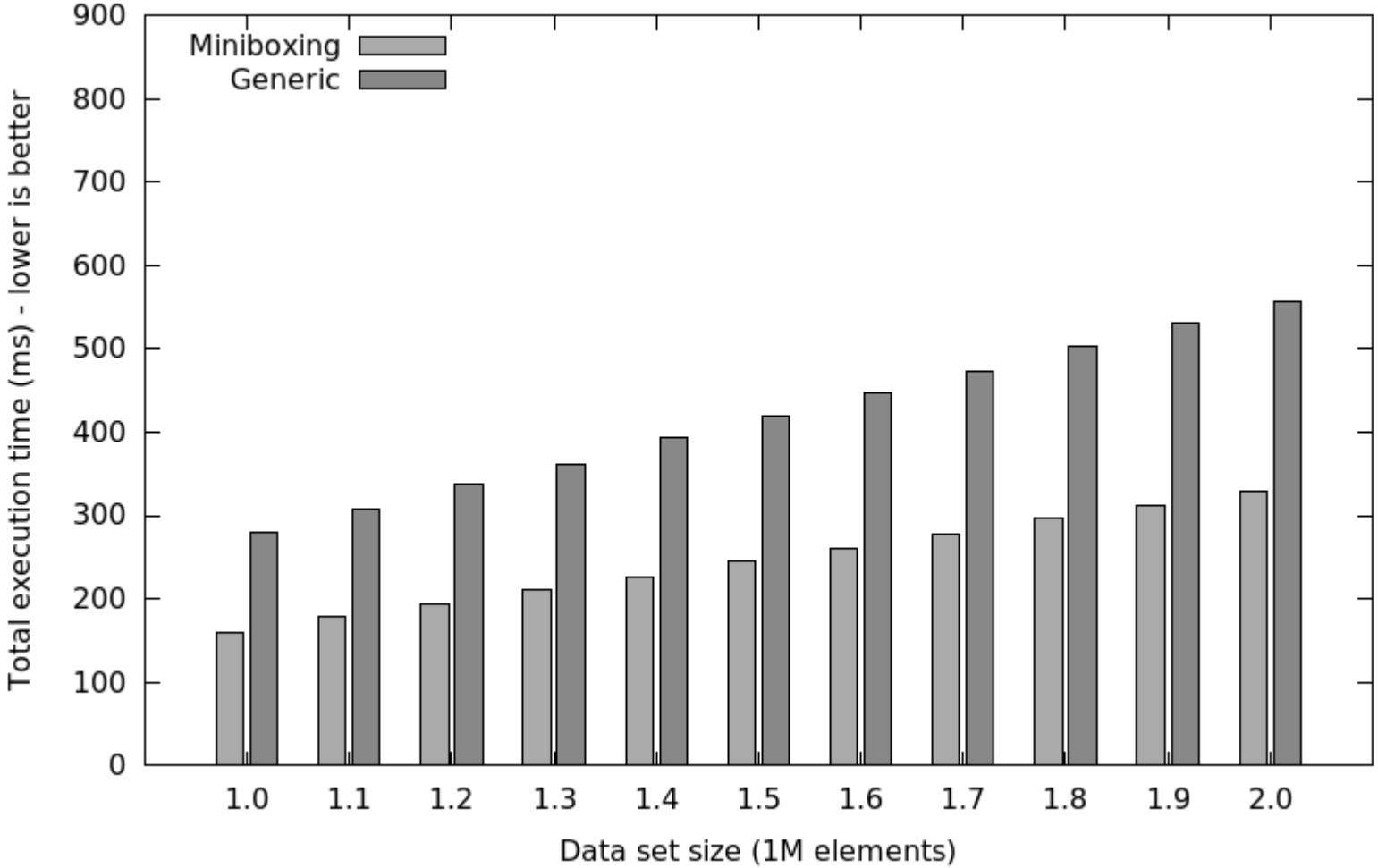
on the Scala library

- benchmark: Least Squares Method



Benchmarks

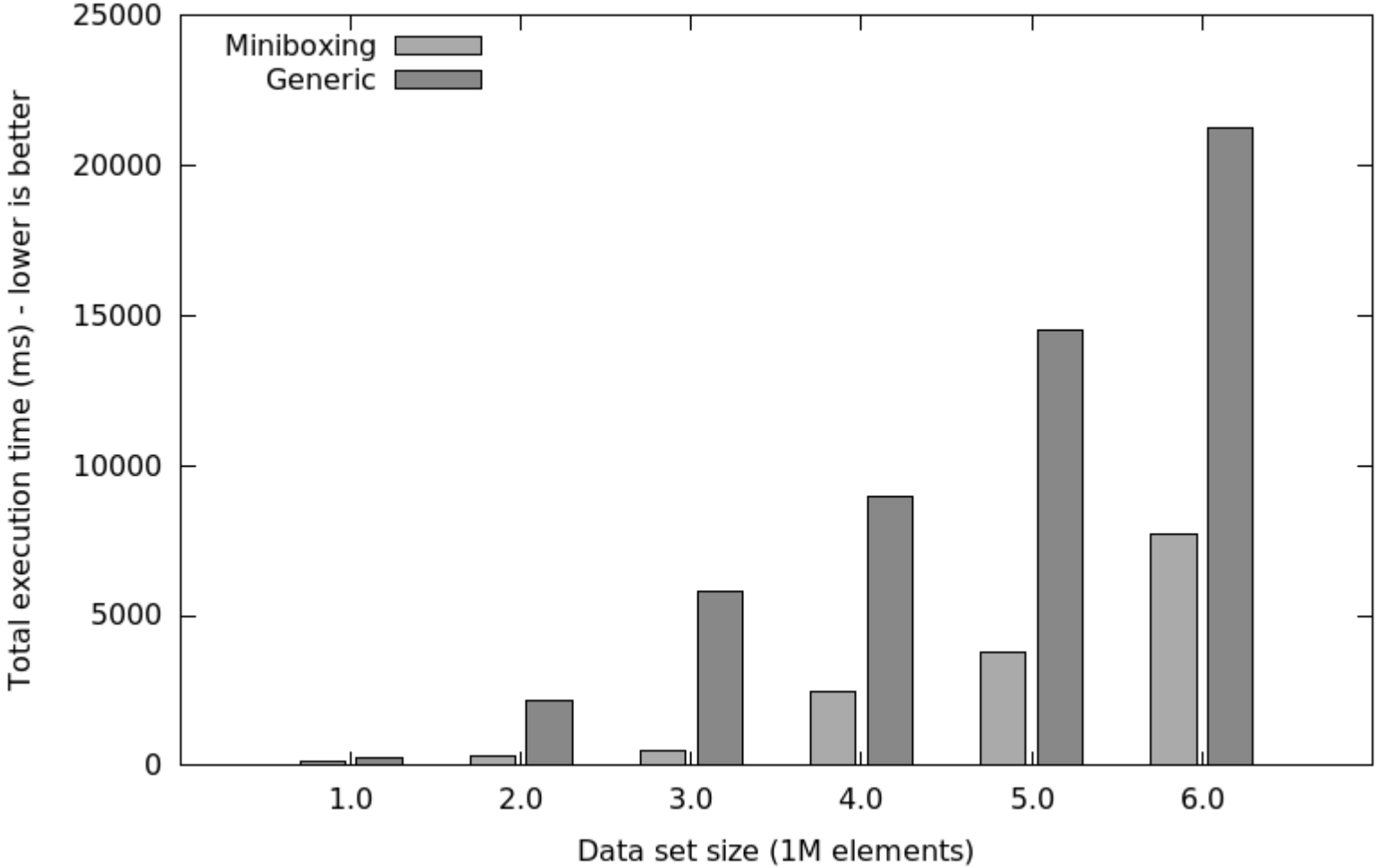
on the Scala library (infinte heap)



45% faster

Benchmarks

on the Scala library (limited heap)



3x faster

Other transformations

on the compiler-side

Other transformations

on the compiler-side

- a theory of **representation transformation**

Other transformations

on the compiler-side

- a theory of **representation transformation**
- which can be used for

Other transformations

on the compiler-side

- a theory of **representation transformation**
- which can be used for
  - auto(un)boxing

Other transformations

on the compiler-side

- a theory of **representation transformation**
- which can be used for
  - auto(un)boxing
  - specialization

Other transformations

on the compiler-side

- a theory of **representation transformation**
- which can be used for
  - auto(un)boxing
  - specialization
  - value classes

Other transformations

on the compiler-side

- a theory of **representation transformation**
- which can be used for
  - auto(un)boxing
  - specialization
  - value classes
  - staging

Other transformations

on the compiler-side

- a theory of **representation transformation**
- which can be used for
  - auto(un)boxing
  - specialization
  - value classes
  - staging
  - function representation

Other transformations

on the compiler-side

- a theory of **representation transformation**
- which can be used for
  - auto(un)boxing
  - specialization
  - value classes
  - staging
  - function representation

prototyped as scalac plugins

Other transformations

on the compiler-side

- a theory of **representation transformation**
- which can be used for
  - auto(un)boxing
  - specialization
  - value classes
  - staging

prototyped as scalac plugins

https://github.com/miniboxing/miniboxing-plugin/blob/wip/docs/2014-03-ldl-draft.pdf?raw=true

Credits

- **Cristian Talau** - developed the initial prototype, as a semester project
- **Eugene Burmako** - the value class plugin based on the LDL transformation
- **Aymeric Genet** - developing collection-like benchmarks for the miniboxing plugin
- Martin Odersky, for his patient guidance
- Eugene Burmako, for trusting the idea enough to develop the value-plugin based on the LDL transformation
- Iulian Dragos, for his work on specialization and many explanations
- Miguel Garcia, for his original insights that spawned the miniboxing idea
- Michel Schinz, for his wonderful comments and enlightening ACC course
- Andrew Myers and Roland Ducournau for the discussions we had and the feedback provided
- Heather Miller for the eye-opening discussions we had
- Vojin Jovanovic, Sandro Stucki, Manohar Jonalagedda and the whole LAMP laboratory in EPFL for the extraordinary atmosphere
- Adriaan Moors, for the miniboxing name which stuck :))
- Thierry Coppey, Vera Salvisberg and George Nithin, who patiently listened to many presentations and provided valuable feedback
- Grzegorz Kossakowski, for the many brainstorming sessions on specialization
- Erik Osheim, Tom Switzer and Rex Kerr for their guidance on the Scala community side
- OOPSLA paper and artifact reviewers, who reshaped the paper with their feedback
- Sandro, Vojin, Nada, Heather, Manohar - reviews and discussions on the LDL paper
- Hubert Plociniczak for the type notation in the LDL paper
- Denys Shabalin, Dmitry Petrashko for their patient reviews of the LDL paper

Thank you!

Vlad Ureche
twitter/github: VladUreche
email: vlad.ureche@epfl.ch

scala-miniboxing.org