scala-miniboxing.org

1st of August 2014
Scala Bay Area Meetup
Linkedin HQ, Mountain View

Vlad URECHE

PhD student in the Scala Team @ EPFL

Miniboxing guy. Also worked on specialization, the backend and scaladoc.


@VladUreche


@VladUreche


vlad.ureche@gmail.com

scala-miniboxing.org

What do **auto(un)boxing**, **specialization** and **value classes** have in common?

What do **auto(un)boxing**, **specialization** and **value classes** have in common?

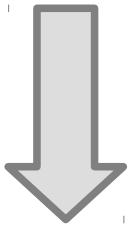
What are they?

Auto(un)boxing

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

Auto(un)boxing

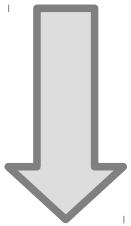
```
def identity[T](t: T): T = t
```



scalac / javac

Auto(un)boxing

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

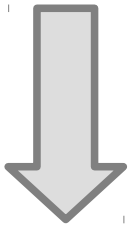


scalac / javac

```
def identity(t: Object): Object = t
```

Auto(un)boxing

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



scalac / javac

```
def identity(t: Object): Object = t
```

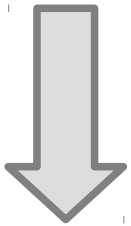
The process is called **erasure**, and is the simplest translation of generics to bytecode.

Auto(un)boxing

identity(5)

Auto(un)boxing

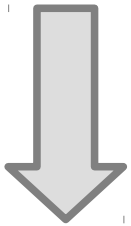
identity(5)



scalac / javac

Auto(un)boxing

identity(5)

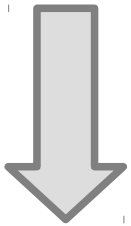


scalac / javac

identity(j.l.Integer.valueOf(5)).intValue

Auto(un)boxing

identity(5)



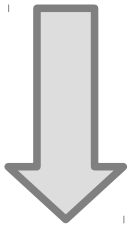
scalac / javac

identity(j.l.Integer.valueOf(5)).intValue

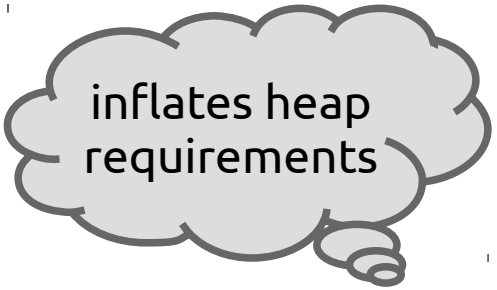
Object representation

Auto(un)boxing

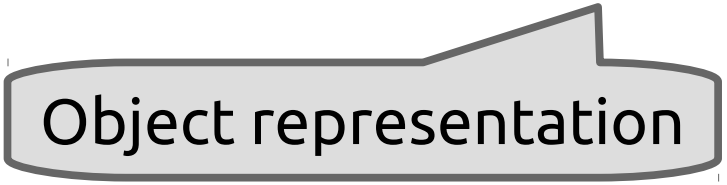
identity(5)



scalac / javac

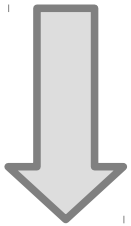


identity(j.l.Integer.valueOf(5)).intValue

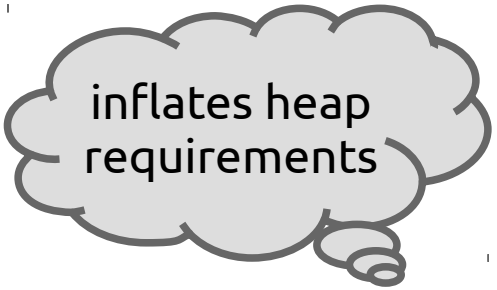


Auto(un)boxing

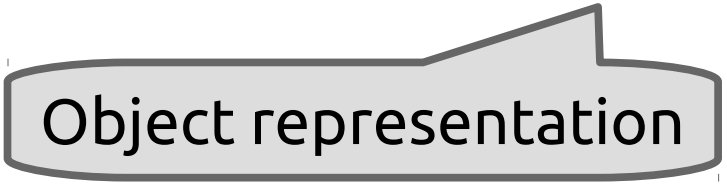
identity(5)



scalac / javac

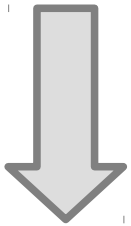


identity(j.l.Integer.valueOf(5)).intValue

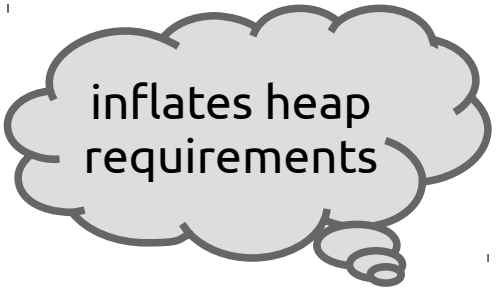


Auto(un)boxing

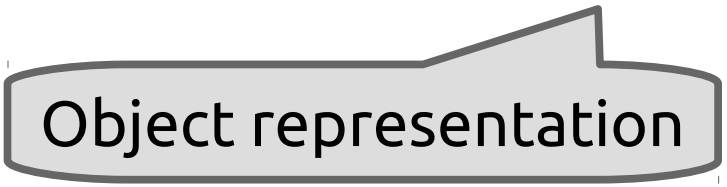
identity(5)

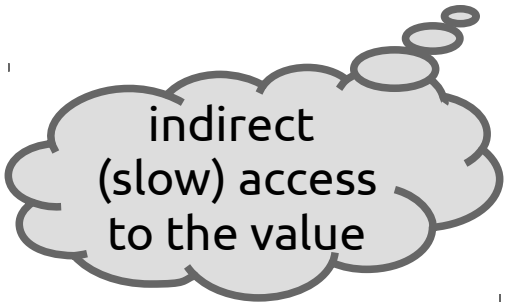


scalac / javac



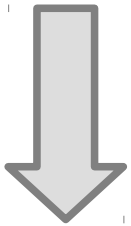
identity(j.l.Integer.valueOf(5)).intValue





Auto(un)boxing

identity(5)



scalac / javac

inflates heap requirements

produces garbage

identity(j.l.Integer.valueOf(5)).intValue

Object representation

indirect (slow) access to the value

breaks locality guarantees

Auto(un)boxing

```
val five: Int = 5
```

Auto(un)boxing

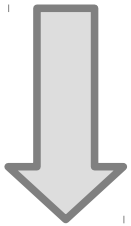
scala.Int behaves like an object
(has methods, can be used with generics)

```
val five: Int = 5
```

Auto(un)boxing

scala.Int behaves like an object
(has methods, can be used with generics)

```
val five: Int = 5
```

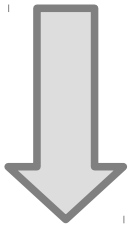


scalac

Auto(un)boxing

scala.Int behaves like an object
(has methods, can be used with generics)

```
val five: Int = 5
```



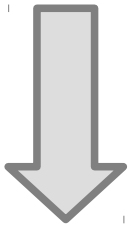
scalac

```
val five: int = 5
```

Auto(un)boxing

scala.Int behaves like an object
(has methods, can be used with generics)

val five: Int = 5



scalac

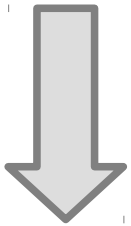
val five: int = 5

Unboxed integer

Auto(un)boxing

scala.Int behaves like an object
(has methods, can be used with generics)

```
val five: Int = 5        five + 3
```



scalac

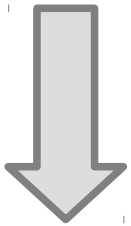
```
val five: int = 5
```

Unboxed integer

Auto(un)boxing

scala.Int behaves like an object
(has methods, can be used with generics)

val five: Int = 5

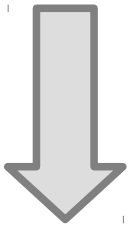


scalac

val five: int = 5

Unboxed integer

five + 3

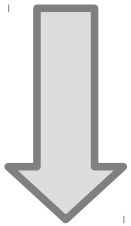


scalac

Auto(un)boxing

scala.Int behaves like an object
(has methods, can be used with generics)

val five: Int = 5

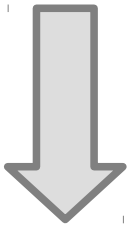


scalac

val five: int = 5

Unboxed integer

five + 3



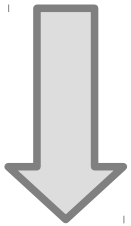
scalac

five + 3

Auto(un)boxing

scala.Int behaves like an object
(has methods, can be used with generics)

val five: Int = 5

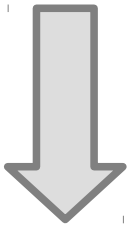


scalac

val five: int = 5

Unboxed integer

five + 3



scalac

five + 3

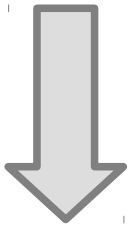
Unboxed addition

Auto(un)boxing

```
val five: Int = identity(5)
```

Auto(un)boxing

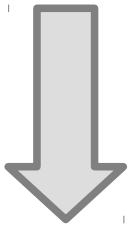
```
val five: Int = identity(5)
```



scalac

Auto(un)boxing

```
val five: Int = identity(5)
```

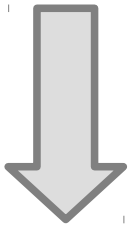


scalac

```
val five: int =
```

Auto(un)boxing

```
val five: Int = identity(5)
```

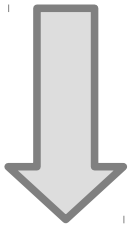


scalac

```
val five: int =
  identity(I.valueOf(5)).intValue
```

Auto(un)boxing

```
val five: Int = identity(5)
```



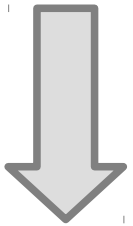
scalac

```
val five: int =
  identity(I.valueOf(5)).intValue
```

Boxing coercion

Auto(un)boxing

```
val five: Int = identity(5)
```



scalac

```
val five: int =
  identity(I.valueOf(5)).intValue
```

Boxing coercion

Unboxing coercion

Auto(un)boxing

scala.Int

Auto(un)boxing

Auto(un)boxing

scala.Int

int

- fast access
- no garbage collection
- locality

Auto(un)boxing

scala.Int

int

- fast access
- no garbage collection
- locality

java.lang.Integer

- indirect access
- object allocation
  - and thus garbage collection
- no locality guarantees
- **compatible with erased generics**

Auto(un)boxing

scala.Int

int

- fast access
- no garbage collection
- locality

java.lang.Integer

- indirect access
- object allocation
  - and thus garbage collection
- no locality guarantees
- **compatible with erased generics**

incompatible
→ **coercions**

What do ~~**auto(un)boxing**~~, **specialization** and **value classes** have in common?

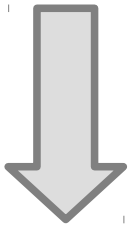
scala-miniboxing.org

Specialization

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

Specialization

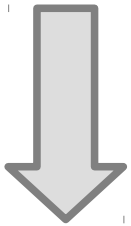
```
def identity[T](t: T): T = t
```



specialization

Specialization

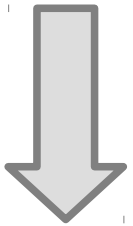
```
def identity[T](t: T): T = t
```




```
def identity(t: Object): Object = t
```

Specialization

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

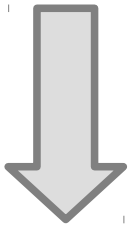


specialization

```
def identity(t: Object): Object = t
def identity_Z(t: bool): bool = t
```

Specialization

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

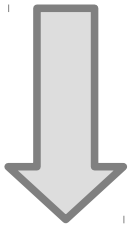


specialization

```
def identity(t: Object): Object = t
def identity_Z(t: bool): bool = t
def identity_C(t: char): char = t
```

Specialization

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



specialization

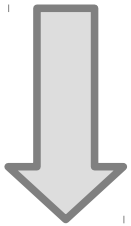
```
def identity(t: Object): Object = t
def identity_Z(t: bool): bool = t
def identity_C(t: char): char = t
... (7 other variants)
```

Specialization

identity(5)

Specialization

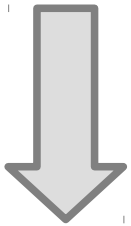
identity(5)



specialization

Specialization

identity(5)

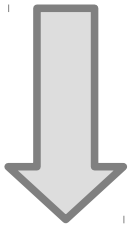


specialization

identity_I(5)

Specialization

identity(5)



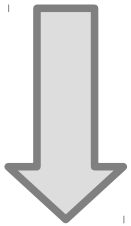
specialization

identity_I(5)

The variant of identity specialized for **int**

Specialization

identity(5)



specialization

identity_I(5) **// no boxing!**

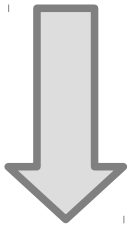
The variant of identity specialized for **int**

Specialization

```
def tupled[T1, T2](t1: T1, t2: T2) ...
```

Specialization

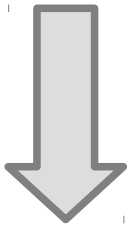
```
def tupled[T1, T2](t1: T1, t2: T2) ...
```



specialization

Specialization

```
def tupled[T1, T2](t1: T1, t2: T2) ...
```

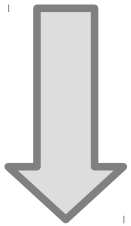


specialization

```
// 100 methods (10²)
```

Specialization

```
def tupled[T1, T2](t1: T1, t2: T2) ...
```

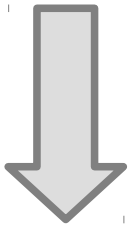


specialization

// 100 methods (10^2)

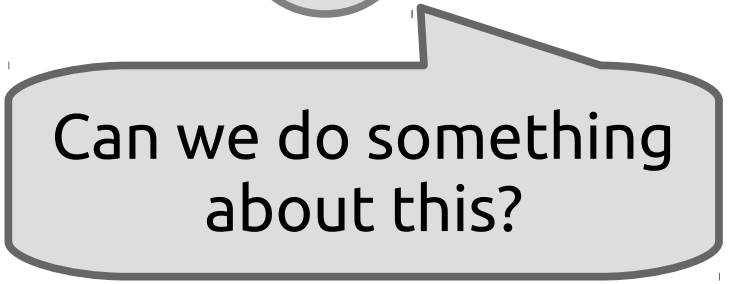
Specialization

```
def tupled[T1, T2](t1: T1, t2: T2) ...
```



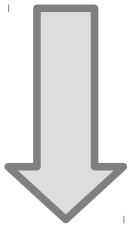
specialization

```
// 100 methods (10²)
```



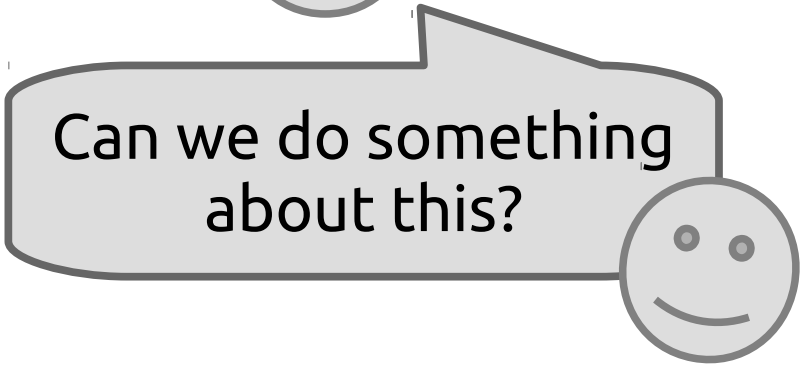
Specialization

```
def tupled[T1, T2](t1: T1, t2: T2) ...
```



specialization

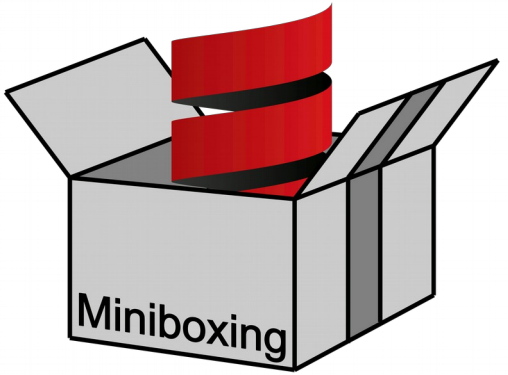
// 100 methods (10^2)



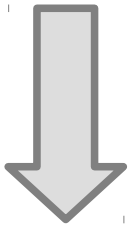
Miniboxing


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

Miniboxing

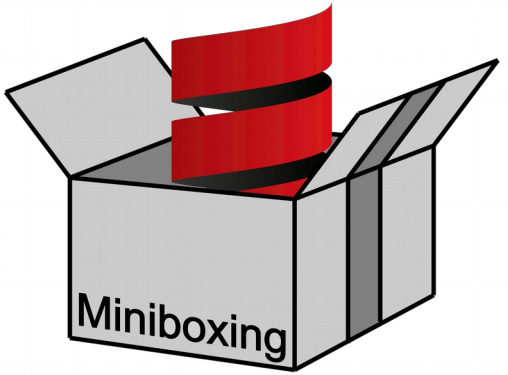



```
def identity[T](t: T): T = t
```

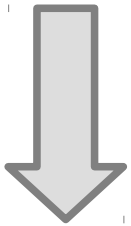


miniboxing

Miniboxing



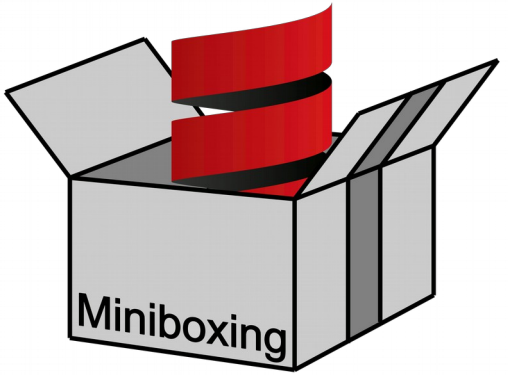

```
def identity[T](t: T): T = t
```



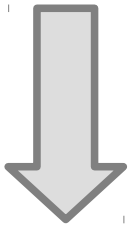
miniboxing

```
def identity(t: Object): Object = t
```

Miniboxing



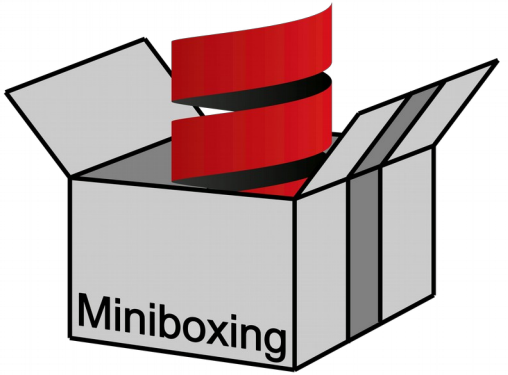

```
def identity[T](t: T): T = t
```



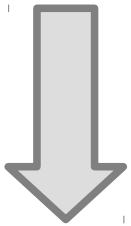
miniboxing

```
def identity(t: Object): Object = t
def identity_M(..., t: long): long = t
```

Miniboxing




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

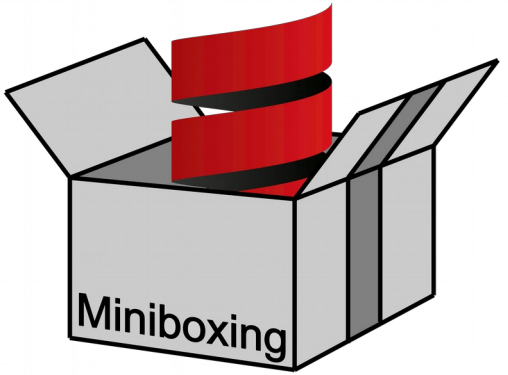


miniboxing

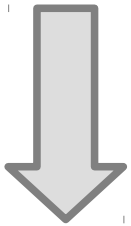
```
def identity(t: Object): Object = t
def identity_M(..., t: long): long = t
```

long **encodes** all primitive types

Miniboxing



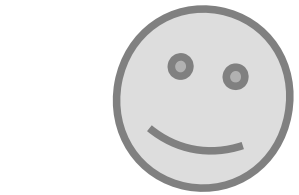

```
def identity[T](t: T): T = t
```



miniboxing

```
def identity(t: Object): Object = t
def identity_M(..., t: long): long = t
```

long **encodes** all primitive types



Only 2^n variants

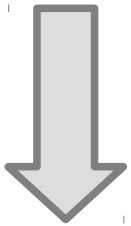
scala-miniboxing.org

Miniboxing

identity(3)

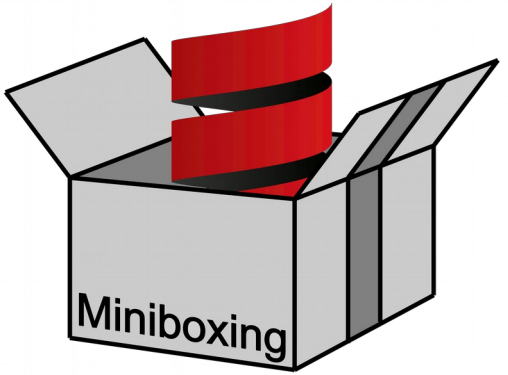
Miniboxing

identity(3)

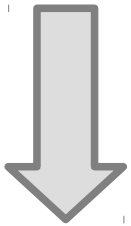


miniboxing

Miniboxing



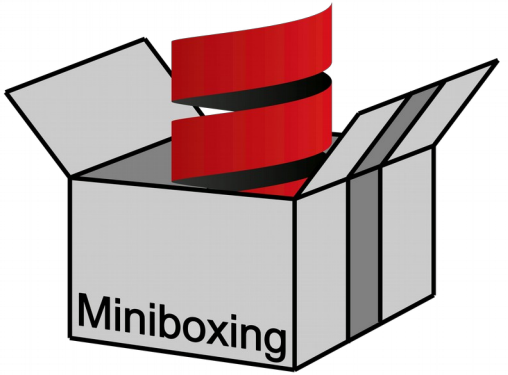
identity(3)



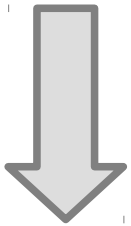
miniboxing

identity_M(..., int2minibox(3))

Miniboxing



identity(3)

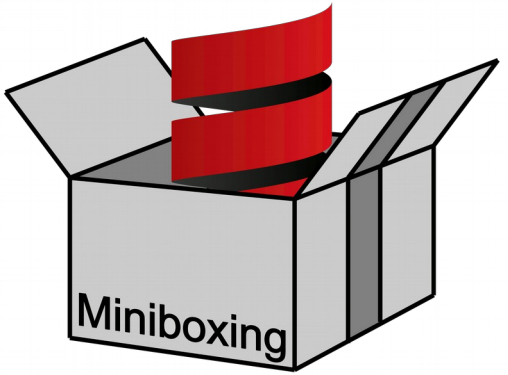


miniboxing

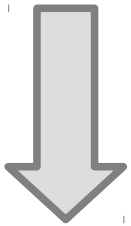
identity_M(..., int2minibox(3))

Coercion

Miniboxing



identity(3)



miniboxing

identity_M(..., int2minibox(3))

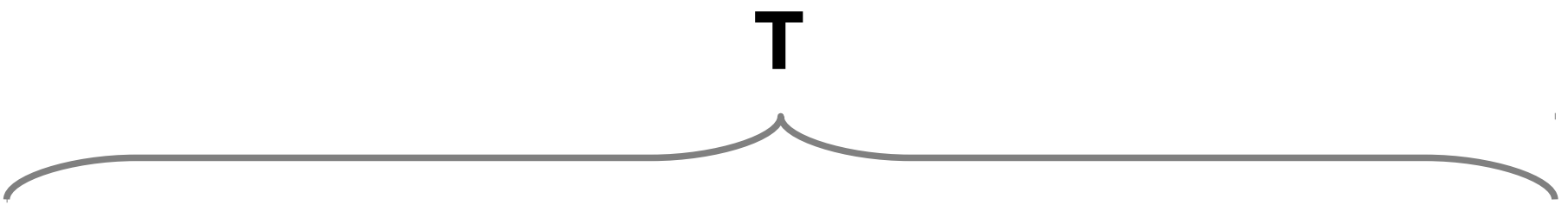
The miniboxed variant of identity

Coercion

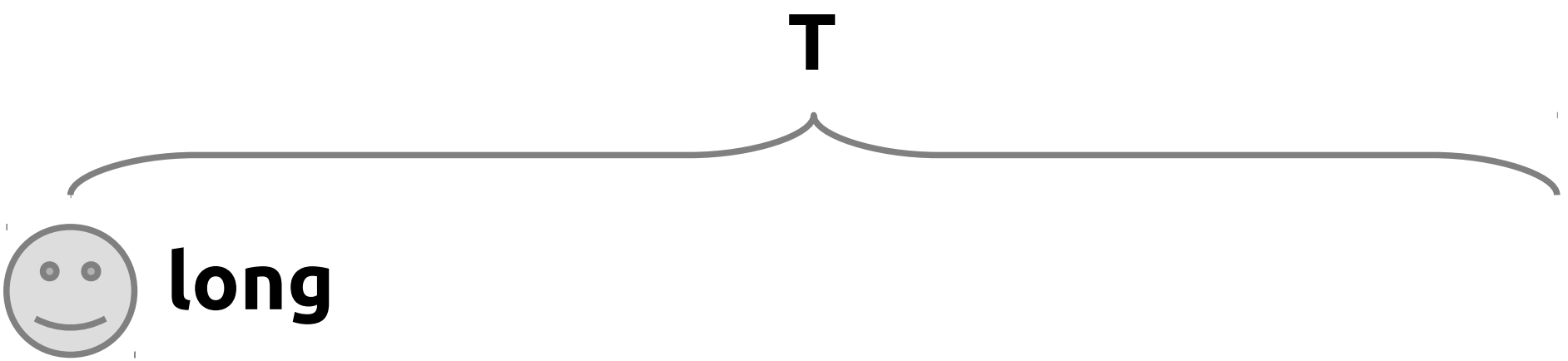
Miniboxing

T

Miniboxing



Miniboxing



long

- preferred encoding

Miniboxing

T

long

- preferred encoding

T (erased to Object)

- fallback encoding
- **compatible with**
  - **method calls**
  - **supertypes**
  - **erased generics**

Miniboxing

T

long

- preferred encoding

T (erased to Object)

- fallback encoding
- **compatible with**
  - **method calls**
  - **supertypes**
  - **erased generics**

incompatible
→ **coercions**

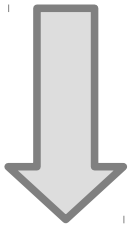
What do ~~**auto(un)boxing**~~, ~~**specialization**~~ and **value classes** have in common?

Value Classes

```
def abs(c: Complex): Double = ...
```

Value Classes

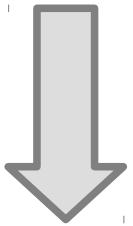
```
def abs(c: Complex): Double = ...
```



value class transformation

Value Classes

```
def abs(c: Complex): Double = ...
```

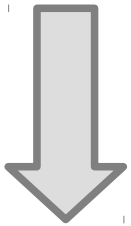


value class transformation

```
def abs(c_re: Double,
        c_im: Double): Double = ...
```

Value Classes

```
def abs(c: Complex): Double = ...
```



value class transformation

```
def abs(c_re: Double,
        c_im: Double): Double = ...
```

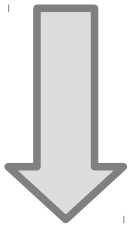
No object created!

Value Classes

```
val c: Complex = Complex(2,1)
```

Value Classes

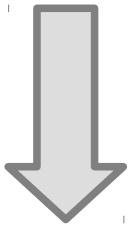
val c: Complex = Complex(2,1)



value class transformation

Value Classes

```
val c: Complex = Complex(2,1)
```

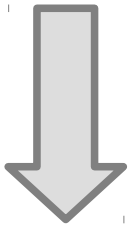


value class transformation

```
val c_re: Double = 2
val c_im: Double = 1
```

Value Classes

```
val c: Complex = Complex(2,1)
```



value class transformation

```
val c_re: Double = 2
val c_im: Double = 1
```

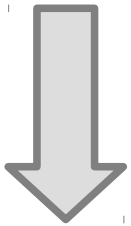
No object created!

Value Classes

```
val a: Any = c
```

Value Classes

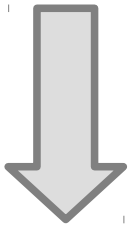
```
val a: Any = c
```



value class transformation

Value Classes

```
val a: Any = c
```

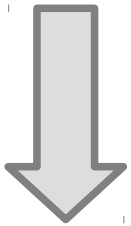


value class transformation

```
val a: Object =
        new Complex(c_re, c_im)
```

Value Classes

```
val a: Any = c
```



value class transformation

```
val a: Object =
        new Complex(c_re, c_im)
```

Coercion!

Value Classes

value class

Value Classes

value class

Value Classes

value class

structure **(by-val)**

- preferred encoding

Value Classes

value class

structure (by-val)

- preferred encoding

class (by-ref)

- fallback encoding
- **compatible with**
  - **supertypes**
  - **erased generics**

Value Classes

value class

structure (by-val)

- preferred encoding

class (by-ref)

- fallback encoding
- **compatible with**
  - **supertypes**
  - **erased generics**

incompatible
→ **coercions**

What do ~~**auto(un)boxing**~~, ~~**specialization**~~ and ~~**value classes**~~ have in common?

What do ~~**auto(un)boxing**~~, ~~**specialization**~~ and ~~**value classes**~~ have in common?

Multi-stage programs too, but we won't go there!

scala-miniboxing.org

What do **auto(un)boxing**, **specialization** and **value classes** have in common?

Auto(un)boxing

scala.Int

int

- fast access
- no garbage collection
- locality

java.lang.Integer

- indirect access
- garbage collection
  - and object allocation
- no locality guarantees
- **compatible with erased generics**

incompatible
→ **coercions**

Miniboxing

T

long

- preferred encoding

T (erased to Object)

- fallback encoding
- **compatible with**
  - **method calls**
  - **supertypes**
  - **erased generics**

incompatible
→ **coercions**

Value Classes

value class

structure (by-val)

- preferred encoding

class (by-ref)

- fallback encoding
- **compatible with**
  - **supertypes**
  - **erased generics**

incompatible
→ **coercions**

What do **auto(un)boxing**, **specialization** and **value classes** have in common?

scala-miniboxing.org

What do **auto(un)boxing**, **specialization** and **value classes** have in common?

Starting from a high-level concept:

What do **auto(un)boxing**, **specialization** and **value classes** have in common?

Starting from a high-level concept:

(1) **split it into multiple representations** based on **external constaints**

What do **auto(un)boxing**, **specialization** and **value classes** have in common?

Starting from a high-level concept:

(1) **split it into multiple representations** based on **external constaints**

(2) introduce the **necessary coercions** when the representation has to be converted

What do **auto(un)boxing**, **specialization** and **value classes** have in common?

Why is this important?

The transformation

Benchmark

Conclusion

scala-miniboxing.org

What do **auto(un)boxing**, **specialization** and **value classes** have in common?

Why is this important?

The transformation

Benchmark

Conclusion

scala-miniboxing.org

Why is this important?

Late Data Layout (LDL)

Late Data Layout (LDL)

concept

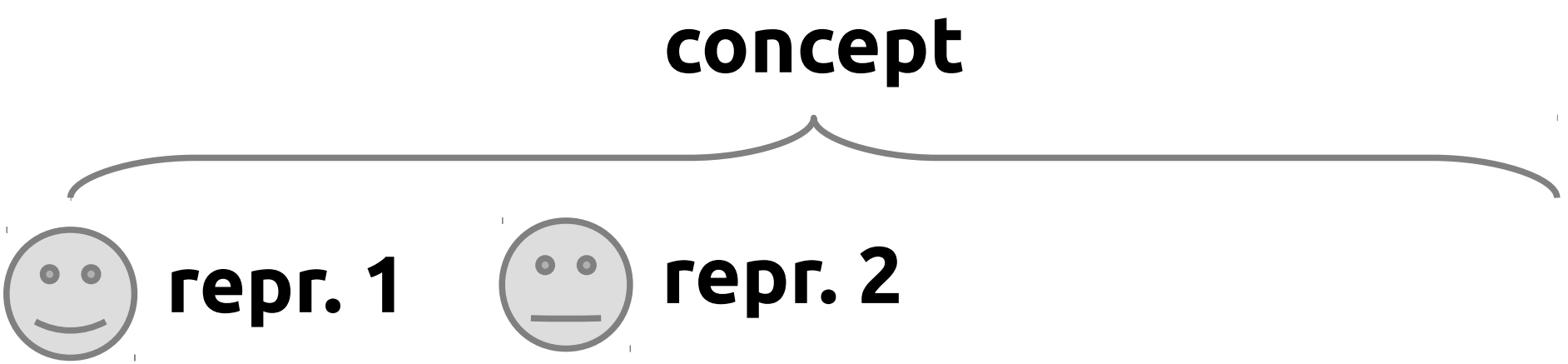
Late Data Layout (LDL)

Late Data Layout (LDL)

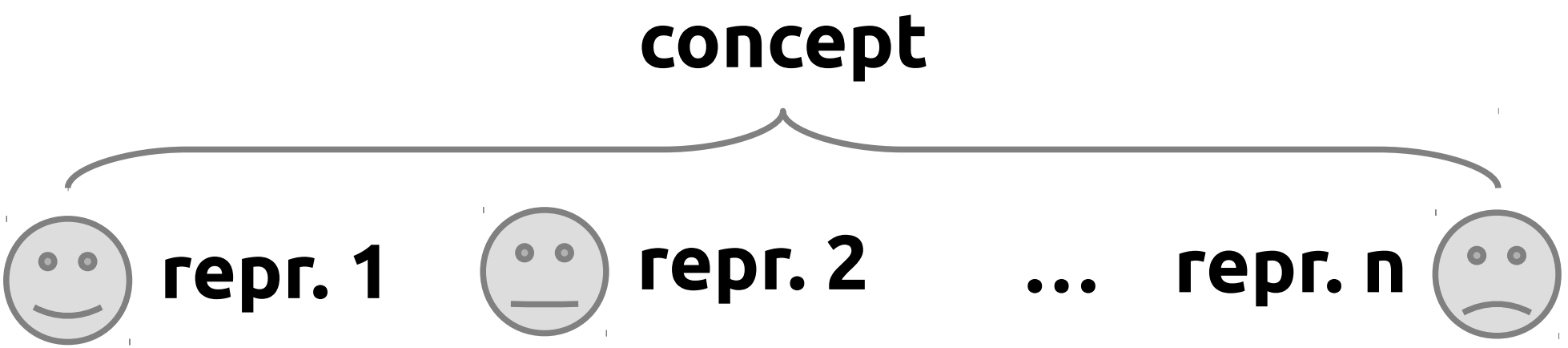
concept

repr. 1

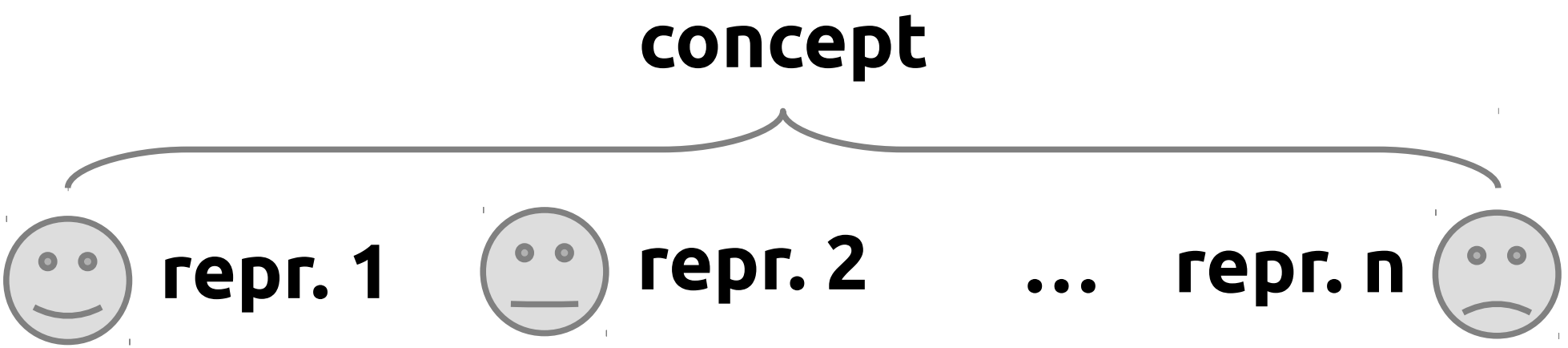
Late Data Layout (LDL)



Late Data Layout (LDL)



Late Data Layout (LDL)



Constraints from the interaction with other language features:

- generics
- subtyping
- virtual dispatch
- DSL semantics (staging)

Late Data Layout (LDL)

concept

repr. 1

repr. 2

...

repr. n

Constraints from the interaction with other language features:
- generics
- subtyping
- virtual dispatch
- DSL semantics (staging)

We've seen this pattern over and over again:
- autounboxing
- specialization
- value classes
- multi-stage programming
- function representation
- **collection representation**

Collection Representation

Collection Representation

List[T]

Stream[T]

- preferred encoding
- for pure comprehensions

Collection Representation

List[T]

Stream[T]

- preferred encoding
- for pure comprehensions

List[T]

- for impure comprehensions
- more expensive (needs to be materialized at each step)

Collection Representation

List[T]

Stream[T]

- preferred encoding
- for pure comprehensions

List[T]

- for impure comprehensions
- more expensive (needs to be materialized at each step)

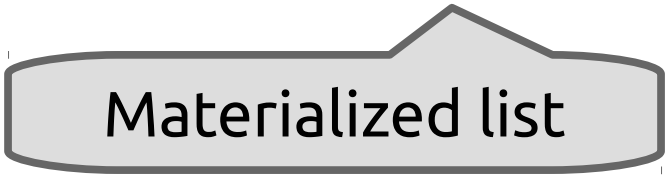
incompatible
→ **coercions**

Collection Representation

```
val c = List(1,2,3)
val d = c.map(_+1).filter(_%2==0)
```

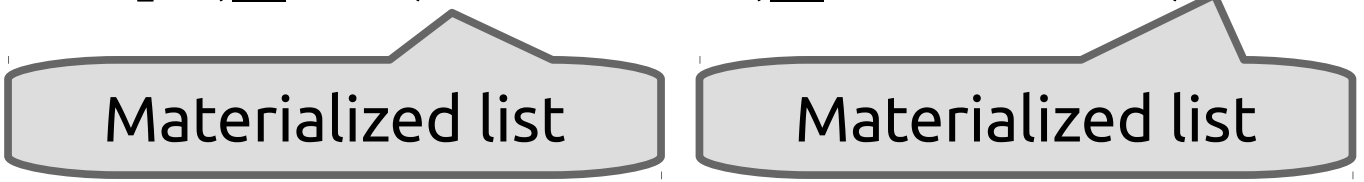
Collection Representation

```
val c = List(1,2,3)
val d = c.map(_+1).filter(_%2==0)
```



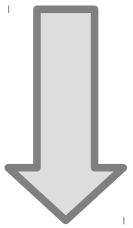
Collection Representation

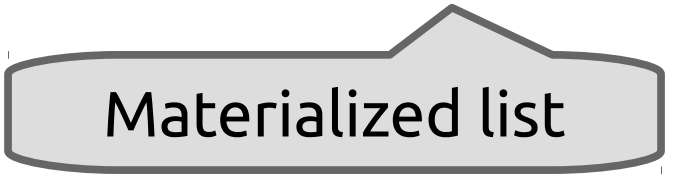
```
val c = List(1,2,3)
val d = c.map(_+1).filter(_%2==0)
```

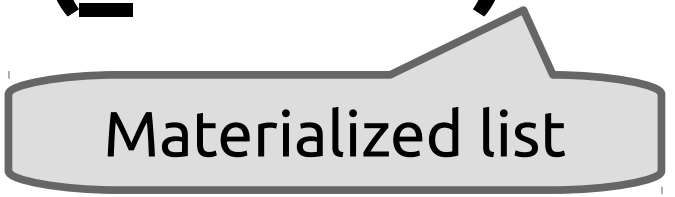


Collection Representation

```
val c = List(1,2,3)
val d = c.map(_+1).filter(_%2==0)
```

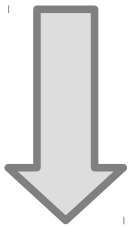






Collection Representation

```
val c = List(1,2,3)
val d = c.map(_+1).filter(_%2==0)
```



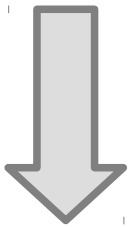
Materialized list

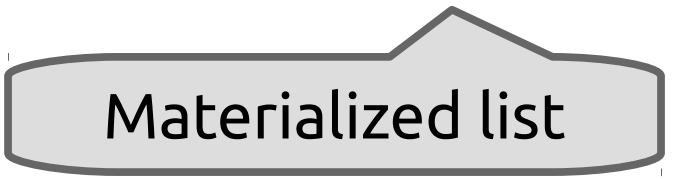
Materialized list

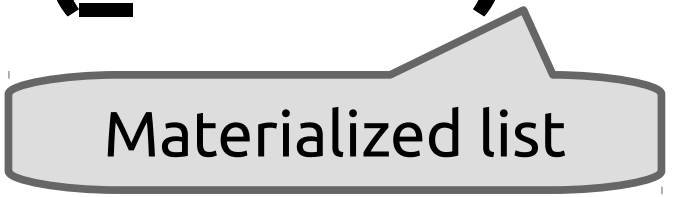
```
val c: Stream[Int] =
       List(1,2,3).toStream
```

Collection Representation

```
val c = List(1,2,3)
val d = c.map(_+1).filter(_%2==0)
```



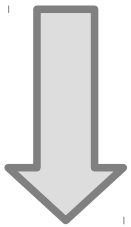





```
val c: Stream[Int] =
       List(1,2,3).toStream
val d: Stream[Int] =
       c.map(_+1).filter(_%2==0)
```

Collection Representation

```
val c = List(1,2,3)
val d = c.map(_+1).filter(_%2==0)
```



Materialized list

Materialized list

```
val c: Stream[Int] =
       List(1,2,3).toStream
val d: Stream[Int] =
       c.map(_+1).filter(_%2==0)
```

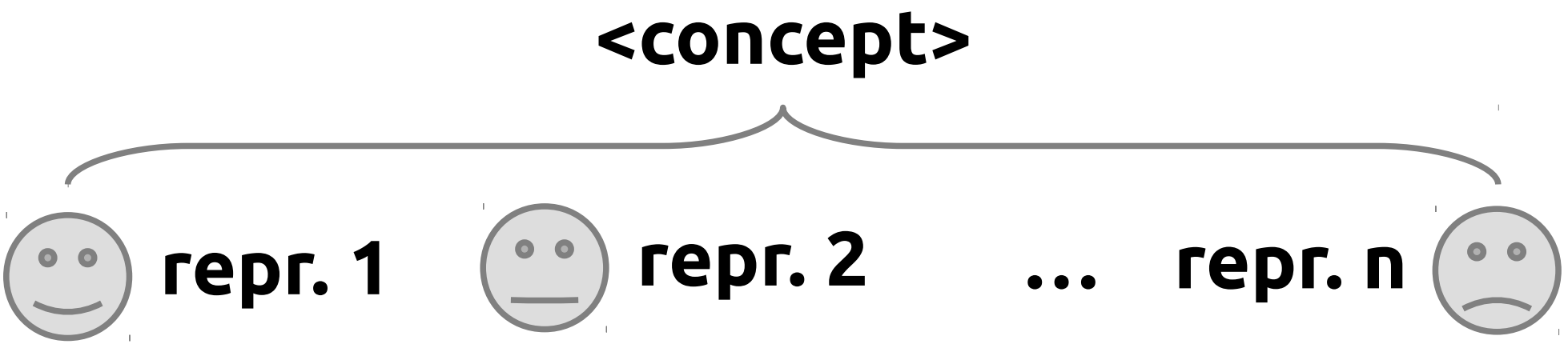
No materialization!

<Your Use Case>

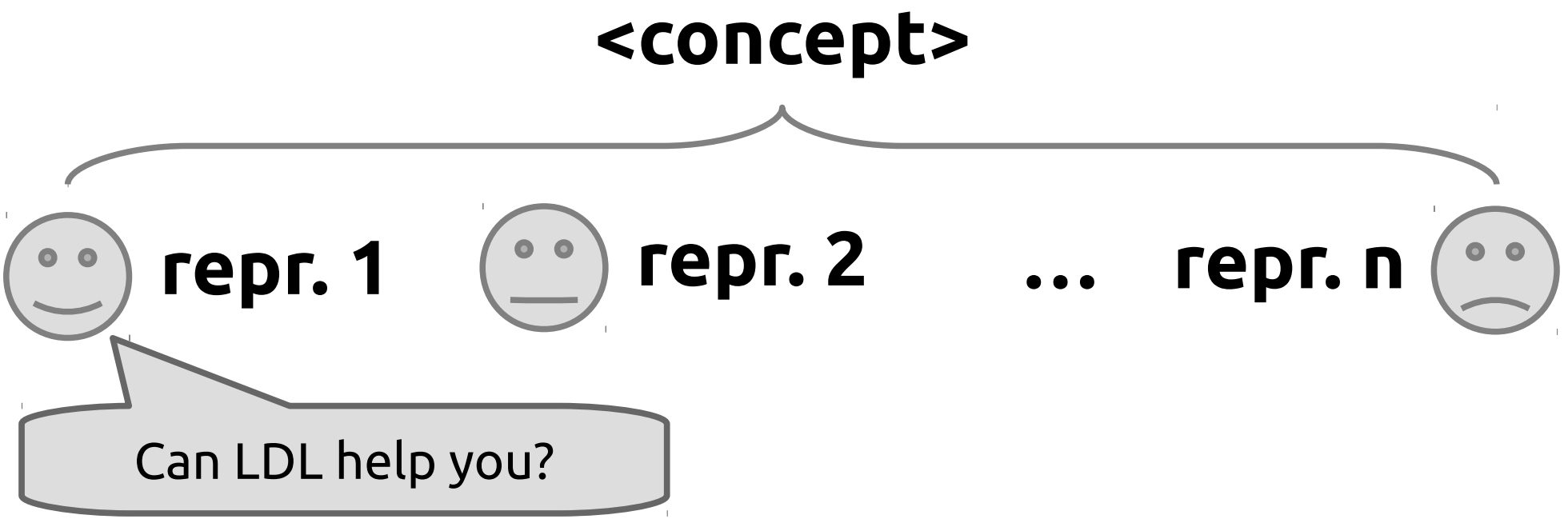
<Your Use Case>

<concept>

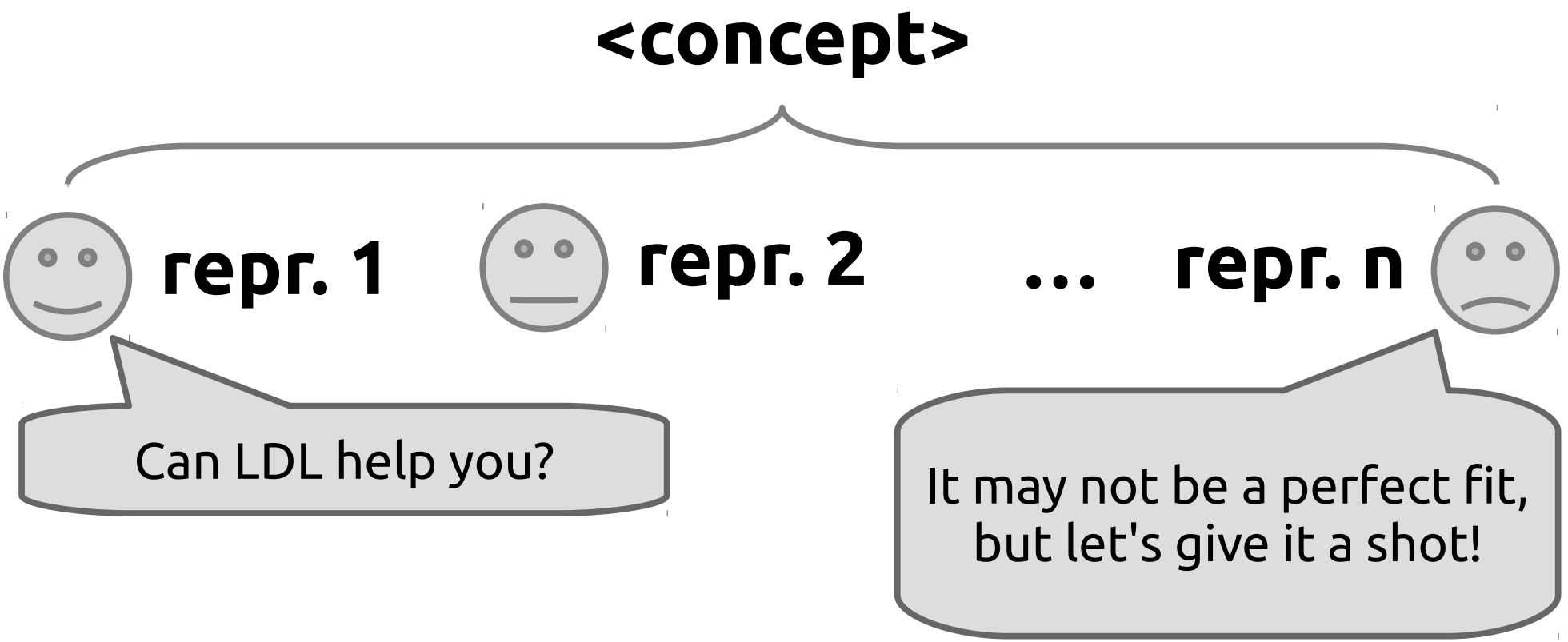
<Your Use Case>



<Your Use Case>



<Your Use Case>



What do **auto(un)boxing**, **specialization** and **value classes** have in common?

Why is this important?

The transformation

Benchmark

Conclusion

scala-miniboxing.org

What do **auto(un)boxing**, **specialization** and **value classes** have in common?

Why is this important?

The transformation

Benchmark

Conclusion

scala-miniboxing.org

How do transform a program?

How do transform a program?

If you understand the high-level picture, you will immediately see if your usecase matches.

How do transform a program?

If you understand the high-level picture, you will immediately see if your usecase matches.

We'll use autounboxing as the running example, to keep things simple

Auto(un)boxing

scala.Int

int

- fast access
- no garbage collection
- locality

java.lang.Integer

- indirect access
- object allocation
  - and thus garbage collection
- no locality guarantees
- **compatible with erased generics**

Auto(un)boxing

scala.Int

int

- fast access
- no garbage collection
- locality

java.lang.Integer

- indirect access
- object allocation
  - and thus garbage collection
- no locality guarantees
- **compatible with erased generics**

incompatible
→ **coercions**

Naive transformation

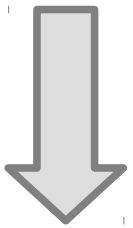
Syntax-based transformation

Type-based LDL transformation

scala-miniboxing.org

Naive transformation

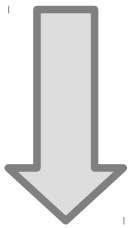
```
val x: Int = List[Int](1, 2, 3).head
val y: List[Int] = List[Int](x)
```



naive unboxing

Naive transformation

```
val x: Int = List[Int](1, 2, 3).head
val y: List[Int] = List[Int](x)
```

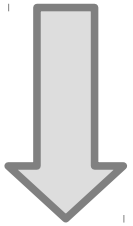


naive unboxing

```
val x: int = List[Int](1, 2, 3).head
val y: List[Int] = List[Int](x)
```

Naive transformation

```
val x: Int = List[Int](1, 2, 3).head
val y: List[Int] = List[Int](x)
```



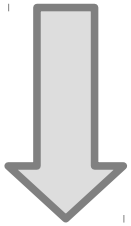
naive unboxing

representation mismatch:
expected: **int**
found: **Int**

```
val x: int = List[Int](1, 2, 3).head
val y: List[Int] = List[Int](x)
```

Naive transformation

```
val x: Int = List[Int](1, 2, 3).head
val y: List[Int] = List[Int](x)
```



naive unboxing

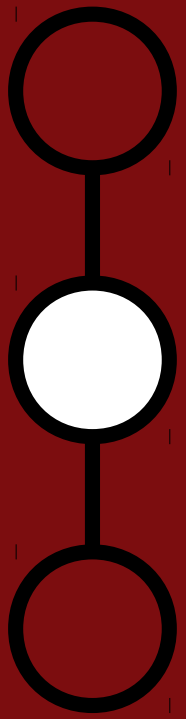
representation mismatch:
expected: **int**
found: **Int**

```
val x: int = List[Int](1, 2, 3).head
val y: List[Int] = List[Int](x)
```

representation mismatch:
expected: **Int**
found: **int**

Naive transformation

- naively replacing representations
  - leads to mismatches
  - which are hard to recover
    (impossible for value classes and miniboxing)
- we need **coercions** between representations



Naive transformation

Syntax-based transformation

Type-based LDL transformation

scala-miniboxing.org

Syntax-based

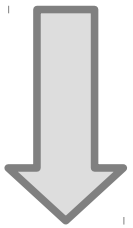
- when transforming a value
  - coerce the definition right-hand side
  - coerce all references to it

Syntax-based

```
val x: Int = List[Int](1, 2, 3).head
```

Syntax-based

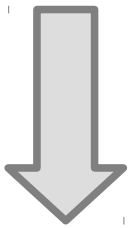
```
val x: Int = List[Int](1, 2, 3).head
```



syntax-based unboxing

Syntax-based

```
val x: Int = List[Int](1, 2, 3).head
```

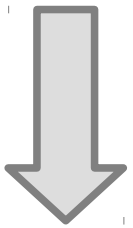


syntax-based unboxing

```
val x: int =
```

Syntax-based

```
val x: Int = List[Int](1, 2, 3).head
```

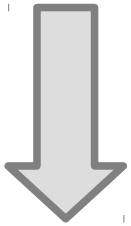


syntax-based unboxing

```
val x: int =
    unbox(List[Int](1, 2, 3).head)
```

Syntax-based

```
val x: Int = List[Int](1, 2, 3).head
```



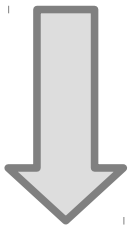
syntax-based unboxing

```
val x: int =
    unbox(List[Int](1, 2, 3).head)
```

There are no references to x,
so there's nothing else to do.

Syntax-based

```
val x: Int = List[Int](1, 2, 3).head
```



syntax-based unboxing

```
val x: int =
    unbox(List[Int](1, 2, 3).head)
```

There are no references to x,
so there's nothing else to do.

Syntax-based

```
val x: Int = List[Int](1, 2, 3).head
val y: Int = x
```

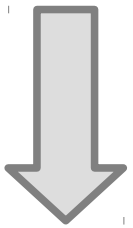
Syntax-based


```
val x: Int = List[Int](1, 2, 3).head
val y: Int = x
```

Syntax-based

Transform one by one

```
val x: Int = List[Int](1, 2, 3).head
val y: Int = x
```

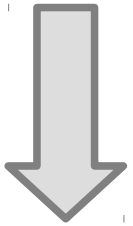


syntax-based unboxing

Syntax-based

Transform one by one

```
val x: Int = List[Int](1, 2, 3).head
val y: Int = x
```



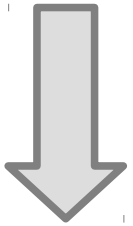
syntax-based unboxing

```
val x: int =
    unbox(List[Int](1, 2, 3).head)
```

Syntax-based

Transform one by one

```
val x: Int = List[Int](1, 2, 3).head
val y: Int = x
```



syntax-based unboxing

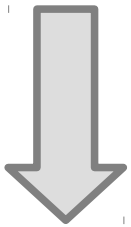
```
val x: int =
       unbox(List[Int](1, 2, 3).head)
val y: Int = box(x)
```

Syntax-based

```
val x: int =
        unbox(List[Int](1, 2, 3).head)
val y: Int = box(x)
```

Syntax-based

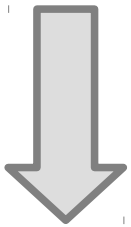
```
val x: int =
      unbox(List[Int](1, 2, 3).head)
val y: Int = box(x)
```



syntax-based unboxing

Syntax-based

```
val x: int =
      unbox(List[Int](1, 2, 3).head)
val y: Int = box(x)
```

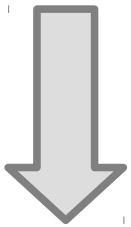


syntax-based unboxing

```
val x: int =
      unbox(List[Int](1, 2, 3).head)
val y: int =
```

Syntax-based

```
val x: int =
      unbox(List[Int](1, 2, 3).head)
val y: Int = box(x)
```

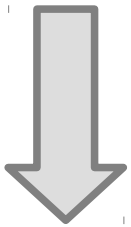


syntax-based unboxing

```
val x: int =
      unbox(List[Int](1, 2, 3).head)
val y: int = unbox(box(x))
```

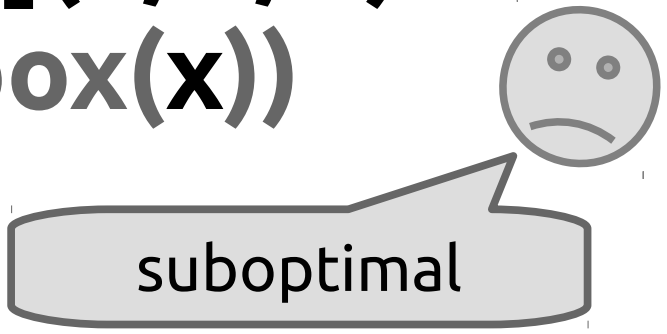
Syntax-based

```
val x: int =
      unbox(List[Int](1, 2, 3).head)
val y: Int = box(x)
```



syntax-based unboxing

```
val x: int =
      unbox(List[Int](1, 2, 3).head)
val y: int = unbox(box(x))
```

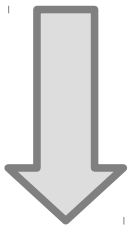


Peephole Optimization

```
val y: int = unbox(box(x))
```

Peephole Optimization

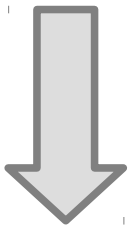
```
val y: int = unbox(box(x))
```



peephole

Peephole Optimization

```
val y: int = unbox(box(x))
```

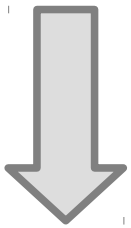


peephole

```
val y: int = x
```

Peephole Optimization

```
val y: int = unbox(box(x))
```

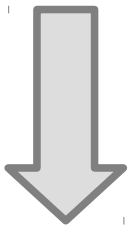


peephole

```
val y: int = x
```

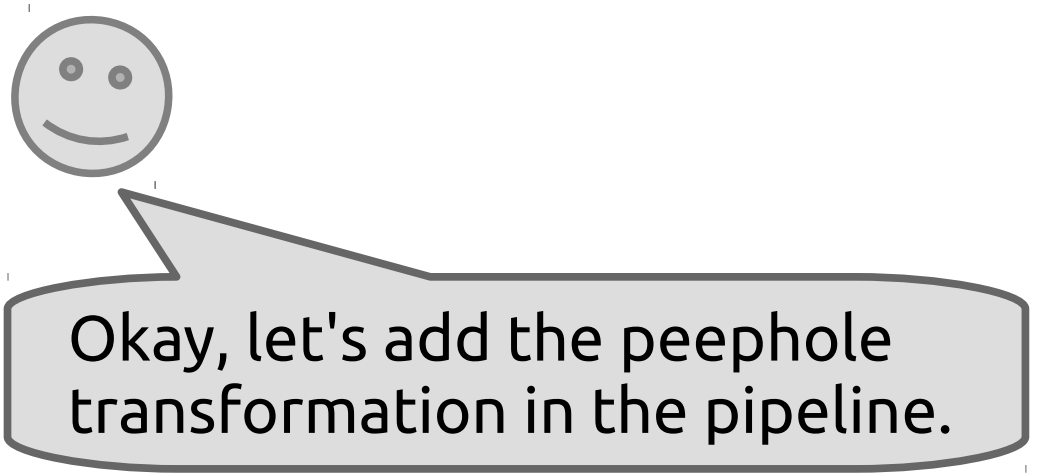
Peephole Optimization

```
val y: int = unbox(box(x))
```



peephole

```
val y: int = x
```



Syntax-based

```
def choice(t1: Int, t2: Int): Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

Syntax-based


```
def choice(t1: Int, t2: Int): Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

Syntax-based

```
def choice(t1: int, t2: Int): Int =
  if (Random.nextBoolean())
    box(t1)
  else
    t2
```

Syntax-based

```
def choice(t1: int, t2: int): Int =
  if (Random.nextBoolean())
    box(t1)
  else
    box(t2)
```

Syntax-based

```
def choice(t1: int, t2: int): Int =
  if (Random.nextBoolean())
    box(t1)
  else
    box(t2)
```

Syntax-based

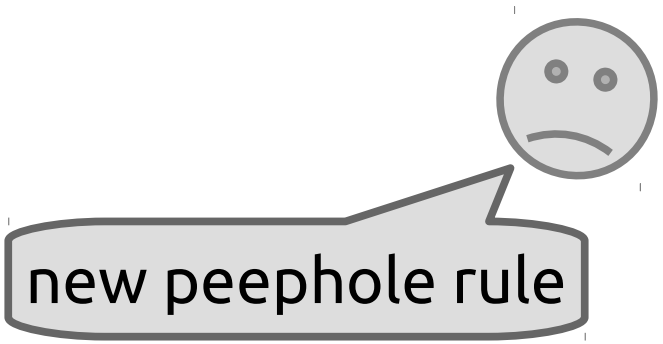
```
def choice(t1: int, t2: int): int =
  unbox(if (Random.nextBoolean())
    box(t1)
  else
    box(t2))
```

Syntax-based

```
def choice(t1: int, t2: int): int =
  unbox(if (Random.nextBoolean())
    box(t1)
  else
    box(t2))
```

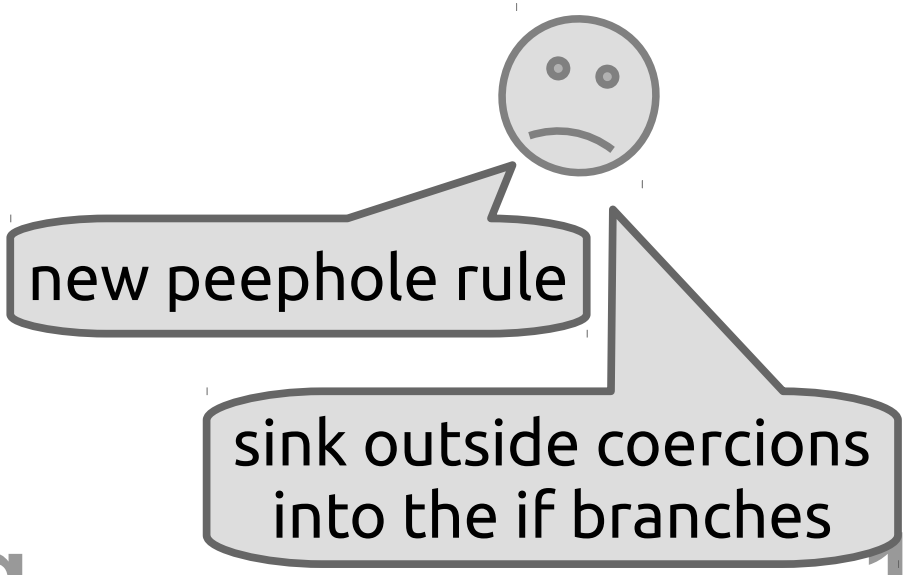
Syntax-based

```
def choice(t1: int, t2: int): int =
  unbox(if (Random.nextBoolean())
    box(t1)
  else
    box(t2))
```



Syntax-based

```
def choice(t1: int, t2: int): int =
  unbox(if (Random.nextBoolean())
    box(t1)
  else
    box(t2))
```



Syntax-based

```
def choice(t1: int, t2: int): int =
  if (Random.nextBoolean())
    unbox(box(t1))
  else
    unbox(box(t2))
```

Syntax-based

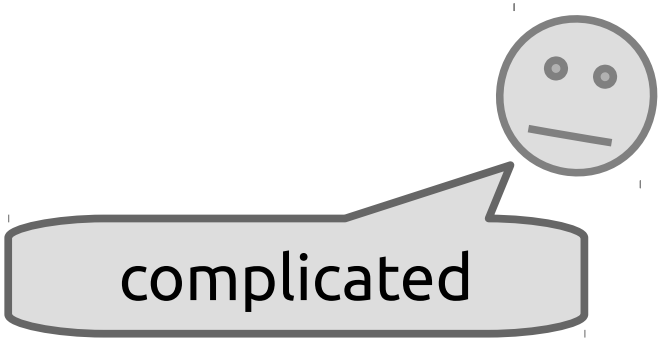
```
def choice(t1: int, t2: int): int =
  if (Random.nextBoolean())
    unbox(box(t1))
  else
    unbox(box(t2))
```

Syntax-based

```
def choice(t1: int, t2: int): int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

Syntax-based

```
def choice(t1: int, t2: int): int =
  if (Random.nextBoolean())
    t1
  else
    t2
```



Syntax-based

- peephole transformation does not scale
  - needs multiple rules for each node
  - needs successive rewriting => slow
  - **impossible to guarantee optimality**

Naive transformation

Syntax-based transformation

Type-based LDL transformation

scala-miniboxing.org

Type-based transformation

- propagate representation information
  - into the type system (based on annotated types)
  - let the type system propagate this information

Type-based transformation

- re-typechecking the tree
  - exposes inconsistencies in the representation
  - so we introduce coercions
    - **only when representations don't match**

Type-based transformation

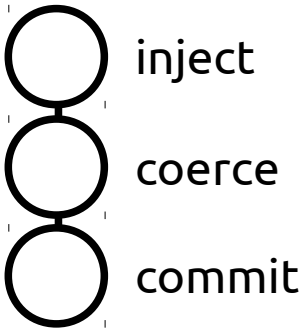
- three-stage mechanism
  - inject → annotate the values to be unboxed
  - coerce → introduce coercion markers
  - commit → commit to the alternative representations

Type-based transformation

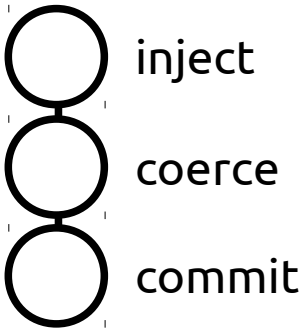
Warning!

Throughout the presentation we'll be writing annotations written **before types** (e.g. “@unboxed Int”), although in the Scala syntax they are written **after the type** (e.g.“Int @unboxed”). This makes it easier to read the types aloud.

Type-based

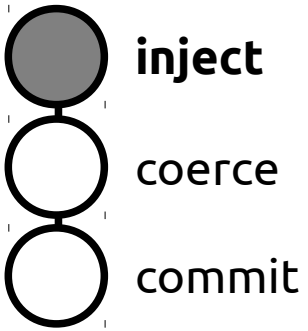


Type-based



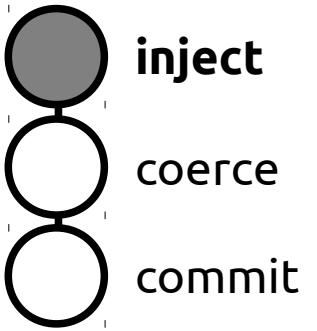

```
def choice(t1: Int,
           t2: Int): Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

Type-based



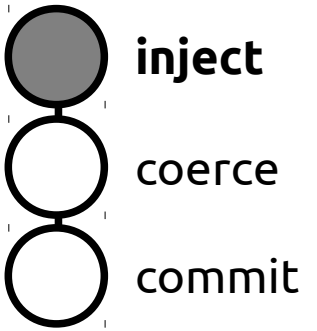

```
def choice(t1: Int,
           t2: Int): Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

Type-based




```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

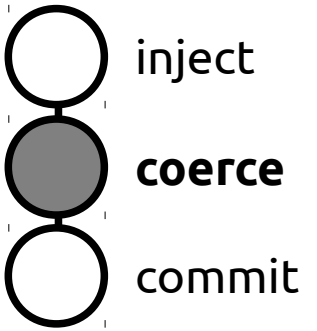
Type-based




```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

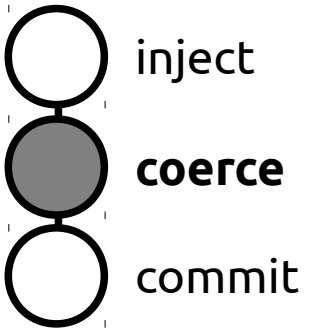
configurable introduction of annotations
based on external constraints

Type-based



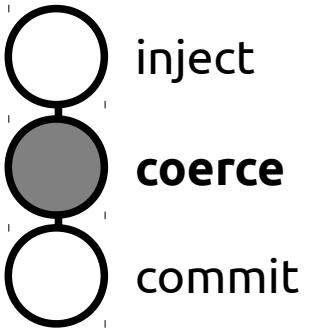

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

Type-based



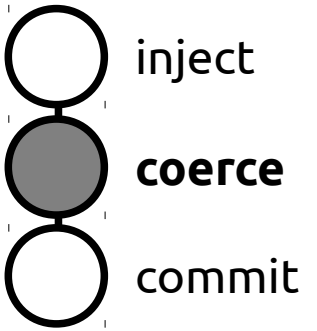

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

Type-based




```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

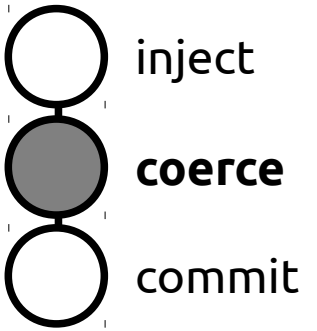
Type-based



the rhs expression must be of type **@unboxed Int**

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

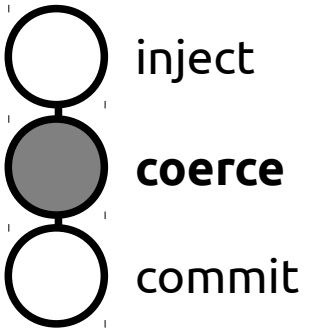
Type-based



the rhs expression must be of type @unboxed Int

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2                                    : @unboxed Int
```

Type-based

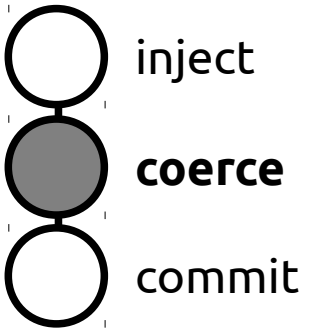


the rhs expression must be of type **@unboxed Int**

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2                                    : @unboxed Int
```

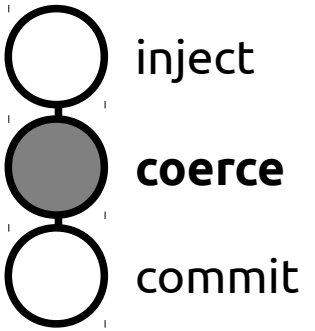
expected type
(part of local type inference)

Type-based




```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean()) : Boolean
    t1
  else
    t2
```

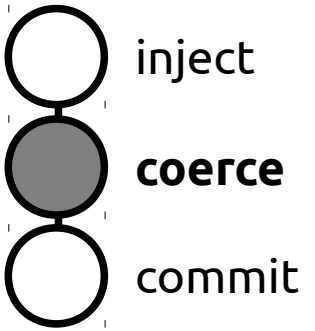
Type-based




```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean()) : Boolean
    t1
  else
    t2
```

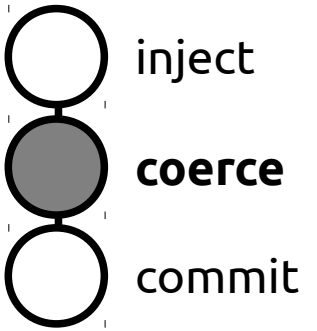
matches:
expected: **Boolean**
found: **Boolean**

Type-based




```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1 : @unboxed Int
  else
    t2
```

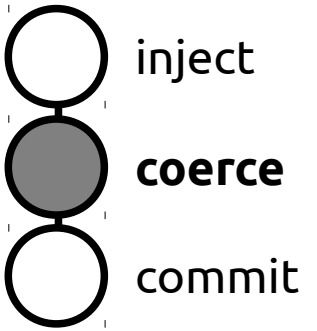
Type-based




```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1 : @unboxed Int
  else
    t2
```

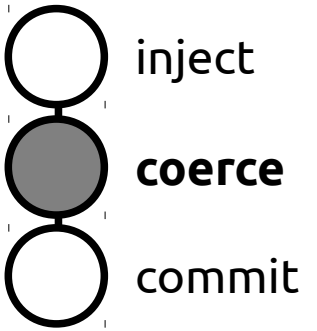
matches:
expected: @unboxed Int
found: @unboxed Int

Type-based




```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2 : @unboxed Int
```

Type-based

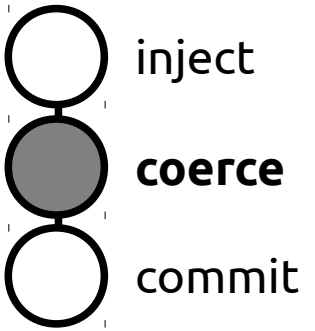



```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2 : @unboxed Int
```

matches:

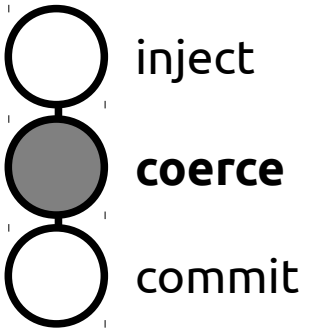
...

Type-based

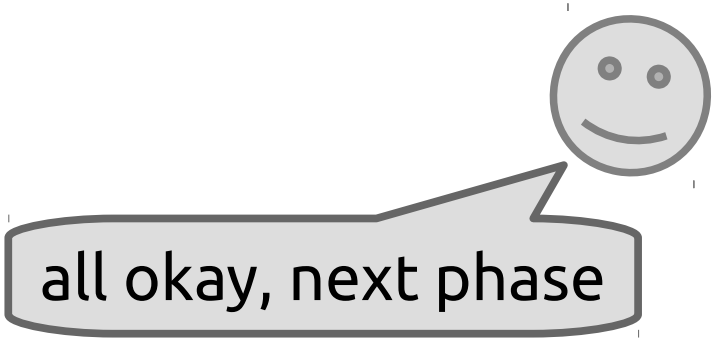



```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

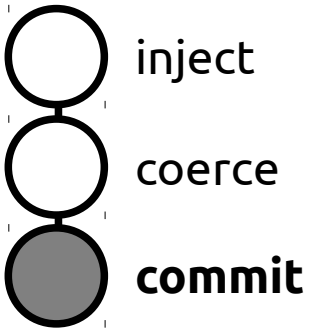
Type-based




```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

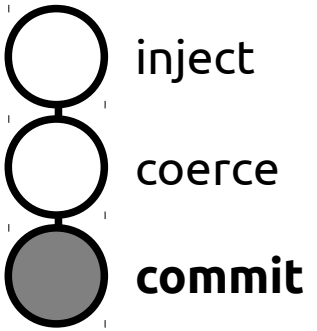


Type-based




```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

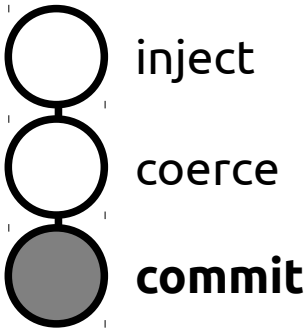
Type-based




```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

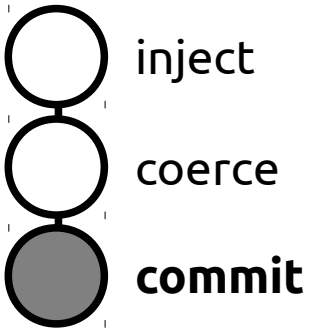
Replace:
@unboxed **Int** → **int**
(not showing **Int** → **j.l.Integer**)

Type-based

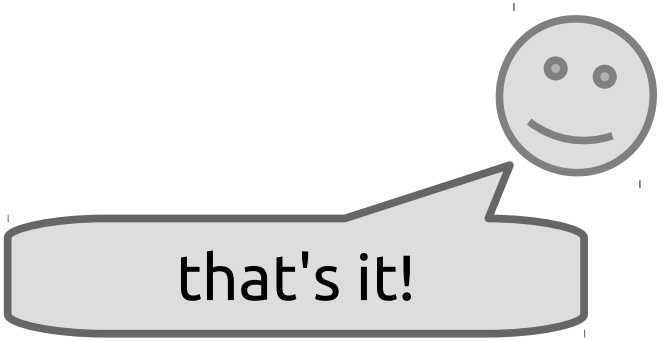



```
def choice(t1: int,
           t2: int): int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

Type-based




```
def choice(t1: int,
           t2: int): int =
  if (Random.nextBoolean())
    t1
  else
    t2
```



Type-based transformation

- three-stage mechanism
  - inject: add annotations
  - coerce: add coercions (based on the annotations)
  - commit: final representation semantics

Type-based transformation

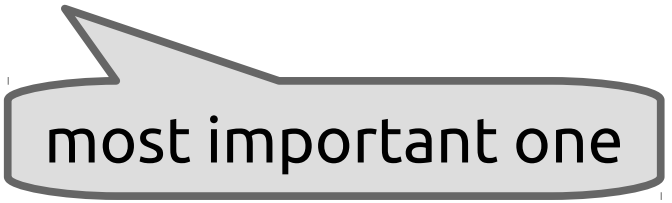
- Scalac's erasure
  - similar transformation
  - less flexible (no annotations)
  - entangled with other transformations
- we took what's good
  - and allowed the transformation to work on other usecases as well

Type-based transformation

- why do this?
  - at the core of:
    - miniboxing (http://scala-miniboxing.org)
    - value classes plugin (https://github.com/miniboxing/value-plugin)
    - multi-stage plugin (https://github.com/miniboxing/staging-plugin)
    - **<your transformation here>**

Type-based transformation

- why do this?
  - at the core of:
    - miniboxing (http://scala-miniboxing.org)
    - value classes plugin (https://github.com/miniboxing/value-plugin)
    - multi-stage plugin (https://github.com/miniboxing/staging-plugin)
    - **<your transformation here>**



Type-based LDL transformation

Properties

Consistency

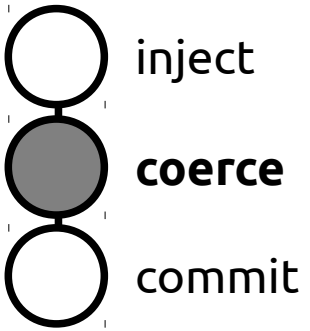
Selectivity

Optimality*

scala-miniboxing.org

* not formally proven yet

Consistency



- representations become explicit in types
  - representation mismatches
    - become type mismatches
    - are exposed by the type system
  - mismatches lead to coercions
    - explicit bridges between representations
    - are introduced automatically
  - regardless of the representations
    - at a meta-level

Type-based LDL transformation

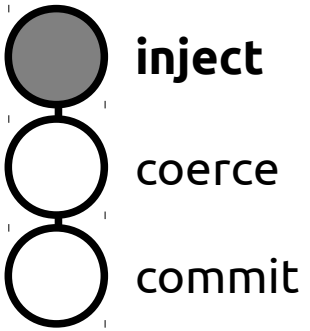
Properties

- Consistency
- Selectivity
- Optimality*

scala-miniboxing.org

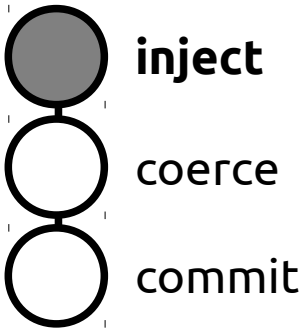
* not formally proven yet

Selectivity



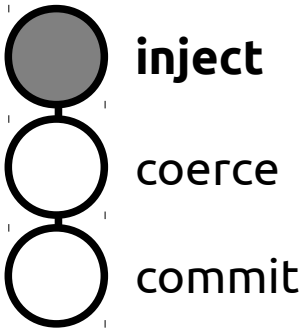
- annotations allow selectively picking the values to be transformed
  - value classes
    - cannot unbox multi-param values in return position (not supported by the JVM platform)
    - bridge methods
  - staging
    - annotations signal **domain-specific knowledge**
      - can occur inside generics (List[@staged Int])

Selectivity




```
def choice(t1: Int,
           t2: Int): Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

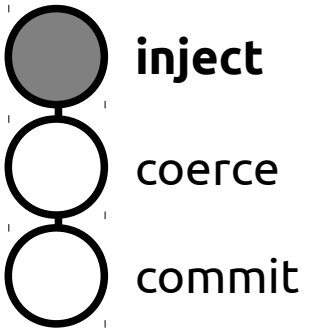
Selectivity



what if we did not annotate t1?

```
def choice(t1: Int,
           t2: Int): Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

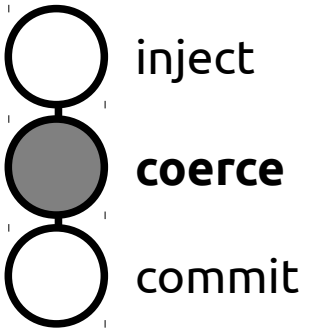
Selectivity



what if we did not annotate t1?

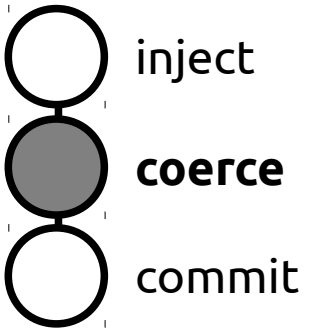
```
def choice(t1: Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

Selectivity



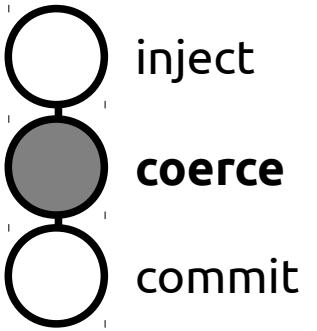

```
def choice(t1: Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2                                    : @unboxed Int
```

Selectivity




```
def choice(t1: Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1 : @unboxed Int
  else
    t2
```

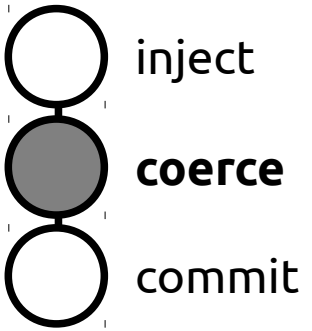
Selectivity




```
def choice(t1: Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1 : @unboxed Int
  else
    t2
```

mismatch:
expected: @unboxed Int
found: Int

Selectivity

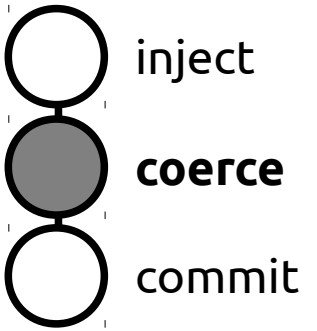



```
def choice(t1: Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1 : @unboxed Int
  else
    t2
```

mismatch:
expected: @unboxed Int
found: Int

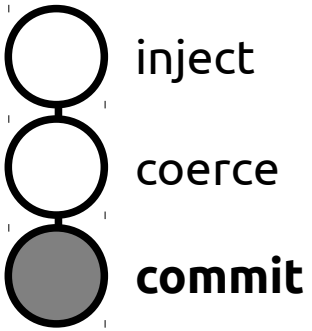
coercion

Selectivity



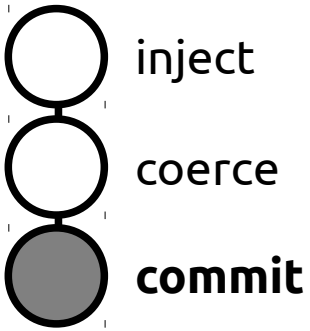

```
def choice(t1: Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    unbox(t1)
  else
    t2
```

Selectivity




```
def choice(t1: Int,
                  t2: @unboxed Int): @unboxed Int =
 if (Random.nextBoolean())
  unbox(t1)
 else
  t2
```

Selectivity




```
def choice(t1: Int,
                  t2: int): int =
  if (Random.nextBoolean())
    t1.intValue
  else
    t2
```

Type-based LDL transformation

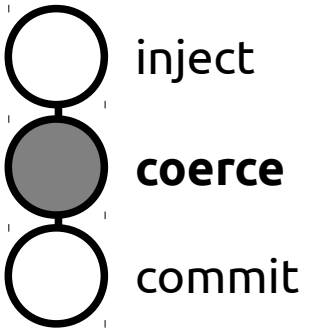
Properties

- Consistency
- Selectivity
- Optimality*

scala-miniboxing.org

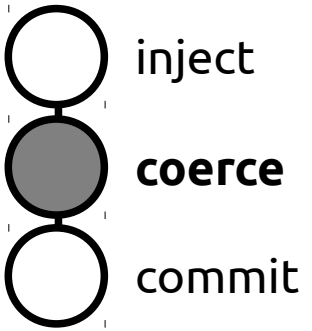
* not formally proven yet

Optimality



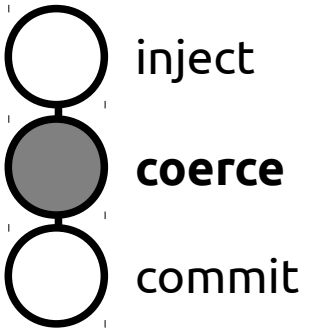

```
def choice(t1: Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    unbox(t1)
  else
    t2
```

Optimality




```
def choice(t1: Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    unbox(t1)
  else
    t2
```

Optimality




```
def choice(t1: Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    unbox(t1)
  else
    t2
```

Coercions are sunk in the tree → excute only if necessary

Type-based LDL transformation

Use cases

- Miniboxing
- Value Classes*
- Multi-stage Programming*

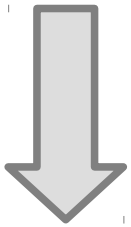
scala-miniboxing.org * early prototypes

LDL for miniboxing

```
def tupled[T1, T2](t1: T1, t2: T2) = ...
```

LDL for miniboxing

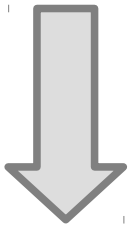
```
def tupled[T1, T2](t1: T1, t2: T2) = ...
```



miniboxing

LDL for miniboxing

```
def tupled[T1, T2](t1: T1, t2: T2) = ...
```

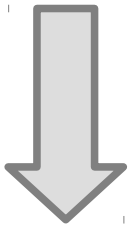


miniboxing

first: duplicate body

LDL for miniboxing

```
def tupled[T1, T2](t1: T1, t2: T2) = ...
```



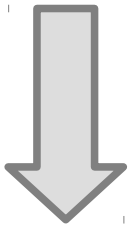
miniboxing

first: duplicate body

```
def tupled[T1, T2](t1: T1, t2: T2) = ...
```

LDL for miniboxing

```
def tupled[T1, T2](t1: T1, t2: T2) = ...
```



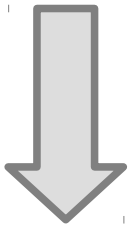
miniboxing

first: duplicate body

```
def tupled[T1, T2](t1: T1, t2: T2) = …
def tupled_ML[T1, T2](..., t1: T1, t2: T2) = …
```

LDL for miniboxing

```
def tupled[T1, T2](t1: T1, t2: T2) = ...
```



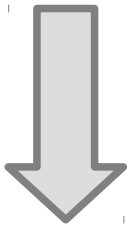
miniboxing

first: duplicate body

```
def tupled[T1, T2](t1: T1, t2: T2) = …
def tupled_ML[T1, T2](..., t1: T1, t2: T2) = …
def tupled_LM[T1, T2](..., t1: T1, t2: T2) = …
```

LDL for miniboxing

```
def tupled[T1, T2](t1: T1, t2: T2) = ...
```



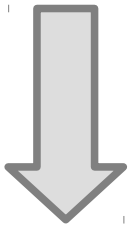
miniboxing

first: duplicate body

```
def tupled[T1, T2](t1: T1, t2: T2) = …
def tupled_ML[T1, T2](..., t1: T1, t2: T2) = …
def tupled_LM[T1, T2](..., t1: T1, t2: T2) = …
def tupled_MM[T1, T2](..., t1: T1, t2: T2) = …
```

LDL for miniboxing

```
def tupled[T1, T2](t1: T1, t2: T2) = ...
```



miniboxing

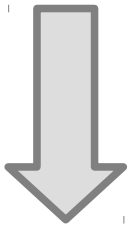
first: duplicate body

```
def tupled[T1, T2](t1: T1, t2: T2) = …
def tupled_ML[T1, T2](..., t1: T1, t2: T2) = …
def tupled_LM[T1, T2](..., t1: T1, t2: T2) = …
def tupled_MM[T1, T2](..., t1: T1, t2: T2) = …
```

second: adapt it

LDL for miniboxing

```
def tupled[T1, T2](t1: T1, t2: T2) = ...
```



miniboxing

first: duplicate body

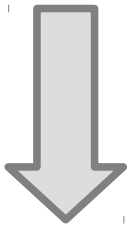
```
def tupled[T1, T2](t1: T1, t2: T2) = …
def tupled_ML[T1, T2](..., t1: @long T1, t2: T2) = …
def tupled_LM[T1, T2](..., t1: T1, t2: @long T2) = …
def tupled_MM[T1, T2](..., t1: @long T1, t2: @long T2) =
```

second: adapt it

* @long is used instead of @storage[Long]

LDL for miniboxing

```
def tupled[T1, T2](t1: T1, t2: T2) = ...
```



miniboxing

first: duplicate body

```
def tupled[T1, T2](t1: T1, t2: T2) = …
def tupled_ML[T1, T2](..., t1: @long T1, t2: T2) = …
def tupled_LM[T1, T2](..., t1: T1, t2: @long T2) = …
def tupled_MM[T1, T2](..., t1: @long T1, t2: @long T2) =
```

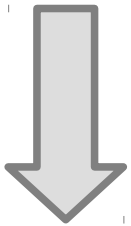
second: adapt it

using the transformation

* @long is used instead of @storage[Long]

LDL for miniboxing

```
def tupled[T1, T2](t1: T1, t2: T2) = ...
```



miniboxing

first: duplicate body

```
def tupled[T1, T2](t1: T1, t2: T2) = …
def tupled_ML[T1, T2](..., t1: @long T1, t2: T2) = …
def tupled_LM[T1, T2](..., t1: T1, t2: @long T2) = …
def tupled_MM[T1, T2](..., t1: @long T1, t2: @long T2) =
```

second: adapt it

using the transformation

body gets adapted**

* @long is used instead of @storage[Long]

Type-based LDL transformation

Use cases

Miniboxing

Value Classes*

Multi-stage Programming*

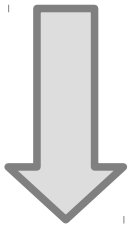
scala-miniboxing.org * early prototypes

LDL for value classes

```
def abs(c: @unboxed Complex): Double = ...
```

LDL for value classes

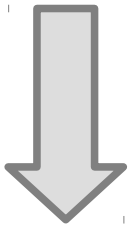
```
def abs(c: @unboxed Complex): Double = ...
```



value class plugin
(commit phase)

LDL for value classes

```
def abs(c: @unboxed Complex): Double = ...
```

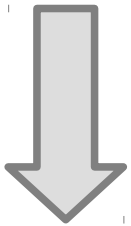


value class plugin
(commit phase)

```
def abs(c_re: Double, c_im: Double): Double = ...
```

LDL for value classes

```
def abs(c: @unboxed Complex): Double = ...
```

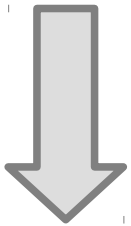


value class plugin
(commit phase)

```
def abs(c_re: Double, c_im: Double): Double = ...
```

LDL for value classes

```
def abs(c: @unboxed Complex): Double = ...
```

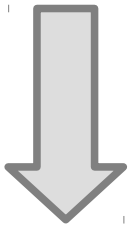


value class plugin
(commit phase)

```
def abs(c_re: Double, c_im: Double): Double = ...
```

LDL for value classes

```
def abs(c: @unboxed Complex): Double = ...
```



value class plugin
(commit phase)

```
def abs(c_re: Double, c_im: Double): Double = ...
```

I'm hiding **a lot of details** here.
But one could talk about this for an entire day

The one is @xeno-by!
He implemented this :)

Type-based LDL transformation

Use cases

- Miniboxing
- Value Classes*
- Multi-stage Programming*

scala-miniboxing.org * early prototypes

Type-based LDL transformation

Use cases

- Miniboxing
- Value Classes*
- Multi-stage Programming*

We won't go into it today, but see
github.com/miniboxing/staging-plugin

scala-miniboxing.org * early prototypes

What do **auto(un)boxing**, **specialization** and **value classes** have in common?

Why is this important?

The transformation

Benchmark

Conclusion

scala-miniboxing.org

What do **auto(un)boxing**, **specialization** and **value classes** have in common?

Why is this important?

The transformation

Benchmark

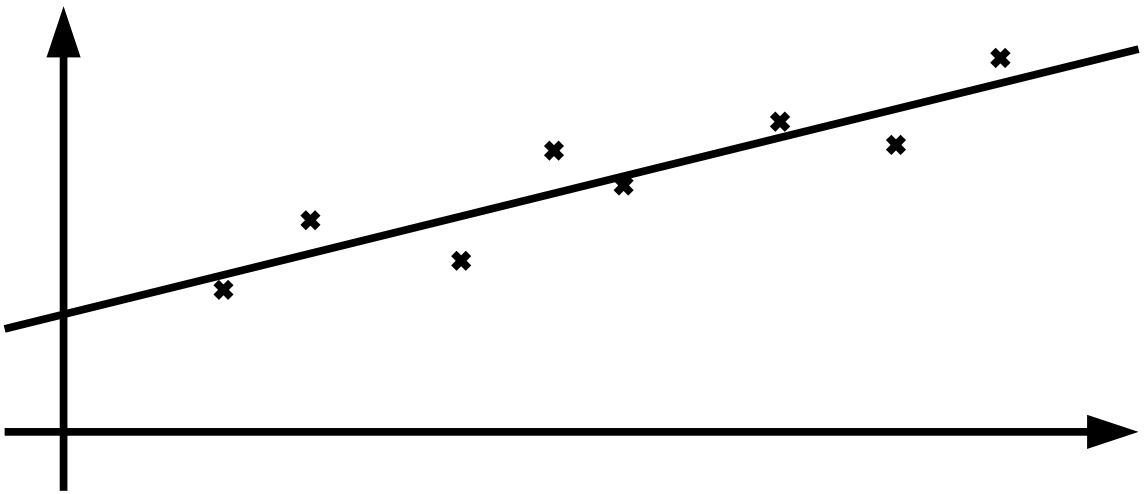
Conclusion

scala-miniboxing.org

Miniboxing Benchmarks

on the Scala library

- benchmark: Least Squares Method
  - using a mockup of **scala.collection.immutable.List**



Miniboxing Benchmarks
on the Scala library

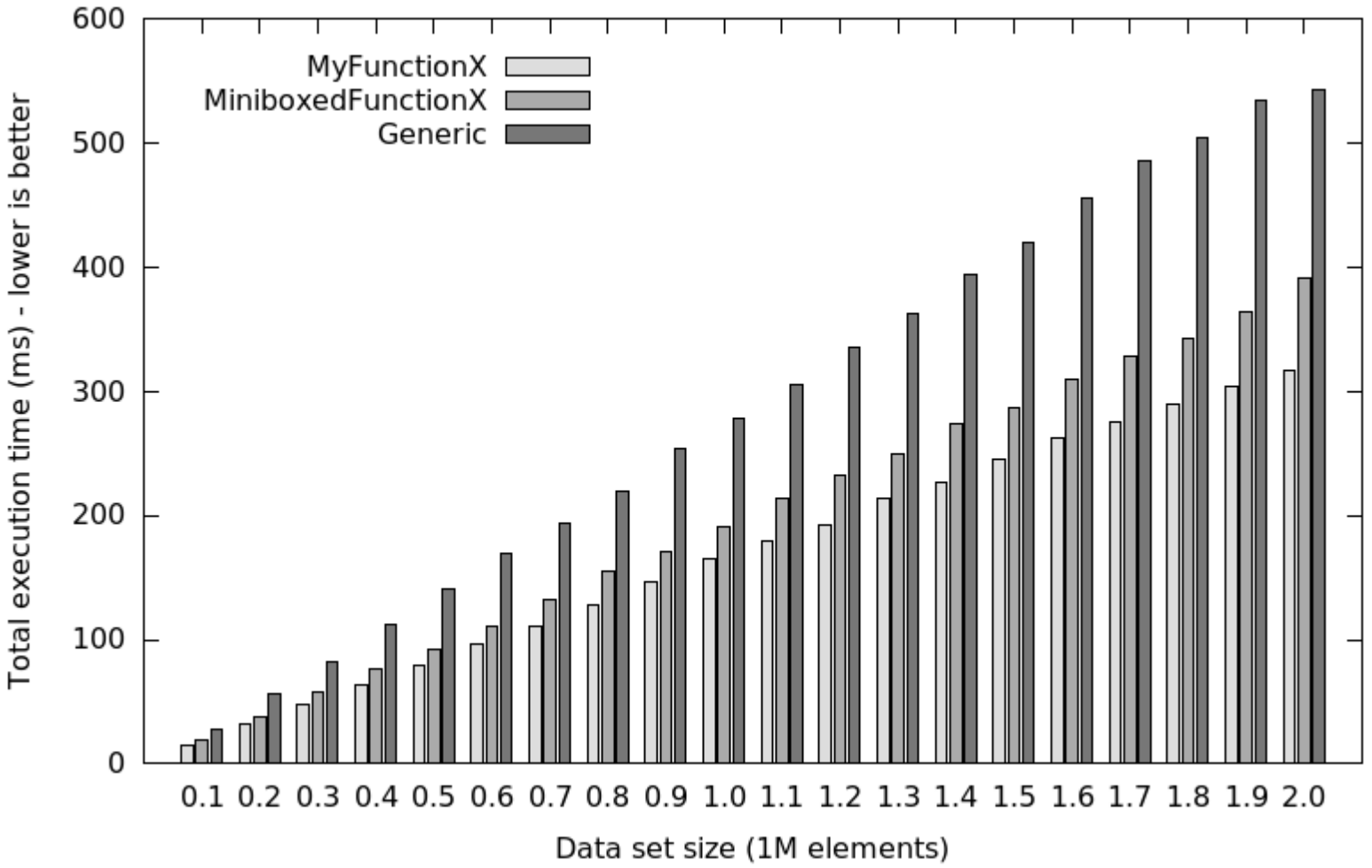
- work with Aymeric Genet (github: @MelodyLucid)
- mock-up of Scala linked List
  - Tuples
  - Traversable/TraversableLike
  - Iterator/Iterable/IterableLike
  - LinearSeqOptimized
  - Builder/CanBuildFrom
- FunctionX has a nice story

Miniboxing Benchmarks
on the Scala library

- **FunctionX** from the Scala library
  - specialized
  - no way to call from miniboxed code without boxing → slow
- **MiniboxedFunctionX ← benchmarked**
  - miniboxed, call without coercing
  - added automatically with another **LDL cycle**
- **MyFunctionX ← benchmarked**
  - miniboxed, call without coercing
  - added by hand to the library

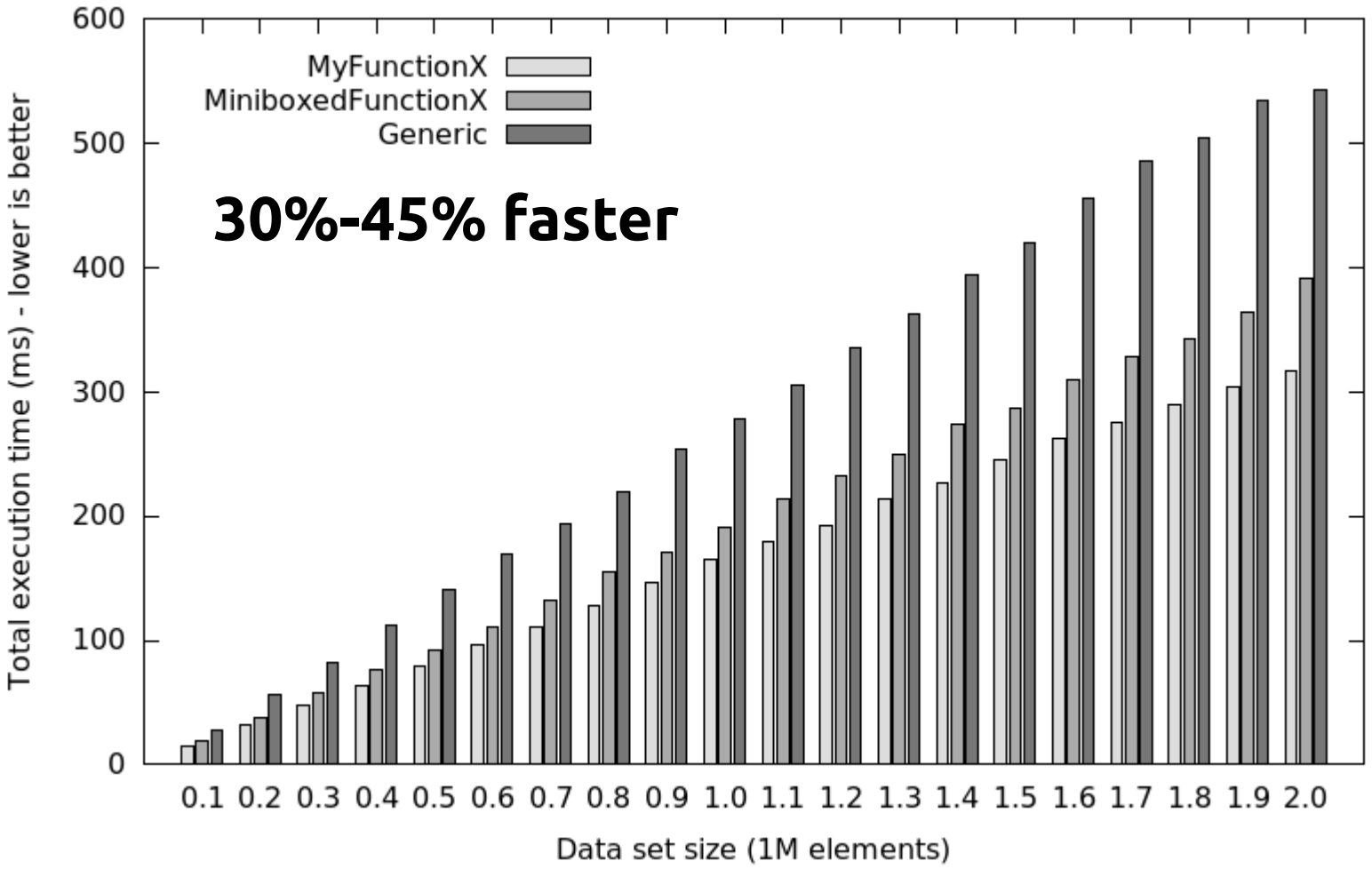
Miniboxing Benchmarks

on the Scala library



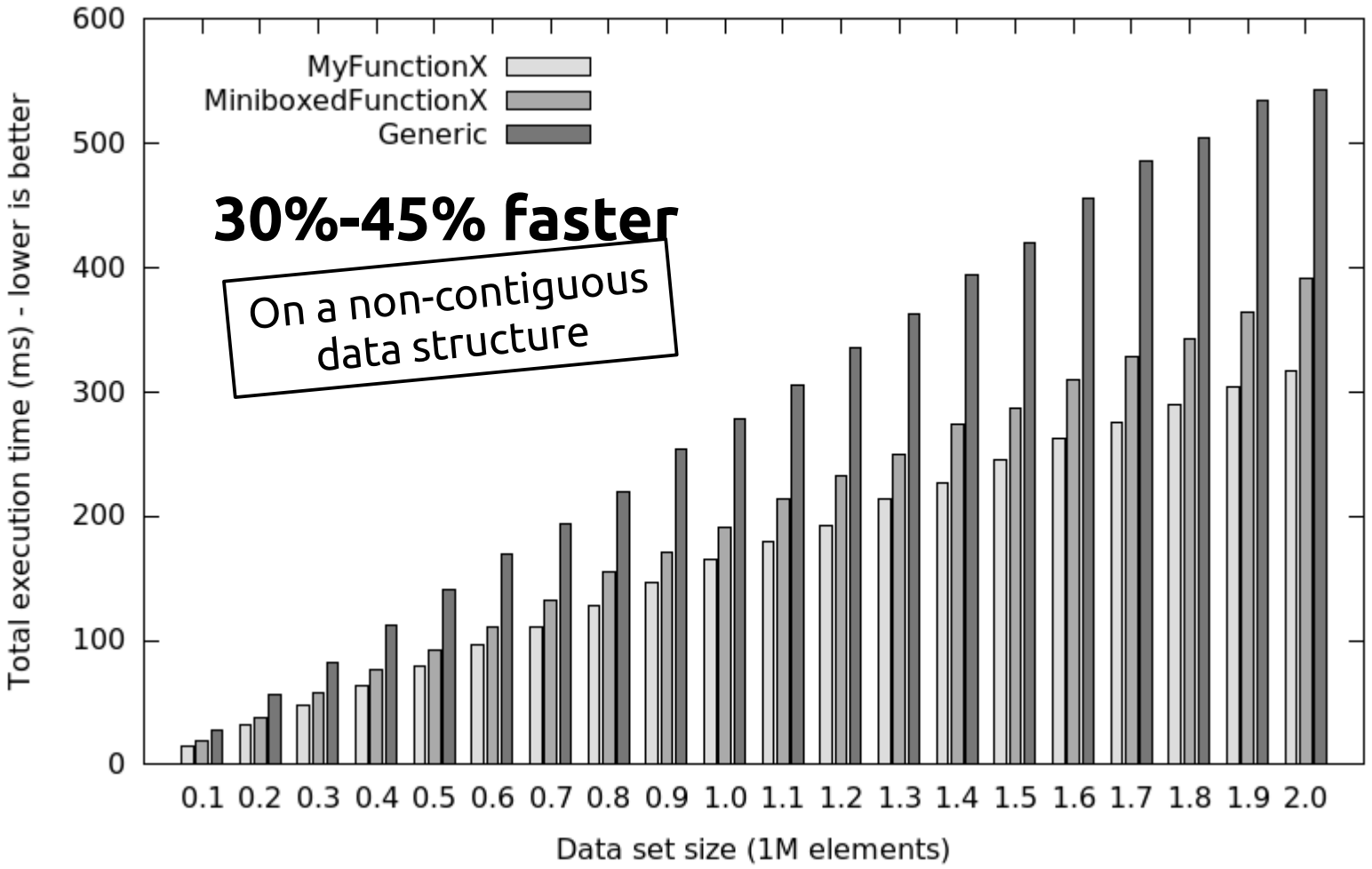
Miniboxing Benchmarks

on the Scala library



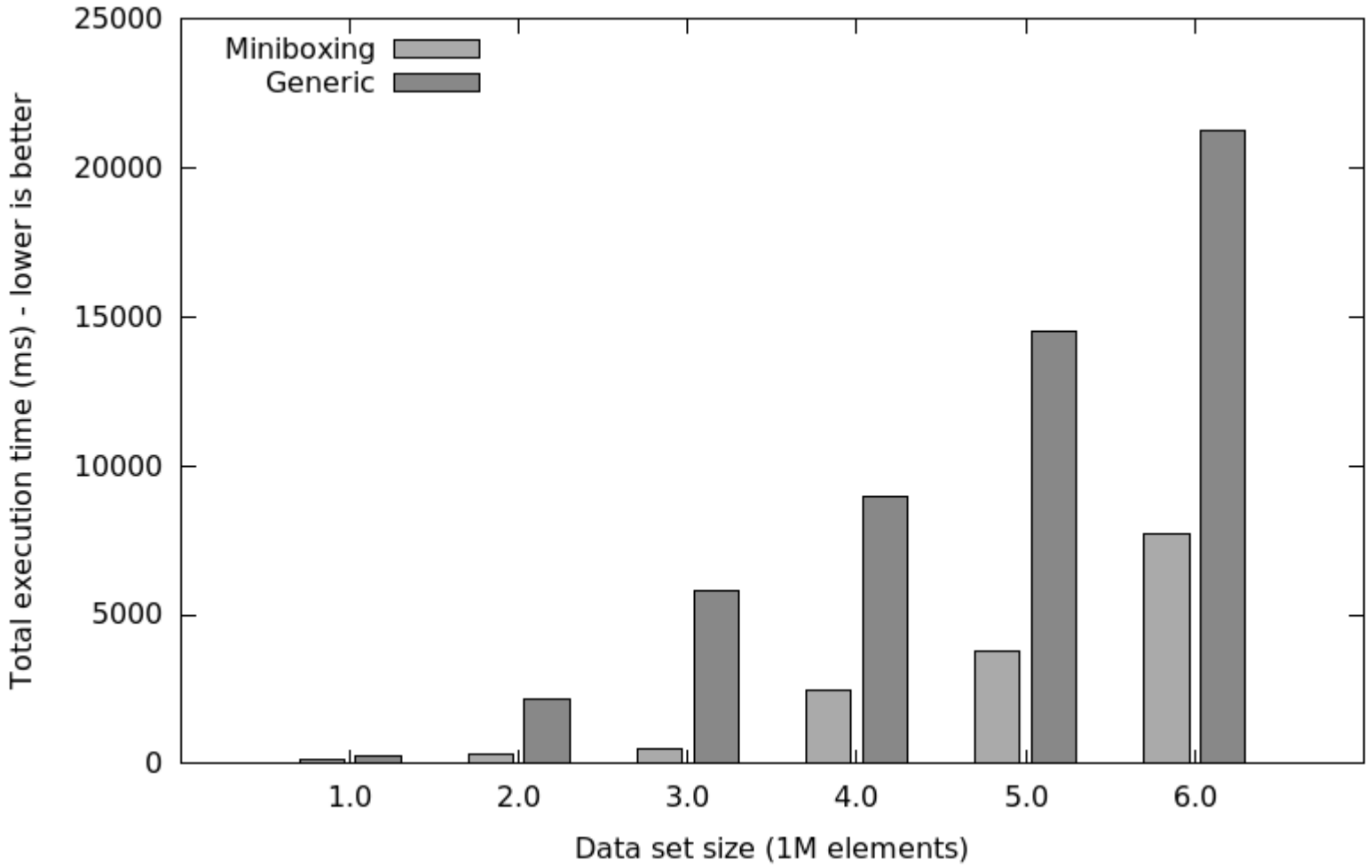
Miniboxing Benchmarks

on the Scala library



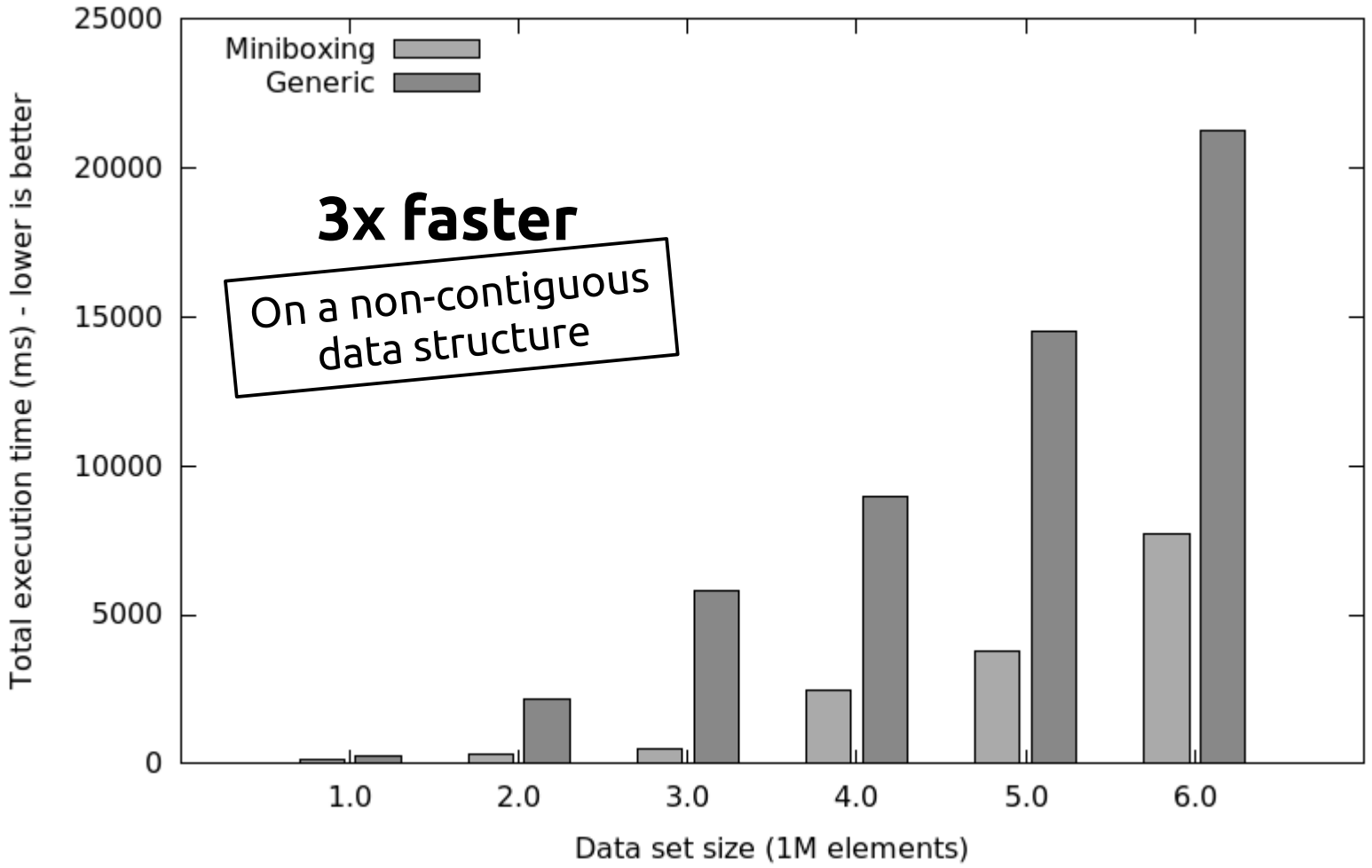
Miniboxing Benchmarks

on the Scala library (with GC)



Miniboxing Benchmarks

on the Scala library (with GC)



What do **auto(un)boxing**, **specialization** and **value classes** have in common?

Why is this important?

The transformation

Benchmark

Conclusion

scala-miniboxing.org

What do **auto(un)boxing**, **specialization** and **value classes** have in common?

Why is this important?

The transformation

Benchmark

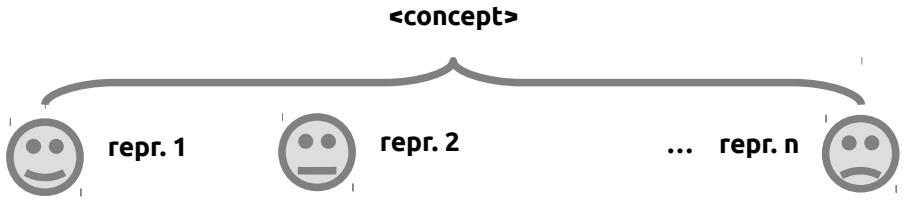
Conclusion

scala-miniboxing.org

Conclusion

LDL is a transformation

- that allows splitting a high-level concept
  - into multiple representations
  - in a **consistent** way (through coercions)
  - in a **selective** way (through annotations)
  - in an **optimal** way (coerce only if necessary)
- use cases covered
  - miniboxing
  - value classes
- what is your use case?



Credits and Thank you-s

- 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

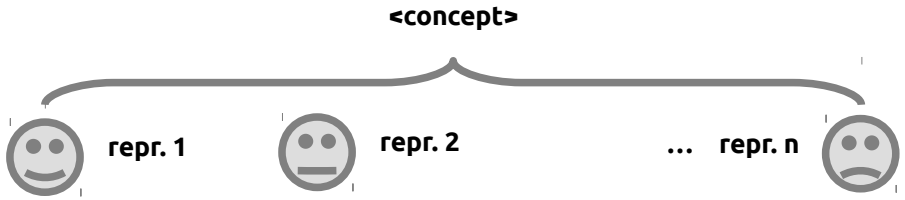
Special thanks to the Scala Community for their support!
(@StuHood, @vpatryshev and everyone else!)

scala-miniboxing.org

Conclusion

LDL is a transformation

- that allows splitting a high-level concept
  - into multiple representations
  - in a **consistent** way (through coercions)
  - in a **selective** way (through annotations)
  - in an **optimal** way (coerce only if necessary)
- use cases covered
  - miniboxing
  - value classes
- what is your use case?



ScalaTeam @ EPFL

scala-miniboxing.org

ScalaTeam @ EPFL

- **Dependent Object Types** calculus
  - core type system of the **dotty** compiler
  - Nada Amin/Tiark Rompf


https://github.com/TiarkRompf/minidot


https://github.com/lampepfl/dotty

ScalaTeam @ EPFL

- **YinYang** multi-stage macro-based frontend
  - replacement for **scala-virtualized**
  - Vojin Jovanovic/Sandro Stucki + others

https://github.com/vjovanov/yin-yang

ScalaTeam @ EPFL

- **Scala.js** backend
  - compiles Scala to JavaScript
  - Sébastien Doeraene/Tobias Schlatter + others

www http://www.scala-js.org/

ScalaTeam @ EPFL

- **Staged Parser-combinators**
  - fast parser combinators through staging
  - Manohar Jonnalagedda + others

https://github.com/manojo/experiments

ScalaTeam @ EPFL

- **Pickling** framework and **Spores**
  - support for **distributed programming**
  - Heather Miller/Philipp Haller + others


https://github.com/scala/pickling


https://github.com/heathermiller/spores

ScalaTeam @ EPFL

- **dotty**
  - experimental compiler based on DOT
  - Martin Odersky/Dmitry Petrashko/Samuel Grütter/Tobias Schlatter + others

https://github.com/lampepfl/dotty

ScalaTeam @ EPFL

- **scala.meta** metaprogramming support
  - Improved reflection, macros, and many more
  - Eugene Burmako/Denys Shabalin + others

http://scalameta.org/

ScalaTeam @ EPFL

- **miniboxing** specialization
  - LDL transformation
  - Vlad Ureche/Aymeric Genêt + others

www http://scala-miniboxing.org/

ScalaTeam @ EPFL

- **scaladyno** plugin
  - giving Scala a dynamic language look and feel
  - Cédric Bastin/Vlad Ureche

https://github.com/scaladyno/scaladyno-plugin

ScalaTeam @ EPFL

- **ScalaBlitz** optimization framework
  - macro-based collection optimization
  - Dmitry Petrashko/Aleksandar Prokopec

www http://scala-blitz.github.io/

ScalaTeam @ EPFL

- **Type debugger** for Scala
  - debugging aid for Scala type errors
  - Hubert Plociniczak

http://lampwww.epfl.ch/~plocinic/type-debugger-tutorial/

ScalaTeam @ EPFL

- **ScalaMeter** benchmarking framework
  - google caliper for scala
  - Aleksandar Prokopec

www http://scalameter.github.io/

ScalaTeam @ EPFL

- the **new scalac backend**
  - good performance gains
  - Miguel Garcia
    - **on the job market right now**

http://magarciaepfl.github.io/scala/

miguel.garcia@tuhh.de

Thank you!

scala-miniboxing.org

1st of August 2014
Scala Bay Area Meetup
Linkedin HQ, Mountain View