Miniboxing

scala-miniboxing.org

21st of October 2014
PDXScala Meetup
Portland, OR

Vlad URECHE

PhD student in the Scala Team @ EPFL

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

@VladUreche

@VladUreche

vlad.ureche@epfl.ch

scala-miniboxing.org

Miniboxing

scala-miniboxing.org

Miniboxing

Theory

Practice

Benchmarks

Conclusion

scala-miniboxing.org

Erased Generics

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

Erased Generics

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

scalac / javac

Erased Generics

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

scalac / javac

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

Erased Generics

identity(5)

Erased Generics

identity(5)

scalac / javac

Erased Generics

identity(5)

scalac / javac

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

Erased Generics

identity(5)

scalac / javac

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

Object representation

Erased Generics

identity(5)

scalac / javac

inflates heap requirements

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

Object representation

Erased Generics

identity(5)

scalac / javac

inflates heap requirements

produces garbage

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

Object representation

Erased Generics

identity(5)

scalac / javac

inflates heap requirements

produces garbage

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

Object representation

indirect (slow) access to the value

Erased Generics

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

Specialization

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

Specialization

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

specialization

Specialization

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

specialization

```
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 (10^2)

Specialization

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

specialization

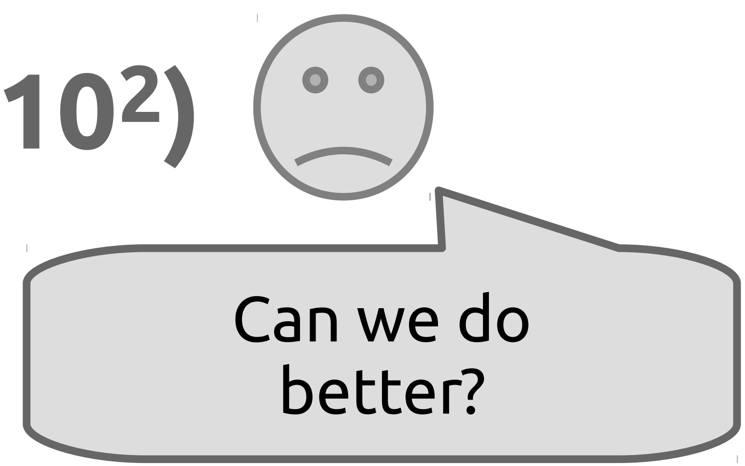
// 100 methods (10^2)

Specialization

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

specialization

```
// 100 methods (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)

$\Big\Downarrow$ miniboxing

Miniboxing

identity(3)

miniboxing

identity_M(..., int2minibox(3))

Miniboxing

identity(3)

miniboxing

identity_M(..., int2minibox(3))

The miniboxed
variant of identity

Miniboxing

identity(3)

miniboxing

identity_M(..., int2minibox(3))

The miniboxed variant of identity

Encoding the integer into a long integer

Miniboxing

$\boxed{\text{Theory}}$

Practice

Benchmarks

Conclusion

scala-miniboxing.org

The Theory of Miniboxing

- Class Transformation
- Late Data Layout
- ...

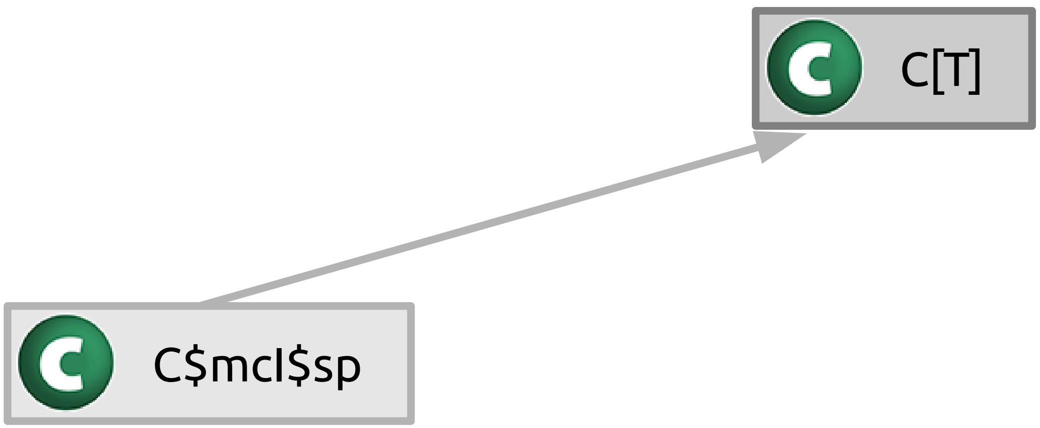
The Theory of Miniboxing

- Class Transformation
- Late Data Layout
- ...

```
class C[@specialized T](t: T)
```

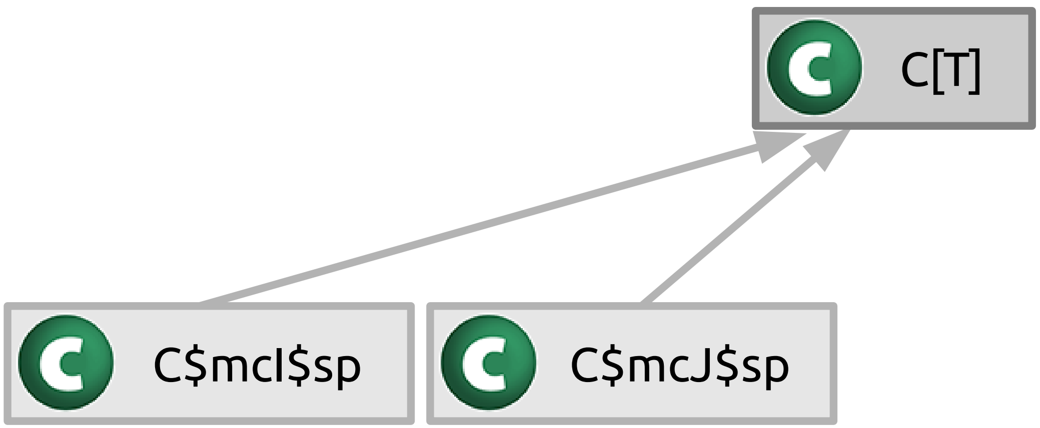
class C[@specialized T](t: T)

class C[@specialized T](t: T)

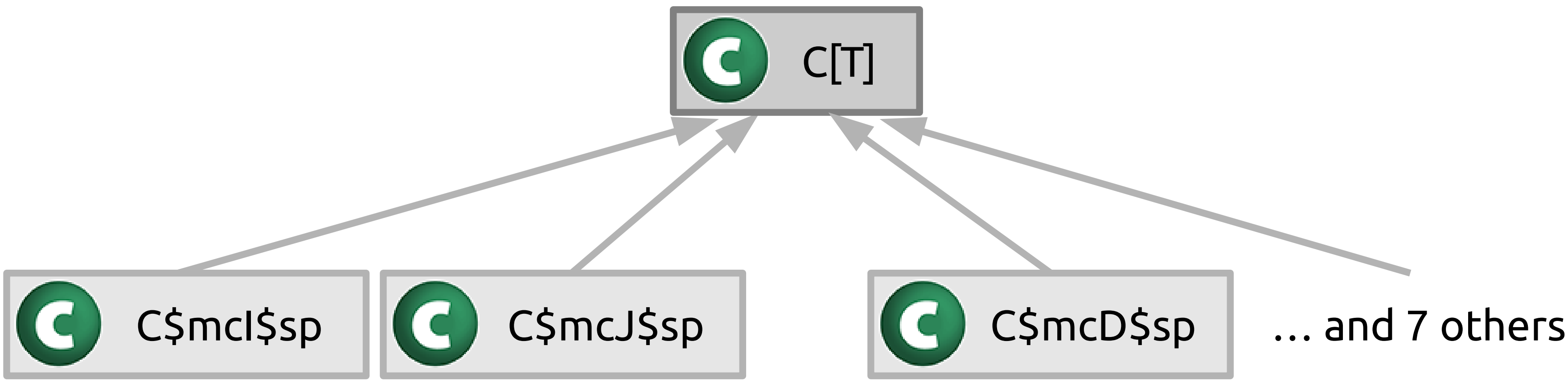


scala-miniboxing.org

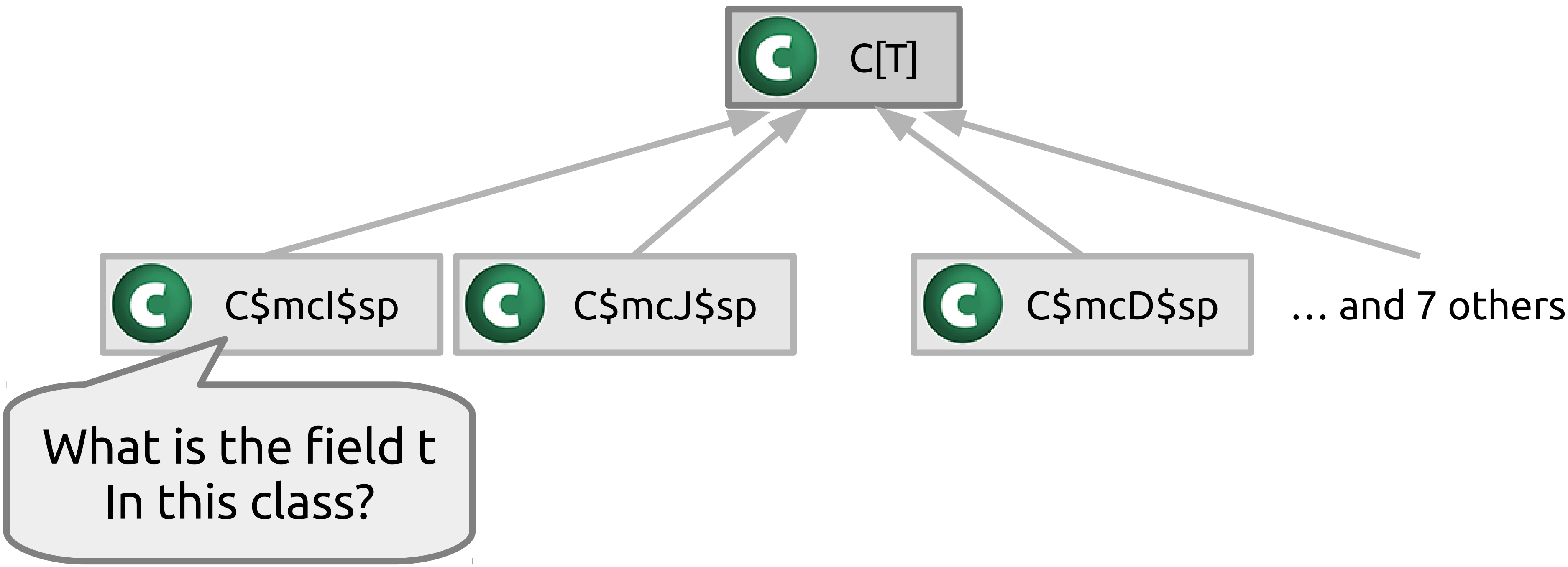
class C[@specialized T](t: T)



class C[@specialized T](t: T)

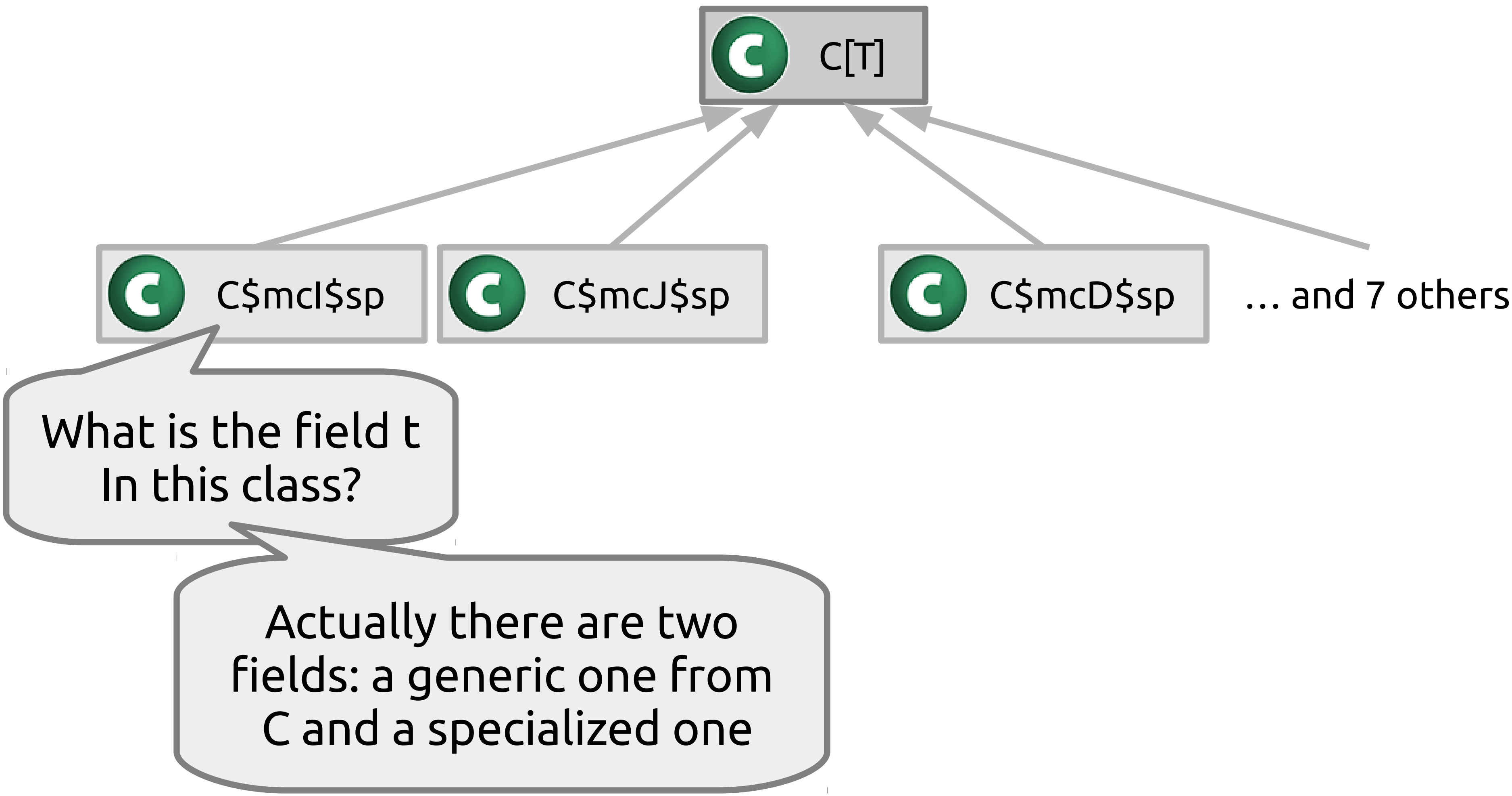


class C[@specialized T](t: T)

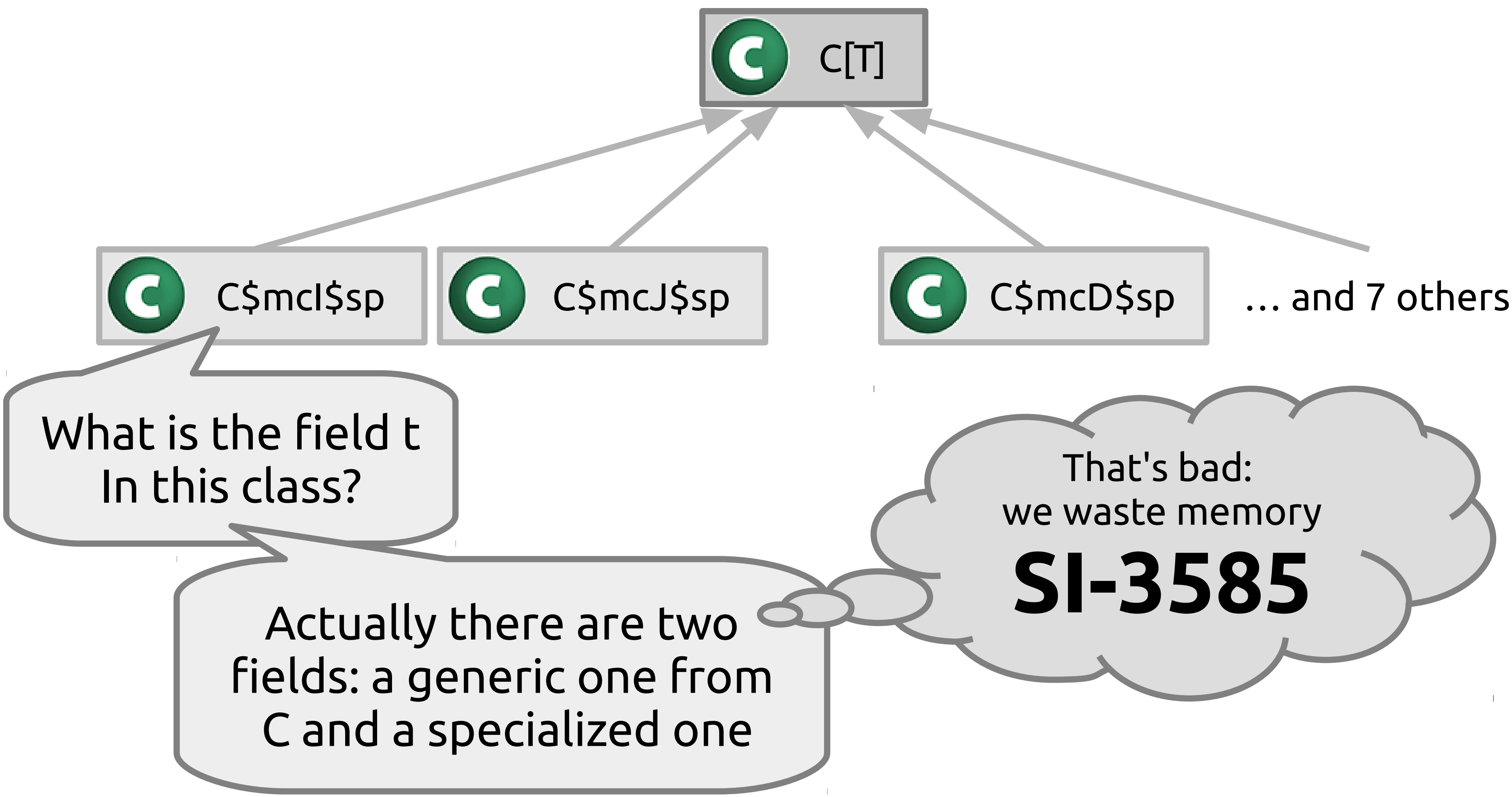


scala-miniboxing.org

class C[@specialized T](t: T)



class C[@specialized T](t: T)

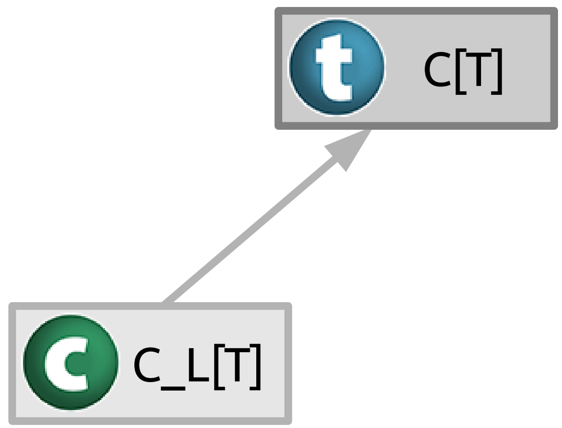


class C[@miniboxed T](t: T)

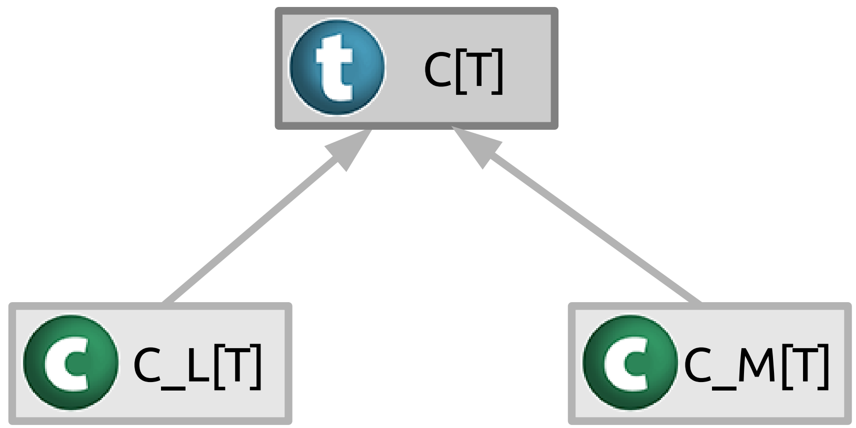
class C[@miniboxed T](t: T)

class C[@miniboxed T](t: T)

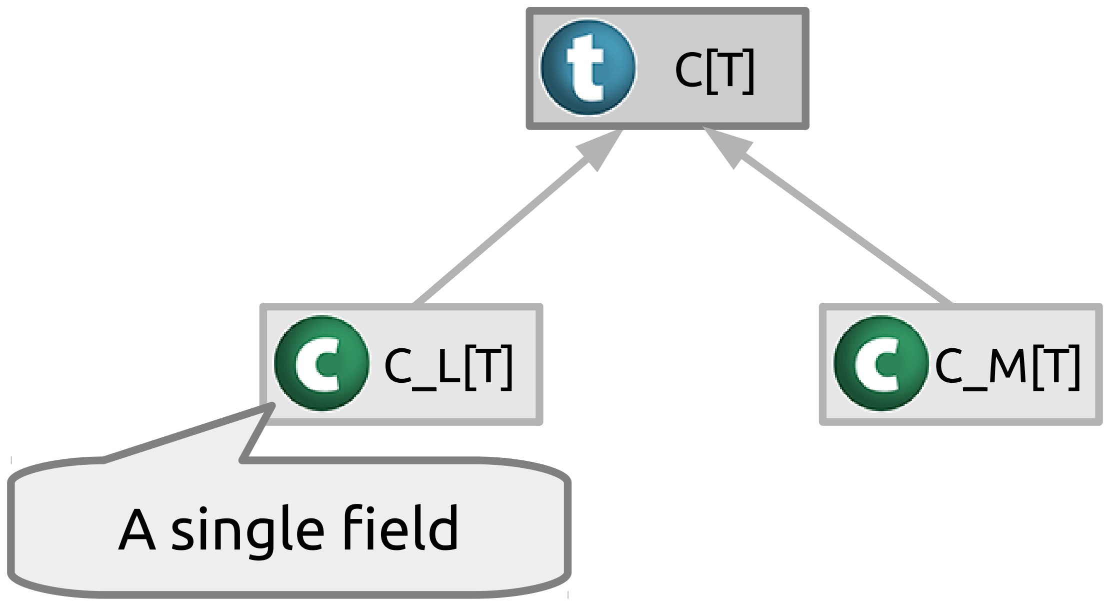
class C[@miniboxed T](t: T)



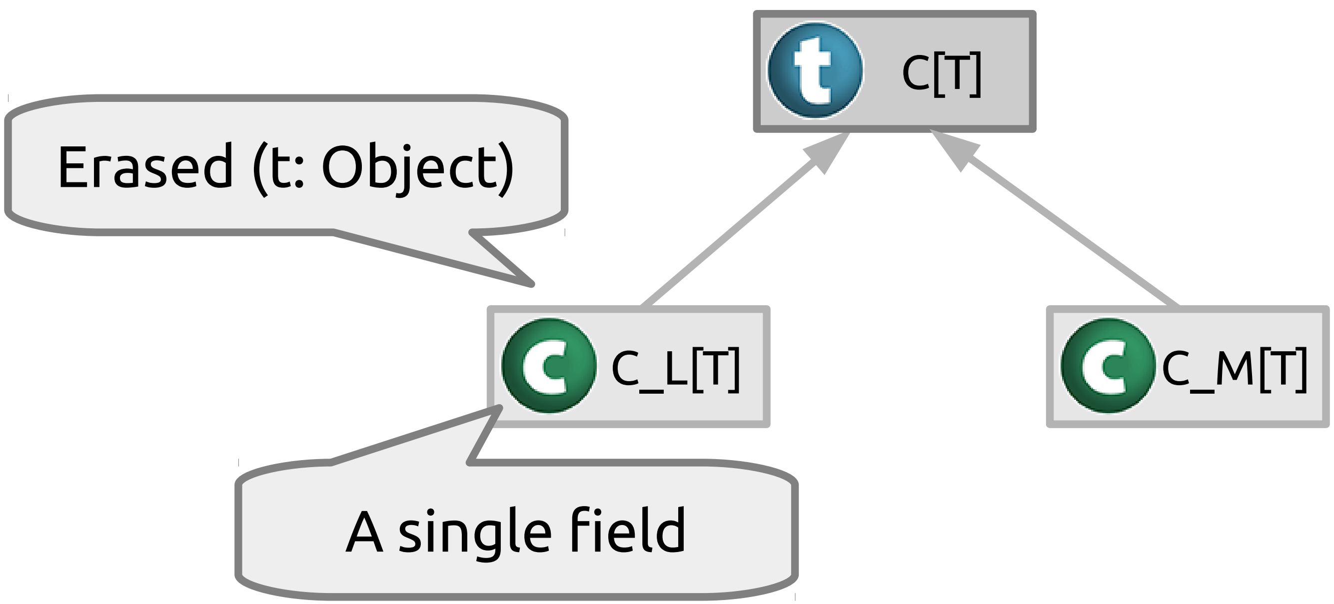
class C[@miniboxed T](t: T)



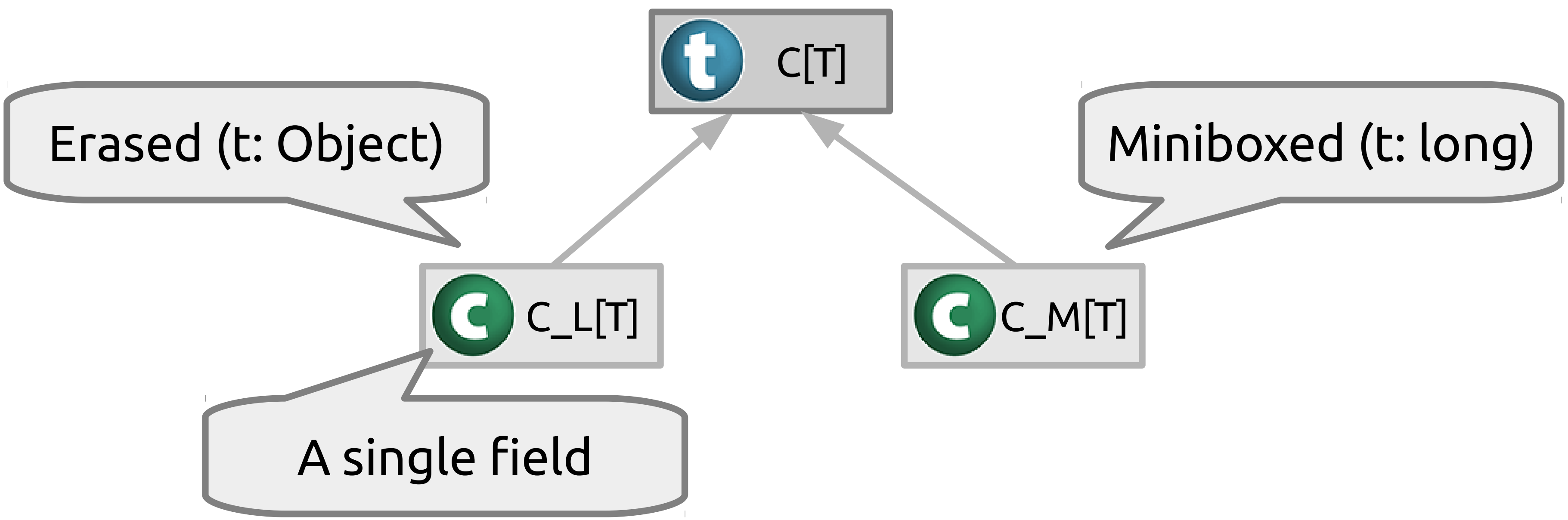
class C[@miniboxed T](t: T)



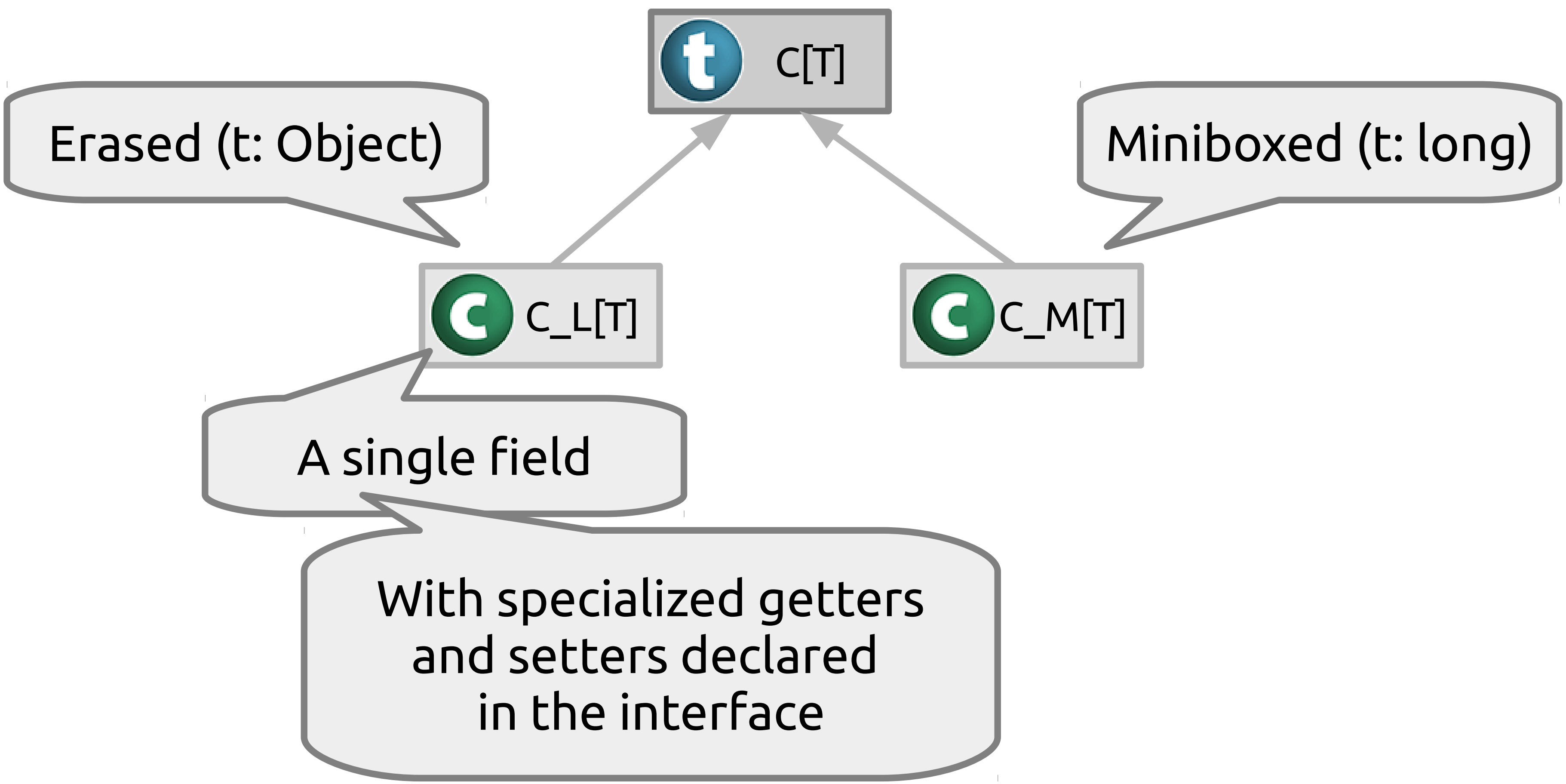
class C[@miniboxed T](t: T)



class C[@miniboxed T](t: T)



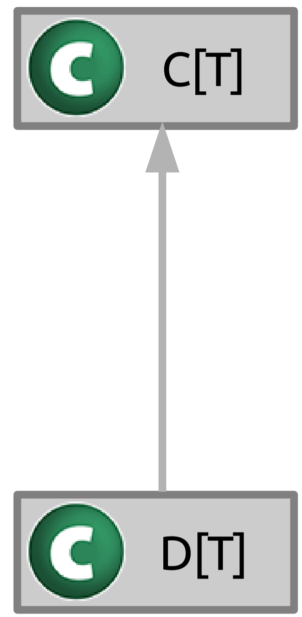
class C[@miniboxed T](t: T)



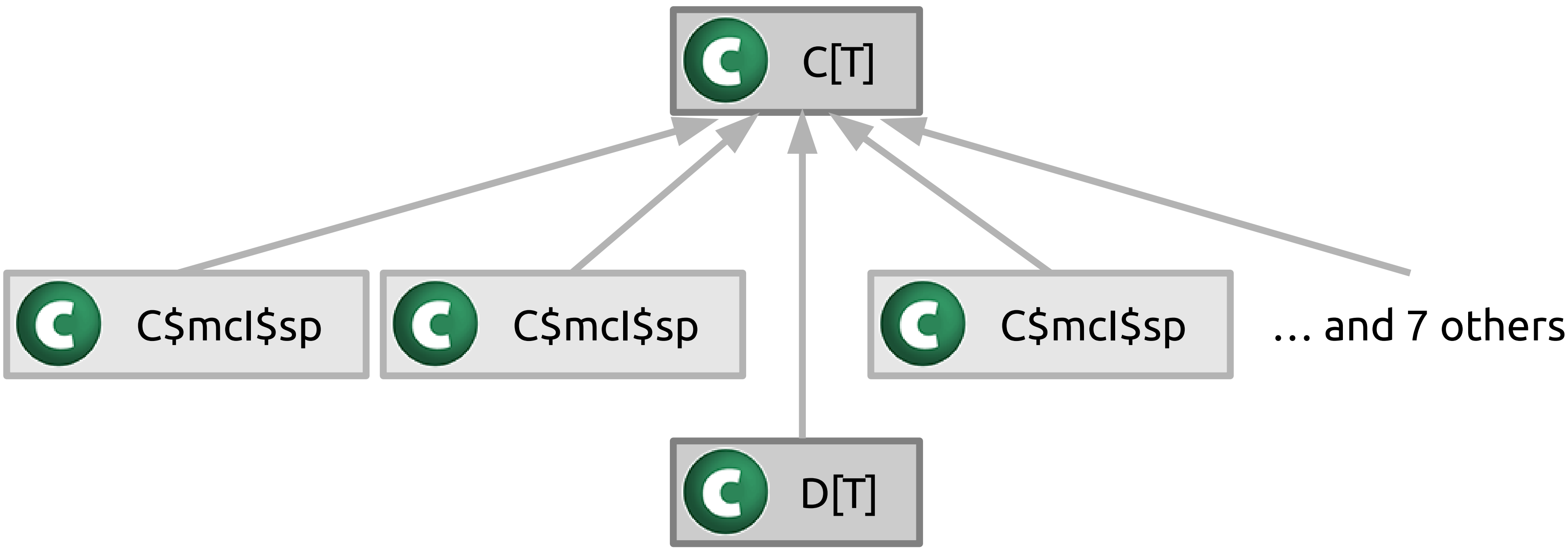
```
class D[@specialized T](t: T)
    extends C[T]
```

class D[@specialized T](t: T) extends C[T]

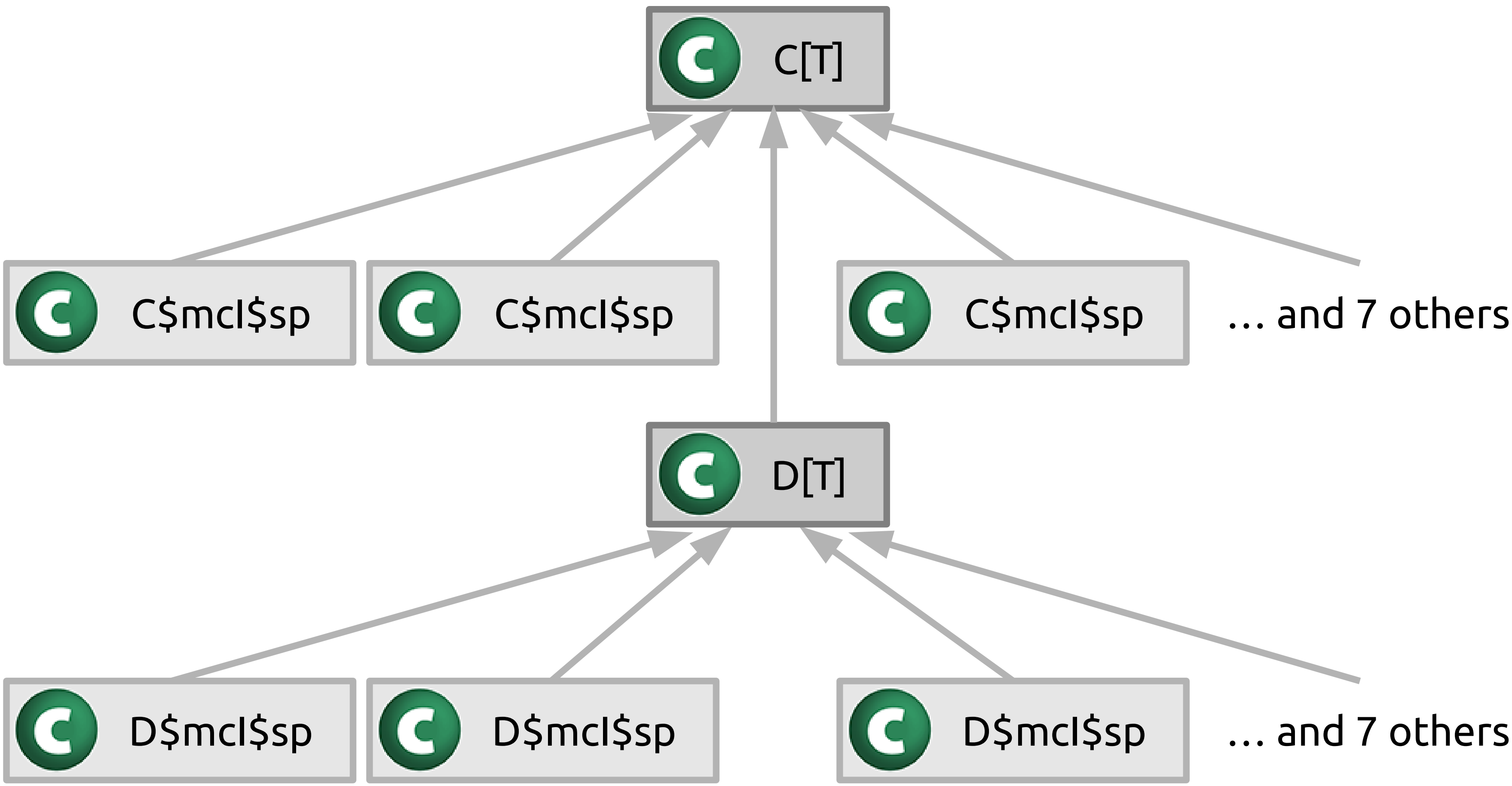
class D[@specialized T](t: T) extends C[T]



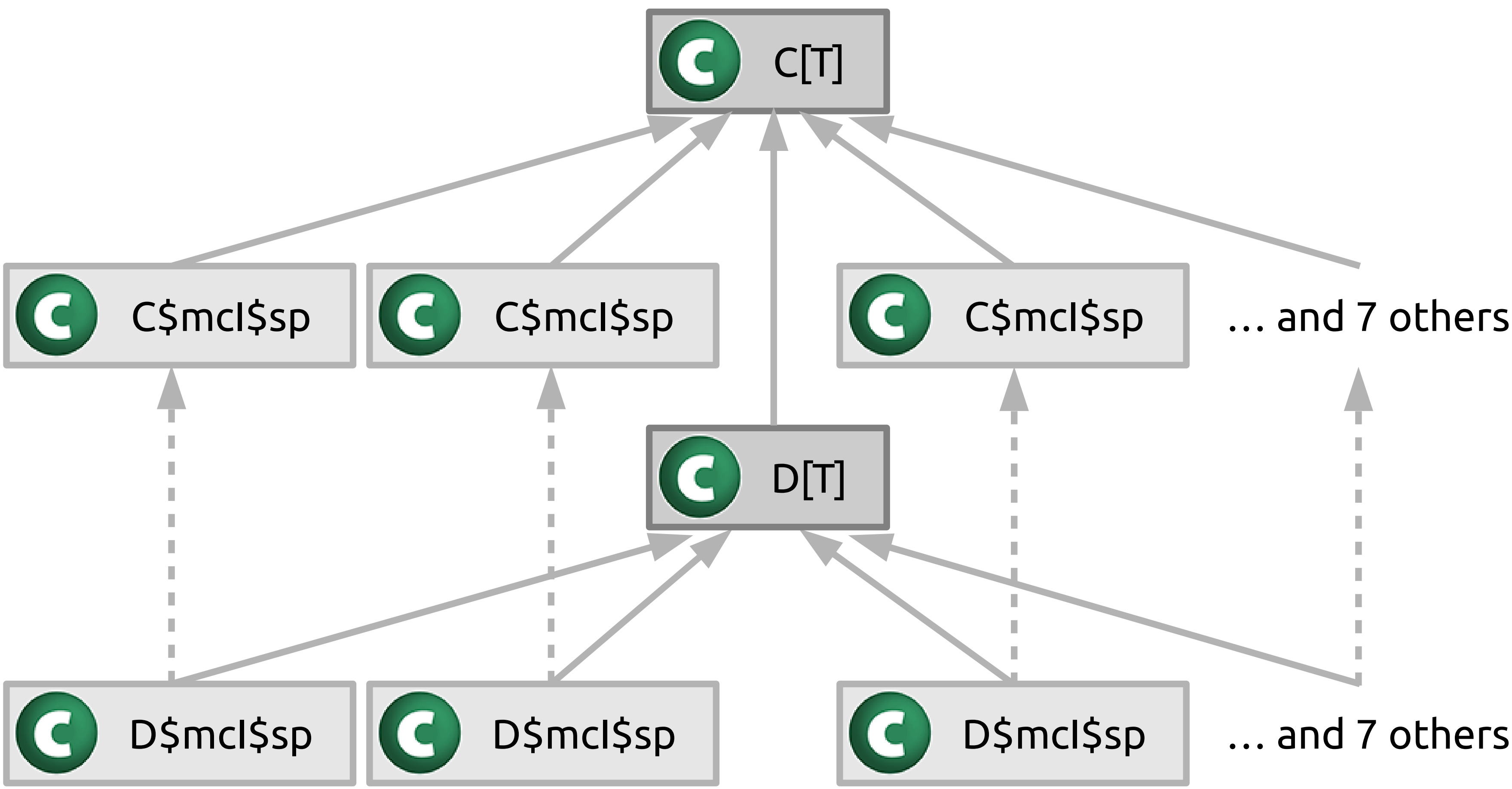
class D[@specialized T](t: T) extends C[T]



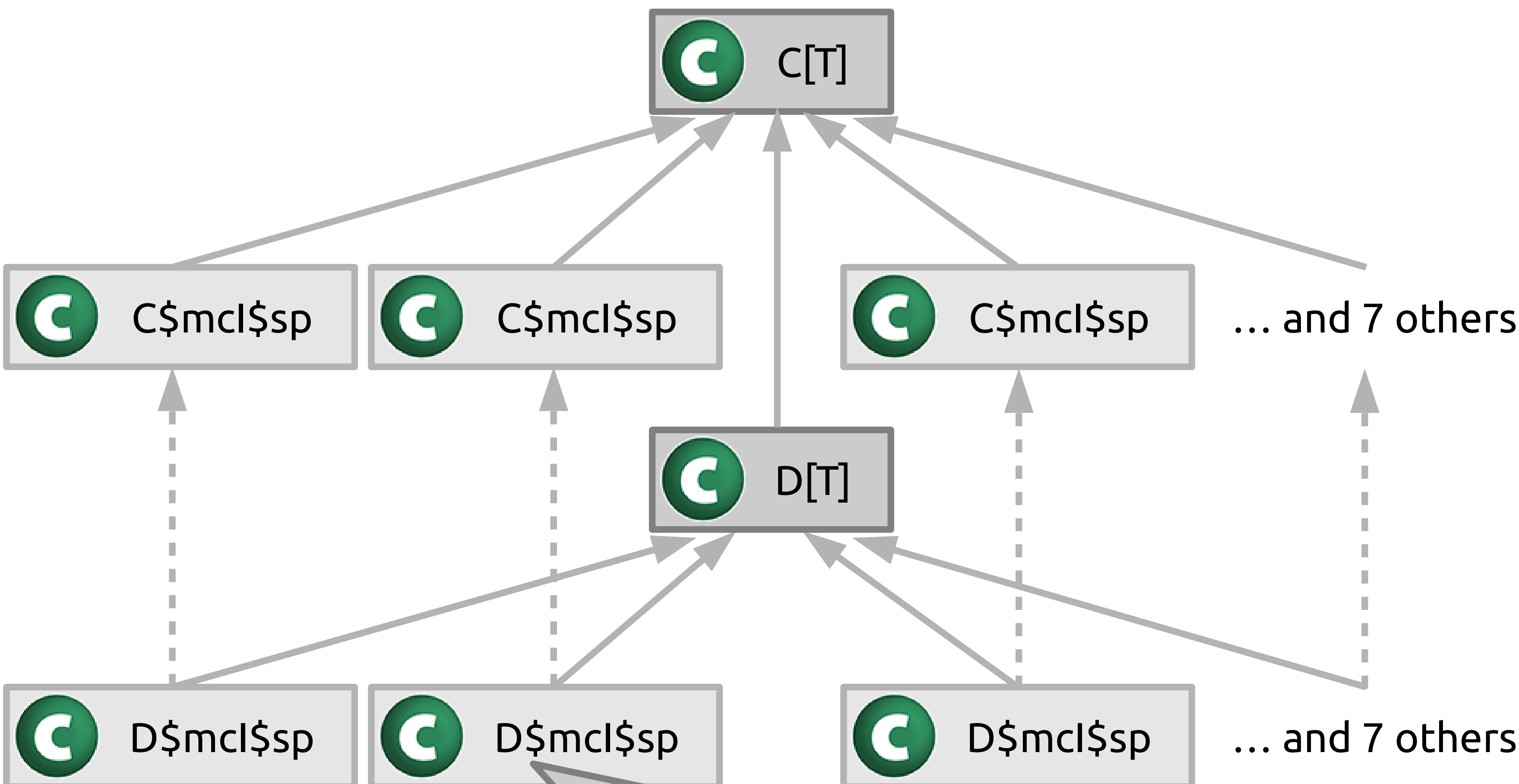
class D[@specialized T](t: T) extends C[T]



class D[@specialized T](t: T) extends C[T]

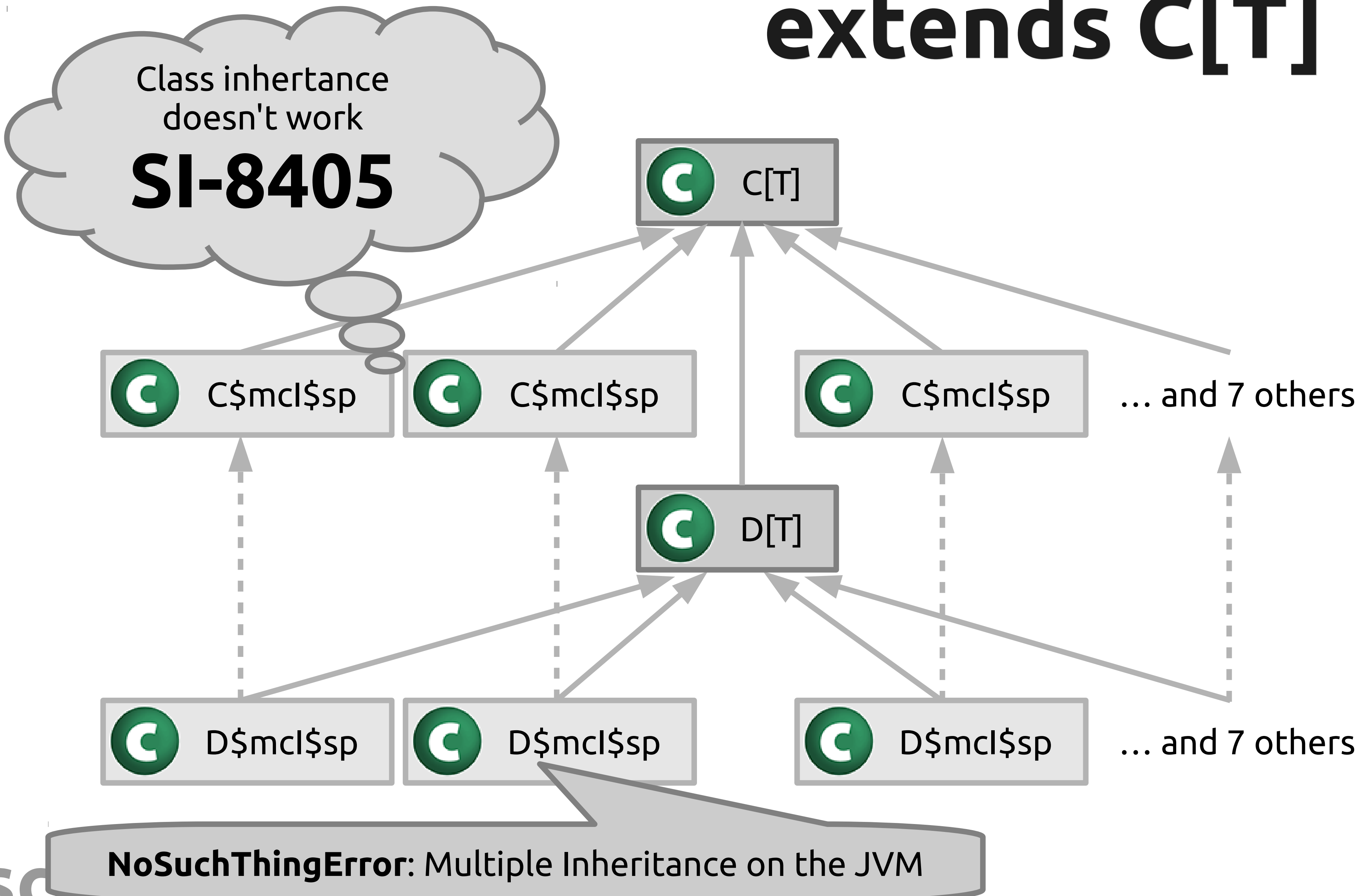


class D[@specialized T](t: T) extends C[T]



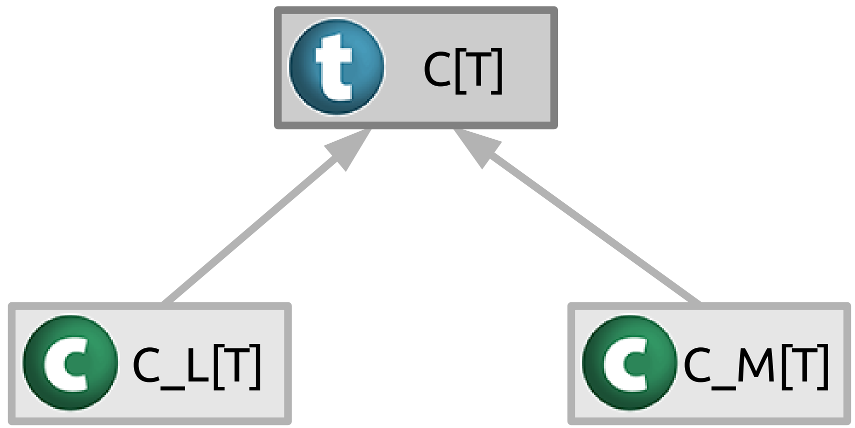
NoSuchThingError: Multiple Inheritance on the JVM

class D[@specialized T](t: T) extends C[T]

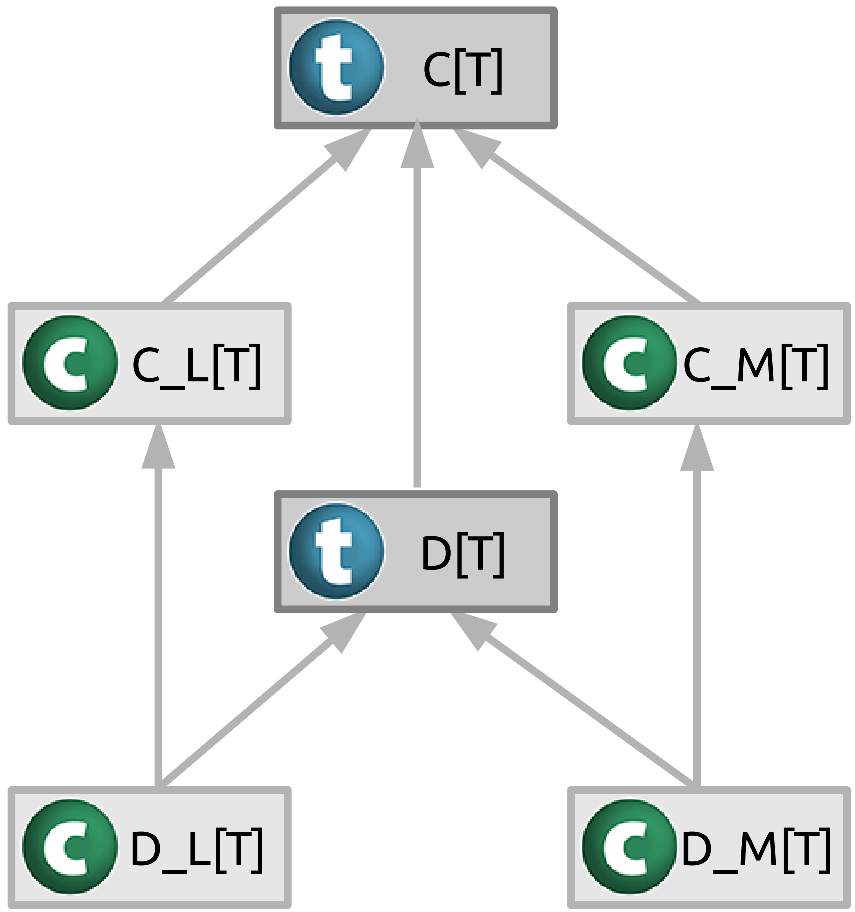


```
class D[@miniboxed T](t: T)
                    extends C[T]
```

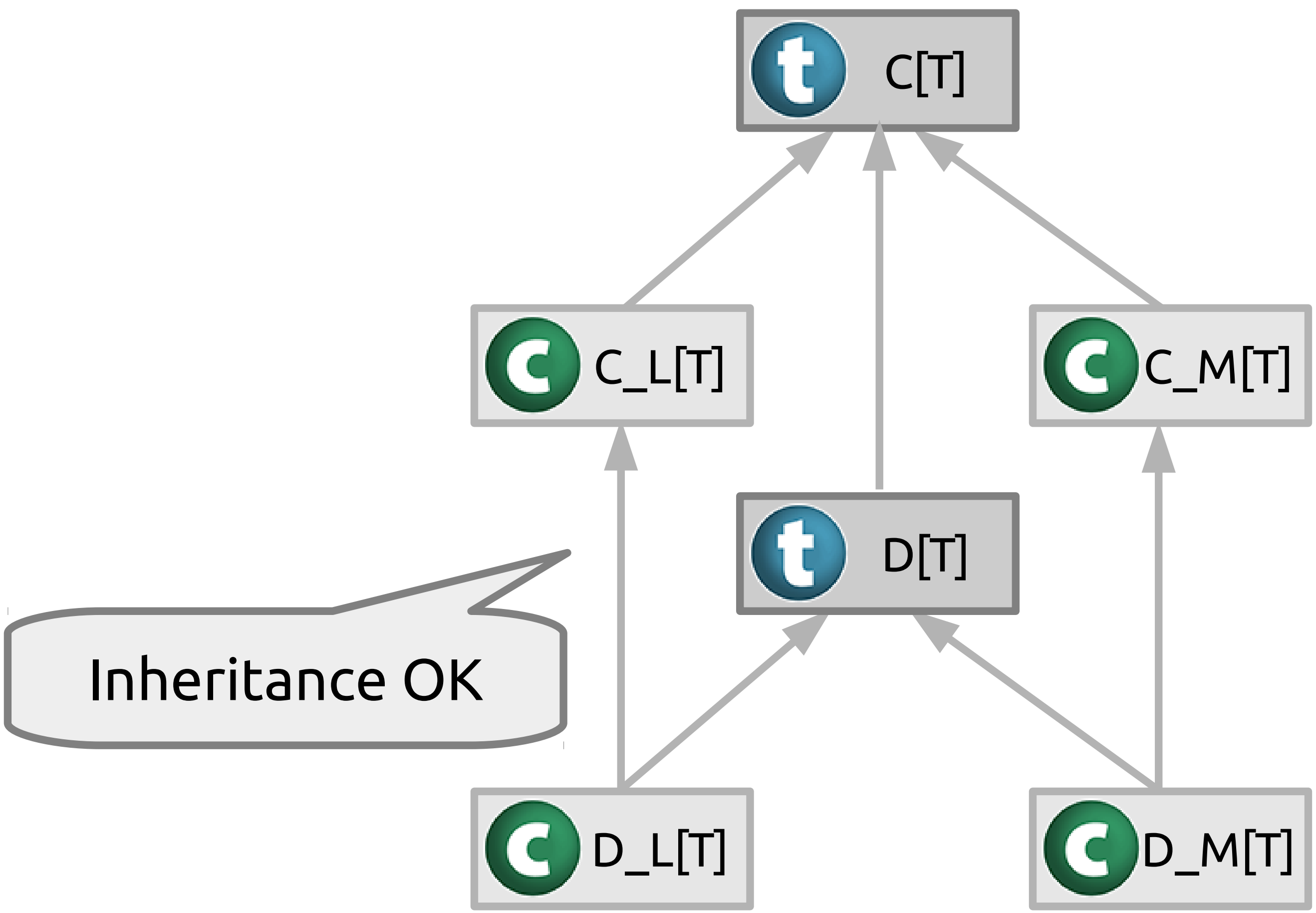
class D[@miniboxed T](t: T) extends C[T]



class D[@miniboxed T](t: T) extends C[T]



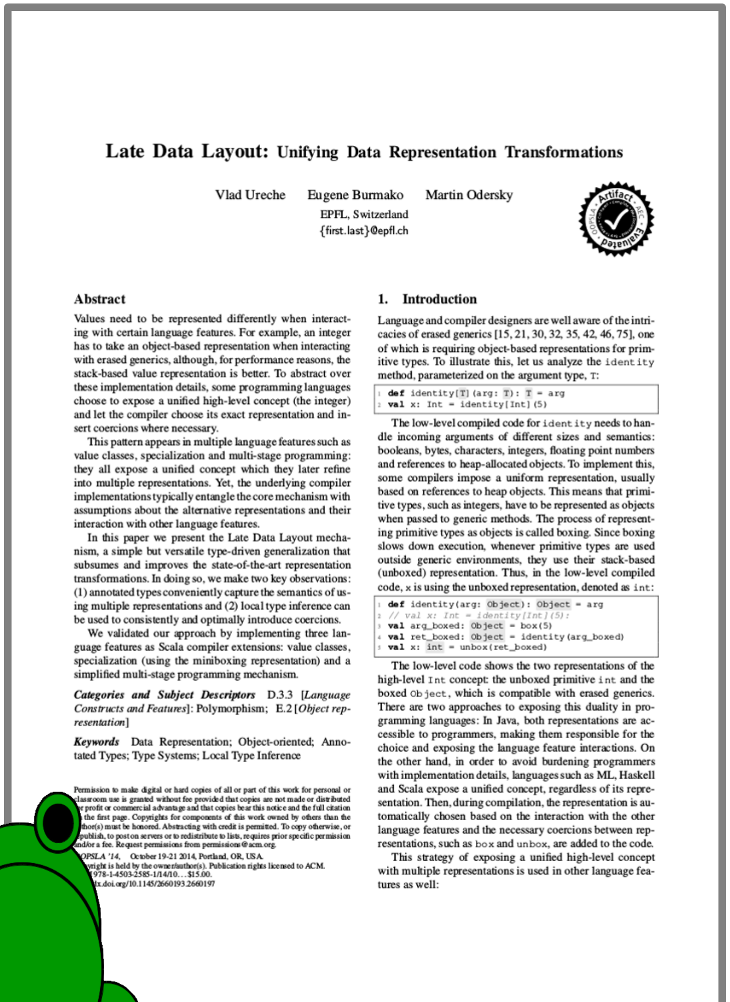
class D[@miniboxed T](t: T) extends C[T]



The Theory of Miniboxing

- Class Transformation
- Late Data Layout
- ...

Late Data Layout



Late Data Layout: Unifying Data Representation Transformations

Vlad Ureche Eugene Burmako Martin Odersky
EPFL, Switzerland
{first.last}@epfl.ch

Abstract

Values need to be represented differently when interacting with certain language features. For example, an integer has to take an object-based representation when interacting with erased generics, although, for performance reasons, the stack-based value representation is better. To abstract over these implementation details, some programming languages choose to expose a unified high-level concept (the integer) and let the compiler choose its exact representation and insert coercions where necessary.

This pattern appears in multiple language features such as value classes, specialization and multi-stage programming: they all expose a unified concept which they later refine into multiple representations. Yet, the underlying compiler implementations typically entangle the core mechanism with assumptions about the alternative representations and their interaction with other language features.

In this paper we present the Late Data Layout mechanism, a simple but versatile type-driven generalization that subsumes and improves the state-of-the-art representation transformations. In doing so, we make two key observations: (1) annotated types conveniently capture the semantics of using multiple representations and (2) local type inference can be used to consistently and optimally introduce coercions.

We validated our approach by implementing three language features as Scala compiler extensions: value classes, specialization (using the miniboxing representation) and a simplified multi-stage programming mechanism.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Polymorphism; E.2 [*Object representation*]

Keywords Data Representation; Object-oriented; Annotated Types; Type Systems; Local Type Inference

OPSLA '14, October 19-21 2014, Portland, OR, USA.

978-1-4503-2585-1/14/10...$15.00.
dx.doi.org/10.1145/2660193.2660197

1. Introduction

Language and compiler designers are well aware of the intricacies of erased generics [15, 21, 30, 32, 35, 42, 46, 75], one of which is requiring object-based representations for primitive types. To illustrate this, let us analyze the `identity` method, parameterized on the argument type, `T`:

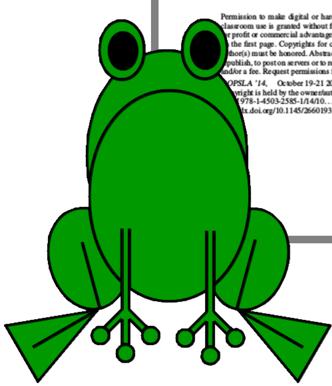
```
def identity[T](arg: T): T = arg
val x: Int = identity[Int](5)
```

The low-level compiled code for `identity` needs to handle incoming arguments of different sizes and semantics: booleans, bytes, characters, integers, floating point numbers and references to heap-allocated objects. To implement this, some compilers impose a uniform representation, usually based on references to heap objects. This means that primitive types, such as integers, have to be represented as objects when passed to generic methods. The process of representing primitive types as objects is called boxing. Since boxing slows down execution, whenever primitive types are used outside generic environments, they use their stack-based (unboxed) representation. Thus, in the low-level compiled code, `x` is using the unboxed representation, denoted as `int`:

```
def identity(arg: Object): Object = arg
// val x: Int = identity[Int](5):
val arg_boxed: Object = box(5)
val ret_boxed: Object = identity(arg_boxed)
val x: int = unbox(ret_boxed)
```

The low-level code shows the two representations of the high-level `Int` concept: the unboxed primitive `int` and the boxed `Object`, which is compatible with erased generics. There are two approaches to exposing this duality in programming languages: In Java, both representations are accessible to programmers, making them responsible for the choice and exposing the language feature interactions. On the other hand, in order to avoid burdening programmers with implementation details, languages such as ML, Haskell and Scala expose a unified concept, regardless of its representation. Then, during compilation, the representation is automatically chosen based on the interaction with the other language features and the necessary coercions between representations, such as `box` and `unbox`, are added to the code.

This strategy of exposing a unified high-level concept with multiple representations is used in other language features as well:



- theory of the transformation
  - how **T** becomes **long**
- generalizes miniboxing
  - value classes
  - staging support
  - function representation
  - etc
- **scala-ldl.org**

The Theory of Miniboxing

- Class Transformation
- Late Data Layout
- ...

Miniboxing

Theory

$\boxed{\text{Practice}}$

Benchmarks

Conclusion

scala-miniboxing.org

Miniboxing in Practice

- Documentation
- Quirks
- Live Demo

scala-miniboxing.org

Miniboxing in Practice

- Documentation
- Quirks
- Live Demo

Start here
Start Page
Introduction
Full Tutorial
Benchmarks

Use it
Sbt config
Command line
Source code
Issue tracker
License

Experiment
Sbt project
Linked List
Functions
Reverse

Discuss
Mailing List
Twitter
News

Introduction

Miniboxing is a compilation scheme that improves the performance of generics in the Scala programming language. The miniboxing transformation is generic enough to be potentially useful for any statically typed language running on one of the Java Virtual Machines, such as Managed X10, Kotlin or Ceylon.

We'll start by following what happens to a generic class, as it gets compiled. Let's take `class C` as an example:

```
class C[T](t: T)
```

After compiling this class to Java Virtual Machine bytecode, under the erasure transformation one would get bytecode which roughly corresponds to:

```
class C {
   var t: Object = _         // field
   def C(t: Object): C = {   // constructor
     this.t = t
   }
 }
```

As you can see, erasure transformed `t` from a generic value into a pointer to a heap object. While this is perfectly suited for storing a string or another class inside `class C`, it becomes suboptimal when dealing with primitive value types, such as booleans, bytes, integers and floating point numbers.

The reason it's suboptimal is because primitive value types are not heap objects but values passed on the stack, so under the Java Virtual Machine it is common to encode them as heap objects, a process known as boxing.

scala-miniboxing.org

La… value classes, staging and many more transformations

Start here

Start Page
Introduction
Full Tutorial
Benchmarks

Use it

Sbt config
Command line
Source code
Issue tracker
License

Experiment

Sbt project
Linked List
Functions
Reverse

Discuss

Mailing List
Twitter
News

Introduction

Miniboxing is a compilation scheme that improves the performance of generics in the Scala programming language. The miniboxing transformation is generic enough to be potentially useful for any statically typed language running on one of the Java Virtual Machines, such as Managed X10, Kotlin or Ceylon.

We'll start by following what happens to a generic class, as it gets compiled. Let's take `class C` as an example:

```
class C[T](t: T)
```

After compiling this class to Java Virtual Machine bytecode, under the erasure transformation one would get bytecode which roughly corresponds to:

```
class C {
   var t: Object = _          // field
   def C(t: Object): C = {    // constructor
     this.t = t
   }
 }
```

As you can see, erasure transformed `t` from a generic value into a pointer to a heap object. While this is perfectly suited for storing a string or another class inside `class C`, it becomes suboptimal when dealing with primitive value types, such as booleans, bytes, integers and floating point numbers.

The reason it's suboptimal is because primitive value types are not heap objects but values passed on the stack, so under the Java Virtual Machine it is common to encode them as heap objects, a process known as boxing.

scala-miniboxing.org

La… value classes, staging and many more transformations

Start here

- Start Page
- Introduction
- Full Tutorial
- Benchmarks

Use it

- Sbt config
- Command line
- Source code
- Issue tracker
- License

Experiment

- Sbt project
- Linked List
- Functions
- Reverse

Discuss

- Mailing List
- Twitter
- News

Introduction

Miniboxing is a compilation scheme that improves the performance of generics in the Scala programming language. The miniboxing transformation is generic enough to be potentially useful for any statically typed language running on one of the Java Virtual Machines, such as Managed X10, Kotlin or Ceylon.

We'll start by following what happens to a generic class, as it gets compiled. Let's take `class C` as an example:

```
class C[T](t: T)
```

After compiling this class to Java Virtual Machine bytecode, under the erasure transformation one would get bytecode which roughly corresponds to:

```
class C {
   var t: Object = _          // field
   def C(t: Object): C = {    // constructor
     this.t = t
   }
 }
```

As you can see, erasure transformed `t` from a generic value into a pointer to a heap object. While this is perfectly suited for storing a string or another class inside `class C`, it becomes suboptimal when dealing with primitive value types, such as booleans, bytes, integers and floating point numbers.

The reason it's suboptimal is because primitive value types are not heap objects but values passed on the stack, so under the Java Virtual Machine it is common to encode them as heap objects, a process known as boxing.

scala-miniboxing.org

tutorials

value classes, staging and many more transformations

Introduction

n scheme that improves the performance of generics in the Scala programming transformation is generic enough to be potentially useful for any statically typed f the Java Virtual Machines, such as Managed X10, Kotlin or Ceylon.

We'll start by following what happens to a generic class, as it gets compiled. Let's take `class C` as an example:

```
class C[T](t: T)
```

After compiling this class to Java Virtual Machine bytecode, under the erasure transformation one would get bytecode which roughly corresponds to:

```
class C {
   var t: Object = _          // field
   def C(t: Object): C = {    // constructor
     this.t = t
   }
 }
```

As you can see, erasure transformed `t` from a generic value into a pointer to a heap object. While this is perfectly suited for storing a string or another class inside `class C`, it becomes suboptimal when dealing with primitive value types, such as booleans, bytes, integers and floating point numbers.

The reason it's suboptimal is because primitive value types are not heap objects but values passed on the stack, so under the Java Virtual Machine it is common to encode them as heap objects, a process known as boxing.

Start here

- Start Page
- Introduction
- Full Tutorial
- Benchmarks

Use it

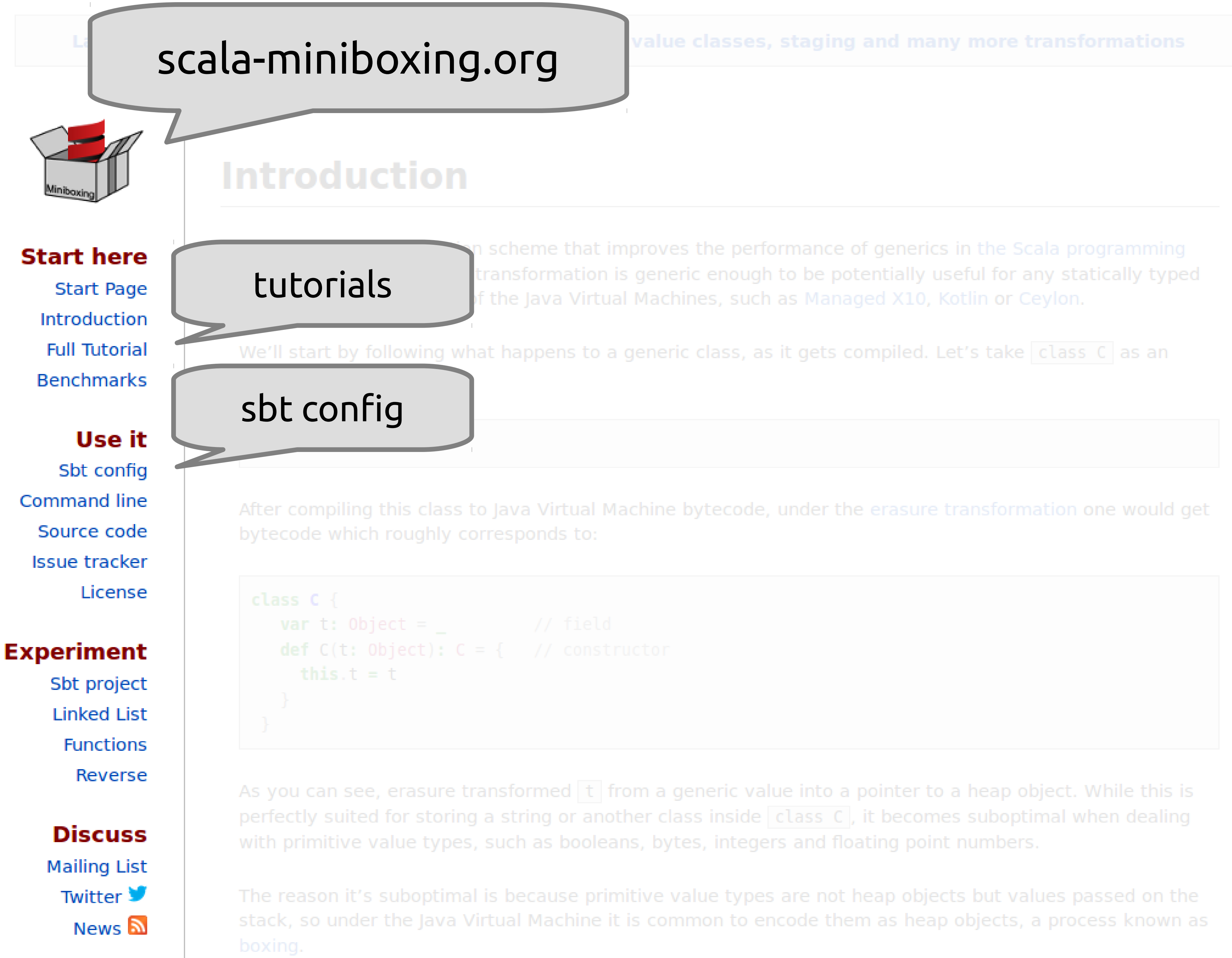
- Sbt config
- Command line
- Source code
- Issue tracker
- License

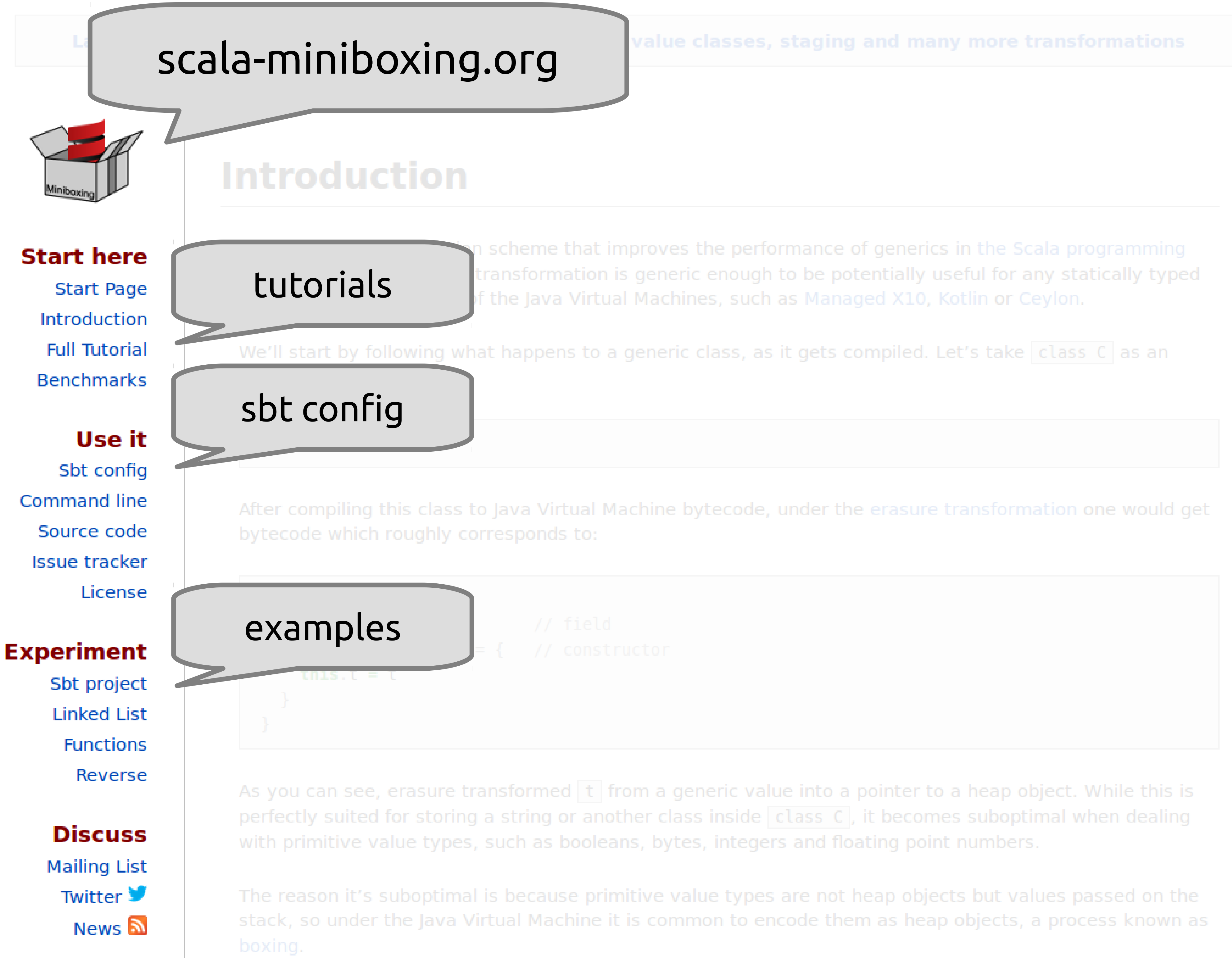
Experiment

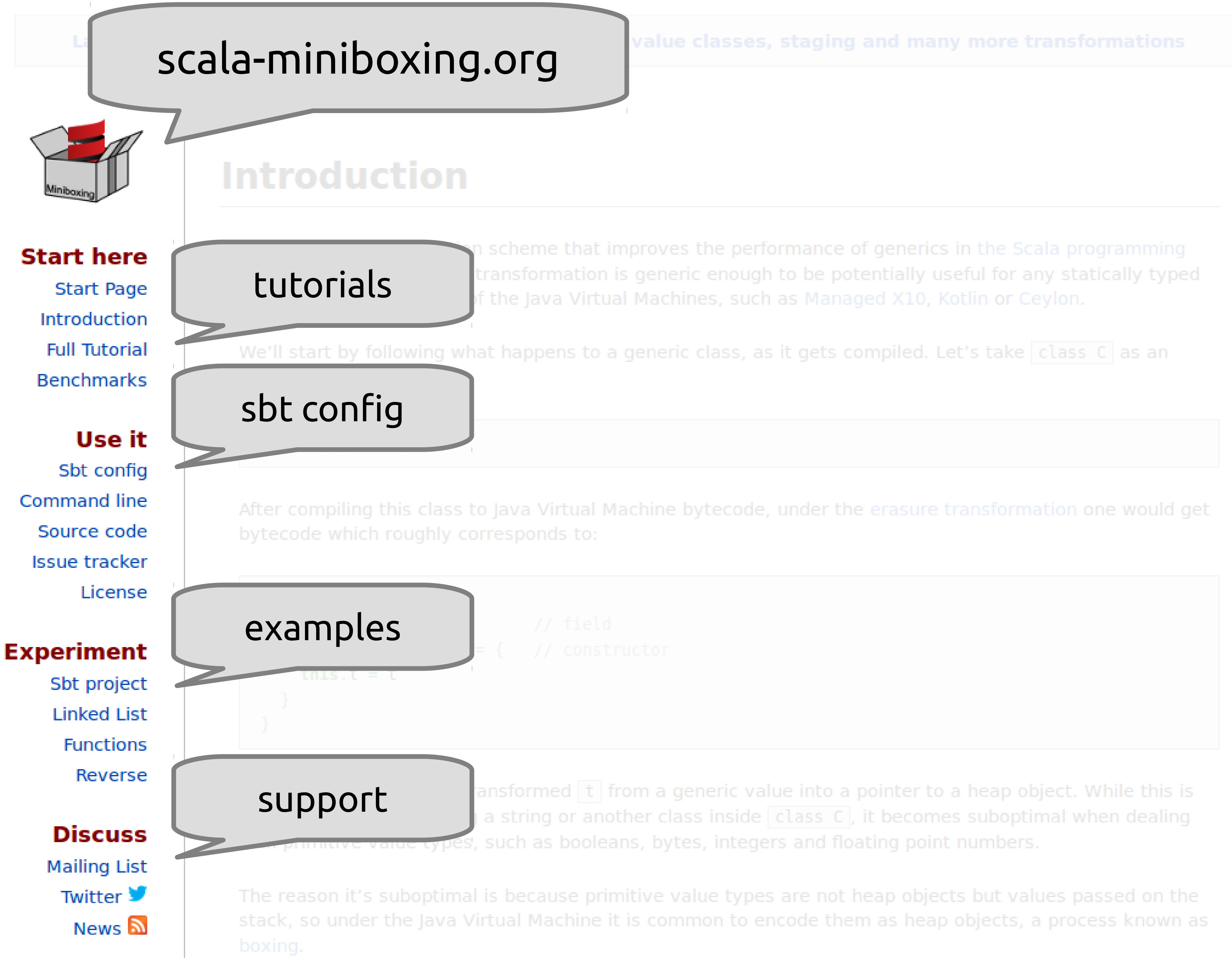
- Sbt project
- Linked List
- Functions
- Reverse

Discuss

- Mailing List
- Twitter
- News


scala-miniboxing.org
tutorials
sbt config
value classes, staging and many more transformations
Miniboxing
Introduction
n scheme that improves the performance of generics in the Scala programming
transformation is generic enough to be potentially useful for any statically typed
f the Java Virtual Machines, such as Managed X10, Kotlin or Ceylon.
We'll start by following what happens to a generic class, as it gets compiled. Let's take class C as an
After compiling this class to Java Virtual Machine bytecode, under the erasure transformation one would get
bytecode which roughly corresponds to:
class C {
var t: Object = _ // field
def C(t: Object): C = { // constructor
this.t = t
}
}
As you can see, erasure transformed t from a generic value into a pointer to a heap object. While this is
perfectly suited for storing a string or another class inside class C, it becomes suboptimal when dealing
with primitive value types, such as booleans, bytes, integers and floating point numbers.
The reason it's suboptimal is because primitive value types are not heap objects but values passed on the
stack, so under the Java Virtual Machine it is common to encode them as heap objects, a process known as
boxing.
Start here
Start Page
Introduction
Full Tutorial
Benchmarks
Use it
Sbt config
Command line
Source code
Issue tracker
License
Experiment
Sbt project
Linked List
Functions
Reverse
Discuss
Mailing List
Twitter
News


scala-miniboxing.org
tutorials
sbt config
examples
Miniboxing
Introduction
Start here
Start Page
Introduction
Full Tutorial
Benchmarks
Use it
Sbt config
Command line
Source code
Issue tracker
License
Experiment
Sbt project
Linked List
Functions
Reverse
Discuss
Mailing List
Twitter
News


scala-miniboxing.org
value classes, staging and many more transformations
Miniboxing
Introduction
tutorials
sbt config
examples
support
Start here
Start Page
Introduction
Full Tutorial
Benchmarks
Use it
Sbt config
Command line
Source code
Issue tracker
License
Experiment
Sbt project
Linked List
Functions
Reverse
Discuss
Mailing List
Twitter
News
n scheme that improves the performance of generics in the Scala programming
transformation is generic enough to be potentially useful for any statically typed
f the Java Virtual Machines, such as Managed X10, Kotlin or Ceylon.
We'll start by following what happens to a generic class, as it gets compiled. Let's take class C as an
After compiling this class to Java Virtual Machine bytecode, under the erasure transformation one would get
bytecode which roughly corresponds to:
// field
// constructor
ansformed t from a generic value into a pointer to a heap object. While this is
a string or another class inside class C, it becomes suboptimal when dealing
, such as booleans, bytes, integers and floating point numbers.
The reason it's suboptimal is because primitive value types are not heap objects but values passed on the
stack, so under the Java Virtual Machine it is common to encode them as heap objects, a process known as
boxing.

Miniboxing in Practice

- Documentation
- Quirks
- Live Demo

Quirks of Scala Specialization

posted by Alex, 03.11.2013.

Having used specialization a lot and having fixed some of its issues, I came across a couple of useful tricks – I want to document them both for myself and others. Specialization is the feature that allows you to generate separate versions of generic classes for primitive types, thus avoiding boxing in most cases. First introduced in Scala 2.8 by Iulian Dragos, by Scala 2.11 specialization has become a pretty robust language feature, and a lot of its issues have been fixed, but there are places where it might stab you in the back if you don't watch out. Problem is, specialization interacts with a some edge-cases in the language and obscure language features in ways that are not expected. Sometimes, these are just unresolved bugs. Here are some tips and tricks that might help you.

Note: when used correctly, this is a powerful and extremely useful feature few JVM languages (if any) can parallel these days. Don't get scared by these tips.

Know the conditions for method specialization

Perhaps you're not aware of this, but even if a method is a part of a specialized class and contains specializable code, it will not really be specialized unless the specialized type appears in its argument list or its return type. For example:

```
def getValue[@specialized T]: T = ???

class Foo[@specialized T] {
  var value: T = _
  def reset() {
    value = getValue
  }
}
```

Above, `reset` is not specialized. This is an elaborate design decision taken in specialization. If you want

Quirks of Scala Specialization

posted by Alex, 03.11.2013.

Alex Prokopec, axel22.github.io

Having used specialization a lot and having fixed some of its issues, I came across a couple of useful tricks – I want to document them both for myself and others. Specialization is the feature that allows you to generate separate versions of generic classes for primitive types, thus avoiding boxing in most cases. First introduced in Scala 2.8 by Iulian Dragos, by Scala 2.11 specialization has become a pretty robust language feature, and a lot of its issues have been fixed, but there are places where it might stab you in the back if you don't watch out. Problem is, specialization interacts with a some edge-cases in the language and obscure language features in ways that are not expected. Sometimes, these are just unresolved bugs. Here are some tips and tricks that might help you.

Note: when used correctly, this is a powerful and extremely useful feature few JVM languages (if any) can parallel these days. Don't get scared by these tips.

Know the conditions for method specialization

Perhaps you're not aware of this, but even if a method is a part of a specialized class and contains specializable code, it will not really be specialized unless the specialized type appears in its argument list or its return type. For example:

```
def getValue[@specialized T]: T = ???

class Foo[@specialized T] {
  var value: T = _
  def reset() {
    value = getValue
  }
}
```

Above, `reset` is not specialized. This is an elaborate design decision taken in specialization. If you want

Quirks of Scala Specialization

posted by Alex, 03.11.2013.

Alex Prokopec, axel22.github.io

Having used specialization a lot and having fixed some of its issues, I came across a couple of useful tricks – I want to document them both for myself and others. Specialization is the feature that allows you to generate separate versions of generic classes for primitive types, thus avoiding boxing in most cases. First introduced in Scala 2.8 by Iulian Dragos, by Scala 2.11 specialization has become a pretty robust language feature, and a lot of its issues have been fixed, but there are places where it might stab you in the back if you don't watch out. Problem is, specialization interacts with a some edge-cases in the language and obscure language features in ways that are not expected. Sometimes, these are just unresolved bugs. Here are some tips and tricks that might help you.

Note: when used correctly, this is a powerful and extremely useful feature few JVM languages (if any) can parallel these days. Don't get scared by these tips.

Know the conditions for method specialization

Perhaps you're not aware of this, but even if a method is a part of a specialized class and contains specializable code, it will not really be specialized unless the specialized type appears in its argument list or its return type. For example:

```
def getValue[@specialized T]: T = ???

class Foo[@specialized T] {
  var value: T = _
  def reset() {
    value = getValue
  }
}
```

Quirks of Scala Specialization

posted by Alex, 03.11.2013.

Alex Prokopec, axel22.github.io

- Know the conditions for method specialization - method specialization
- Know the conditions for method specialization - class specialization
- Initialize specialized values outside constructor body
- Resolve access problems using the package-private modifier
- Use traits where possible
- Avoid traits where possible
- Make your classes as flat as possible
- Avoid super calls
- Be wary of vars
- Think about the primitive types you really care about
- Avoid using specialization and implicit classes

Quirks of Scala Specialization

posted by Alex, 03.11.2013.

Alex Prokopec, axel22.github.io

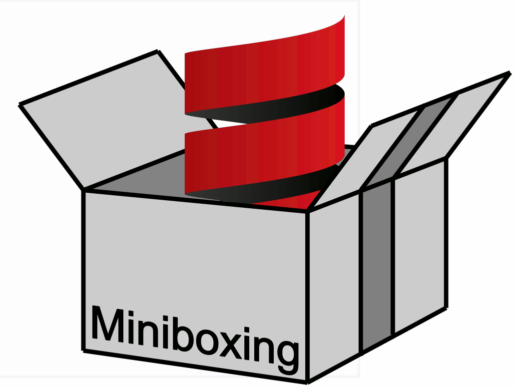
- Know the conditions for method specialization - method specialization
- Know the conditions for method specialization - class specialization
- Initialize specialized values outside constructor body
- Resolve access problems using the package-private modifier
- Use traits where possible
- Avoid traits where possible
- Make your classes as flat as possible
- Avoid super calls
- Be wary of vars
- Think about the primitive types you really care about
- Avoid using specialization and implicit classes

Quirks of Scala Specialization

posted by Alex, 03.11.2013.

Alex Prokopec, axel22.github.io

- Know the conditions for method specialization - method specialization
- Know the conditions for method specialization - class specialization
- Initialize specialized values outside constructor body
- Resolve access problems using the package-private modifier
- Use traits where possible
- Avoid traits where possible
- Make your classes as flat as possible
- Avoid super calls
- Be wary of vars
- Think about the primitive types you really care about
- Avoid using specialization and implicit classes



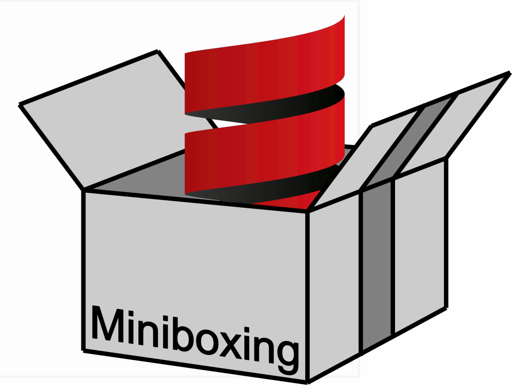
Quirks of Scala Specialization

posted by Alex, 03.11.2013.

Alex Prokopec, axel22.github.io

🟡 = fixed by the 0.4 release

- 🟢 Know the conditions for method specialization - method specialization
- 🟢 Know the conditions for method specialization - class specialization
- 🟡 Initialize specialized values outside constructor body
- 🟡 Resolve access problems using the package-private modifier
- 🟢 Use traits where possible
- 🟢 Avoid traits where possible
- 🟡 Make your classes as flat as possible
- 🔴 Avoid super calls
- 🟡 Be wary of vars
- 🟢 Think about the primitive types you really care about
- 🟢 Avoid using specialization and implicit classes



Quirks of Scala Specialization

posted by Alex, 03.11.2013.

Alex Prokopec, axel22.github.io

Having used specialization a lot and having fixed some of its issues, I came across a couple of useful tricks – I want to document them both for myself and others. Specialization is the feature that allows you to generate separate versions of generic classes for primitive types, thus avoiding boxing in most cases. First introduced in Scala 2.8 by Iulian Dragos, by Scala 2.11 specialization has become a pretty robust language feature, and a lot of its issues have been fixed, but there are places where it might stab you in the back if you don't watch out. Problem is, specialization interacts with a some edge-cases in the language and obscure language features in ways that are not expected. Sometimes, these are just unresolved bugs. Here are some tips and tricks that might help you.

Quirk

Note: when used correctly, this is a powerful and extremely useful feature few JVM languages (if any) can parallel these days. Don't get scared by these tips.

Know the conditions for method specialization

Perhaps you're not aware of this, but even if a method is a part of a specialized class and contains specializable code, it will not really be specialized unless the specialized type appears in its argument list or its return type. For example:

```
def getValue[@specialized T]: T = ???

class Foo[@specialized T] {
  var value: T = _
  def reset() {
    value = getValue
  }
}
```

Quirks of Scala Specialization

posted by Alex, 03.11.2013.

Alex Prokopec, axel22.github.io

Having used specialization a lot and having fixed some of its issues, I came across a couple of useful tricks – I want to document them both for myself and others. Specialization is the feature that allows you to generate separate versions of generic classes for primitive types, thus avoiding boxing in most cases. First introduced in Scala 2.8 by Iulian Dragos, by Scala 2.11 specialization has become a pretty robust language feature, and a lot of its issues have been fixed, but there are places where it might stab you in the back if you don't watch out. Problem is, specialization interacts with a some edge-cases in the language and obscure language features in ways that are not expected. Sometimes, these are just unresolved bugs. Here are some tips and tricks that might help you.

Quirk = limitation

Note: when used correctly, this is a powerful and extremely useful feature few JVM languages (if any) can parallel these days. Don't get scared by these tips.

Know the conditions for method specialization

Perhaps you're not aware of this, but even if a method is a part of a specialized class and contains specializable code, it will not really be specialized unless the specialized type appears in its argument list or its return type. For example:

```
def getValue[@specialized T]: T = ???

class Foo[@specialized T] {
  var value: T = _
  def reset() {
    value = getValue
  }
}
```

Above, `reset` is not specialized. This is an elaborate design decision taken in specialization. If you want

Quirks of Scala Specialization

posted by Alex, 03.11.2013.

Alex Prokopec, axel22.github.io

Having used specialization a lot and having fixed some of its issues, I came across a couple of useful tricks – I want to document them both for myself and others. Specialization is the feature that allows you to generate separate versions of generic classes for primitive types, thus avoiding boxing in most cases. First introduced in Scala 2.8 by Iulian Dragos, by Scala 2.11 specialization has become a pretty robust language feature, and a lot of its issues have been fixed, but there are places where it might stab you in the back if you don't watch out. Problem is, specialization interacts with a some edge-cases in the language and obscure language features in ways that are not expected. Sometimes, these are just unresolved bugs. Here are some tips and tricks that might help you.

Quirk = limitation + silent failure

Note: when used correctly, this is a powerful and extremely useful feature few JVM languages (if any) can parallel these days. Don't get scared by these tips.

Know the conditions for method specialization

Perhaps you're not aware of this, but even if a method is a part of a specialized class and contains specializable code, it will not really be specialized unless the specialized type appears in its argument list or its return type. For example:

```
def getValue[@specialized T]: T = ???

class Foo[@specialized T] {
  var value: T = _
  def reset() {
    value = getValue
  }
}
```

Quirks of Scala Specialization

posted by Alex, 03.11.2013.

Alex Prokopec, axel22.github.io

Having used specialization a lot and having fixed some of its issues, I came across a couple of useful tricks – I want to document them both for myself and others. Specialization is the feature that allows you to generate separate versions of generic classes for primitive types, thus avoiding boxing in most cases. First introduced in Scala 2.8 by Iulian Dragos, by Scala 2.11 specialization has become a pretty robust language feature, and a lot of its issues have been fixed, but there are places where it might stab you in the back if you don't watch out. Problem is, specialization interacts with a some edge-cases in the language and obscure language features in ways that are not expected. Sometimes, these are just unresolved bugs. Here are some tips and tricks that might help you.

Quirk = limitation + silent failure

Note: when used correctly, this is a powerful and extremely useful feature few JVM languages (if any) can parallel these days. Don't get scared by these tips.

let's make the limitations transparent

Know the conditions for method specialization

Perhaps you're not aware of this, but even if a method is a part of a specialized class and contains specializable code, it will not really be specialized unless the specialized type appears in its argument list or its return type. For example:

```
def getValue[@specialized T]: T = ???

class Foo[@specialized T] {
  var value: T = _
  def reset() {
    value = getValue
  }
}
```

Miniboxing in Practice

- Documentation
- Quirks
- Live Demo

Miniboxing

Theory

Practice

$\boxed{\text{Benchmarks}}$

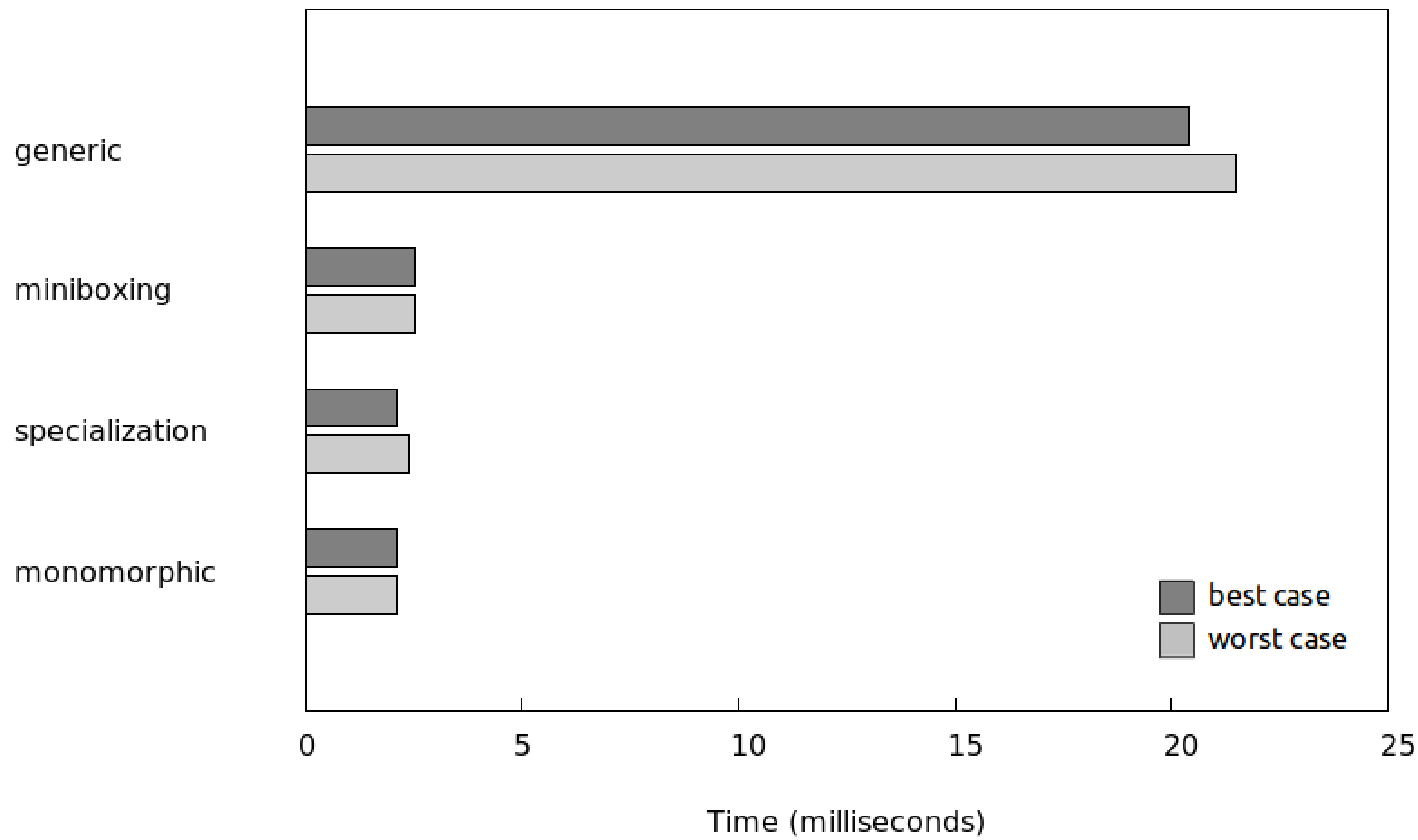
Conclusion

scala-miniboxing.org

Benchmarks

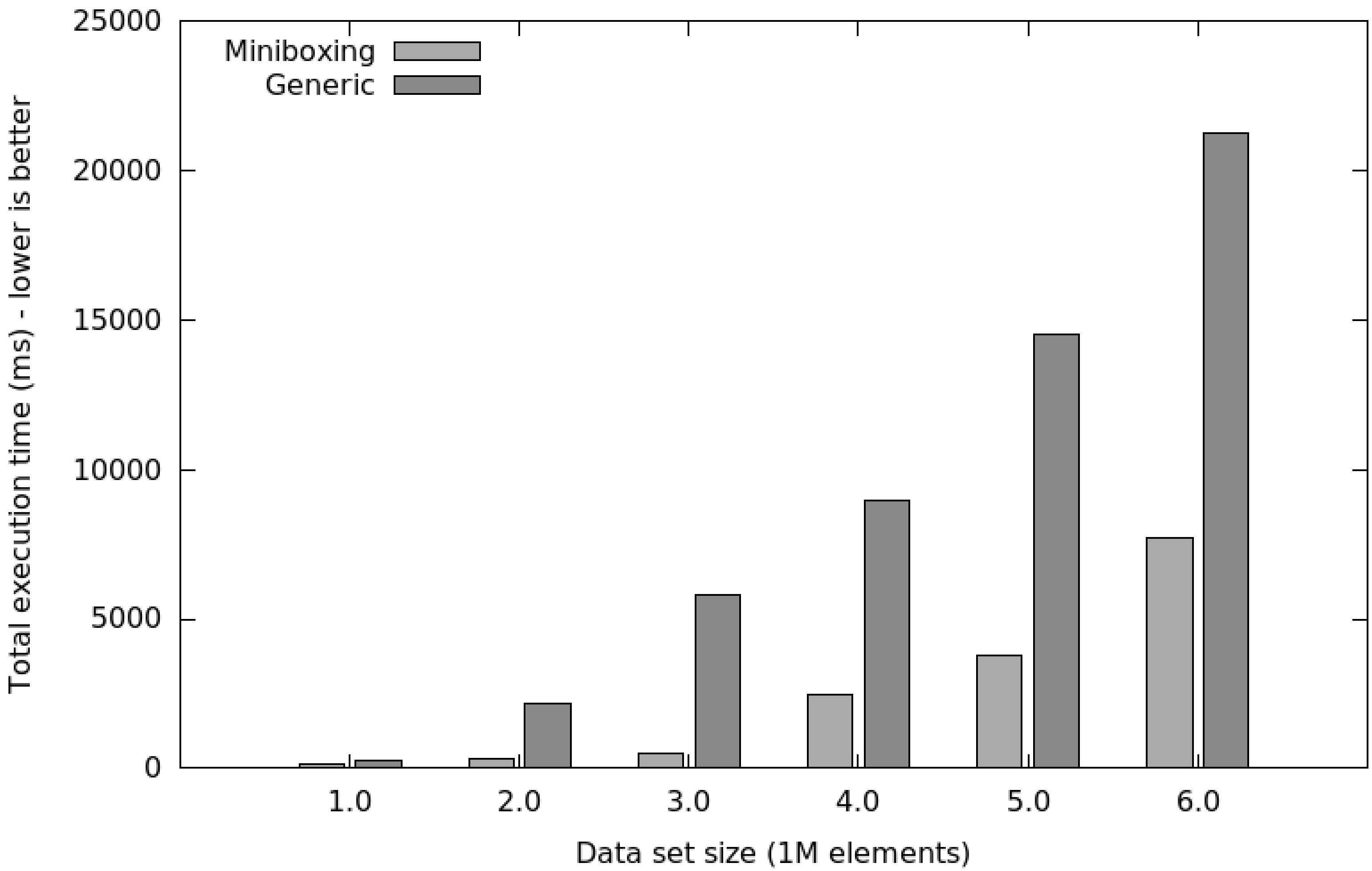
- Microbenchmarks
- Collections
- Spire

Miniboxing Benchmarks
on array buffers

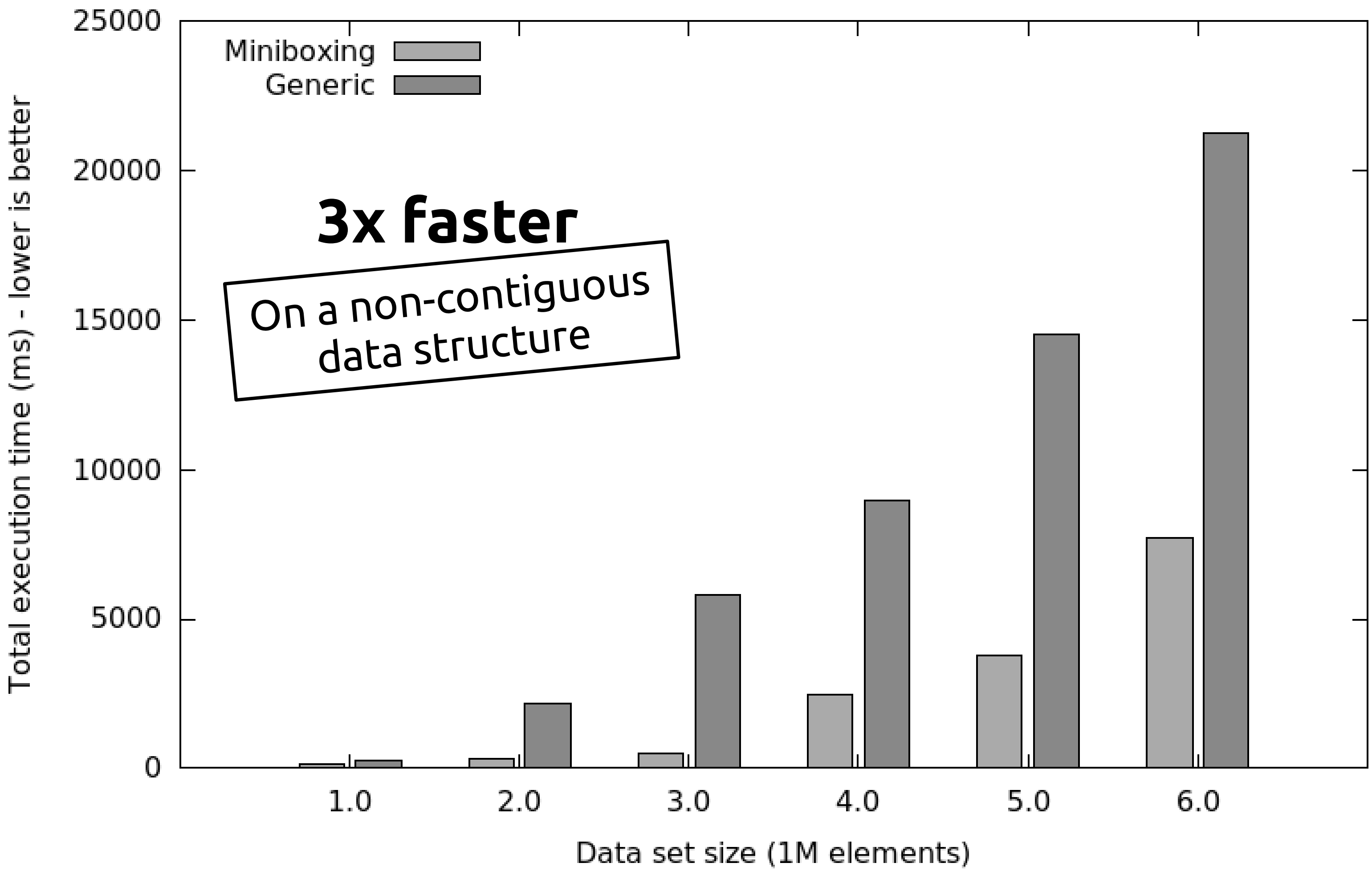


scala-miniboxing.org

Miniboxing Benchmarks
on linked lists

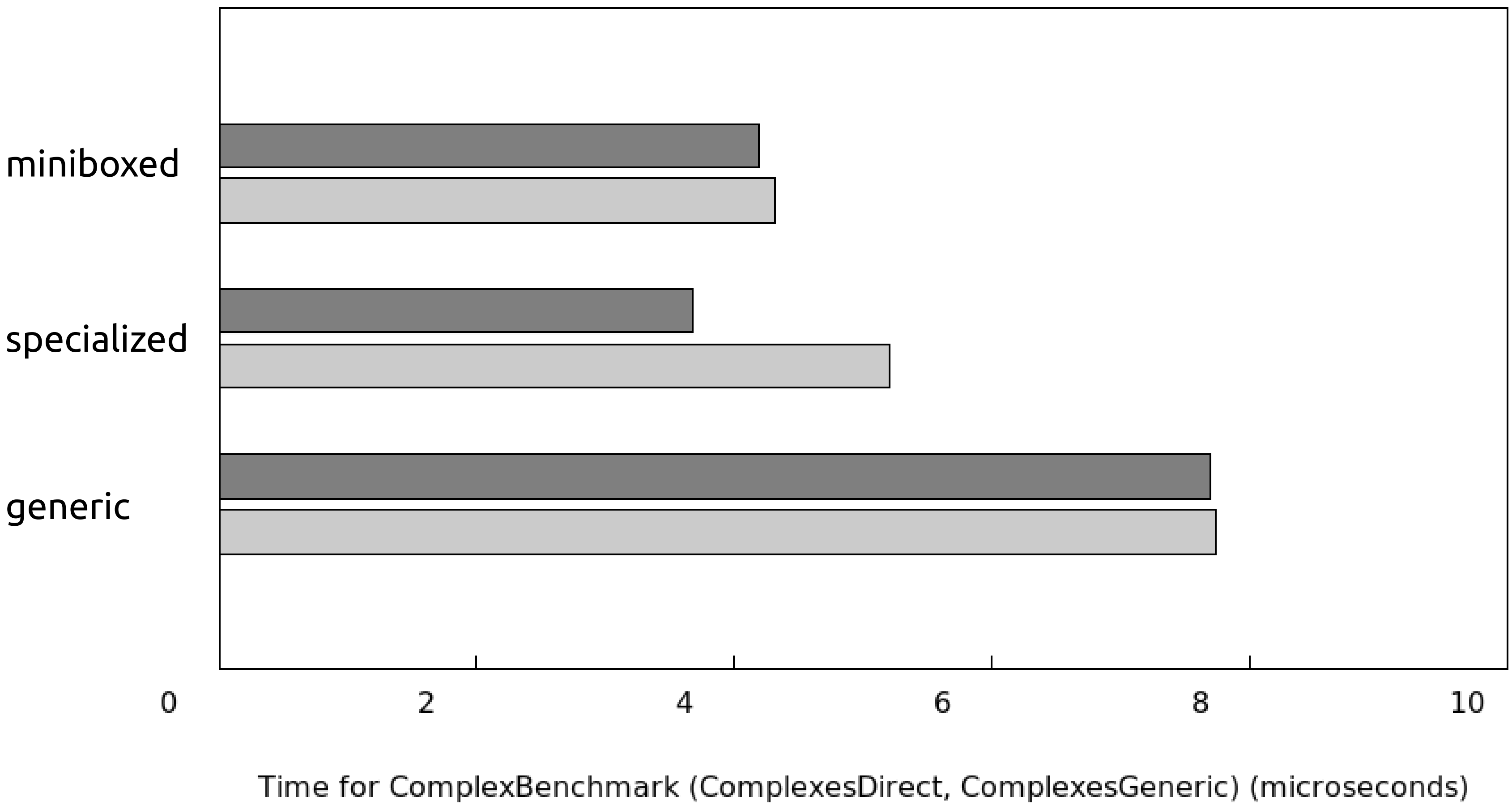


Miniboxing Benchmarks
on linked lists



Miniboxing Benchmarks

on the Spire library (RexBench)



Miniboxing

Theory

Practice

Benchmarks

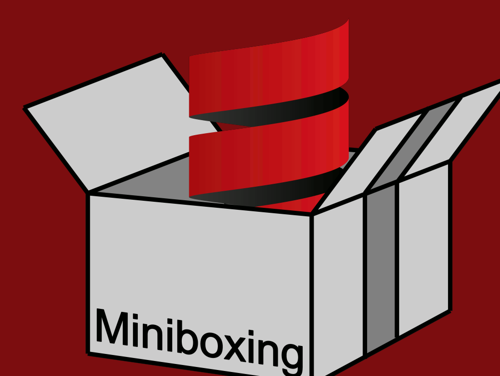
Conclusion

scala-miniboxing.org

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

Miniboxing

scala-miniboxing.org

ScalaTeam @ EPFL

scala-miniboxing.org

ScalaTeam @ EPFL

- **YinYang** frontend multi-stage execution
  - based on macro transformations
  - Vojin Jovanovic/Sandro Stucki

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

ScalaTeam @ EPFL

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

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

ScalaTeam @ EPFL

- **Lightweight Modular Staging**
  - program optimization
  - Tiark Rompf + pretty much everyone

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

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

- **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

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

https://github.com/manojo/experiments

ScalaTeam @ EPFL

- **dotty** compiler
  - compiler for Scala but with the DOT type system
  - Martin Odersky/Dmitry Petrashko/Tobias Schlatter

https://github.com/lampepfl/dotty

ScalaTeam @ EPFL

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

www http://scalameta.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

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

www http://scala-miniboxing.org/

ScalaTeam @ EPFL

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

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

ScalaTeam @ EPFL

- **LMS-Kappa** protein simulator
  - using multi-stage programming for performance
  - Sandro Stucki

https://github.com/sstucki/lms-kappa

ScalaTeam @ EPFL

- **Odds** probabilistic programming framework
  - using scala-virtualized and macros
  - Sandro Stucki

https://github.com/sstucki/odds

ScalaTeam @ EPFL

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

www 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

- **Vector** implementation using RRB trees
  - improved performance for Scala collections
  - Nicolas Stucki

www https://github.com/nicolasstucki/scala-rrb-vector

ScalaTeam @ EPFL

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

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

@ miguel.garcia@tuhh.de