Miniboxing

Fast Generics on the JVM

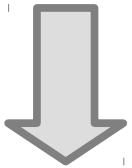
ScalaDays, 18th of June 2014

Vlad Ureche
gh/twitter:@VladUreche

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

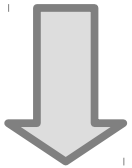
class C[T](t: T)

scalac / javac



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

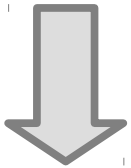
scalac / javac



```
class C(t: Object)
```

class C[T](t: T)

scalac / javac

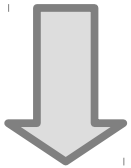


The process is called **erasure**, and it replaces type paramerers by their upper bound

class C(t: Object)

class C[T](t: T)

scalac / javac



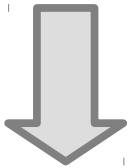
The process is called **erasure**, and it replaces type paramerers by their upper bound

class C(t: Object)

Requires boxed primitive types (java.lang.Integer, ...)

class C[T](t: T)

scalac / javac



The process is called **erasure**, and it replaces type paramerers by their upper bound

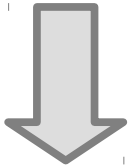
class C(t: Object)

Requires boxed primitive types (java.lang.Integer, ...)

You don't see it: **scala.Int** can be either **int** or **j.l.Integer**. Scalac does the work for you!

class C[T](t: T)

scalac / javac



The process is called **erasure**, and it replaces type paramerers by their upper bound

class C(t: Object)

Requires boxed primitive types (java.lang.Integer, ...)

You don't see it: **scala.Int** can be either **int** or **j.l.Integer**. Scalac does the work for you!

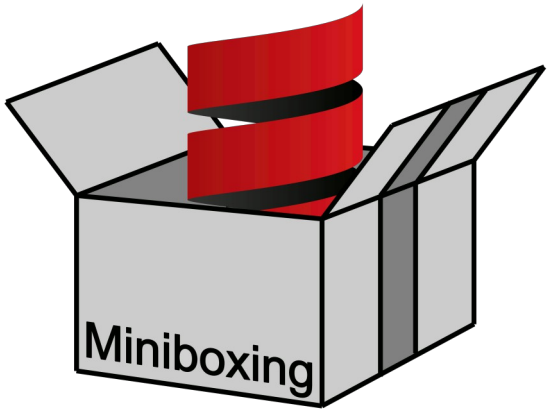
Yet boxing degrades performance
- heap allocations / GC cycles ...
- indirect reads, broken locality

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

@specialized

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

@miniboxed

What is 

?

@miniboxed =

@miniboxed = @specialized

@miniboxed = @specialized

- the limitations

@miniboxed = @specialized

- the limitations
- bytecode bloat

opens the way to
@miniboxed
Scala collections

@miniboxed = @specialized

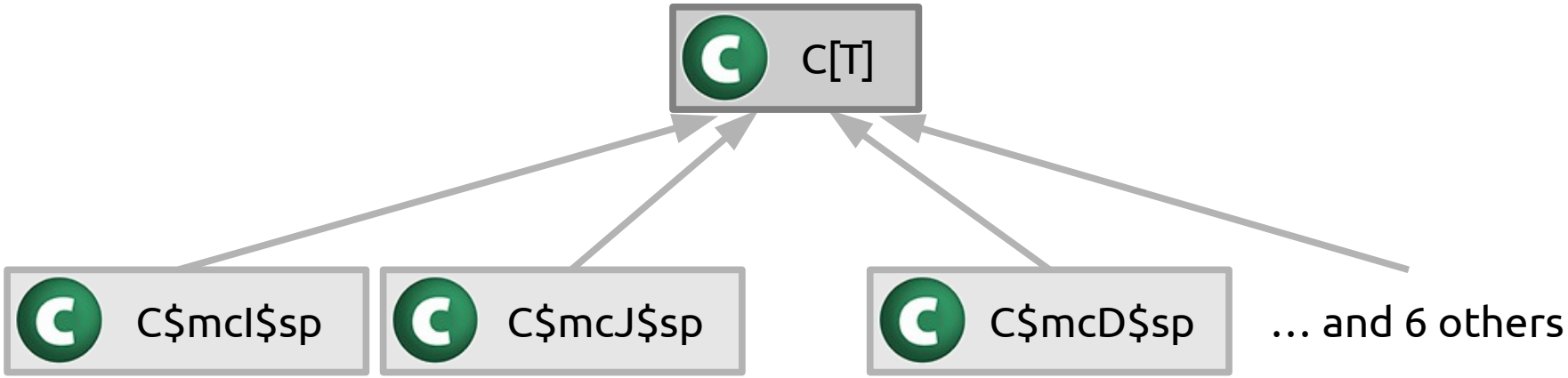
- the limitations
- bytecode bloat

@specialized

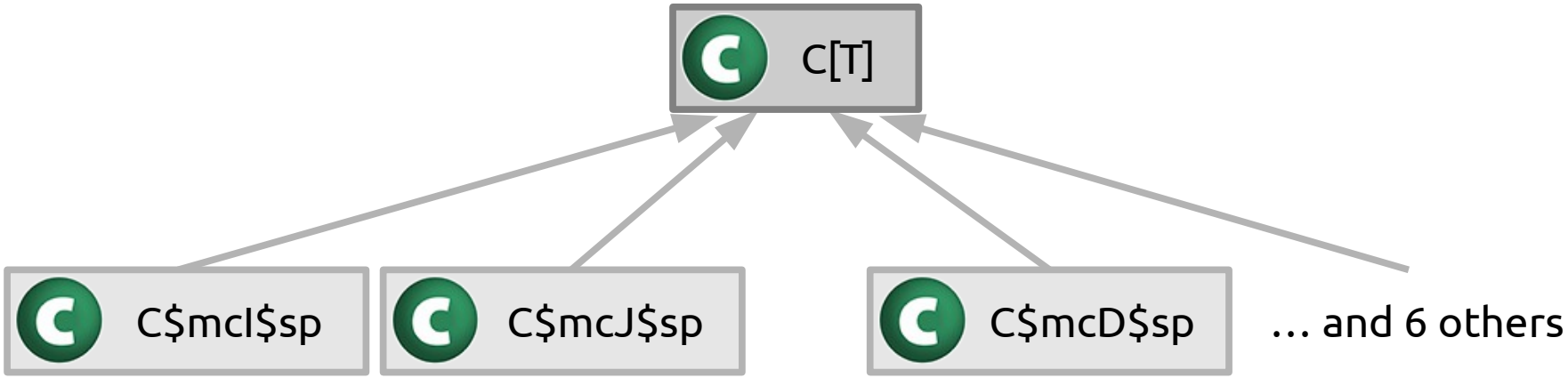
```
class C[@specialized T](t: T)
```

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

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

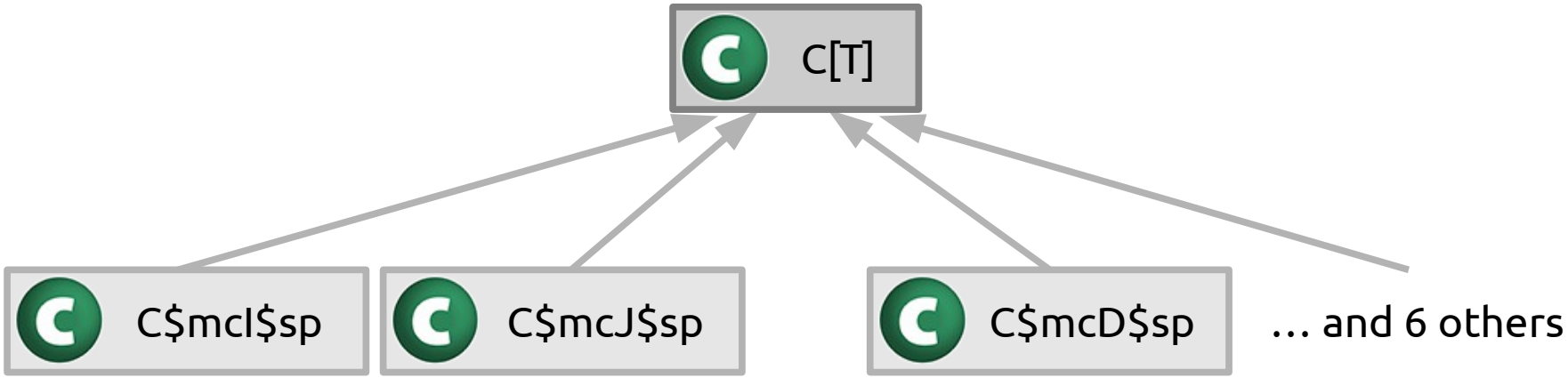


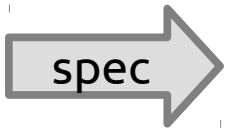
class C[@specialized T](t: T)



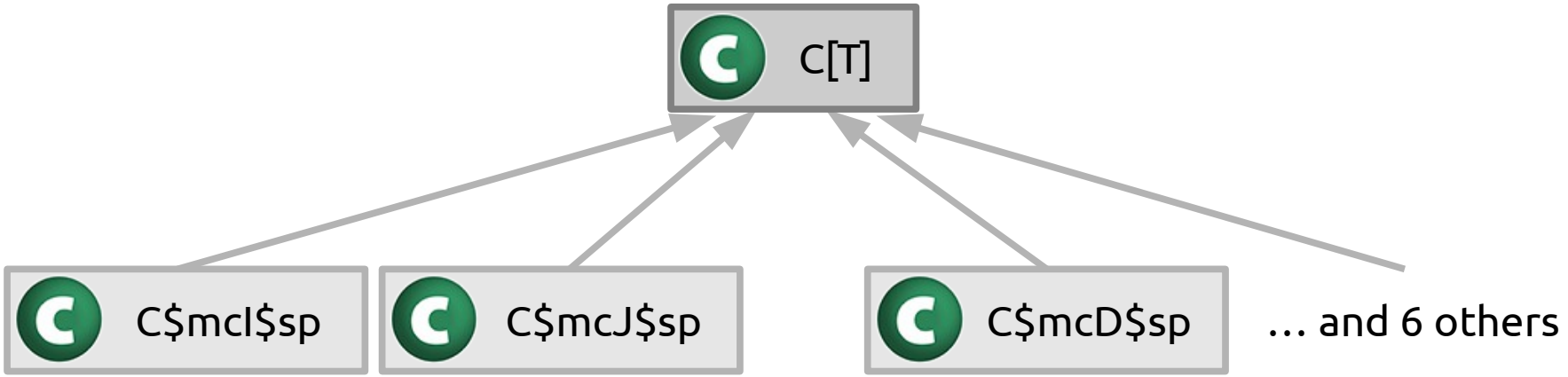
new C[**Int**](4)

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



new C[**Int**](4)  new C$mc**I**$sp(4)

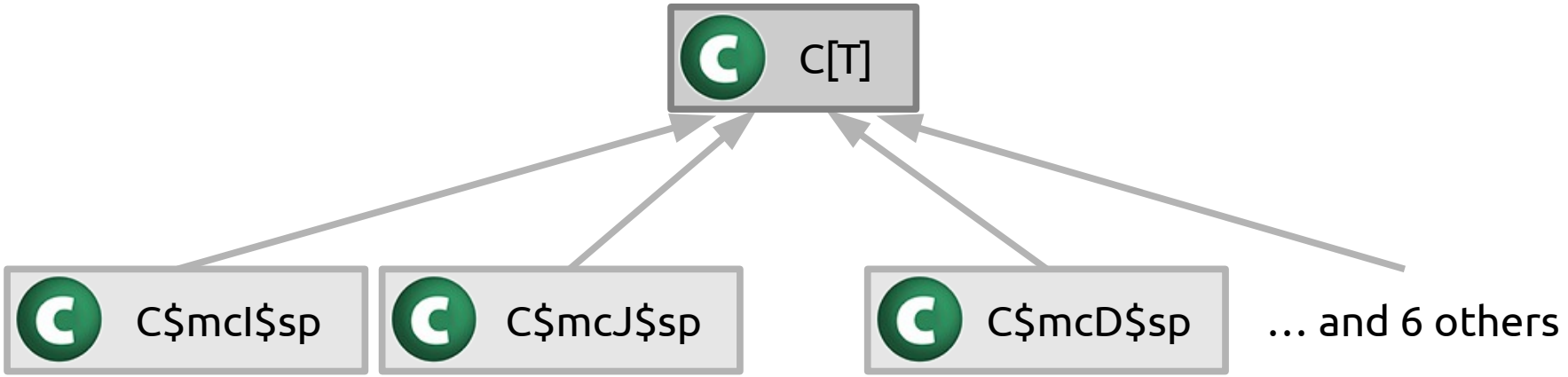
class C[@specialized T](t: T)



new C[**Int**](4) spec new C$mc**I**$sp(4)

Adapted to integers **(tmcIsp: int)**

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

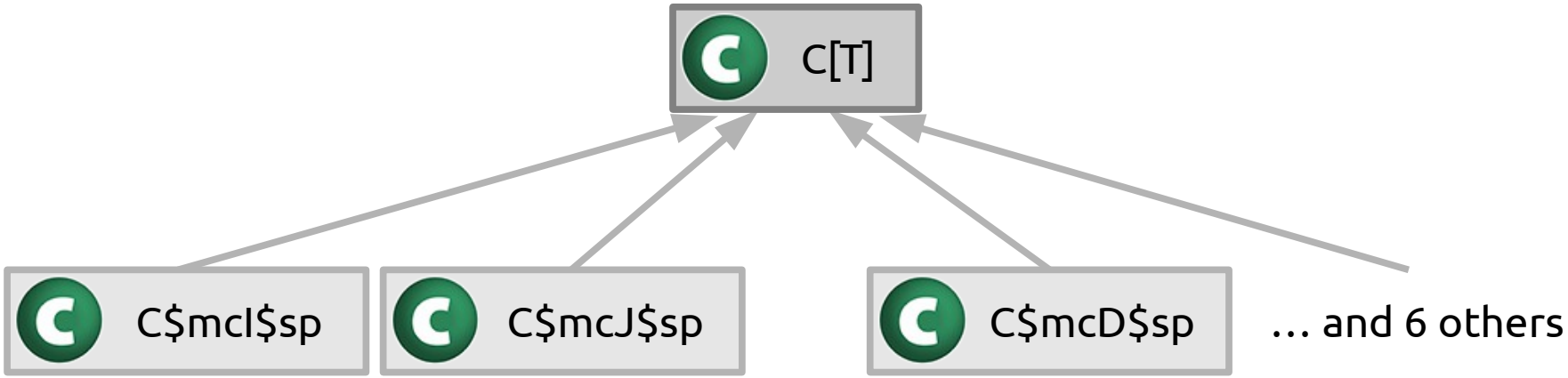


new C[**Int**](4) spec new C$mc**I**$sp(4)

Adapted to integers **(tmcIsp: int)**

new C("abc")

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

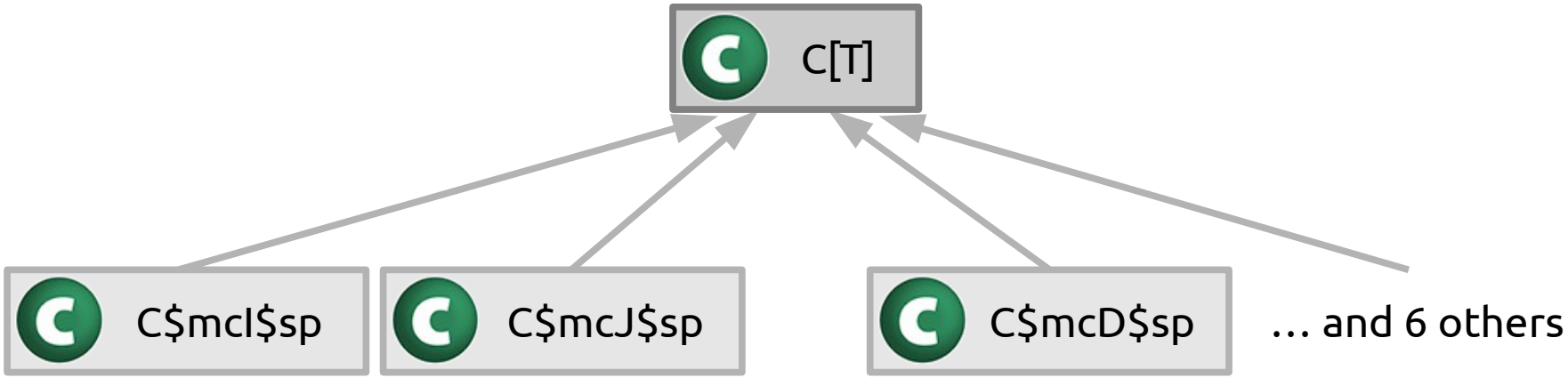


new C[**Int**](4) → spec → new C$mc**I**$sp(4)

Adapted to integers **(tmcIsp: int)**

new C("abc") → spec → new C[String]("abc")

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



new C[**Int**](4) → spec → new C$mc**I**$sp(4)

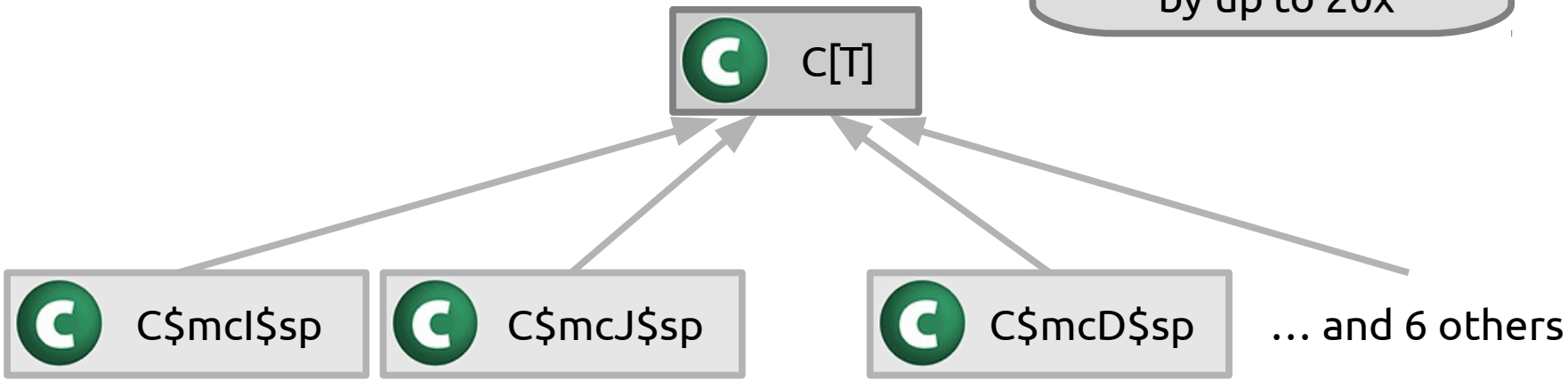
Adapted to integers **(tmcIsp: int)**

new C("abc") → spec → new C[String]("abc")

* similar transformation for methods

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

Can speed up
certain code patterns
by up to 20x



new C[**Int**](4) → spec → new C$mc**I**$sp(4)

Adapted to integers **(tmcIsp: int)**

new C("abc") → spec → new C[String]("abc")

* similar transformation for methods

@miniboxed = @specialized

- the limitations
- bytecode bloat

@miniboxed = @specialized

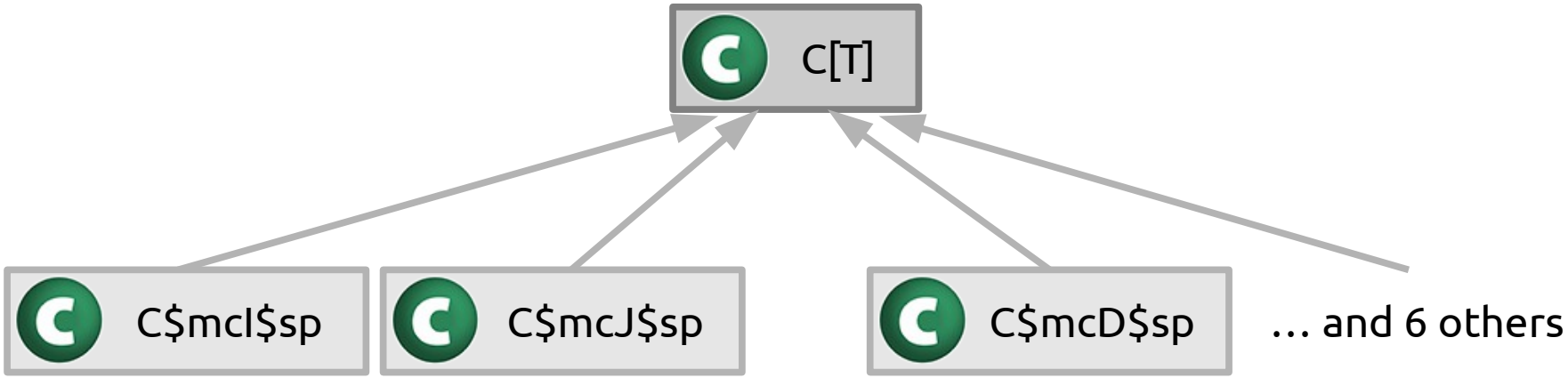
- the limitations
- bytecode bloat

- duplicate fields
- broken inheriance

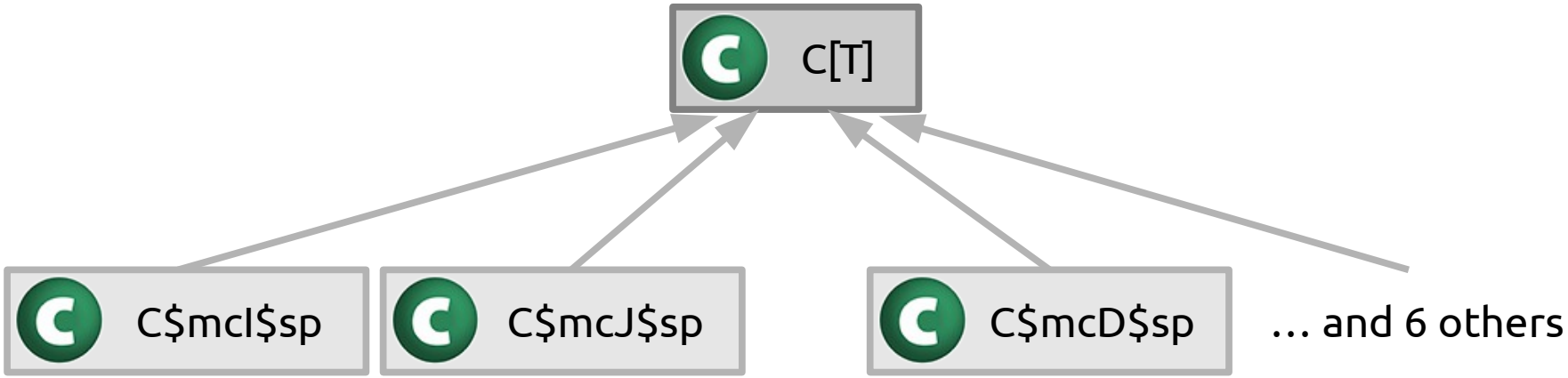
```
class C[@specialized T](t: T)
```

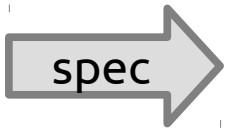
class C[@specialized T](t: T)

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



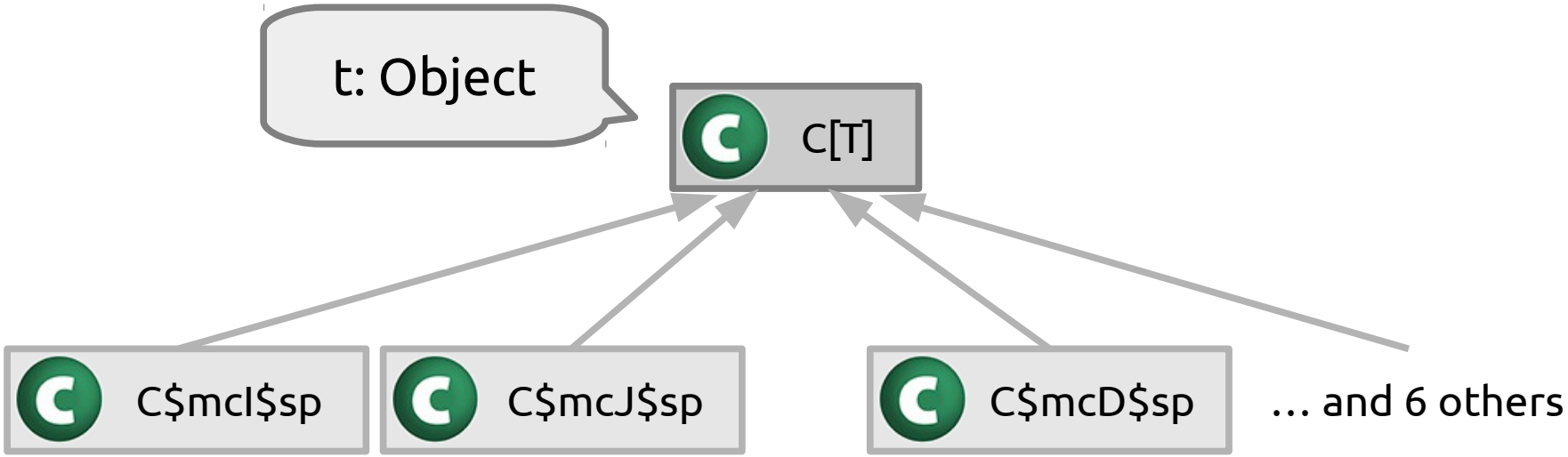
class C[@specialized T](t: T)



new C("abc") 

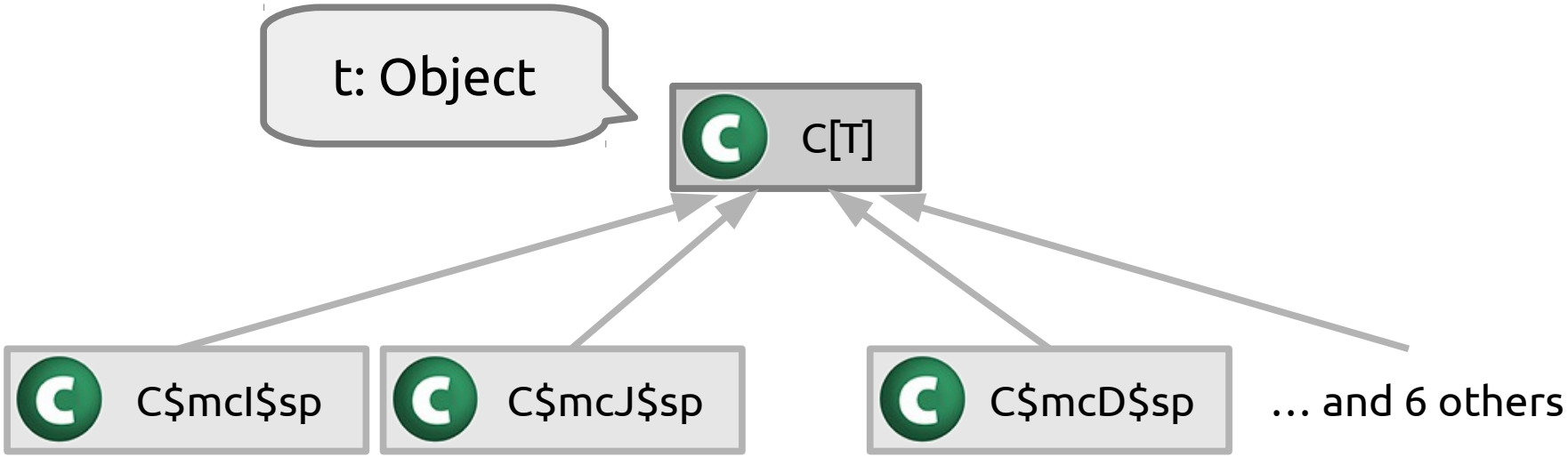
new C[String]("abc")

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



new C("abc") spec new C[String]("abc")

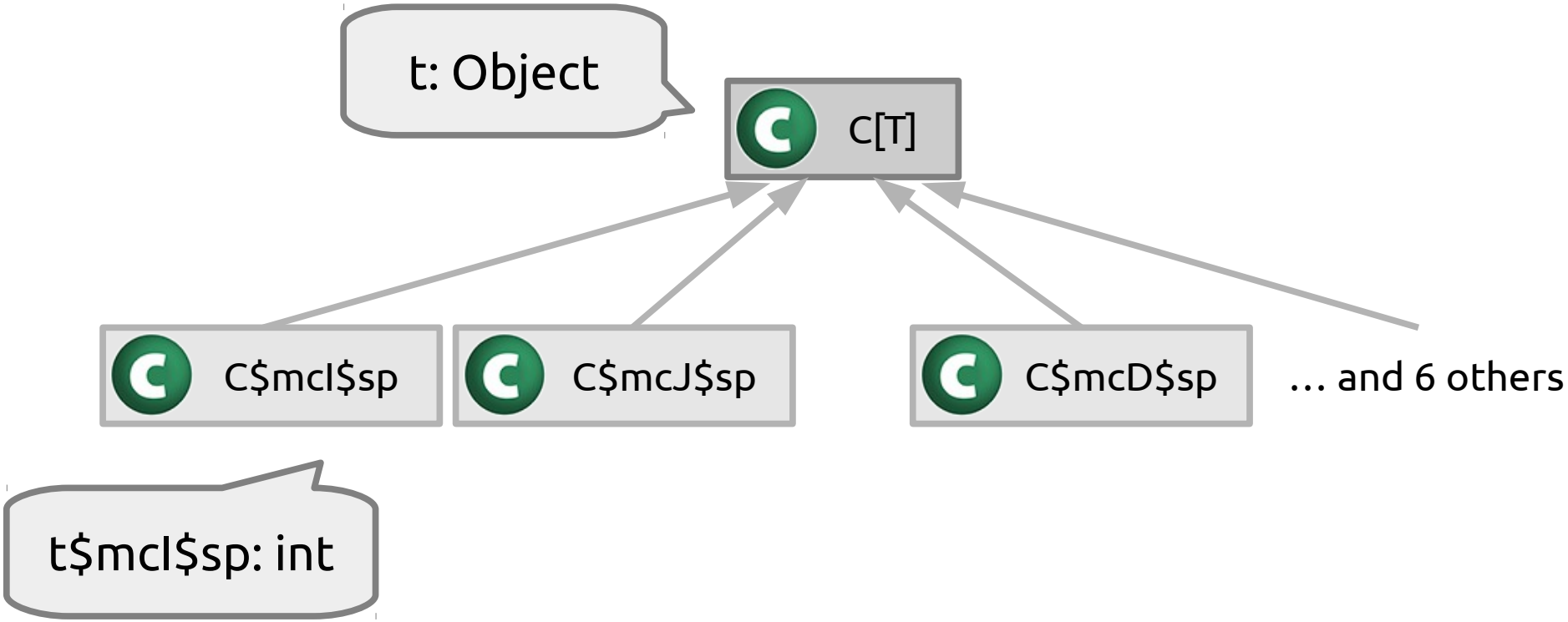
class C[@specialized T](t: T)



new C("abc") spec new C[String]("abc")

new C[**Int**](4) spec new C$mc**I**$sp(4)

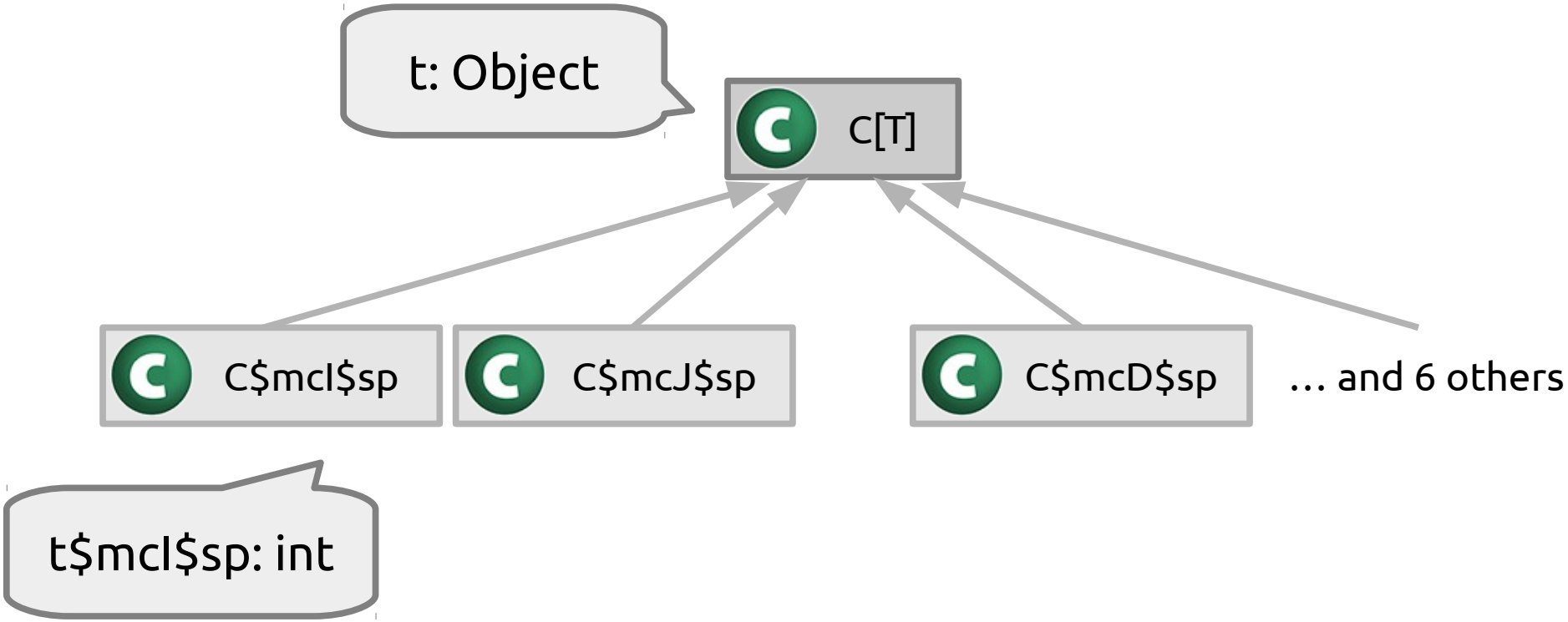
class C[@specialized T](t: T)



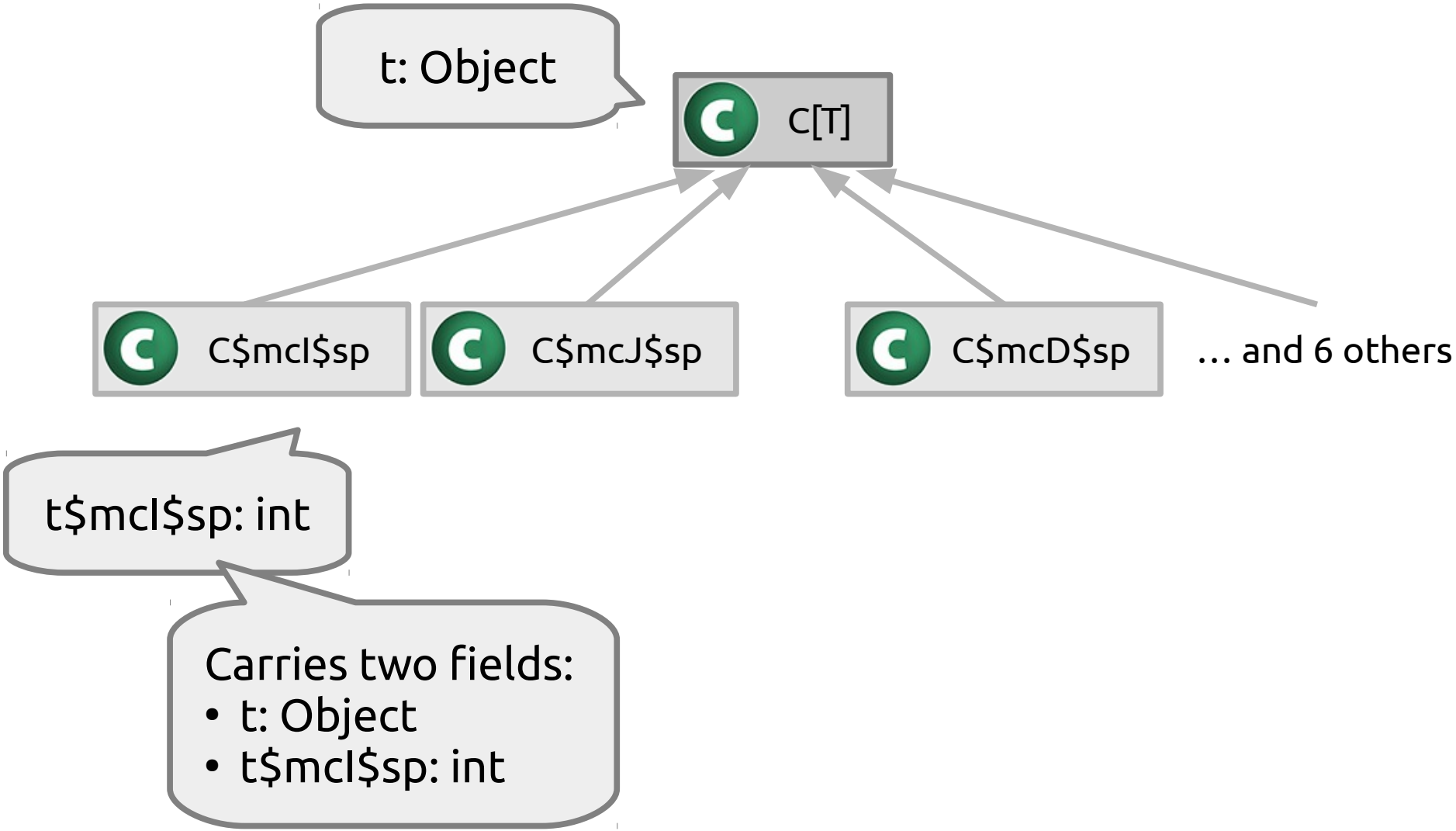
new C("abc") spec new C[String]("abc")

new C[**Int**](4) spec new C$mc**I**$sp(4)

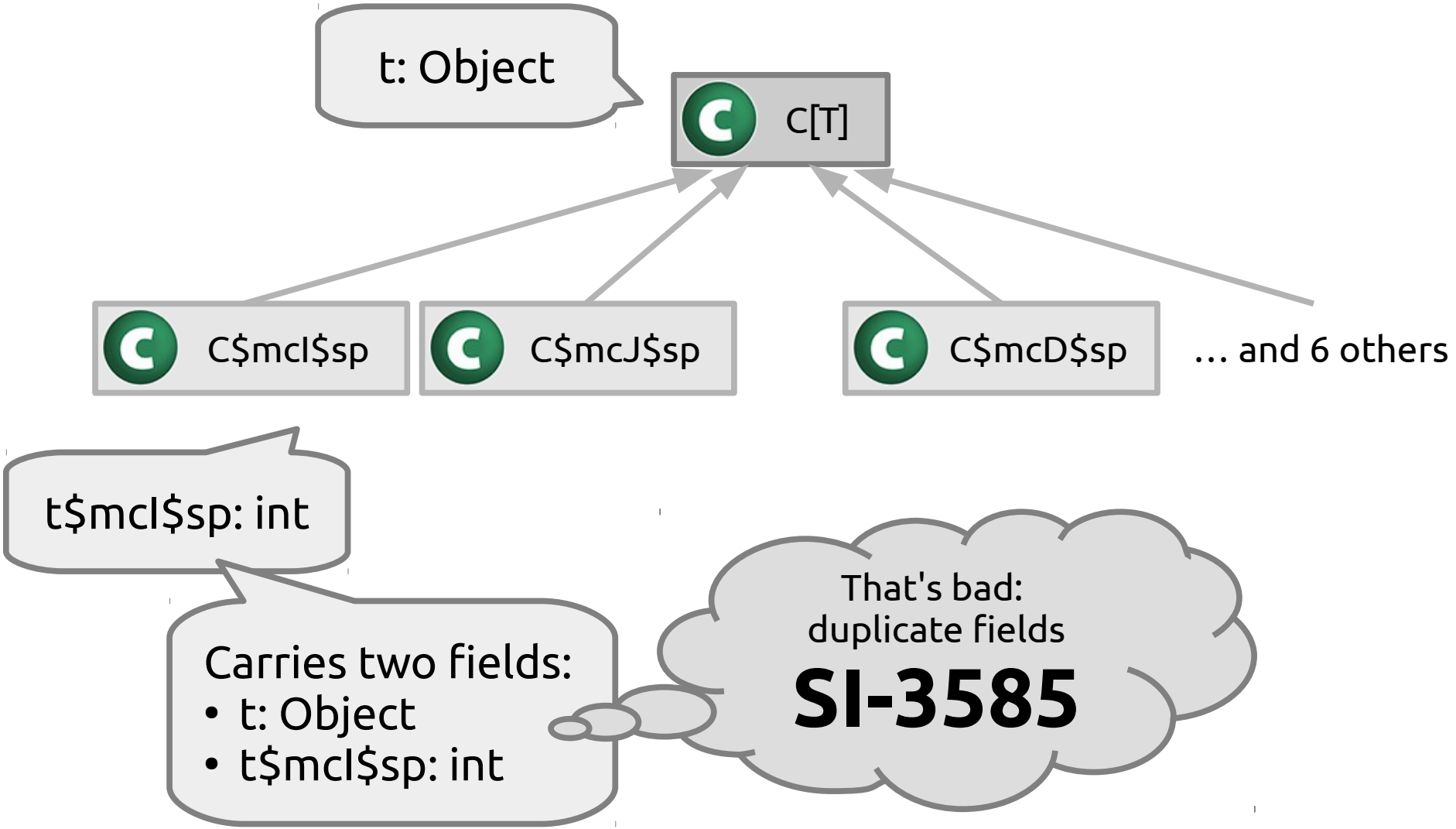
class C[@specialized T](t: T)



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

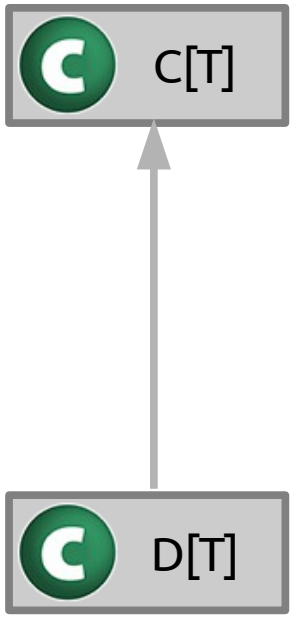


class C[@specialized T](t: T)

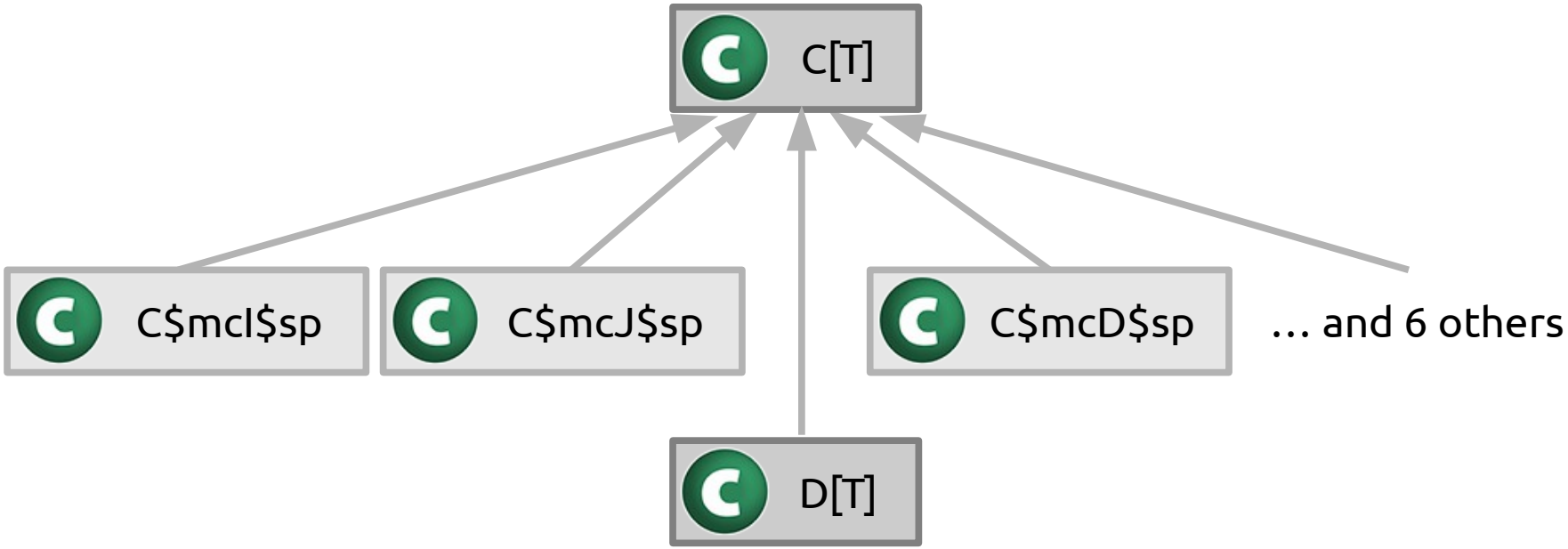


```
class D[@specialized T](t: T)
extends C[T]
```

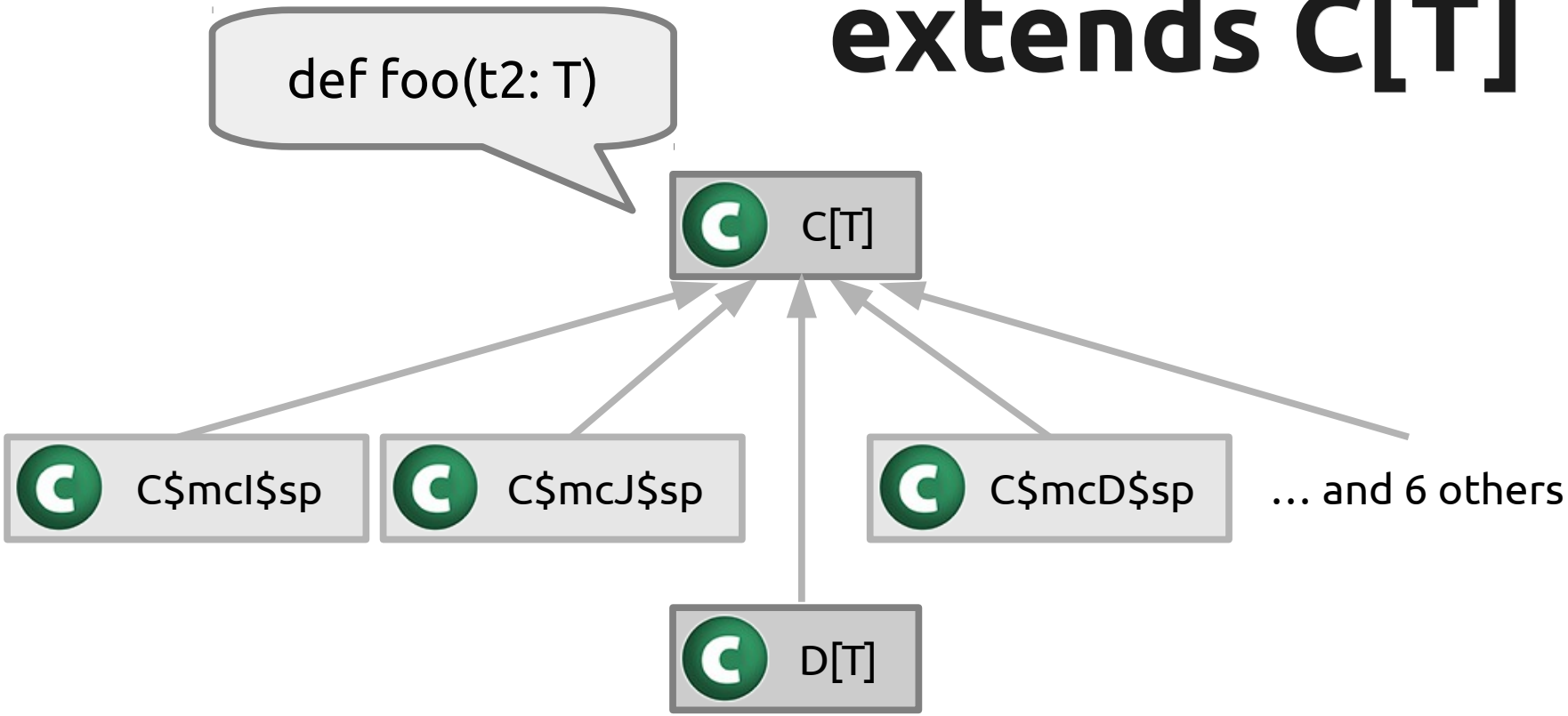
class D[@specialized T](t: T) extends C[T]



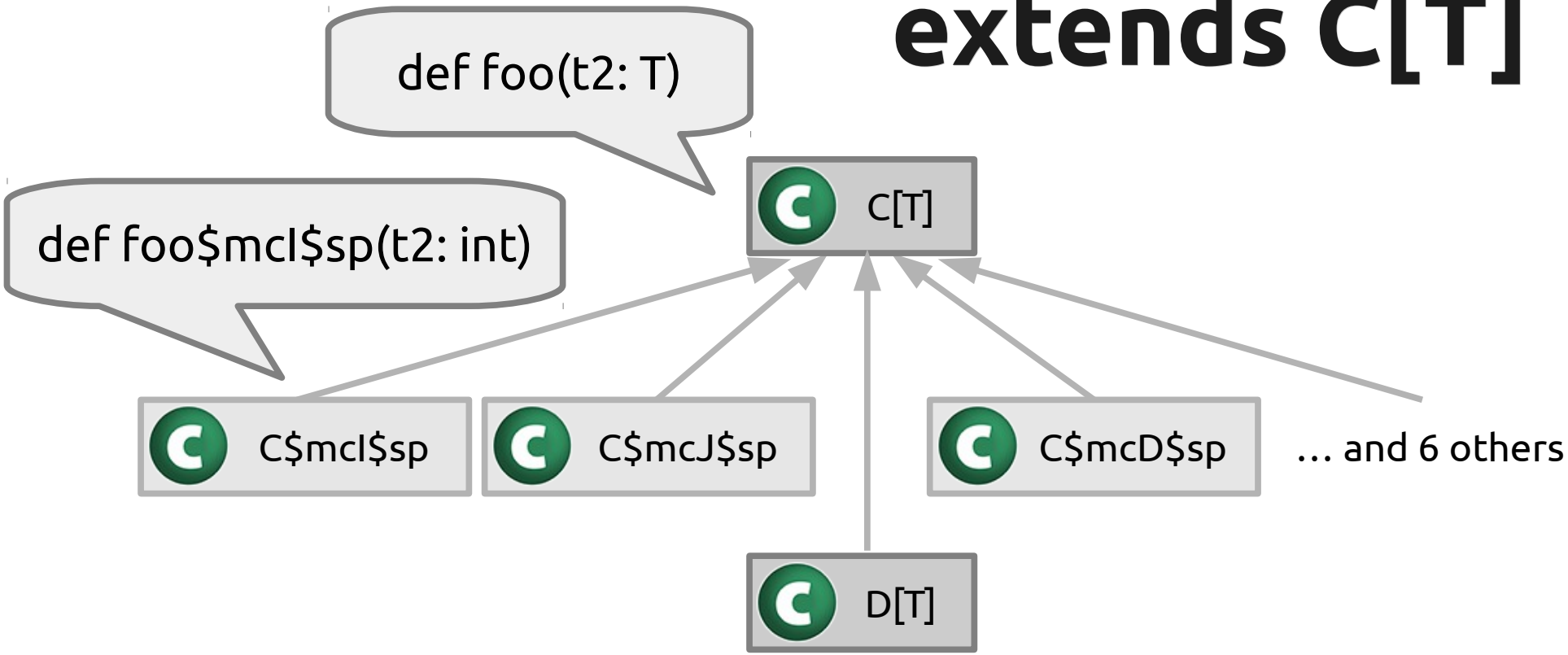
class D[@specialized T](t: T) extends C[T]



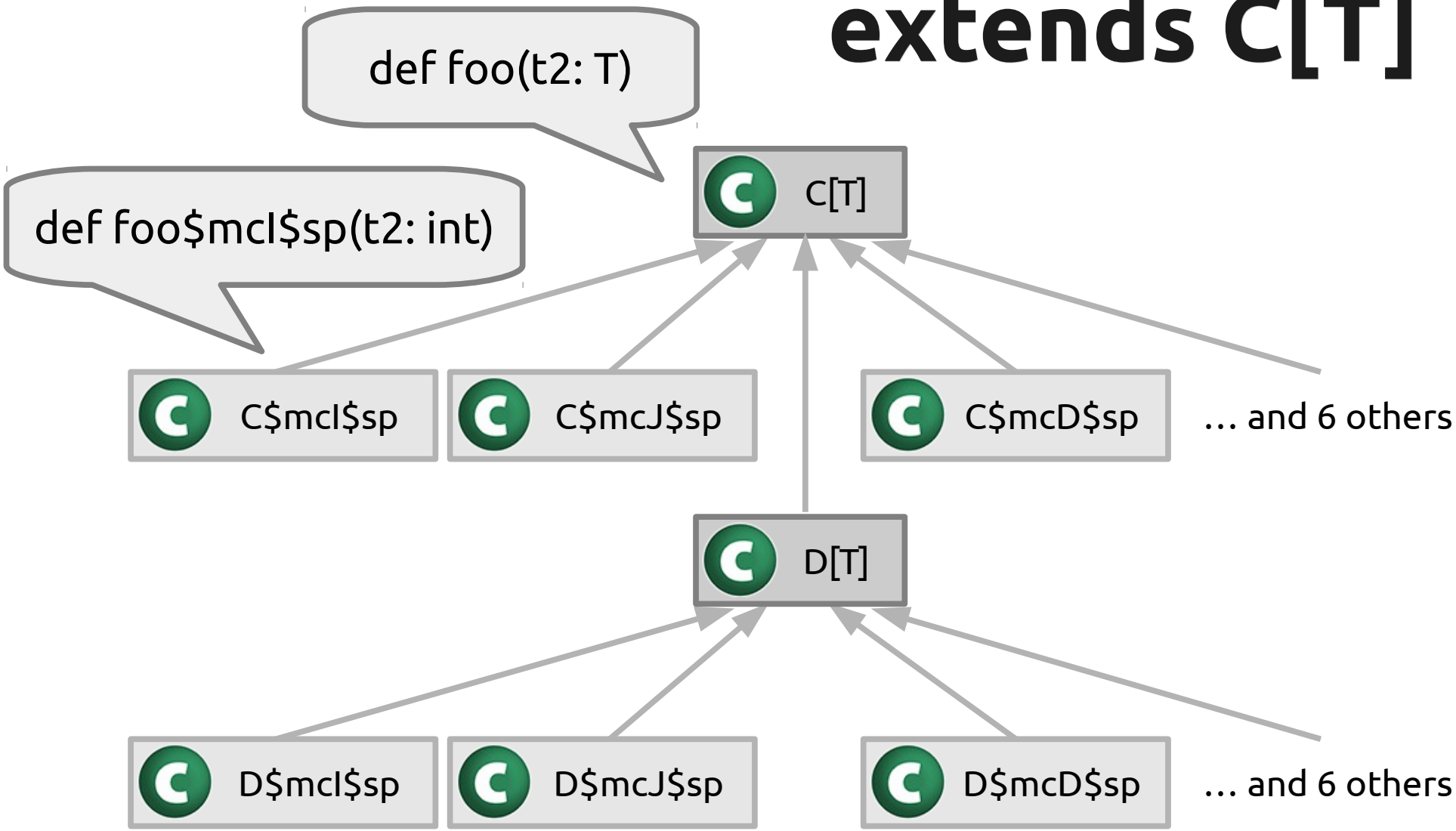
class D[@specialized T](t: T) extends C[T]



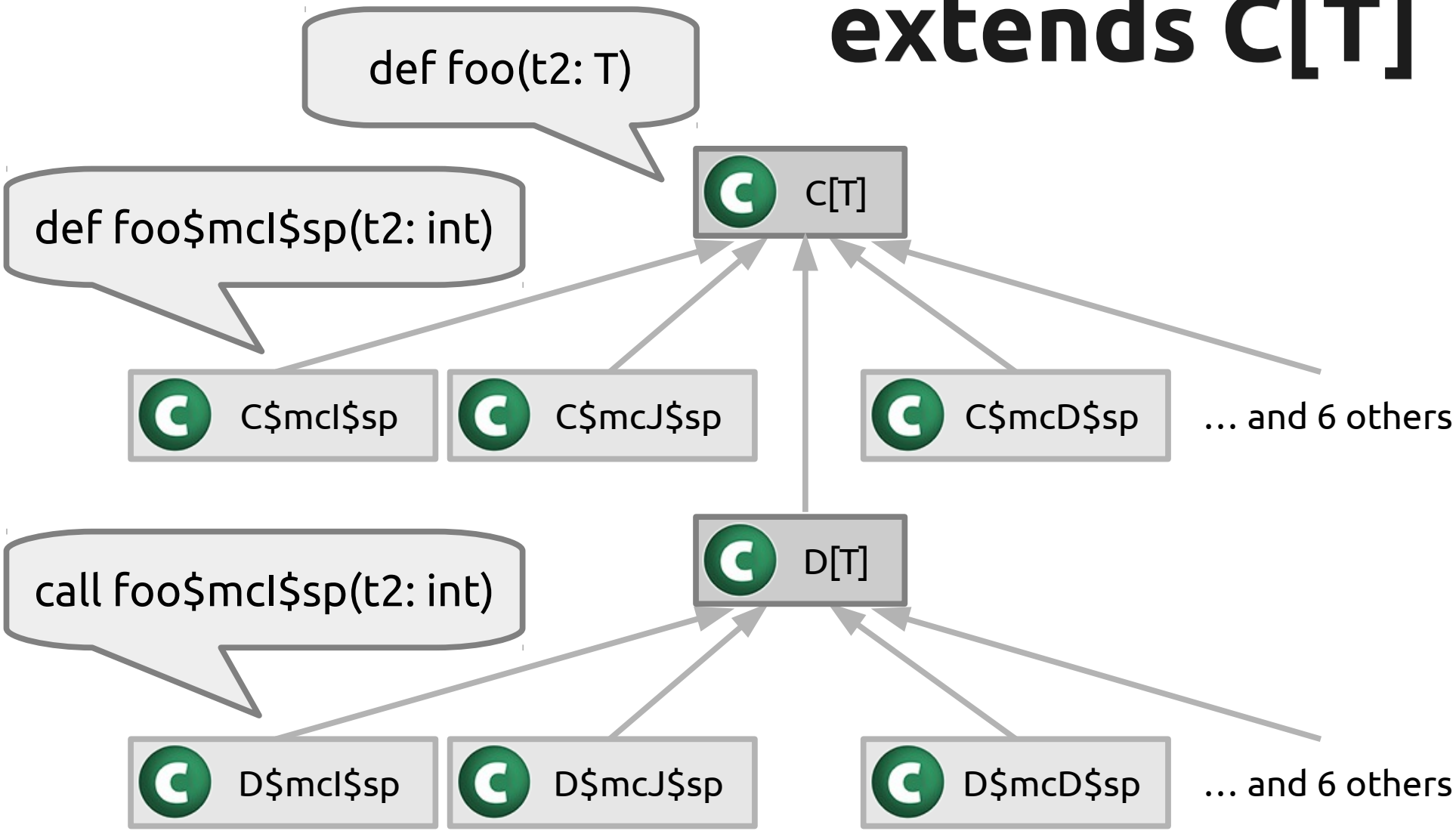
class D[@specialized T](t: T) extends C[T]



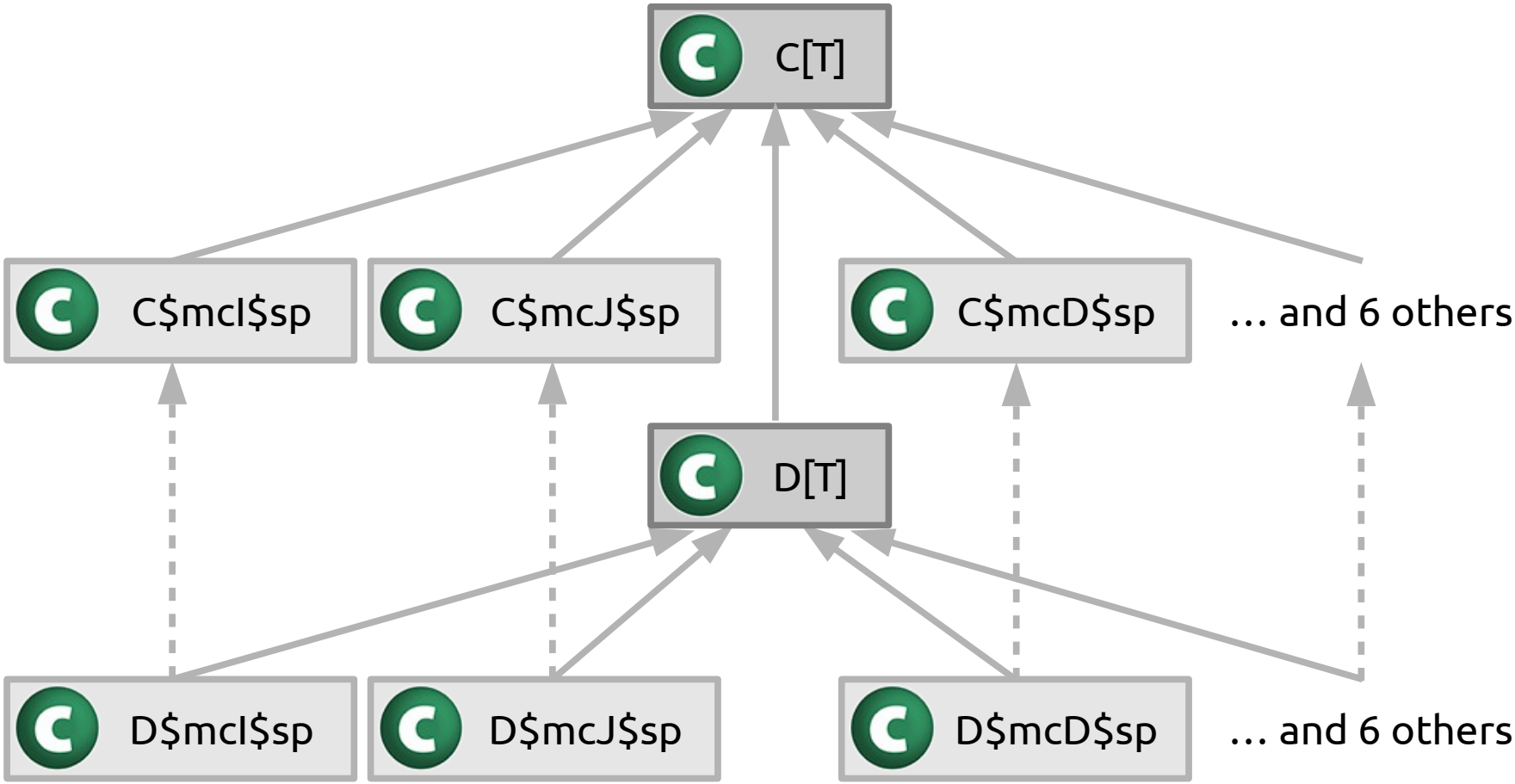
class D[@specialized T](t: T) extends C[T]



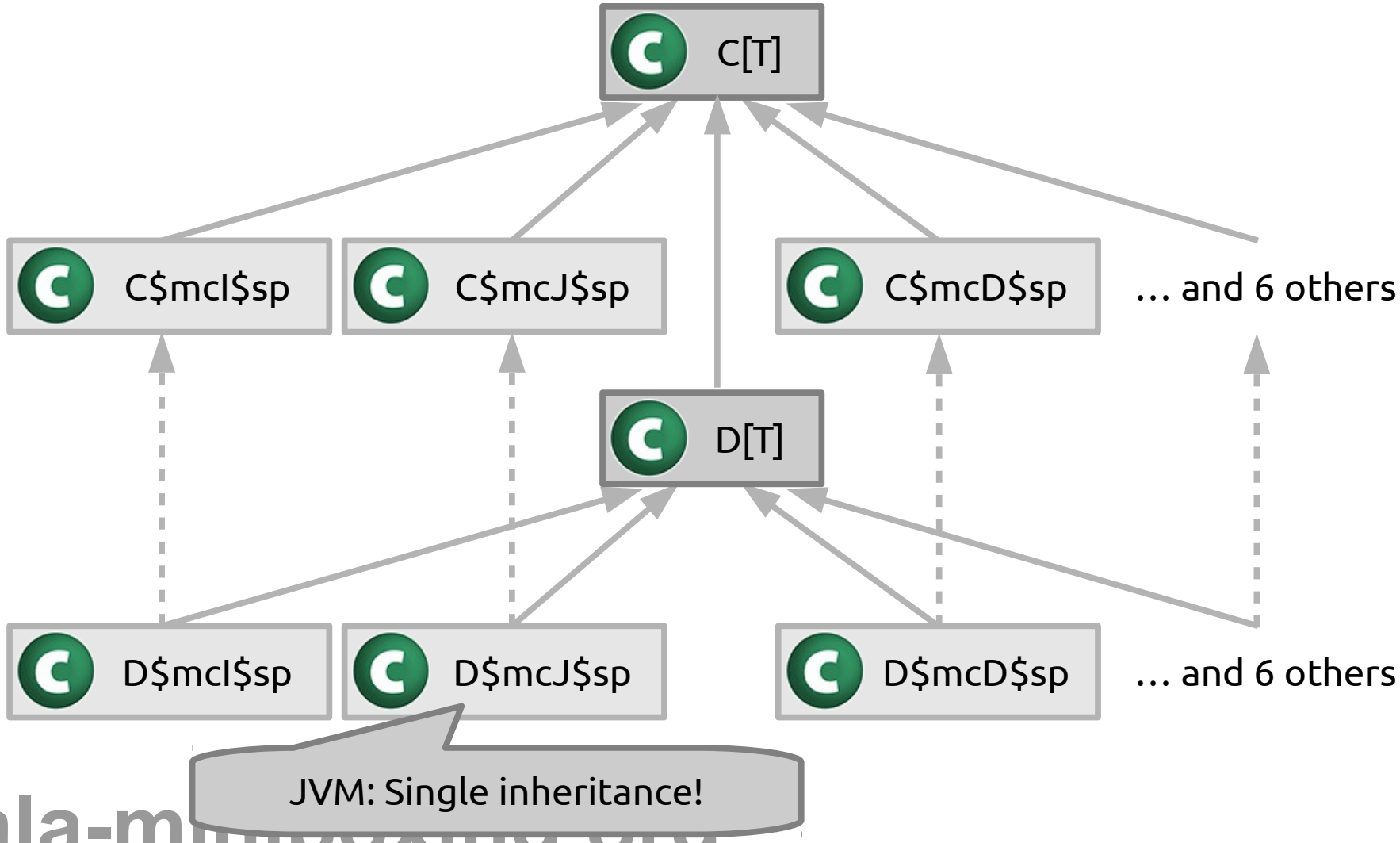
class D[@specialized T](t: T) extends C[T]



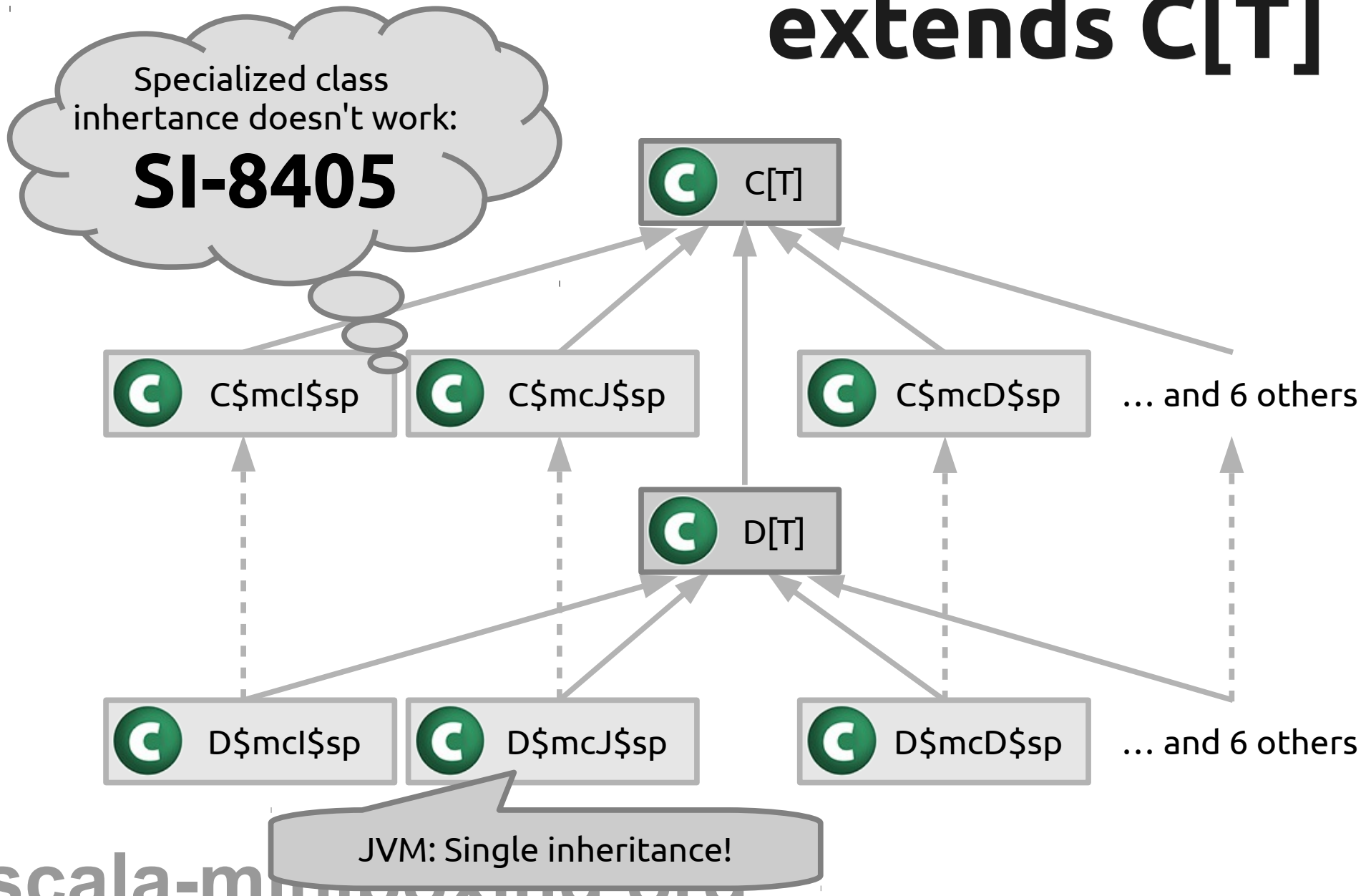
class D[@specialized T](t: T) extends C[T]



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



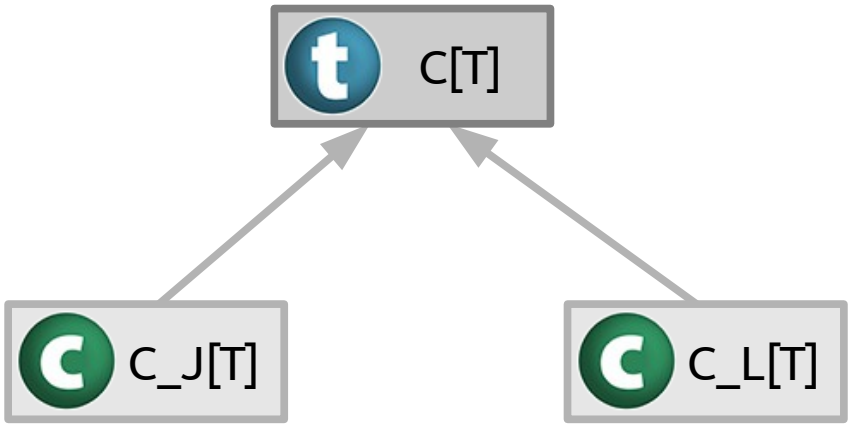
class D[@specialized T](t: T) extends C[T]



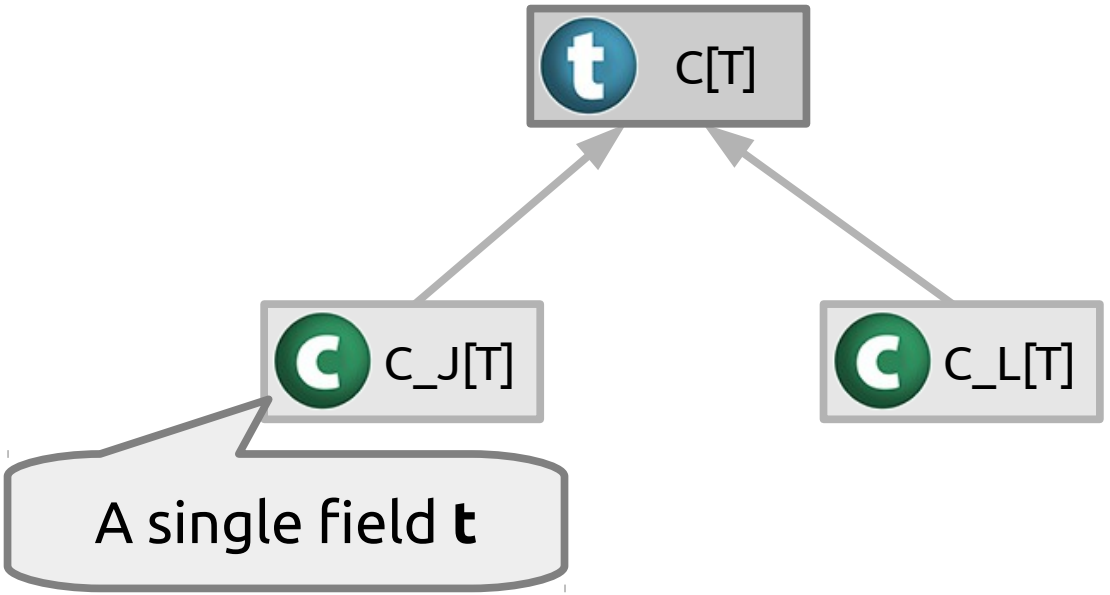
@miniboxing

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

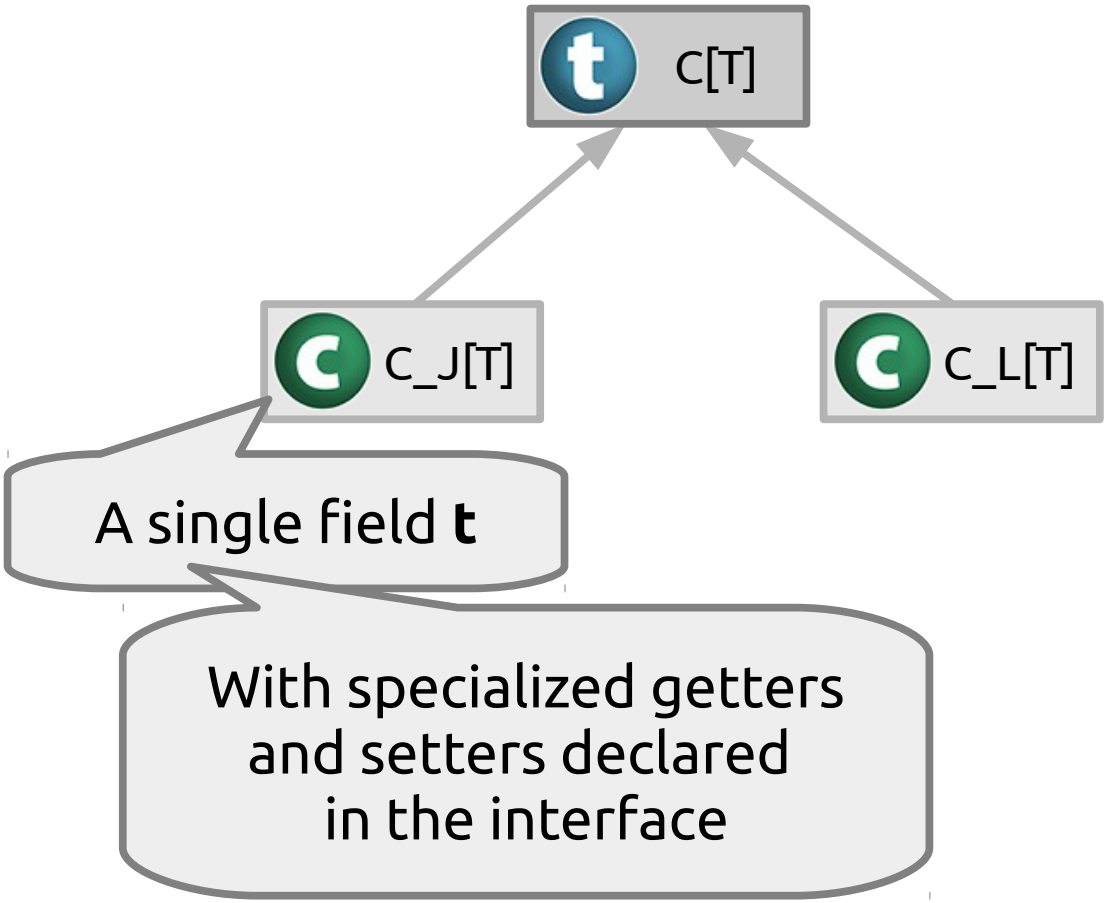
class C[@miniboxed T](t: T)



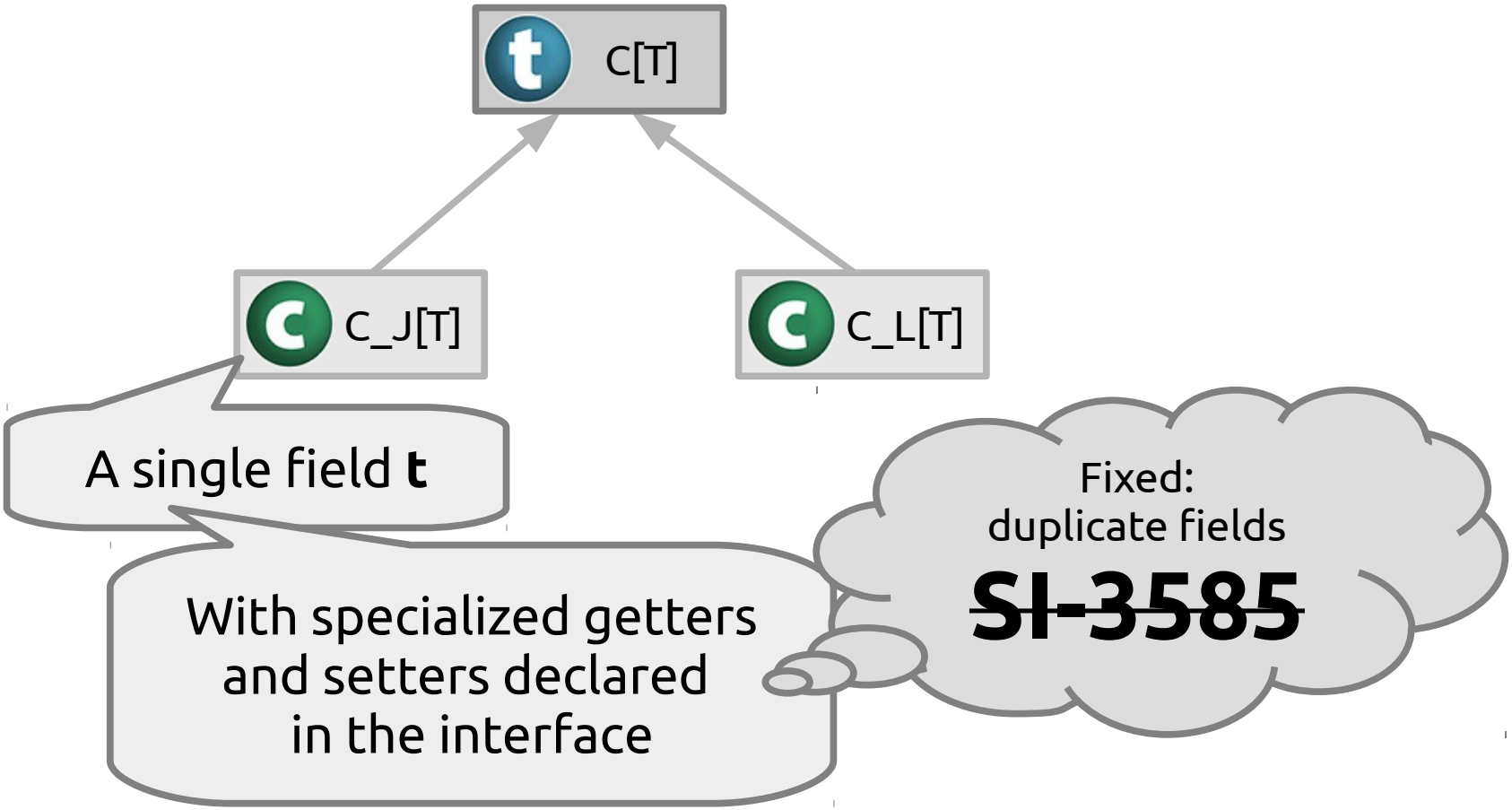
class C[@miniboxed T](t: T)



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

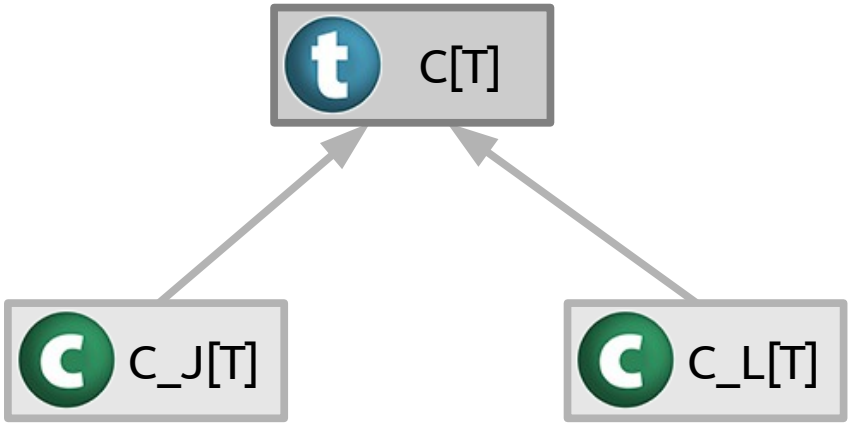


class C[@miniboxed T](t: T)

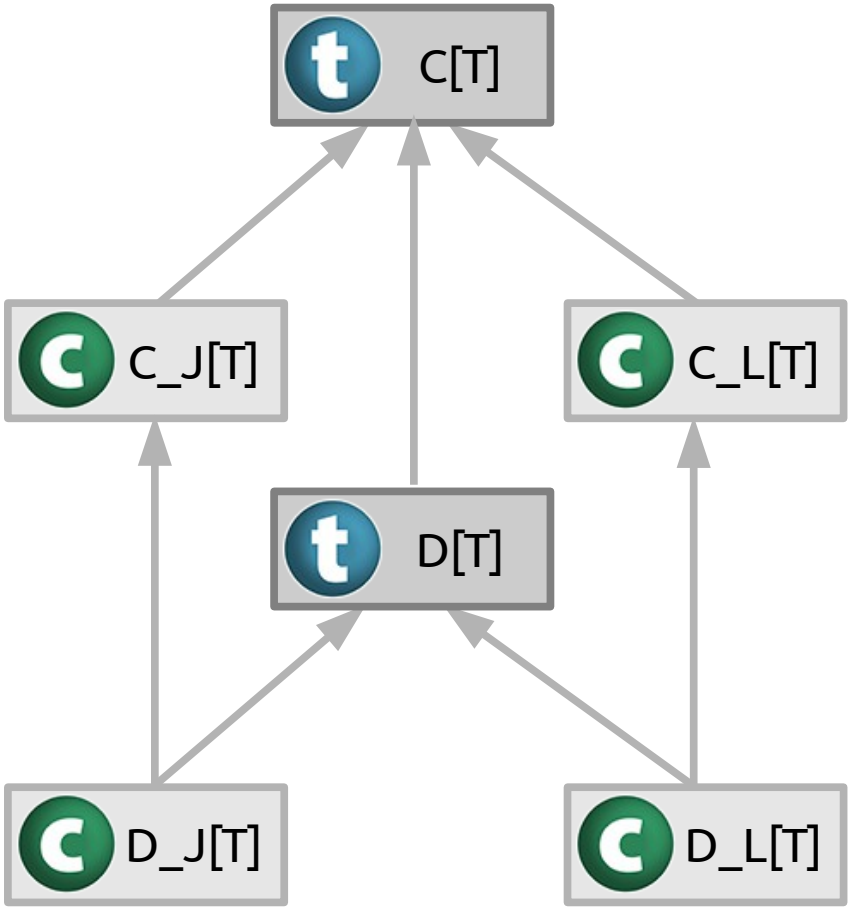


```
class D[@miniboxed T](t: T)
extends C[T]
```

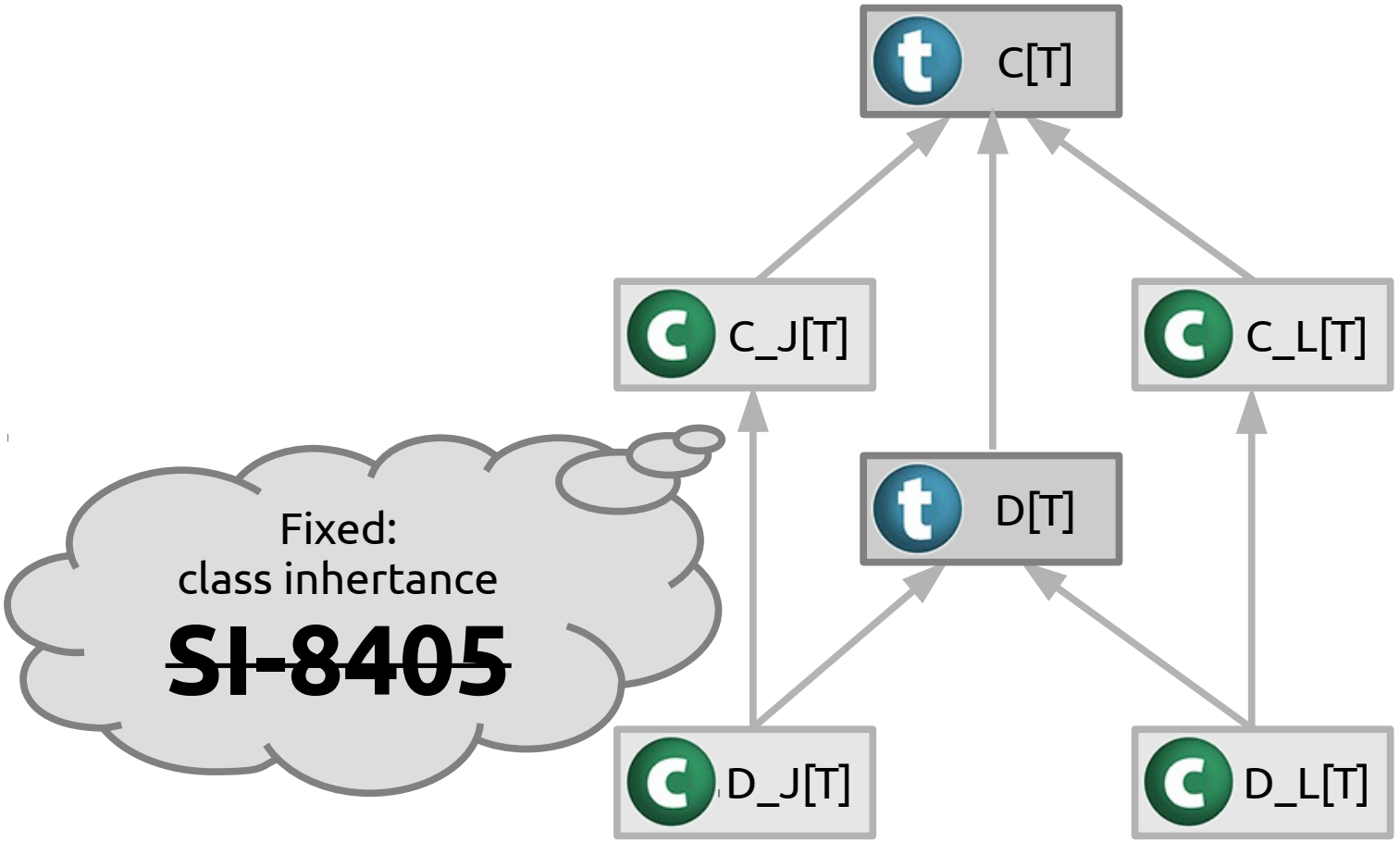
class D[@miniboxed T](t: T) extends C[T]



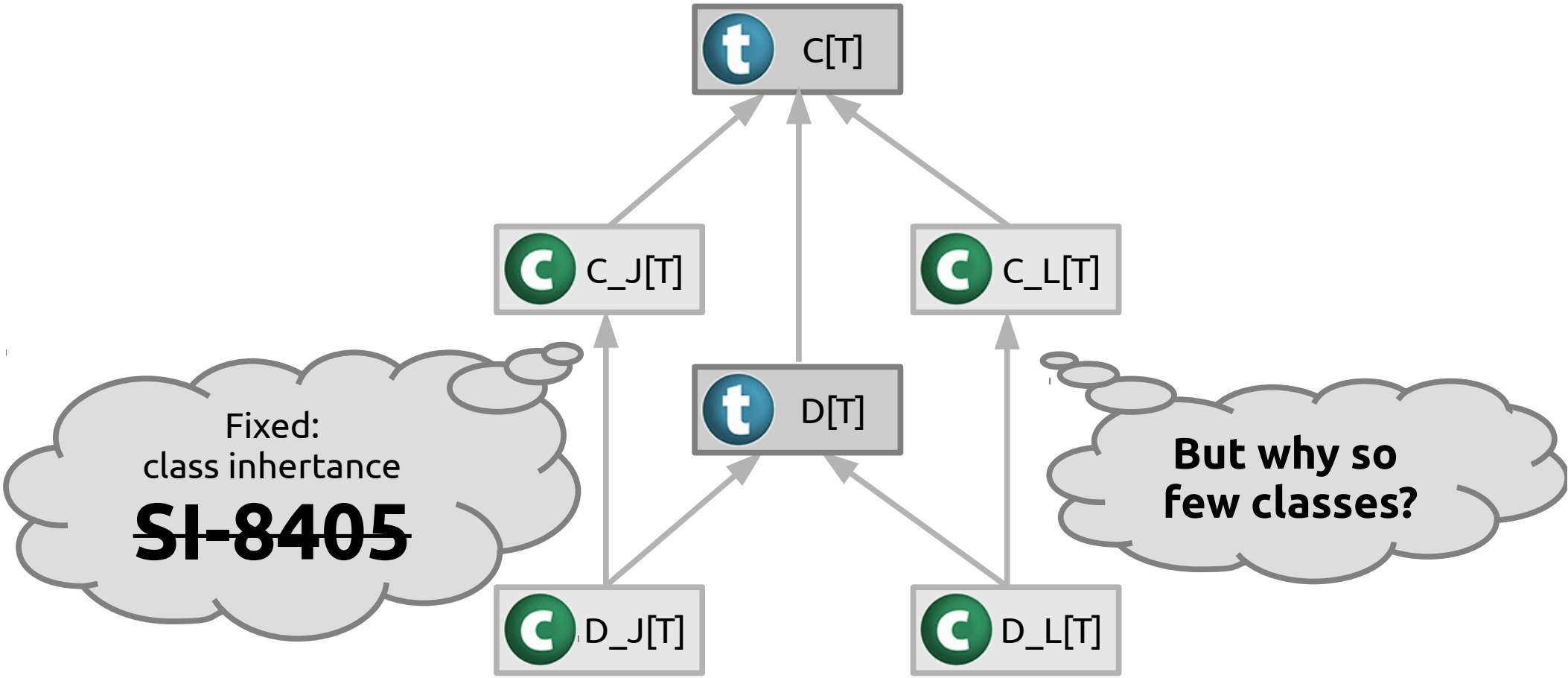
class D[@miniboxed T](t: T) extends C[T]



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



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



@miniboxed = @specialized

- the limitations
- bytecode bloat

Where's the

bytecode bloat?

```
trait Function2[-T1, -T2, +R]
```

Where's the

bytecode bloat?

```
trait Function2[-T1, -T2, +R]
```

Unit, Boolean, Byte, Char, Short,
Int, Long, Float, Double, Object

Where's the

bytecode bloat?

```
trait Function2[-T1, -T2, +R]
```

Unit, Boolean, Byte, Char, Short,
Int, Long, Float, Double, Object

- fully specializing `Function2` → **10^3 traits**

Where's the

bytecode bloat?

```
trait Function2[-T1, -T2, +R]
```

Unit, Boolean, Byte, Char, Short,
Int, Long, Float, Double, Object

- fully specializing `Function2` → **10^3 traits**
- **upfront bytecode (not on-demand)**

Where's the bytecode bloat?

```
trait Function2[-T1, -T2, +R]
```

Unit, Boolean, Byte, Char, Short,
Int, Long, Float, Double, Object

- fully specializing `Function2` → **10^3 traits**
- **upfront bytecode (not on-demand)**
- too much for the Scala library

Where's the
bytecode bloat?

```
trait Function2[-T1, -T2, +R]
```

Unit, Boolean, Byte, Char, Short,
Int, Long, Float, Double, Object

- fully specializing `Function2` → **10^3 traits**
- **upfront bytecode (not on-demand)**
- too much for the Scala library

Still want to distribute it
via maven, not **via torrents**

But...

we can do better

But...

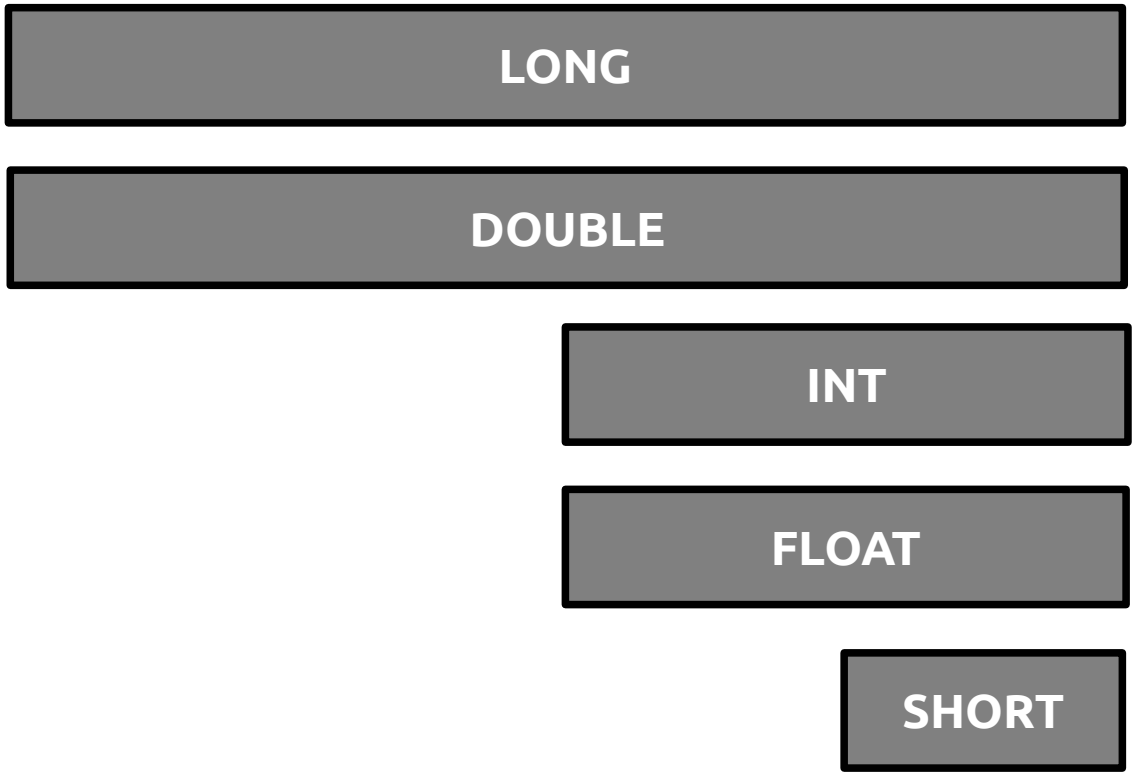
we can do better

One day in 2012 Miguel Garcia walked into my office and said: *"From a low-level perspective, there are only values and pointers. Maybe you can use that!"*

But...

we can do better

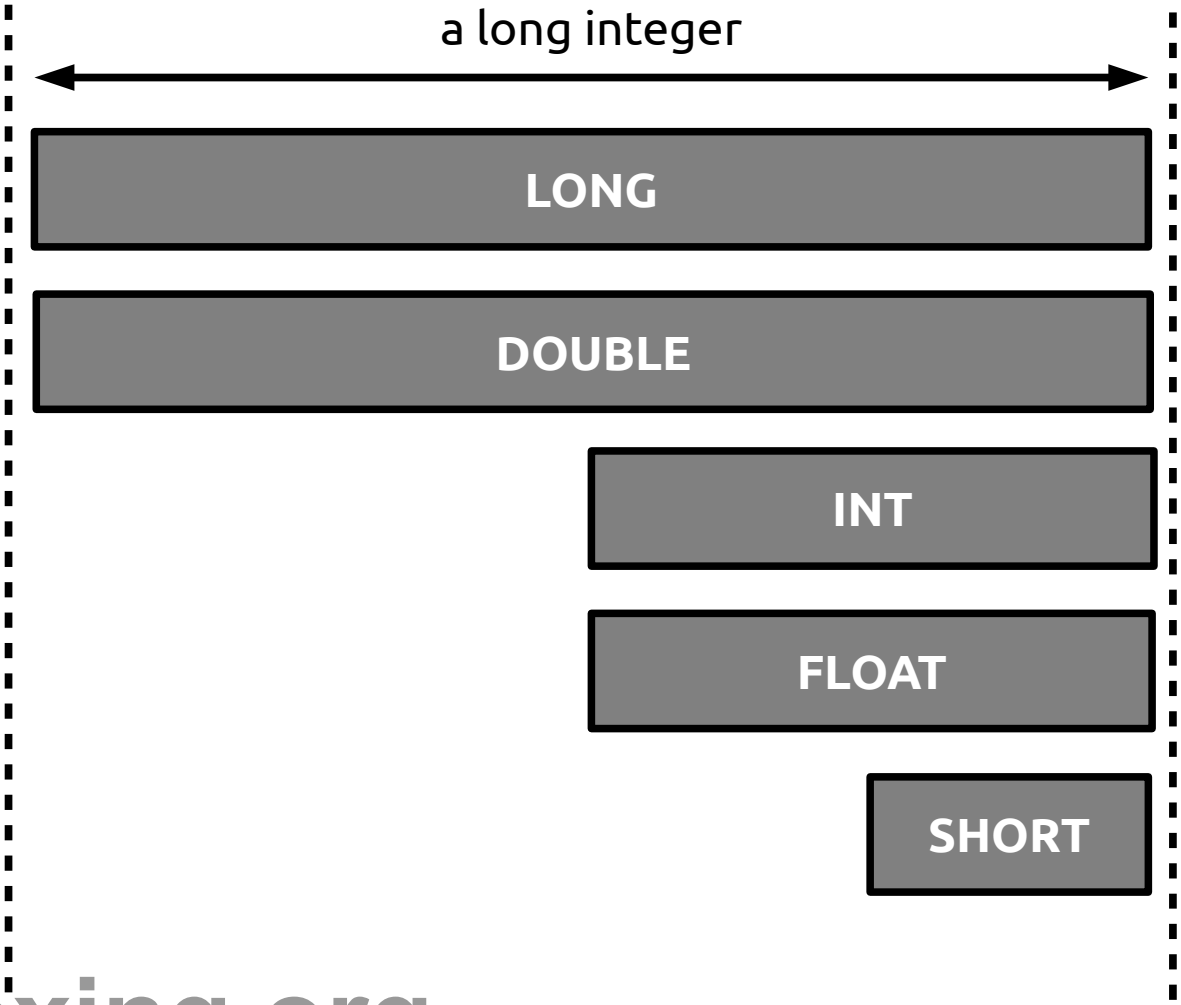
One day in 2012 Miguel Garcia walked into my office and said: *"From a low-level perspective, there are only values and pointers. Maybe you can use that!"*



But...

we can do better

One day in 2012 Miguel Garcia walked into my office and said: *"From a low-level perspective, there are only values and pointers. Maybe you can use that!"*



And then the

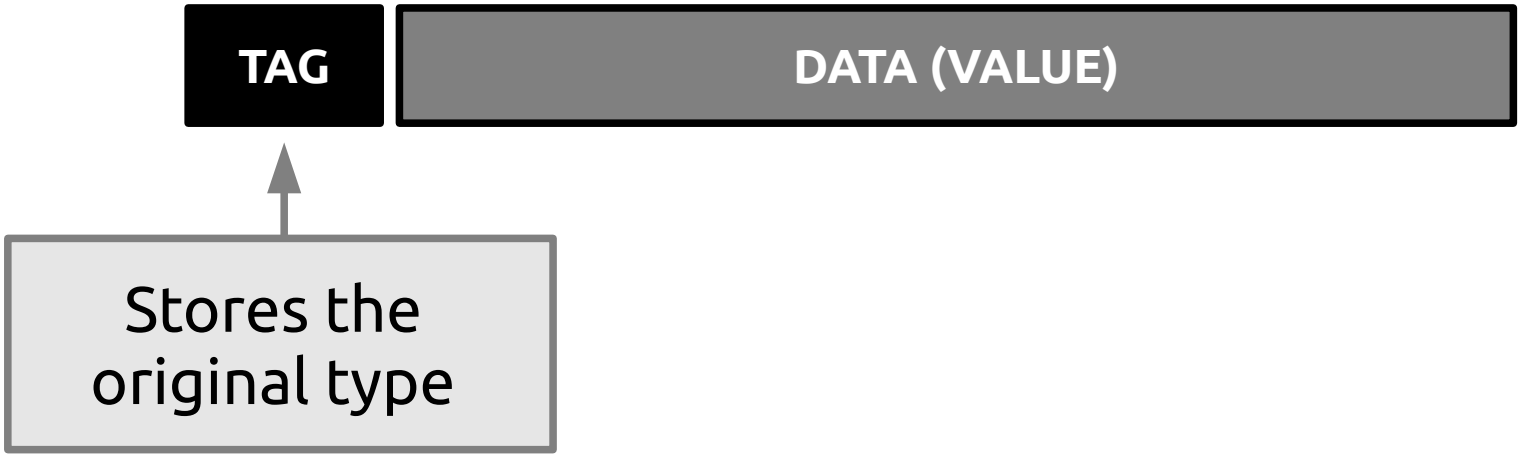
And then the
idea was born

And then the idea was born

it started from the **tagged union**

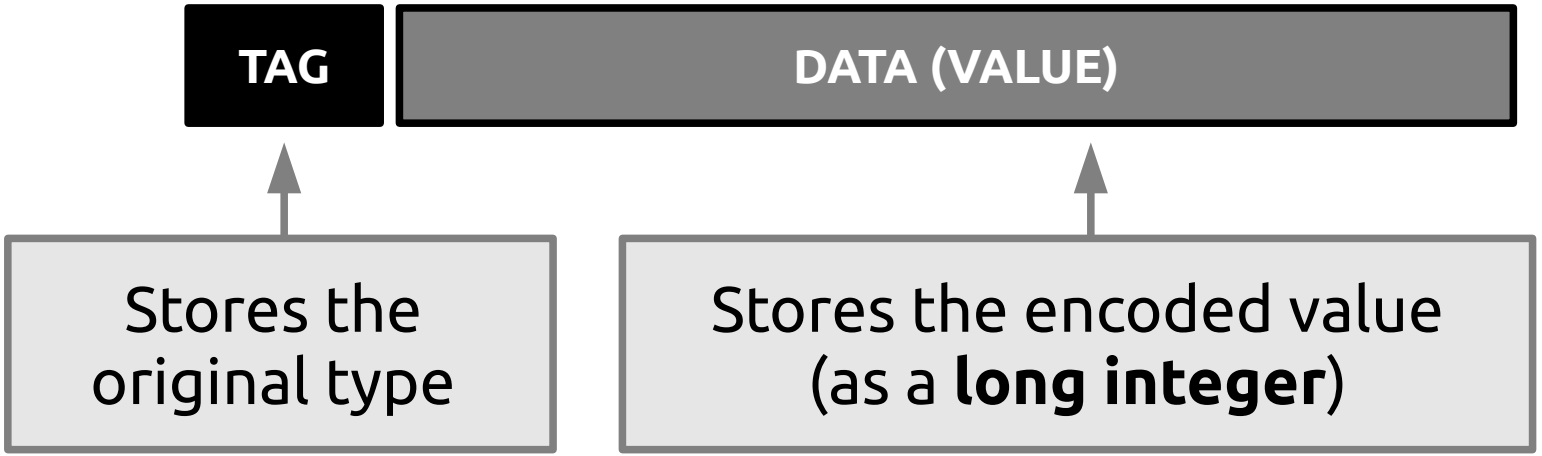
And then the idea was born

it started from the **tagged union**



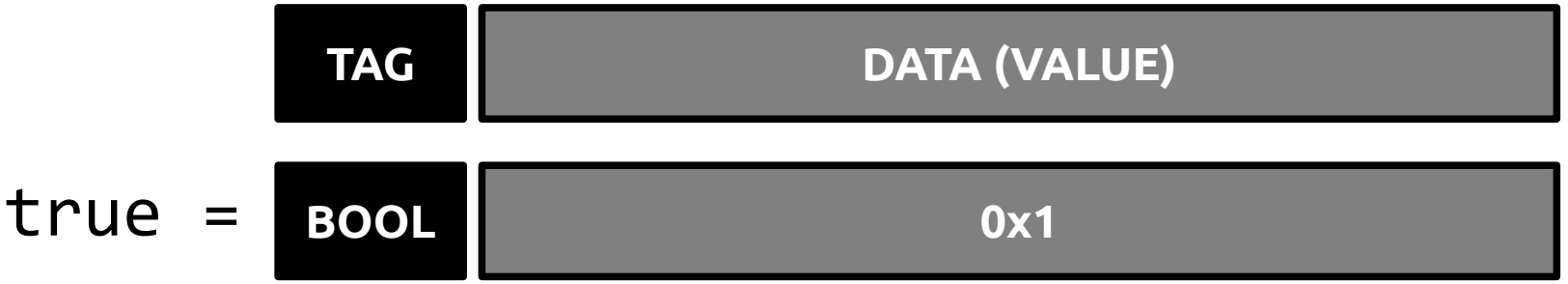
And then the idea was born

it started from the **tagged union**



And then the idea was born

it started from the **tagged union**



And then the idea was born

it started from the **tagged union**

	TAG	DATA (VALUE)
`true` =	BOOL	0x1
`42` =	INT	0x2A

And then the idea was born

it started from the **tagged union**

	TAG	DATA (VALUE)
`true` =	BOOL	0x1
`42` =	INT	0x2A
`5.0f` =	FLOAT	bit representation

And then the idea was born

it started from the **tagged union**

- somewhat similar to a boxed object

And then the idea was born

it started from the **tagged union**

- somewhat similar to a boxed object
- but not in the heap memory

And then the idea was born

it started from the **tagged union**

- somewhat similar to a boxed object
- but not in the heap memory
- direct access to the value

And then the idea was born

it started from the **tagged union**

- somewhat similar to a boxed object
- but not in the heap memory
- direct access to the value

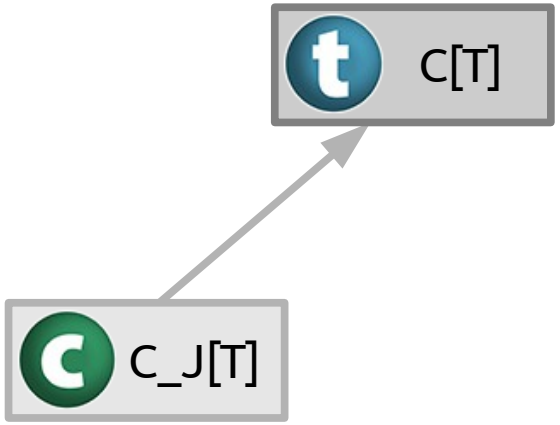
Same benefits as for unboxed values

And then the idea was born

we can reduce the number of variants

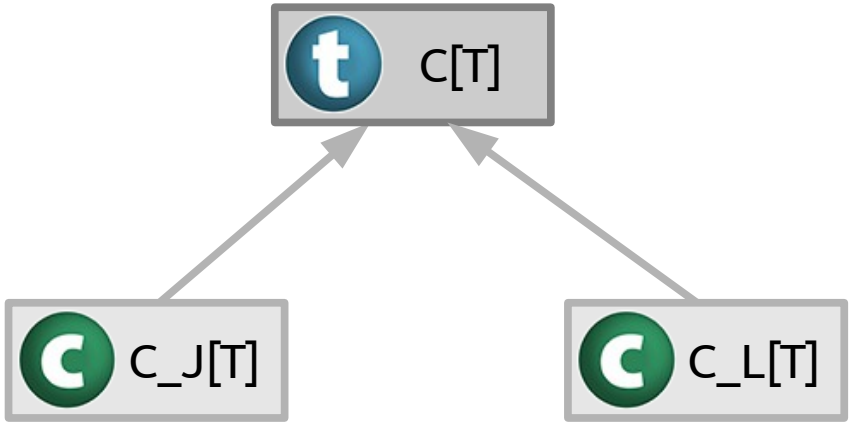
And then the idea was born

we can reduce the number of variants



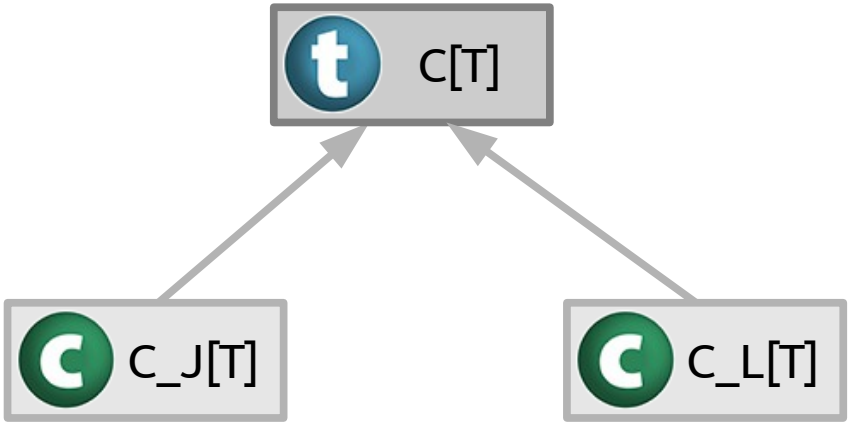
And then the idea was born

we can reduce the number of variants



And then the idea was born

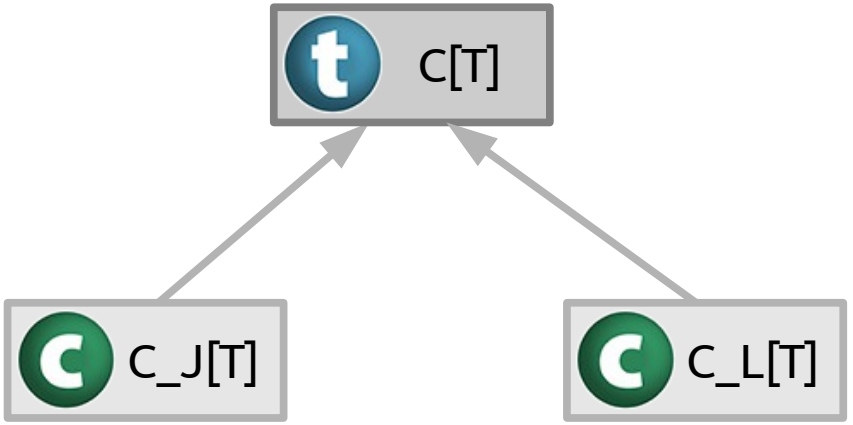
we can reduce the number of variants



"From a low-level perspective, there are only values and pointers."

And then the idea was born

we can reduce the number of variants



Let's take an example

"From a low-level perspective, there are only values and pointers."

Let's take

an example


```
def choice[@miniboxed T](t1: T,
                         t2: T): T =
  if (util.Random.nextBoolean())
    t1
  else
    t2
```

Let's take
an example


```
def choice[@miniboxed T](t1: T,
                         t2: T): T =
  if (util.Random.nextBoolean())
    t1
  else
    t2
```

Let's take
an example


```
def choice[@miniboxed T](t1: T,
                         t2: T): T =
  if (util.Random.nextBoolean())
    t1
  else
    t2
```

We'll have a version for primitive types

Let's take

an example


```
def choice_J[T](t1: ...,
                t2: ...): ... =
  if (util.Random.nextBoolean())
    t1
  else
    t2
```

Let's take
an example


```
def choice_J[T](t1: ...,
                t2: ...): ... =
  if (util.Random.nextBoolean())
    t1
  else
    t2
```

But what's the signature?

Let's take

an example


```
def choice_J[T](t1: (Tag,Value),
                t2: (Tag,Value)):(Tag,Value)=
  if (util.Random.nextBoolean())
    t1
  else
    t2
```

Let's take
an example


```
def choice_J[T](t1: (Tag,Value),
                t2: (Tag,Value)):(Tag,Value)=
  if (util.Random.nextBoolean())
    t1
  else
    t2
```

This is naive tagged union

Let's take

an example

That's wasteful: we carry the tag for T twice

```
def choice_J[T](t1: (Tag,Value),
                t2: (Tag,Value)):(Tag,Value)=
  if (util.Random.nextBoolean())
    t1
  else
    t2
```

This is naive tagged union

Let's take

an example

That's wasteful: we carry the tag for T twice

```
def choice_J[T](t1: (Tag,Value),
                t2: (Tag,Value)):(Tag,Value)=
  if (util.Random.nextBoolean())
    t1
  else
    t2
```

And we even return it, despite the caller having passed it

This is naive tagged union

Let's take

an example

That's wasteful: we carry the tag for T twice

```
def choice_J[T](t1: (Tag,Value),
                t2: (Tag,Value)):(Tag,Value)=
  if (util.Random.nextBoolean())
    t1
  else
    t2
```

And we even return it, despite the caller having passed it

Insight: we're in a statically typed language, **use that**!

Let's take

an example


```
def choice_J[T](T_Tag: Tag, t1: Value,
                            t2: Value):Value=
  if (util.Random.nextBoolean())
    t1
  else
    t2
```

Let's take

an example

T_Tag corresponds to the type parameter

```
def choice_J[T](T_Tag: Tag, t1: Value,
                            t2: Value):Value=
  if (util.Random.nextBoolean())
    t1
  else
    t2
```

Let's take an example

Sort of a class tag

T_Tag corresponds to the type parameter

```
def choice_J[T](T_Tag: Tag, t1: Value,
                            t2: Value):Value=
  if (util.Random.nextBoolean())
    t1
  else
    t2
```

Let's take an example

Sort of a class tag

T_Tag corresponds to the type parameter

```
def choice_J[T](T_Tag: Tag, t1: Value,
                            t2: Value):Value=
  if (util.Random.nextBoolean())
    t1
  else
    t2
```

Encoded as **Long**

Let's take

an example


```
def choice_J[T](T_Tag: Byte, t1: Long,
                              t2: Long): Long=
  if (util.Random.nextBoolean())
    t1
  else
    t2
```

So, back to the bytecode issue


```
trait Function2[-T1, -T2, +R]
```

- with specialization this produces **10^3 traits**

So, back to the bytecode issue


```
trait Function2[-T1, -T2, +R]
```

- with specialization this produces **10^3 traits**
- with miniboxing only **2^3** (100x less bytecode)

So, back to the bytecode issue


```
trait Function2[-T1, -T2, +R]
```

- with specialization this produces **10^3 traits**
- with miniboxing only **2^3** (100x less bytecode)
- so we expect it will be usable on the library

So, back to the bytecode issue


```
trait Function2[-T1, -T2, +R]
```

- with specialization this produces **10^3 traits**
- with miniboxing only **2^3** (100x less bytecode)
- so we expect it will be usable on the library

But before we wrap this up

@miniboxed = @specialized

- the limitations
- bytecode bloat

@miniboxed = @specialized

- the limitations
- bytecode bloat

before the benchmarks:
some internals

this is where it
gets complicated

this is where it
gets complicated

time for the
hardhats

Let's revisit

the example


```
def choice[@ miniboxed T](t1: T,
                          t2: T): T =
  if (nextBoolean() && t1.hashCode != 13)
    t1
  else
    t2
```

Let's revisit

the example


```
def choice_J[T](T_Tag: Tag, t1: Long,
                                t2: Long): Long =
  if (nextBoolean() && t1.hashCode != 13)
    t1
  else
    t2
```

Let's revisit
the example


```
def choice_J[T](T_Tag: Tag, t1: Long,
                            t2: Long): Long =
  if (nextBoolean() && t1.hashCode != 13)
    t1
  else
    t2
```

Let's revisit

the example


```
t1.hashCode
```

Let's revisit
the example


```
t1.hashCode
```

- how to translate this?

Let's revisit
the example


```
t1.hashCode
```

- how to translate this?
- when t1 is miniboxed?

Let's revisit

the example


```
scala> true.hashCode
res0: Int = 1231

scala> false.hashCode
res1: Int = 1237
```

Let's revisit
the example


```
scala> true.hashCode
res0: Int = 1231

scala> false.hashCode
res1: Int = 1237
```

So calling hashCode on the Long won't work

Let's revisit

the example


```
minibox2box[T](T_Tag, t).hashCode
```

Let's revisit
the example


```
minibox2box[T](T_Tag, t).hashCode
```

- conversions: minibox2box, box2minibox

Let's revisit
the example


```
minibox2box[T](T_Tag, t).hashCode
```

- conversions: minibox2box, box2minibox
- hash code

Let's revisit
the example


```
minibox2box[T](T_Tag, t).hashCode
```

- conversions: minibox2box, box2minibox
- hash code
  - box the value back

Let's revisit
the example


```
minibox2box[T](T_Tag, t).hashCode
```

- conversions: minibox2box, box2minibox
- hash code
  - box the value back
  - execute the hashCode method

Performance-wise

Performance-wise

this is okay

Performance-wise

this is okay

- conversions

Performance-wise

this is okay

- conversions
  - between minboxed and unboxed integer types
    - free on x64

Performance-wise

this is okay

- conversions
  - between minboxed and unboxed integer types
    - free on x64
  - between miniboxed and floating point types
    - low overhead (not free*)

* improved translation (thanks Rex!)

Performance-wise

this is okay

- conversions
  - between minboxed and unboxed integer types
    - free on x64
  - between miniboxed and floating point types
    - low overhead (not free*)
  - between miniboxed and boxed values
    - avoided by @miniboxed!

* improved translation (thanks Rex!)

the last example
before the benchmarks

The last example

before the benchmarks


```
def list[@miniboxed T](t1: T,
                       t2: T): List[T] =
  List[T](t1, t2)
```

The last example

before the benchmarks


```
def list[@miniboxed T](t1: T,
                       t2: T): List[T] =
  List[T](t1, t2)
```

```
  List.apply[T](t1, t2)
```

The last example

before the benchmarks


```
def list[@miniboxed T](t1: T,
                       t2: T): List[T] =
  List.apply[T](t1, t2)
```

The last example

before the benchmarks


```
def list[@miniboxed T](t1: T,
                       t2: T): List[T] =
  List.apply[T](t1, t2)
```

```
def list_J[T](T_Tag: Byte, t1: Long,
              t2: Long): ... =
  List[T].apply(t1, t2)
```

The last example

before the benchmarks


```
def list[@miniboxed T](t1: T,
                       t2: T): List[T] =
  List.apply[T](t1, t2)
```

```
def list_J[T](T_Tag: Byte, t1: Long,
              t2: Long): ... =
  List[T].apply(t1, t2)
```

How to transform List[T]?

The last example

before the benchmarks


```
List[T]
```

what is List[T] when T is miniboxed?

The last example

before the benchmarks


```
List[T]
```

what is List[T] when T is miniboxed?

- for specialization:

  [T ← Int] List[T] => List[Int]

The last example

before the benchmarks


```
List[T]
```

what is List[T] when T is miniboxed?

- for specialization:

  [T ← Int] List[T] => List[Int]

- for miniboxing:

  still **List[T]**

The last example
before the benchmarks


```
List[T]
```

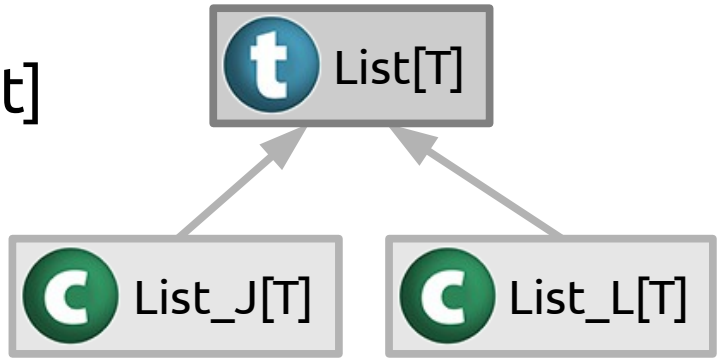
what is List[T] when T is miniboxed?

- for specialization:

  [T ← Int] List[T] => List[Int]

- for miniboxing:

  still **List[T]**



The last example
before the benchmarks


```
List[T]
```

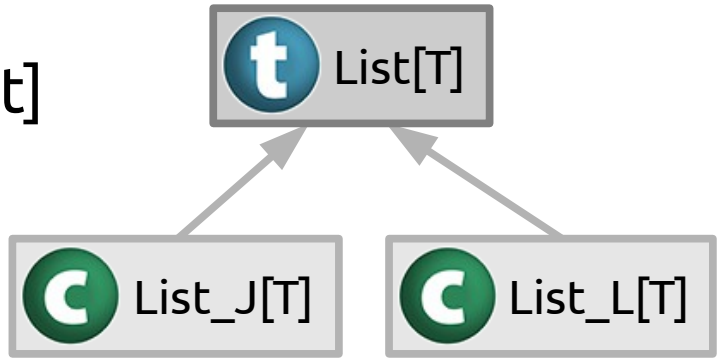
what is List[T] when T is miniboxed?

- for specialization:

  [T ← Int] List[T] => List[Int]

- for miniboxing:

  still **List[T]**



But List[T] is an interface,
we can still have an adapted
implementation class (List_J)

The last example

before the benchmarks


```
def list_J[T](T_Tag: Byte, t1: Long,
              t2: Long): List[T] =
  List.apply[T](t1, t2)
```

The last example

before the benchmarks


```
def list_J[T](T_Tag: Byte, t1: Long,
              t2: Long): List[T] =
  List.apply[T](t1, t2)
```

The last example

before the benchmarks


```
def list_J[T](T_Tag: Byte, t1: Long,
              t2: Long): List[T] =
  List.apply[T](t1, t2)
```

```
expected: T
found:    Long
```

The last example

before the benchmarks


```
def list_J[T](T_Tag: Byte, t1: Long,
              t2: Long): List[T] =
  List.apply[T](minibox2box[T](T_Tag,t1),
                minibox2box[T](T_Tag,t2))
```

The last example

before the benchmarks


```
def list_J[T](T_Tag: Byte, t1: Long,
              t2: Long): List[T] =
  List.apply[T](minibox2box[T](T_Tag,t1),
                minibox2box[T](T_Tag,t2))
```

The last example

before the benchmarks


```
def list_J[T](T_Tag: Byte, t1: Long,
              t2: Long): List[T] =
  List.apply[T](minibox2box[T](T_Tag,t1),
                minibox2box[T](T_Tag,t2))
```

```
def list_J[T](T_Tag: Byte, t1: Long,
              t2: Long): List[T] =
  List.apply_J[T](T_Tag, t1, t2)
```

It turns out there's

more here

It turns out there's more here

- essentially a very general mechanism

It turns out there's more here

- essentially a very general mechanism
- and Eugene (xeno-by) gave it a shot

It turns out there's more here

- essentially a very general mechanism
- and Eugene (xeno-by) gave it a shot
  - in **a week** he implemented

It turns out there's more here

- essentially a very general mechanism
- and Eugene (xeno-by) gave it a shot
  - in **a week** he implemented
  - a **value class** plugin

It turns out there's more here

- essentially a very general mechanism
- and Eugene (xeno-by) gave it a shot
  - in **a week** he implemented
  - a **value class** plugin
  - with **multi-param** value classes

It turns out there's more here

- essentially a very general mechanism
- and Eugene (xeno-by) gave it a shot
  - in **a week** he implemented
  - a **value class** plugin
  - with **multi-param** value classes

https://github.com/miniboxing/value-plugin

It turns out there's more here

- essentially a very general mechanism
- and Eugene (xeno-by) gave it a shot
  - in **a week** he implemented
  - a **value class** plugin
  - with **multi-param** value classes

https://github.com/miniboxing/value-plugin

But I won't go into the theory, that's another talk

Benchmarks

Linked List

Benchmarks

on the Scala library

- work with Aymeric Genet (github: @MelodyLucid)
- mock-up of Scala linked list
  - Function1 / Function2 / Tuple2
  - Traversable / TraversableLike
  - Iterator / Iterable / IterableLike
  - LinearSeqOptimized
  - Builder / CanBuildFrom

Benchmarks

on the Scala library

- work with Aymeric Genet (github: @MelodyLucid)
- mock-up of Scala linked list
  - Function1 / Function2 / Tuple2
  - Traversable / TraversableLike
  - Iterator / Iterable / IterableLike
  - LinearSeqOptimized
  - Builder / CanBuildFrom

All the things you hate about the library are there!

Benchmarks

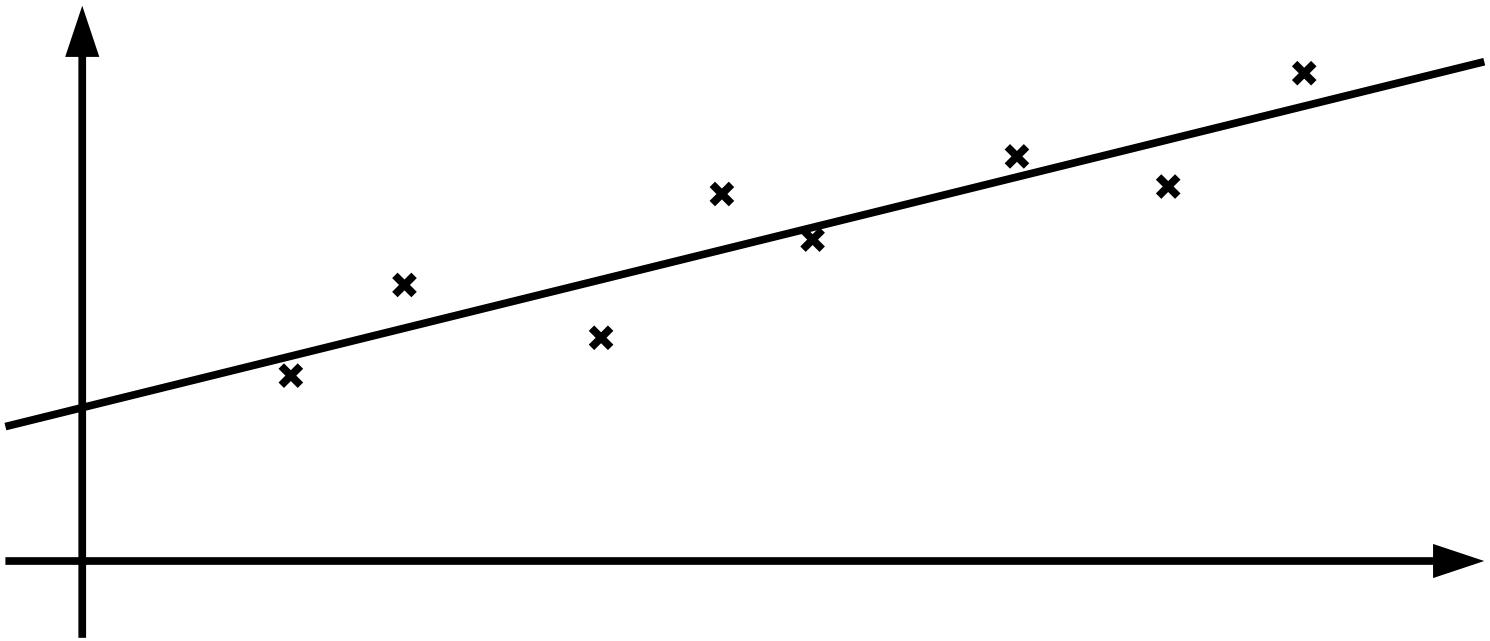
on the Scala library

- benchmark: Least Squares Method

Benchmarks

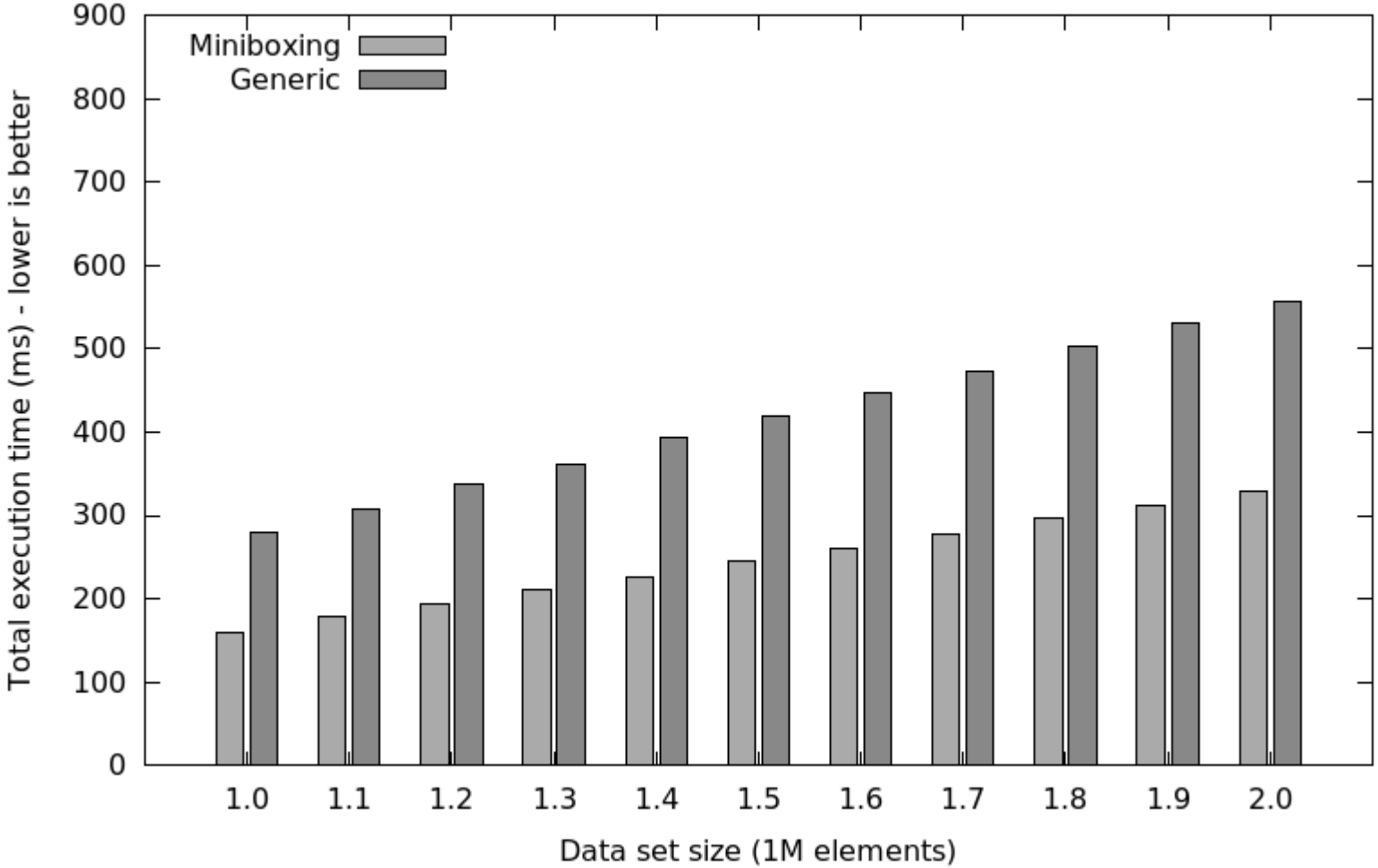
on the Scala library

- benchmark: Least Squares Method



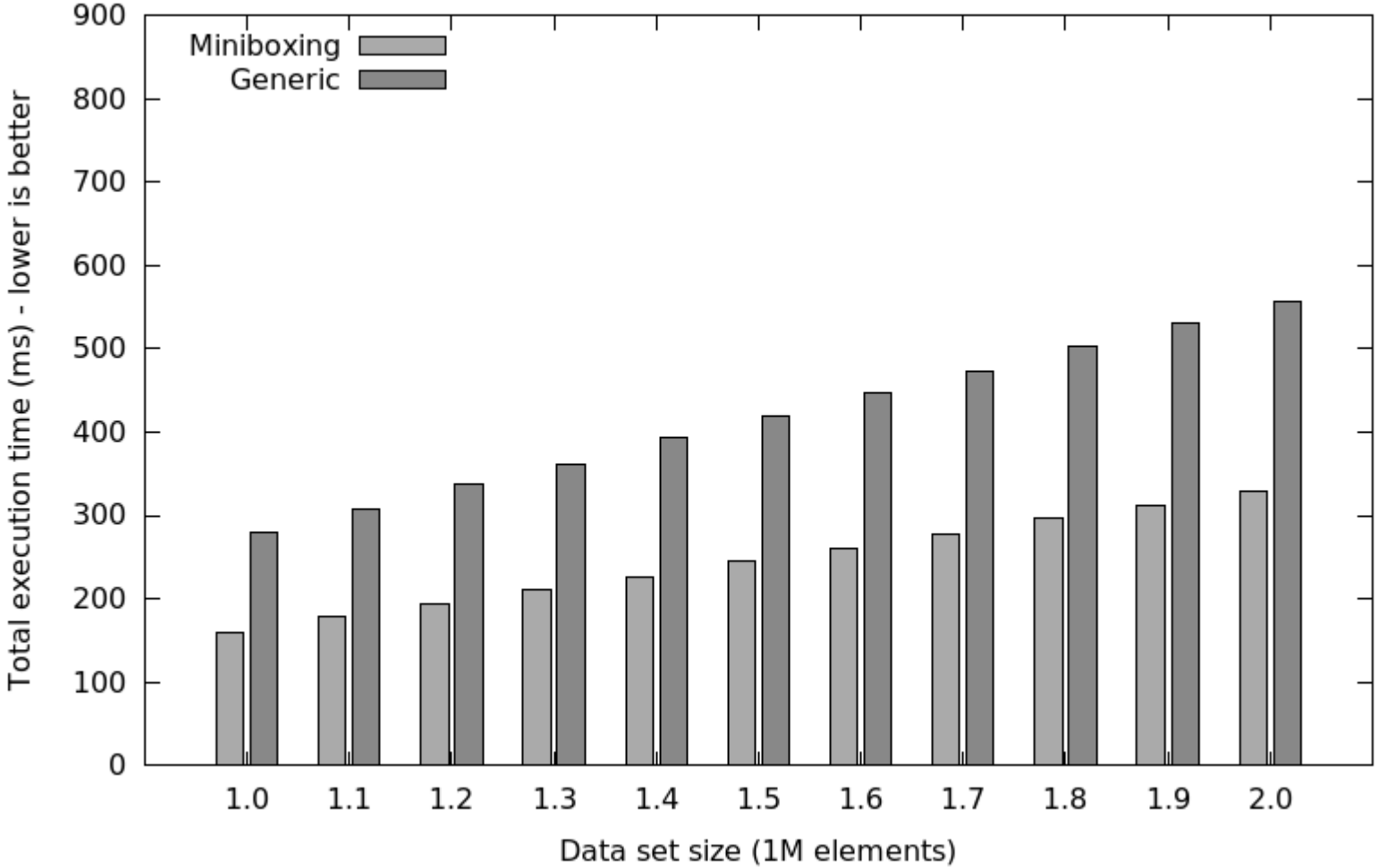
Benchmarks

on the Scala library (inf. heap)



Benchmarks

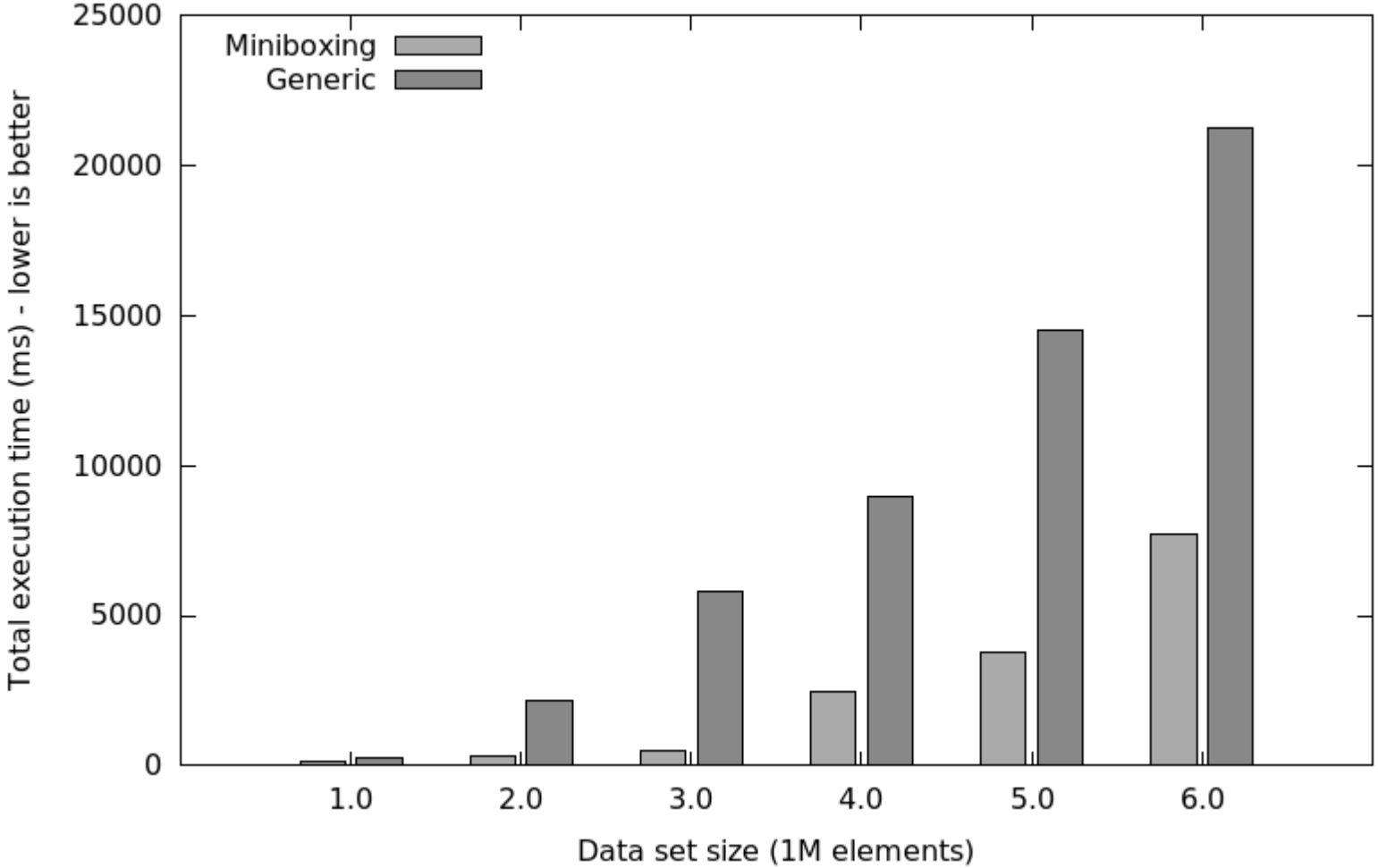
on the Scala library (inf. heap)



1.7x faster

Benchmarks

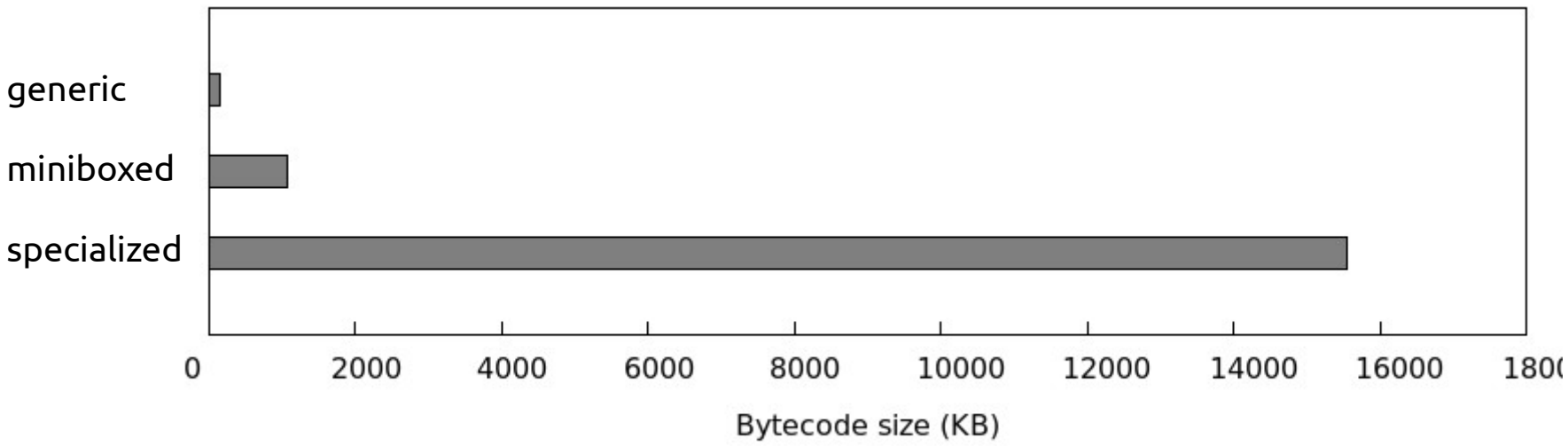
on the Scala library (with GC)



3x faster

Benchmarks

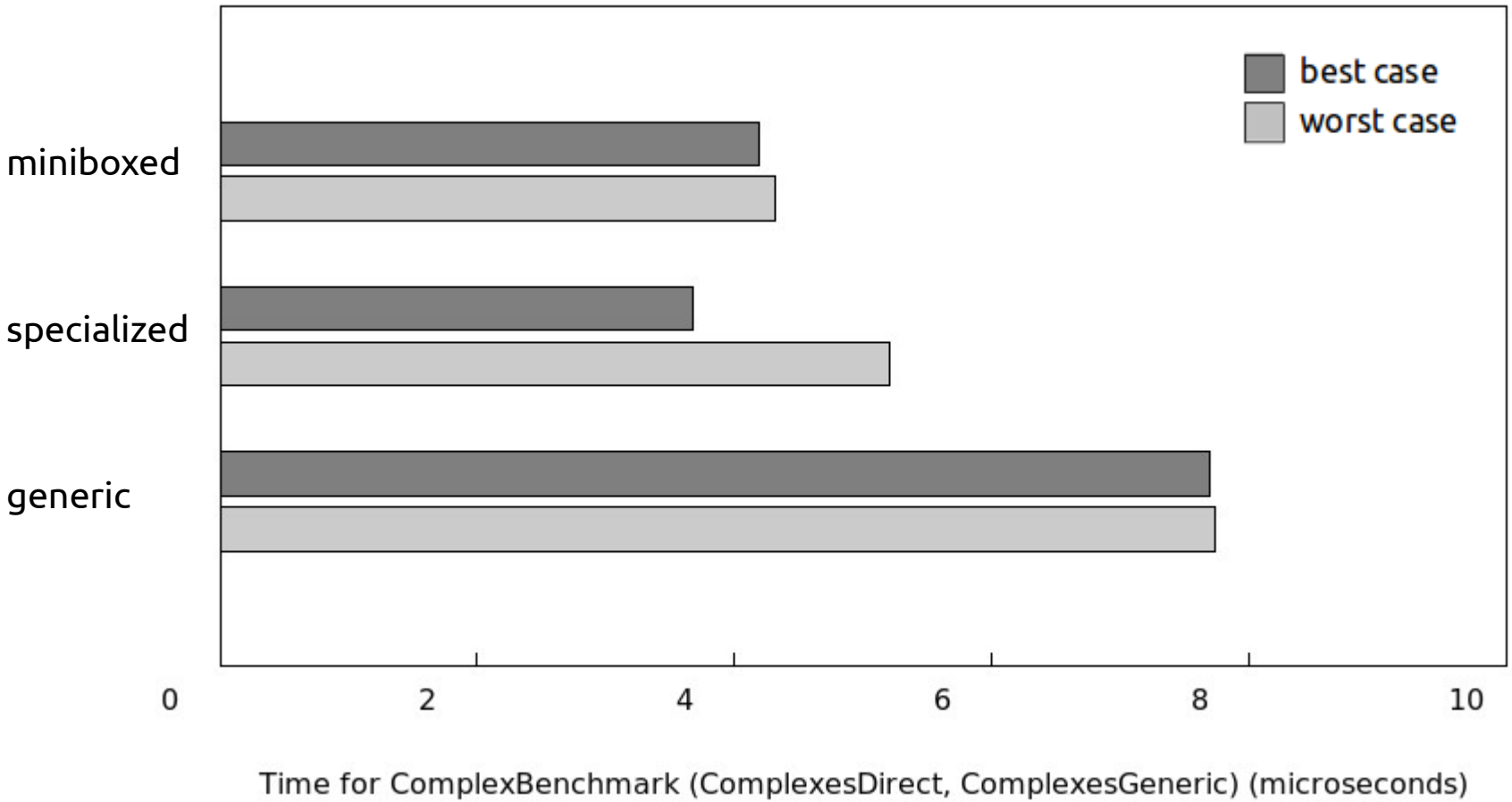
on the Scala library (bytecode)



non/spire

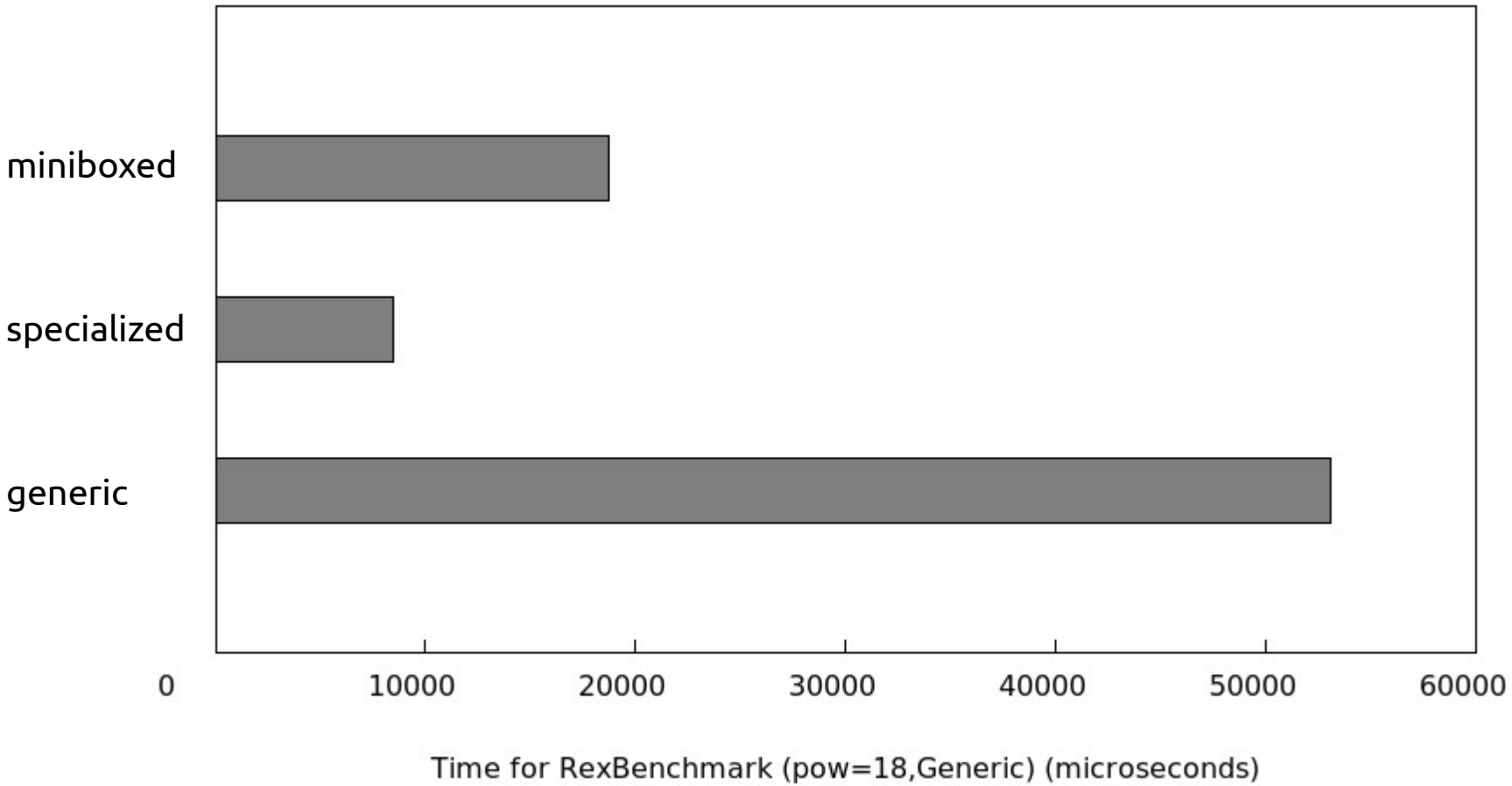
Benchmarks

on the Spire library (Complex)



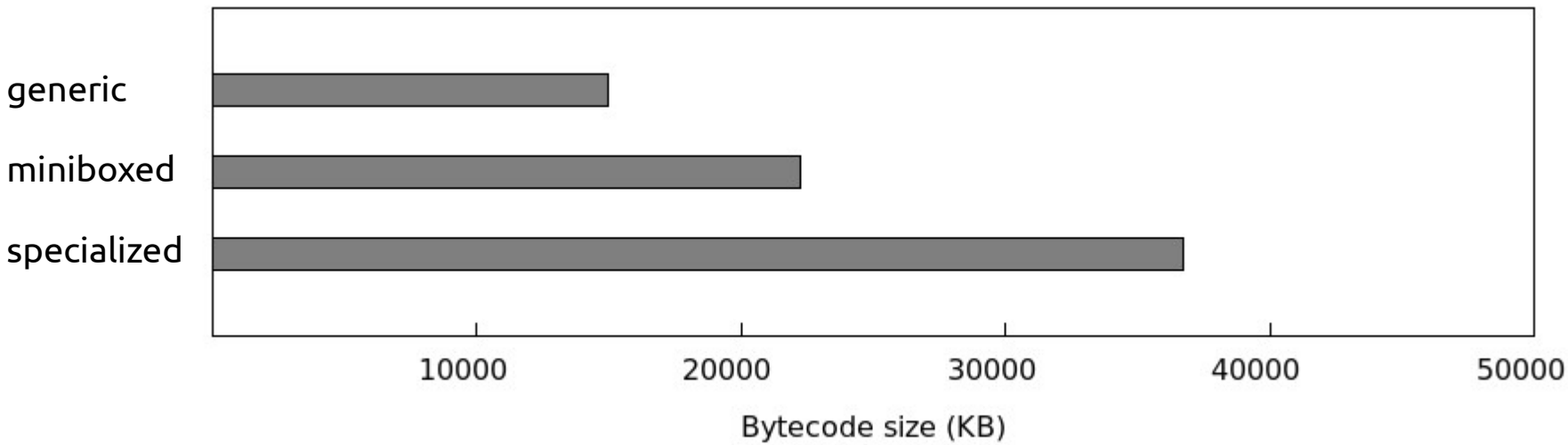
Benchmarks

on the Spire library (RexBench)



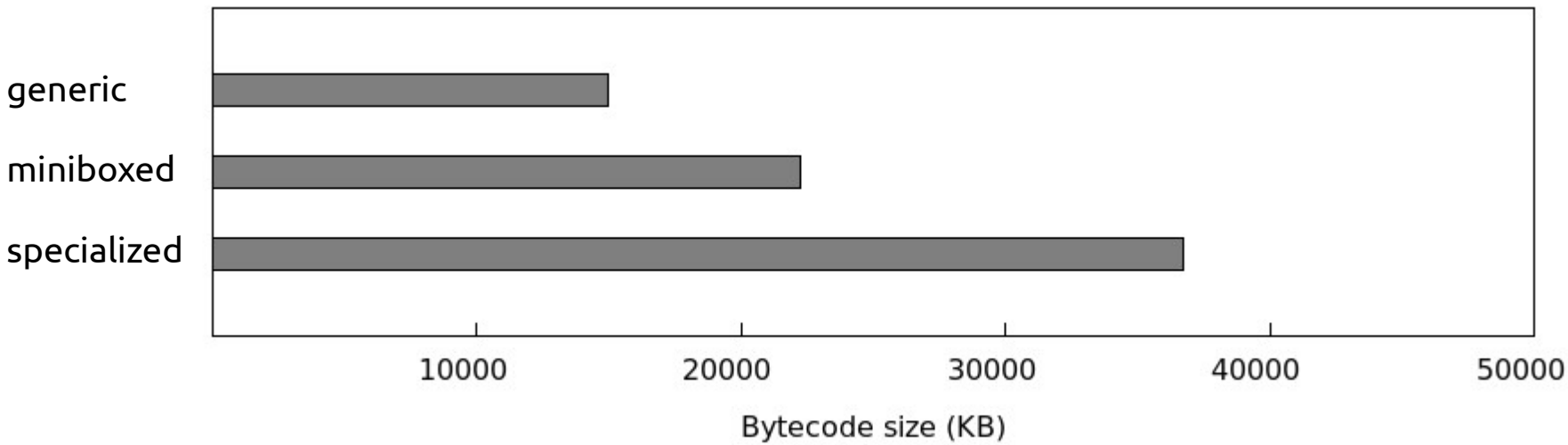
Benchmarks

on the Spire library (bytecode)



Benchmarks

on the Spire library (bytecode)



Spire is optimized for specialization

Credits

- Cristian Talau - developed the initial prototype, as a semester project
- Eugene Burmako - the value class plugin based on the LDL transformation
- Aymeric Genet - developing collection-like benchmarks for the miniboxing plugin
- Martin Odersky, for his patient guidance
- 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

@miniboxed = @specialized

- the limitations
- bytecode bloat

for Scala collections!

visit scala-miniboxing.org!