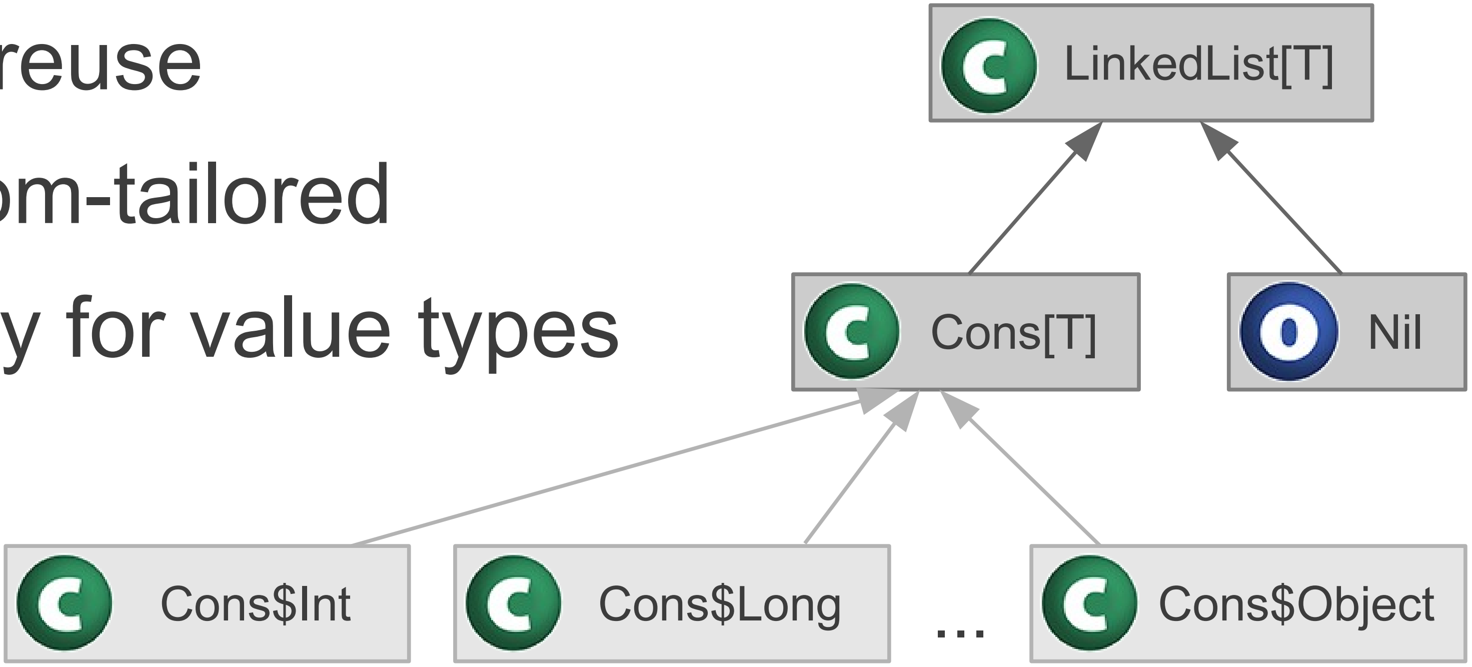
Miniboxing

Runtime Specialization on the JVM

Vlad Ureche
vlad.ureche@epfl.ch

Generic Code

- enables reuse
- not custom-tailored
- especially for value types



We are looking for specialization

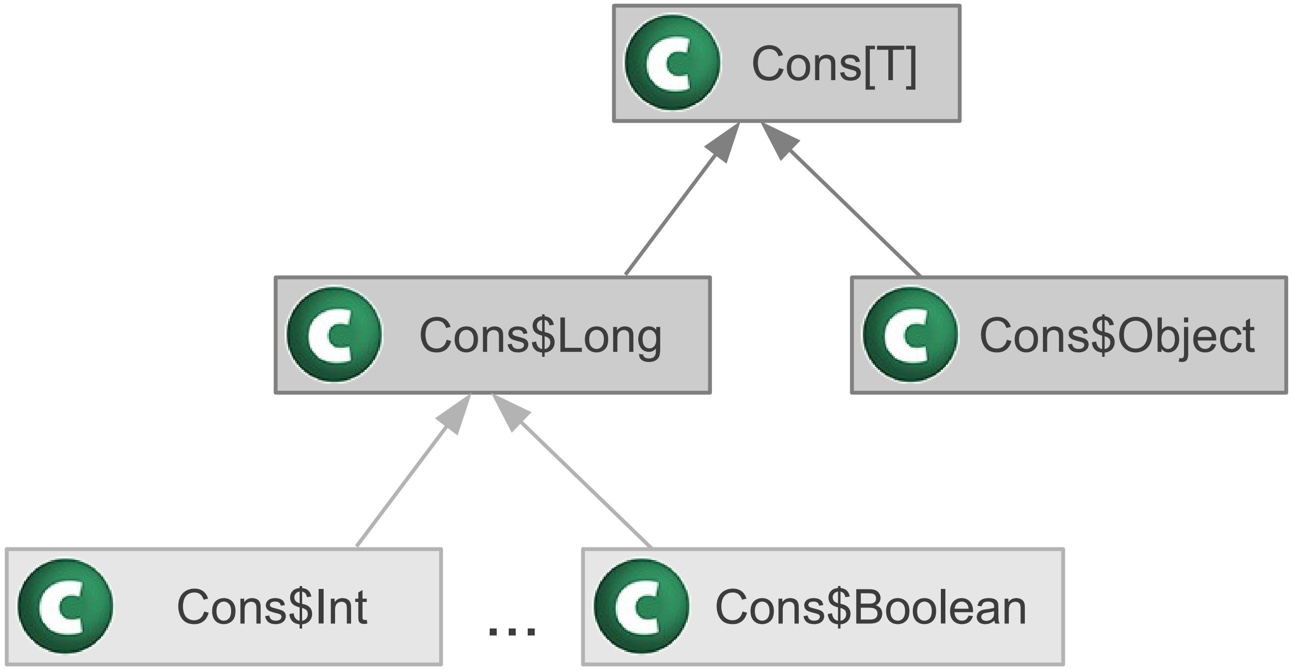
State of the Art

- dot net
  - runtime specialization, very little overhead
  - doesn't apply to Java bytecode - erasure
- pizza
  - runtime specialization via classloader
  - complex changes, reflection, slow
- static specialization
  - static specialization during compilation
  - massive amounts of bytecode – Map[K,V]

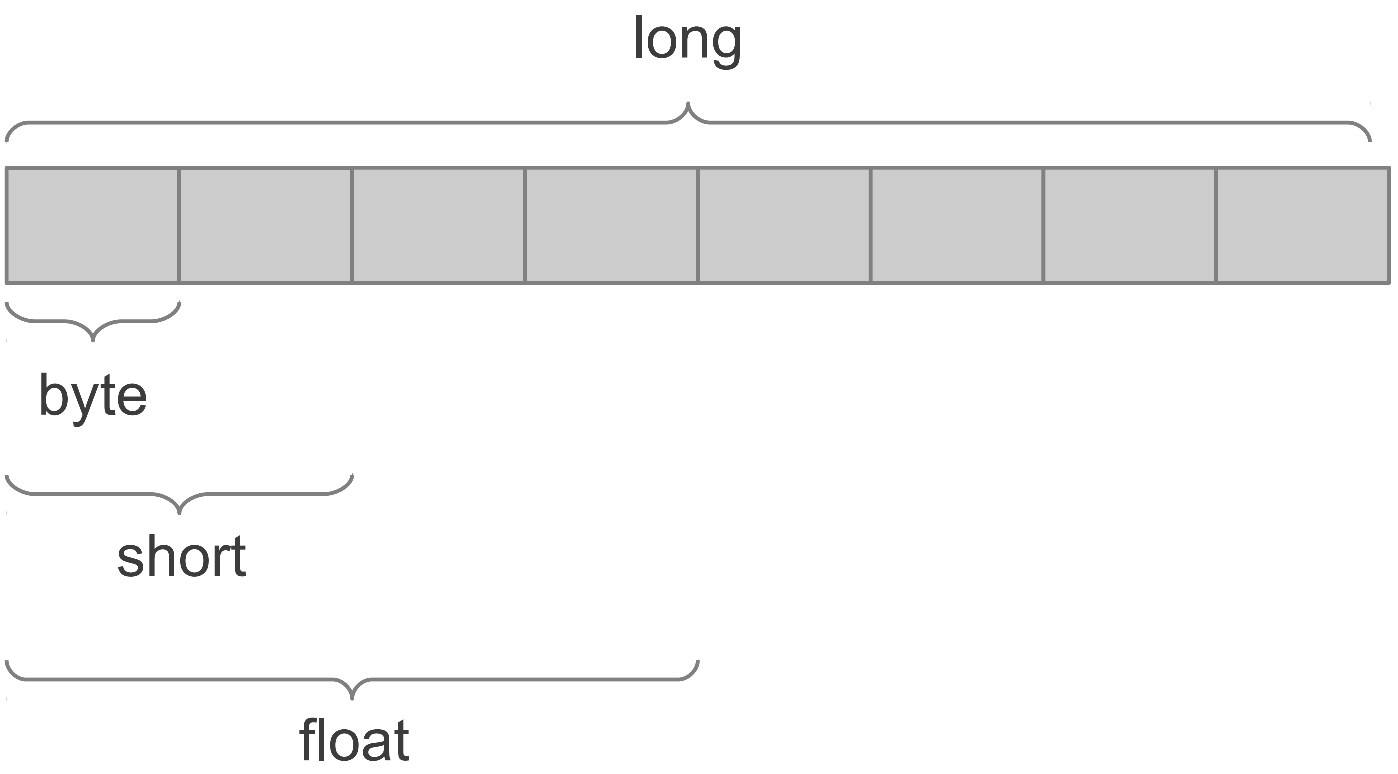
Hard problem

Miniboxing

- dot net-like runtime specialization
  - two stages: compile-time and runtime
  - reduces specialization to constant propagation



Insight



One **large** value type is enough

Main slowdown

- Restore the original type
- Restore the boxed type

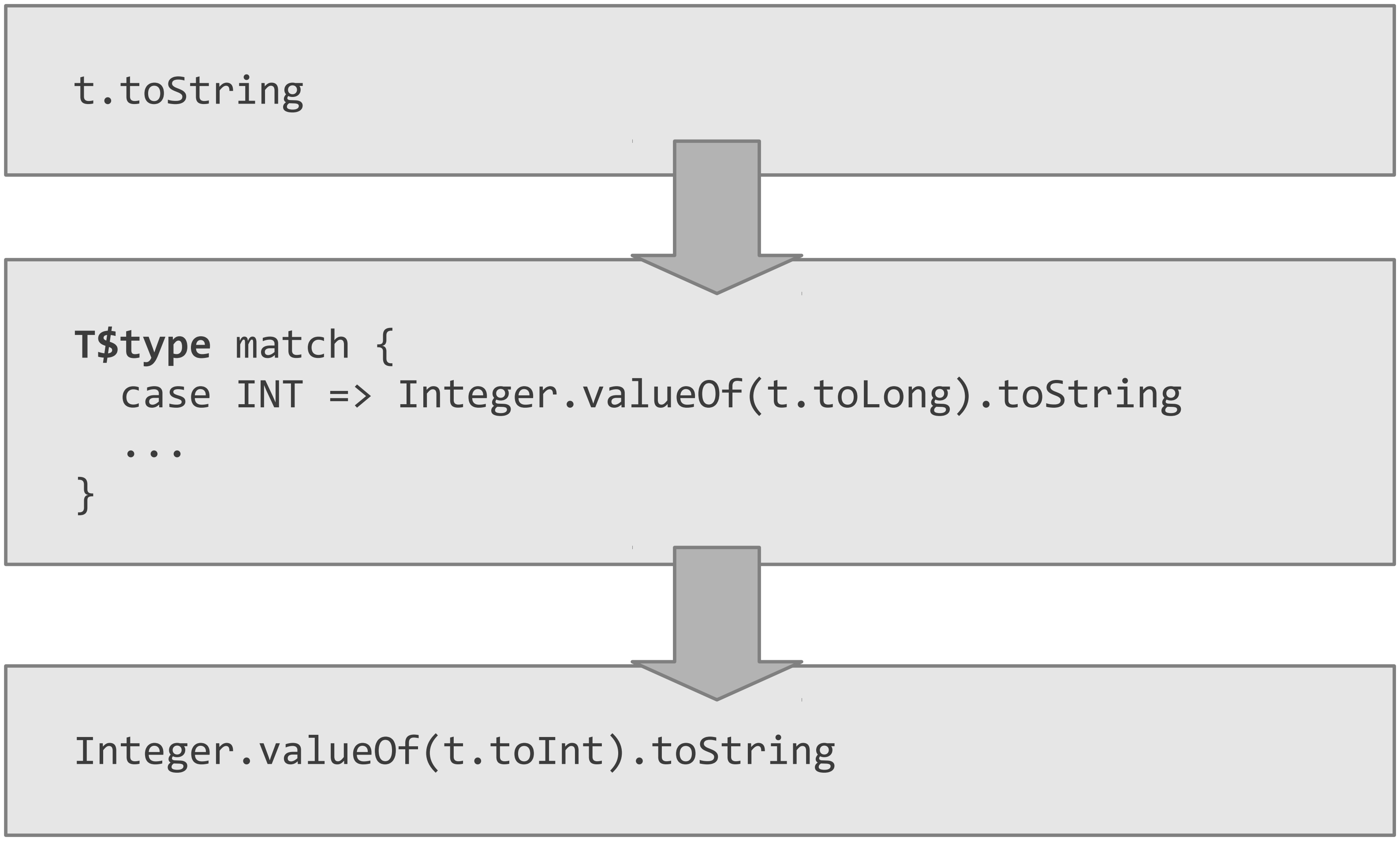
```
t.toString
```

```
T$type match {
  case INT => Integer.valueOf(t.toLong).toString
  ...
}
```

Runtime Specialization

- `T$type`
  - constant value
  - particular for each instance
  - embed as a constant in the class file
  - using a runtime classloader
- JVM
  - constant propagation
  - dead code elimination

Runtime Specialization

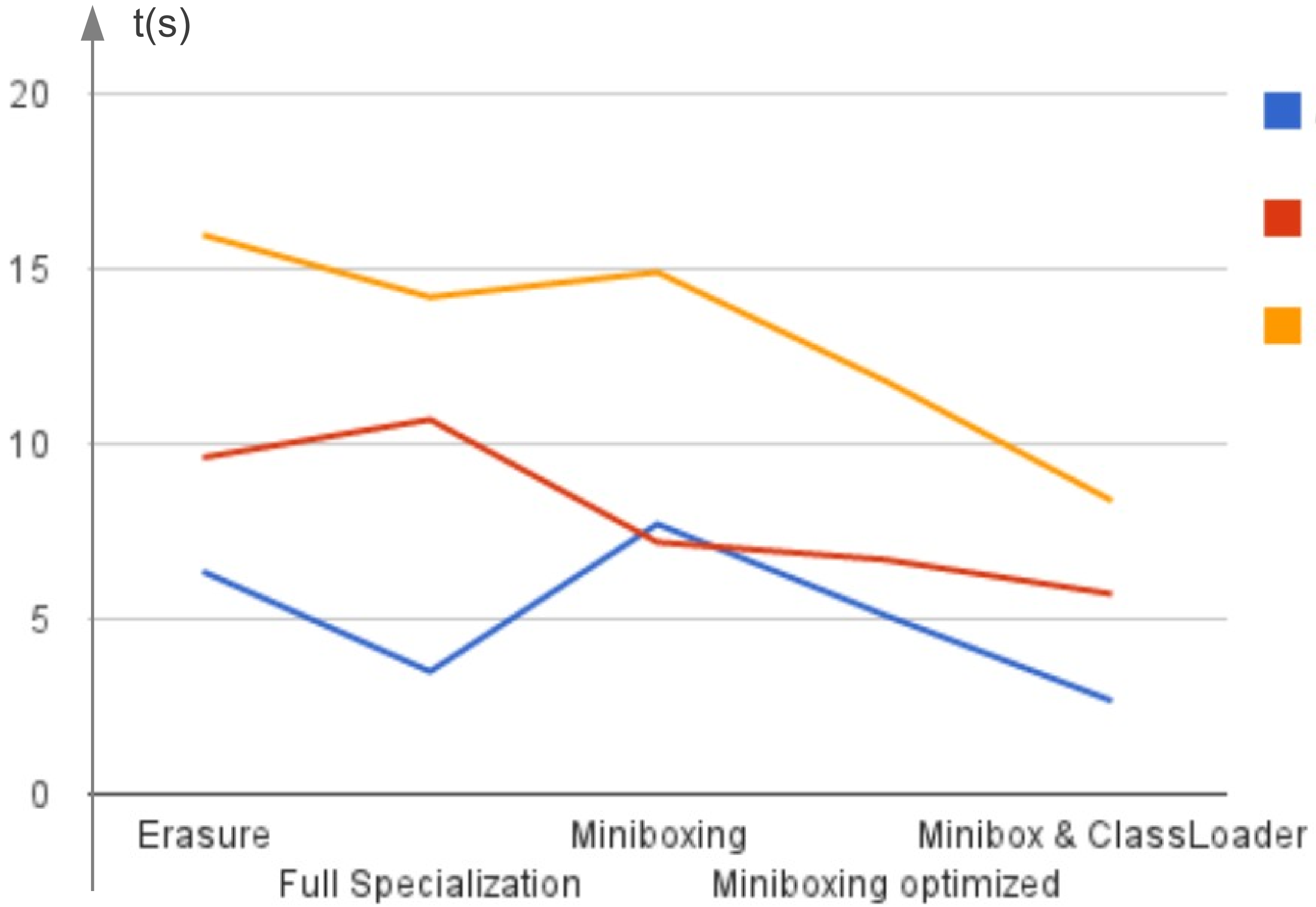


```
t.toString
```

```
T$type match {
  case INT => Integer.valueOf(t.toLong).toString
  ...
}
```

```
Integer.valueOf(t.toInt).toString
```

Benchmarks


t(s)
20
15
10
5
0
Erasure
Full Specialization
Miniboxing
Miniboxing optimized
Minibox & ClassLoader
🟦 Array operations
🟥 List operations
🟧 Overall

MethodHandles

- Classloader approach is complex
- MethodHandles could help

```
def foo[T](t: T) = ???
```

```
def foo(T$Type: Byte, t: Long) = ???

val fooChar = fooMH.bindTo(CHAR)
val fooLong = fooMH.bindTo(LONG)
```

Thank you!

github.com/miniboxing