The Optimizer in Scala 2.12

Lukas Rytz, Scala Team @ Typesafe

Scala

Typesafe

Why?

- The JVM optimizer (HotSpot) has 17 years of tuning
- More powerful: run-time program statistics, speculative optimization
- javac does not perform *any* optimizations

HotSpot is not Perfect

- HotSpot fails to optimize well-known patterns
- Much more common in Scala – Java 8 catching up
- Recognized by JVM experts

Prominent Issues

- Megamorphic dispatch (JDK-8015416)
  - Click: The inlining problem
  - Rose: Profile pollution (talk on "JVM Challenges")
  - Shipilëv: The Black Magic of Method Dispatch
- Value boxing
  - Project Valhalla: specialization, value types
  - Rose's talk again

Megamorphic Callsites

```scala
class Range {
  def foreach(f: Int => Unit) = {
    while(..) { .. f.apply(i) .. }
  }
}





(1 to 10) foreach (x => foo)
(2 to 20) foreach (x => bar)
(3 to 30) foreach (x => baz)
```

Megamorphic Callsites

```
class Range {
  def foreach(f: Int => Unit) = {
    while(..) { .. f.apply(i) .. }
  }
}

(1 to 10) foreach (x => foo)
(2 to 20) foreach (x => bar)
(3 to 30) foreach (x => baz)
```

Virtual call:

- Run-time type of `f` defines which code to run
- Megamorphic callsite, varying targets
- Method lookup on every loop iteration

Solution: Inlining

```
(1 to 10) foreach (x => foo)
```

Scala optimizer inlines `foreach`

```
val _this = 1 to 10
val _f = (x: Int) => foo
while(..) { .. _f.apply(i) .. }
```

Solution: Inlining

```
(1 to 10) foreach (x => foo)
```

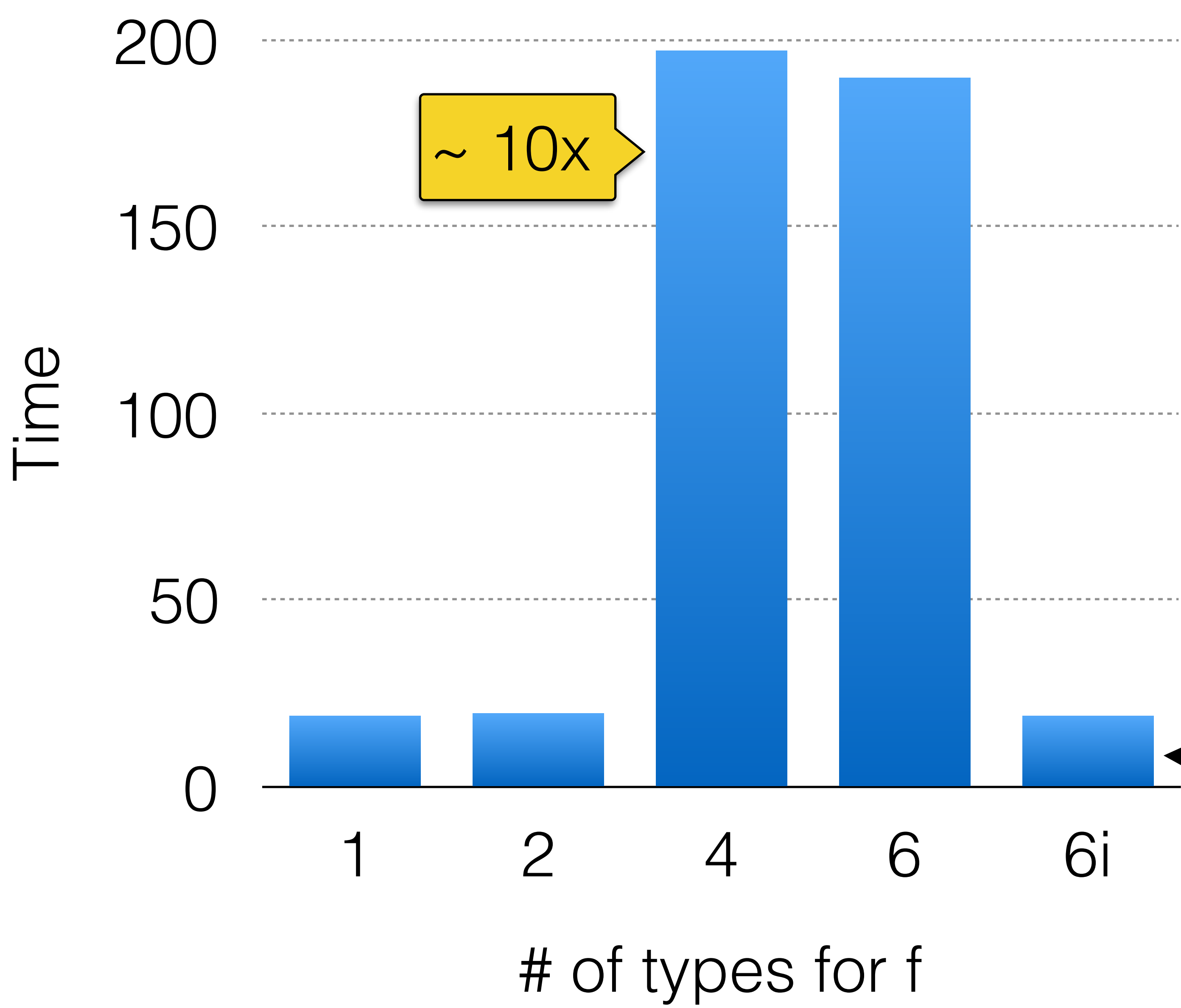
Scala optimizer inlines `foreach`

```
val _this = 1 to 10
val _f = (x: Int) => foo
while(..) { .. _f.apply(i) .. }
```

Monomorphic callsite enables JVM optimizations:

- Skip method lookup
- Inlining `apply` enables further optimizations

Megamorphic Callsites




```
while(..) {
  .. (x => foo).apply() ..
}
while(..) {
  .. (x => bar).apply() ..
}
...
```

github.com/lrytz/benchmarks

A Scala Problem?

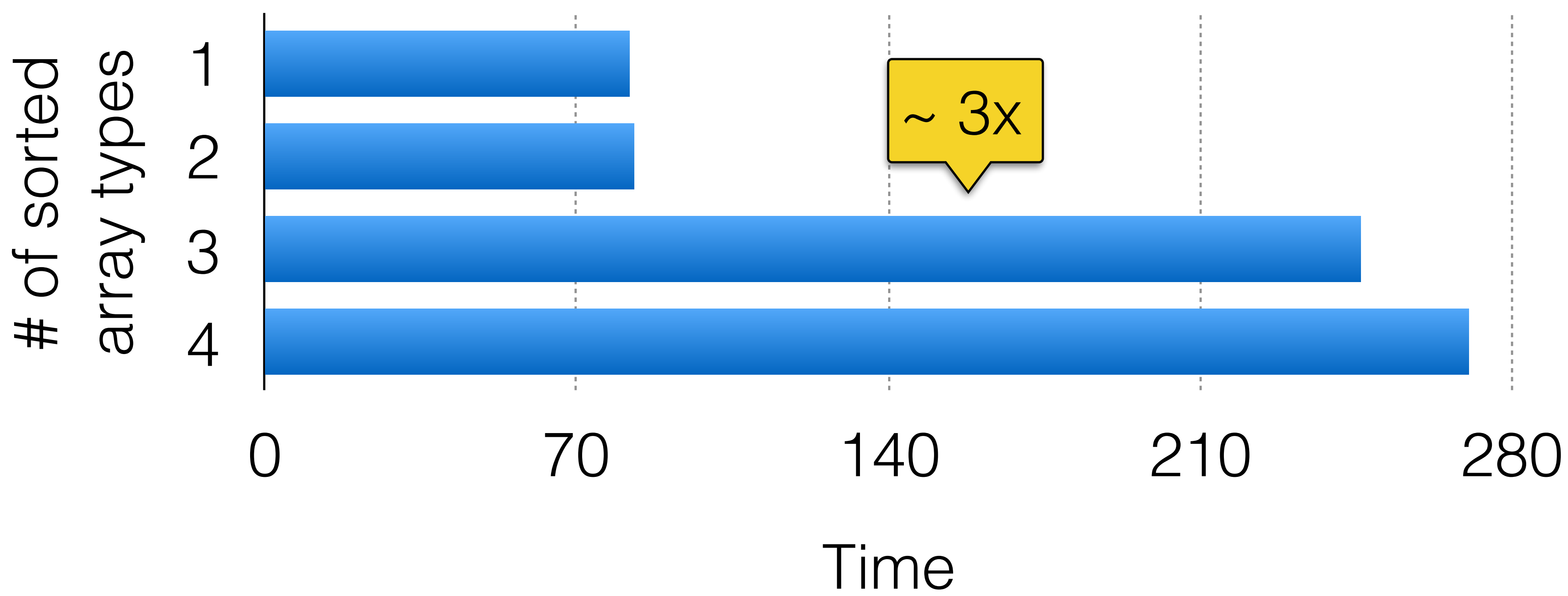
```
class A(val x: Int) // similar: B, C, ..
object AC extends Comparator[A] { .. }
java.util.Arrays.sort(aArr, 0, N, AC)
// java.util.Arrays.sort(bArr, 0, N, BC)
```

Scala

Typesafe

A Scala Problem?

```
class A(val x: Int) // similar: B, C, ..
object AC extends Comparator[A] { .. }
java.util.Arrays.sort(aArr, 0, N, AC)
// java.util.Arrays.sort(bArr, 0, N, BC)
```



Value Boxing

```
var r = 0
(1 to 10000) foreach { x => r += x }
```

```
val r = IntRef(0)
val f = new anonfun(r)
(1 to 10000) foreach f
```

```
class anonfun(r: IntRef) {
  def apply(x: Int) {
    r.elem += x
  }
}
```

Value Boxing

```scala
var r = 0
(1 to 10000) foreach { x => r += x }
```

```scala
val r = IntRef(0)
val f = new anonfun(r)
(1 to 10000) foreach f
```

Slow

- Why? Not obvious..

```scala
class anonfun(r: IntRef) {
  def apply(x: Int) {
    r.elem += x
  }
}
```

Inlining

```scala
val r = IntRef(0)
val f = new anonfun(r)
(1 to 10000) foreach f
```

Inline `foreach` and function body

```scala
val r = IntRef(0)
val f = new anonfun(r)
var x = 0
while (x < 10000) {
  r.elem += x
}
```

Inlining

```
val r = IntRef(0)
val f = new anonfun(r)
(1 to 10000) foreach f
```

Inline `foreach` and function body

```
val r = IntRef(0)
val f = new anonfun(r)
var x = 0
while (x < 10000) {
  r.elem += x
}
```

Still slow (same as before)!

- Why? `IntRef`
- Escape analysis fails..

Closure Elimination

```
val r = IntRef(0)
val f = new anonfun(r)
var x = 0
while (x < 10000) {
  r.elem += x
}
```

Eliminate the closure allocation

```
val r = IntRef(0)
var x = 0
while (x < 10000) {
  r.elem += x
}
```

Scala

Typesafe

Closure Elimination

```
val r = IntRef(0)
val f = new anonfun(r)
var x = 0
while (x < 10000) {
  r.elem += x
}
```

Eliminate the closure allocation

```
val r = IntRef(0)
var x = 0
while (x < 10000) {
  r.elem += x
}
```

Fast! JVM escape analysis kicks in.

Scala

Typesafe

Box Elimination

```
val r = IntRef(0)
var x = 0
while (x < 10000) {
  r.elem += x
}
```

Local var instead of IntRef

```
var r = 0
var x = 0
while (x < 10000) {
  r += x
}
```

Box Elimination

```
val r = IntRef(0)
var x = 0
while (x < 10000) {
  r.elem += x
}
```

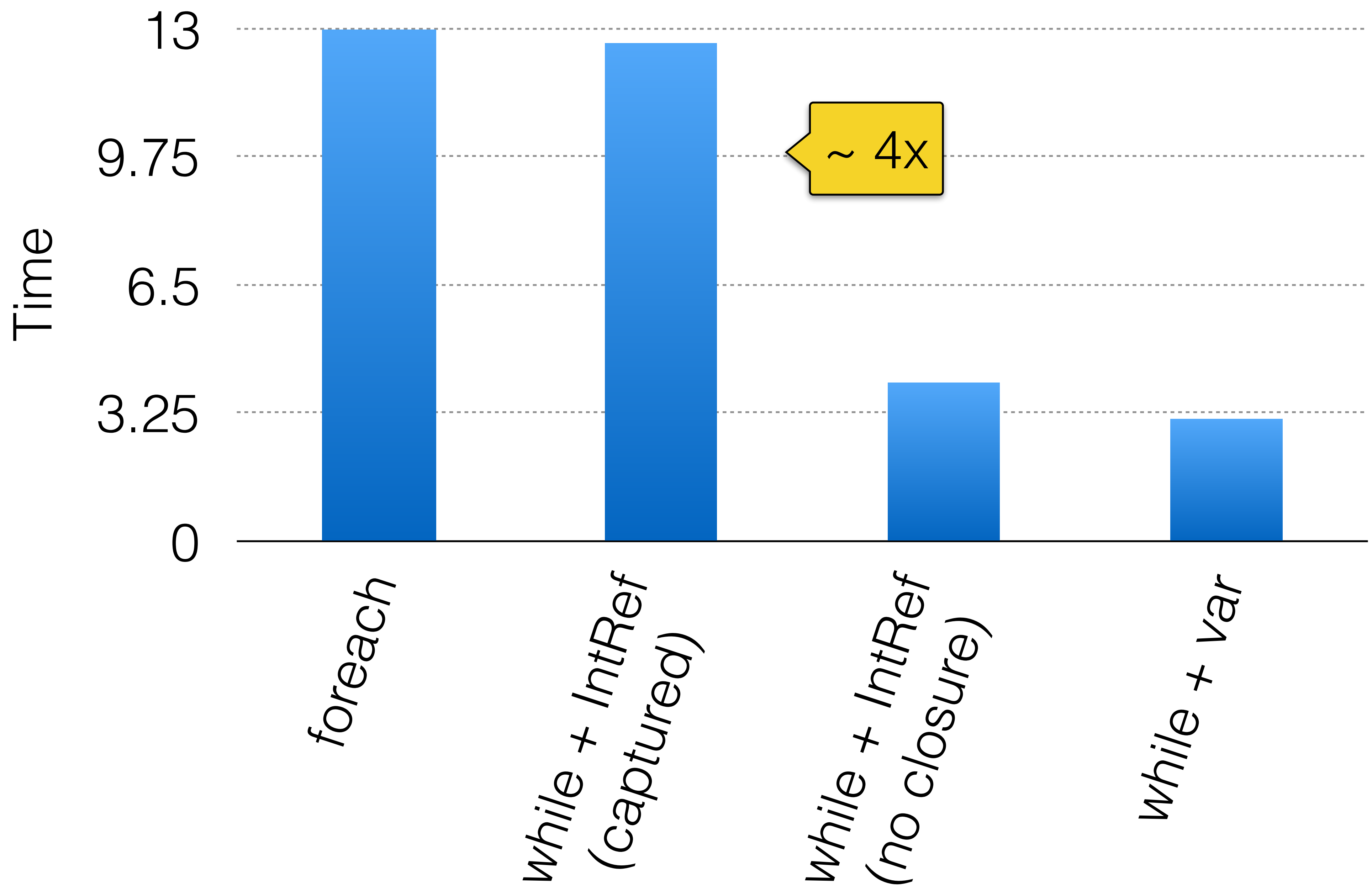
Local `var` instead of `IntRef`

```
var r = 0
var x = 0
while (x < 10000) {
  r += x
}
```

Same as before!
JVM optimizes the IntRef just fine.

Scala Typesafe

Bars



13
9.75
6.5
3.25
0
Time
~ 4x
foreach
while + IntRef (captured)
while + IntRef (no closure)
while + var

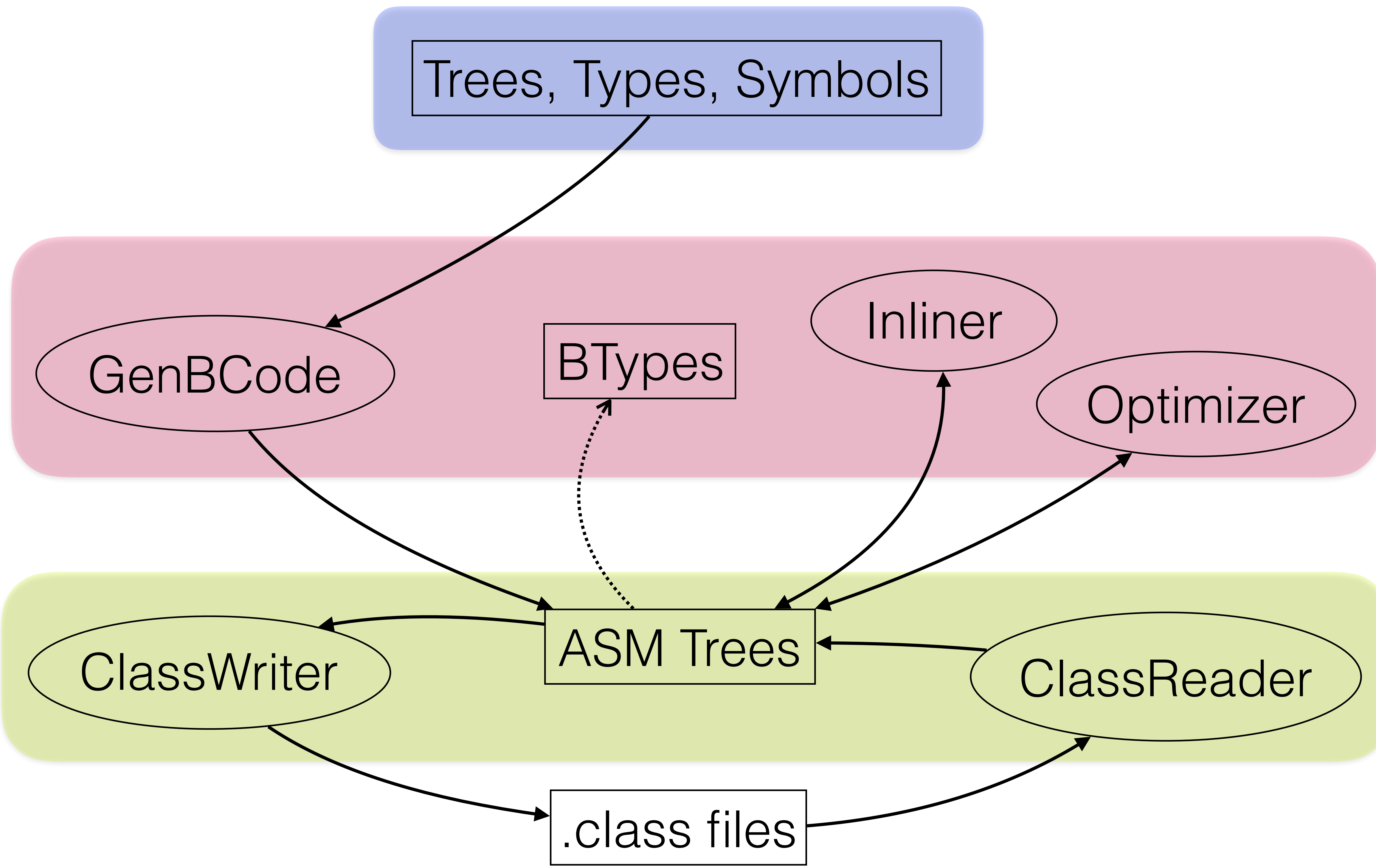
Compile-time Optimizer

- Goal: transform the code to make it please the JVM
- Don't perform optimizations that the JVM does well
- Avoid fruitless inlining: degrades performance
  - JVM optimizer is sensitive to method size

Agenda

- Compilation Pipeline Overview
- Local Optimizations
- Inlining and Heuristics
- Limitations
- Outlook, comparison with Scala.js

2.12 Backend: GenBCode



Local Optimizations

```scala
trait BooleanOrdering extends Ordering[Boolean] {
  def compare(x: Boolean, y: Boolean) =
    (x, y) match {
      case (false, true) => -1
      case (true, false) => 1
      case _ => 0
    }
}
```

Nullness

```
Tuple2.mcZZ.sp sp2 = new Tuple2.mcZZ.sp(x, y);
if (sp2 != null) {
    boolean bl = sp2._1$mcZ$sp();
    boolean bl2 = sp2._2$mcZ$sp();
    if (!bl) {
        if (true == bl2) return -1;
    }
}
if (sp2 == null) return 0;
boolean bl = sp2._1$mcZ$sp();
boolean bl3 = sp2._2$mcZ$sp();
if (!bl) return 0;
if (false == bl3) return 1;
return 0;
```

Nullness

```
Tuple2.mcZZ.sp sp2 = new Tuple2.mcZZ.sp(x, y);
if (sp2 != null) {
    boolean bl = sp2._1$mcZ$sp();
    boolean bl2 = sp2._2$mcZ$sp();
    if (!bl) {
        if (true == bl2) return -1;
    }
}
if (sp2 == null) return 0;
boolean bl = sp2._1$mcZ$sp();
boolean bl3 = sp2._2$mcZ$sp();
if (!bl) return 0;
if (false == bl3) return 1;
return 0;
```

Scala

Typesafe

Box-Unbox

```
int n;
Tuple2.mcZZ.sp sp2 = new Tuple2.mcZZ.sp(x, y);
boolean bl = sp2._1$mcZ$sp();
boolean bl2 = sp2._2$mcZ$sp();
if (!bl && bl2) {
    n = -1;
} else {
    boolean bl3 = sp2._1$mcZ$sp();
    boolean bl4 = sp2._2$mcZ$sp();
    n = bl3 && !bl4 ? 1 : 0;
}
return n;
```

Box-Unbox

```
int n;
Tuple2.mcZZ.sp sp2 = new Tuple2.mcZZ.sp(x, y);
boolean bl = sp2._1$mcZ$sp();
boolean bl2 = sp2._2$mcZ$sp();
if (!bl && bl2) {
    n = -1;
} else {
    boolean bl3 = sp2._1$mcZ$sp();
    boolean bl4 = sp2._2$mcZ$sp();
    n = bl3 && !bl4 ? 1 : 0;
}
return n;
```

Redundant Locals

```
int n;
boolean bl = y;
boolean bl2 = x;
Tuple2.mcZZ.sp sp2 = new Tuple2.mcZZ.sp(bl2, bl);
boolean bl3 = bl2;
boolean bl4 = bl;
if (!bl3 && bl4) {
    n = -1;
} else {
    boolean bl5 = bl2;
    boolean bl6 = bl;
    n = bl5 && !bl6 ? 1 : 0;
}
return n;
```

Scala

Typesafe

Redundant Locals

```
int n;
boolean bl = y;
boolean bl2 = x;
Tuple2.mcZZ.sp sp2 = new Tuple2.mcZZ.sp(bl2, bl);
boolean bl3 = bl2;
boolean bl4 = bl;
if (!bl3 && bl4) {
    n = -1;
} else {
    boolean bl5 = bl2;
    boolean bl6 = bl;
    n = bl5 && !bl6 ? 1 : 0;
}
return n;
```

Unused Values

```
Tuple2.mcZZ.sp sp2 = new Tuple2.mcZZ.sp(x, y);
int n = false == x && true == y ? -1 :
        (true == x && false == y ? 1 : 0);
return n;
```

Unused Values

```
Tuple2.mcZZ.sp sp2 = new Tuple2.mcZZ.sp(x, y);
int n = false == x && true == y ? -1 :
        (true == x && false == y ? 1 : 0);
return n;
```

Unused Values

```
Tuple2.mcZZ.sp sp2 = new Tuple2.mcZZ.sp(x, y);
int n = false == x && true == y ? -1 :
        (true == x && false == y ? 1 : 0);
return n;
```

```
int n = false == x && true == y ? -1 :
        (true == x && false == y ? 1 : 0);
return n;
```

Scala

Typesafe

Local Optimizations

- Nullness
- Unreachable Code
- Box-Unbox
- Redundant Locals
- Redundant Casts
- Unused Values
- Simplify Jumps

Local Optimizations

- Nullness
- Unreachable Code
- Box-Unbox
- Redundant Locals
- Redundant Casts
- Unused Values
- Simplify Jumps

Primitive Boxes, Tuples, Refs

Local Optimizations

- Nullness
- Unreachable Code
- Box-Unbox — Primitive Boxes, Tuples, Refs
- Redundant Locals
- Redundant Casts
- Unused Values — Boxes, Closures
- Simplify Jumps

Scala

Typesafe

Closure Elimination

```
(1 to 10) foreach (x => foo)
```

```
val _this = 1 to 10
val _f = (x: Int) => foo
while(..) { .. _f.apply(i) .. }
```

```
val _this = 1 to 10
while(..) { .. foo .. }
```

Scala

Typesafe

Agenda

- Compilation Pipeline Overview
- Local Optimizations
- Inlining and Heuristics
- Limitations
- Outlook, comparison with Scala.js

Inlining

- Local optimizations often enabled by inlining
  - Inlining `foreach` allows eliminating the closure
- Challenge: when to inline (heuristics)
  - Method size impacts JVM performance

Heuristics

- Goal: predict elimination of megamorphic callsites and value boxing
- How to identify callsites to inline?
  - Argument types (Functions, type classes)
  - Analysis of the callee (is an argument function invoked? passed to another call? captured?)
  - Backtracking
- Other considerations: method size, call frequency

Agenda

- Compilation Pipeline Overview
- Local Optimizations
- Inlining and Heuristics
- Limitations
- Outlook, comparison with Scala.js

Open World Assumption

- Inline only methods that cannot be overridden
- No whole-program analysis, support reflection
  - Don't know what are subclasses, overrides
  - Don't know what types are instantiated

A Magic Wand?

- The optimizer is by no means a magic wand
  - Most likely no speedup for an average program
- Instead: a tool for experts
  - Allows using high-level patterns in performance-critical code

Performance Example

- Richards from Scala.js benchmark suite
  - Scala.js optimizer: 3x speedup (on V8)
  - 2.12 optimizer: no speedup (JVM)
- TODO: find out why!
  - Scala.js heuristics better than ours?
  - Open-world assumption prevents optimizations?
  - JVM better than V8, more to gain for Scala.js?

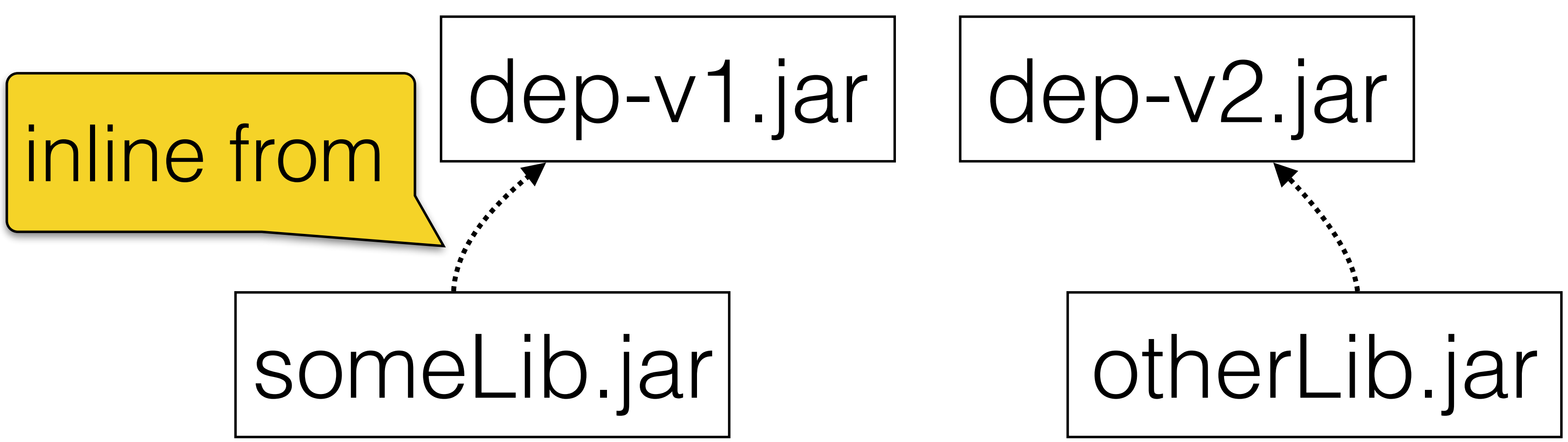
Binary Compatibility

- Inlining from a library enforces a specific version
  - Assumption: consistent run-time classpath
- Problematic for library authors: forces specific versions for dependencies

Binary Compatibility

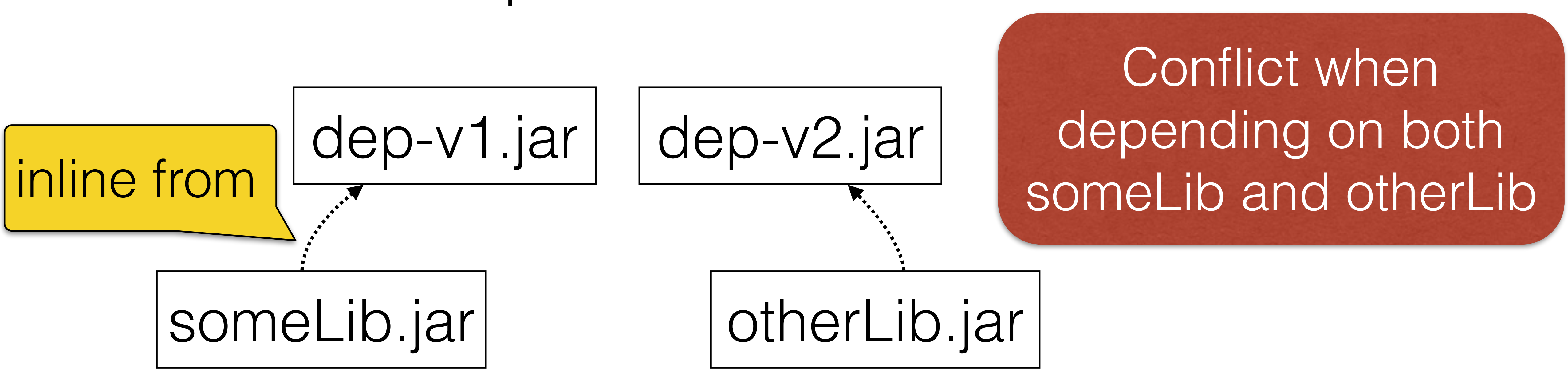
- Inlining from a library enforces a specific version
  - Assumption: consistent run-time classpath

- Problematic for library authors: forces specific versions for dependencies



Binary Compatibility

- Inlining from a library enforces a specific version
  - Assumption: consistent run-time classpath
- Problematic for library authors: forces specific versions for dependencies



Scala

Typesafe

Binary Compatibility

- Library authors: don't inline from the classpath
  - `Range.foreach` is slow
- Deployed applications: optimize freely
  - Ensure same classpath at runtime
  - Consider building dependencies from source

Agenda

- Compilation Pipeline Overview
- Local Optimizations
- Inlining and Heuristics
- Limitations
- Outlook, comparison with Scala.js

Global Optimization

- Assumptions for whole-program optimization:
  - ↣ Entire program available (including libraries)
  - ↣ Entry-point(s) known
  - ↣ Can change / remove code, no more linking
  - ↣ Restrictions in using reflection (may be "none")

Advantages

- Optimizer causes no binary compatibility issues
- Global knowledge: instantiated types, subclasses
    - ↣ Enables inlining more non-final callsites
- Examples:
    - ↣ Scala.js, dotty linker (in the works), dart2js
    - ↣ JVM, but more time-constrained

Thank You!