JVM Backend and Optimizer in Scala 2.12

Lukas Rytz, Scala Team @ Typesafe
Scala 2.12 on one Slide

• Move to Java 8: enjoy new VM and library features
  ➔ Interop for functions: source and bytecode
  ➔ Make use of default methods
  ➔ Interop with Java (parallel) streams

• New optimizer
  ➔ Configurable, more reliable, better diagnostics
  ➔ Fewer bugs (inline trait methods)
Agenda

• Move to Java 8
  → Interop for functions: source and bytecode
  → Default methods for compiling traits

• New backend
  → Simplified compilation pipeline
  → New optimizer: capabilities and constraints
Agenda

• Move to Java 8
  ➔ Interop for functions: source and bytecode
  ➔ Default methods for compiling traits

• New backend
  ➔ Simplified compilation pipeline
  ➔ New optimizer: capabilities and constraints
Function Interoperability

• Source code: interoperability in both directions

```scala
// Scala code:
new Thread(() => println("hi!")).run()
```

```java
// Java code:
scalaCollection.foreach(x -> println(x));
```

• Bytecode: generate the same as Java
Use Java APIs

- No explicit function types in Java 8
  - Lambda syntax for functional (SAM) interfaces

```java
interface Runnable { void run(); }
class Thread { Thread(Runnable r) }
new Thread(() -> println("hi")).run();
```
Use Java APIs

• No explicit function types in Java 8
  → Lambda syntax for functional (SAM) interfaces

```java
interface Runnable { void run(); }
class Thread { Thread(Runnable r) }
new Thread(() -> println("hi")).run();
```

• SAM support in Scala 2.12
  → Try it with 2.11.6 -Xexperimental

```scala
new Thread(() => println("hi!")).run()
```
// Java example:
stream.filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
SAMs in Scala

// Java example:
stream.filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);

Welcome to Scala version 2.11.6 -Xexperimental
scala> import java.util.stream.Stream
scala> def s = Stream of ("a1","a2","b1","c2","c1")
scala> s.filter(s => s.startsWith("c"))
    .map(_.toUpperCase).sorted.forEach(println)
C1
C2
Write Java APIs

- Scala 2.12: FunctionN are functional interfaces

```java
// Java code:
scalaCollection.foreach(x -> println(x));
```
Write Java APIs

- Scala 2.12: `FunctionN` are functional interfaces

```java
// Java code:
scalaCollection.foreach(x -> println(x));
```

- Scala 2.11: compatibility layer (*)

  ➔ Defines `JFunctionN` functional interfaces

```java
import static scala.compat.java8.JFunction.*;
scalaCollection.foreach(func(x -> println(x)));
```

(*) [github.com/scala/scala-java8-compat](https://github.com/scala/scala-java8-compat)
Bytecode: Scala 2.11

l.reduce((x, y) => x + y)
Bytecode: Scala 2.11

```scala
l.reduce(((x, y) => x + y))

class anonfun$1 extends Function2 {
  def apply(x: Int, y: Int): Int = x + y
}

l.reduce(new anonfun$1())
```
interface IIIFun { int apply(int x, int y); }
abstract class Test {
    abstract int reduce(IIIFun f);
    int test() { return reduce((x, y) -> x + y); }
}
interface IIIIFun { int apply(int x, int y); }
abstract class Test {
    abstract int reduce(IIIIFun f);
    int test() { return reduce((x, y) -> x + y); }
}

private static int lambda$test$0(int x, int y) {
    return x + y;
}
return reduce(
    magicClosure("lambda$test$0", "IIIIFun::apply"))
interface IIIFun { int apply(int x, int y); }  
abstract class Test {  
    abstract int reduce(IIIFun f);  
    int test() { return reduce((x, y) -> x + y); }  
}  

private static int lambda$test$0(int x, int y) {  
    return x + y;  
}  
return reduce(  
    magicClosure("lambda$test$0", "IIIFun::apply"))  
InvokeDynamic + LambdaMetaFactory
Scala 2.11 with
-Ydelambdrafy:method

l.reduce((x, y) => x + y)
Scala 2.11 with -Ydelambdafy:method

```scala
l.reduce((x, y) => x + y)

<static> def $anonfun$1(x: Int, y: Int): Int = x + y

class lambda$1 extends Function2 {
  def apply(x: Int, y: Int): Int = $anonfun$1(x, y)
}

l.reduce(new lambda$1())
```
Scala 2.11 with -Ydelambdafy:method

```
l.reduce((x, y) => x + y)

<static> def $anonfun$1(x: Int, y: Int): Int = x + y

class lambda$1 extends Function2 {
  def apply(x: Int, y: Int): Int = $anonfun$1(x, y)
}
l.reduce(new lambda$1())
```

2.12: InvokeDynamic + LambdaMetaFactory
Why InDy+LMF

• No classfiles for functions – smaller JARs
• Let the JVM know what values are functions
  ➔ Might lead to better optimizations
• Be a good JVM citizen
Agenda

• Move to Java 8
  ➔ Interop for functions: source and bytecode
  ➔ Default methods for compiling traits

• New backend
  ➔ Simplified compilation pipeline
  ➔ New optimizer: capabilities and constraints
trait F1[-T, +R] {
  def apply(v: T): R
  def compose[A](g: A => T): A => R = ...
}

interface F1 {
  def apply(v: Object): Object
  def compose(g: F1): F1
}

class F1$class {
  <static> def compose($this: F1, g: F1): F1 = ...
}
Trait Compilation

trait F1[-T, +R] {
  def apply(v: T): R
  def compose[A](g: A => T): A => R = ...
}

interface F1 {
  def apply(v: Object): Object
  def compose(g: F1): F1
}

class F1$class {
  <static> def compose($this: F1, g: F1): F1 = ...
}
SAM FunctionN interfaces

• Not possible in Scala 2.11
  ➔ Targets Java 1.6, no default methods

• Options for Scala 2.12
  ➔ Write scala.FunctionN in Java
  ➔ Special treatment for FunctionN
  ➔ Use default methods to compile traits
Default Methods for Traits

• Write SAM interfaces in Scala
  ➔ Better Java APIs in written Scala

• Binary compatibility: allow some changes to traits
  ➔ Add a method to a trait without recompiling subclasses

• Status: performed a few experiments (*)

(*) github.com/lrytz/traits-default-methods
    github.com/scala/scala-java8-compat
    github.com/lampepfl/dotty
Default Methods for Traits

• Simple solution: forwarders to implementation class

```scala
interface F1 {
  def apply(v: Object): Object
  default def compose(g: F1): F1 = F1$class.compose(this, g)
}
```

• Ambitious solution: no more implementation classes, use only defaults
Agenda

• Move to Java 8
  ➞ Interop for functions: source and bytecode
  ➞ Default methods for compiling traits

• New backend
  ➞ Simplified compilation pipeline
  ➞ New optimizer: capabilities and constraints
2.11 Backend: GenASM

- Trees
- Types, Symbols
- GenICode
- GenASM
- ClassWriter
- ASM Trees
- Bytecode
- ICode
- Optimizer
- Inliner
- ICodeReader
2.12 Backend: GenBCode

Trees

Types, Symbols

GenBCode

BTypes

ASM Trees

Inliner

Optimizer

ClassWriter

Bytecode

ClassReader
2.12 Backend: GenBCode

Available in 2.11
-Ybackend:GenBCode

Types, Symbols → Trees → GenBCode → BTypes → ASM Trees → ClassWriter

ASM Trees → ClassReader

ASM Trees → Inliner

ASM Trees → Optimizer

ASM Trees → Bytecode
New Backend

• Fewer components to maintain

• Thread-safe representation (ASM Trees + BTypes)
  ➞ Parallelize classfile serialization, local optimizations

• Better platform to implement optimizations
  ➞ ASM has built-in tools for code analysis
  ➞ Fewer invariants (no explicit basic blocks)
Agenda

• Move to Java 8
  → Interop for functions: source and bytecode
  → Default methods for compiling traits

• New backend
  → Simplified compilation pipeline
  → New optimizer: capabilities and constraints
Optimizer in GenBCode

- Original prototype by @magarciaEPFL

- Work in progress
  - Optimizer being added to GenBCode backend under 2.11.x
  - Local optimizations in 2.11.6
  - Inliner in 2.11.7
  - Future: allocation elimination, heuristics, ...

- Thorough testing
Optimizer Sub-Agenda

• Why do we need a compile-time optimizer?
• Features and roadmap
• Inherent limitations
• Distant future: a whole-program optimizer?
Optimizer Sub-Agenda

• Why do we need a compile-time optimizer?
• Features and roadmap
• Inherent limitations
• Distant future: a whole-program optimizer?
Can we beat the JVM?

- The JVM is a powerful optimizing runtime
  - Run-time statistics available (counts, types)
- Fails to optimize certain common Scala patterns
- Goal: help out the JVM to do a great job
  - Avoid blind, premature "optimization"
    - Example: slowdown when inlining too much
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)
class Range {
    def foreach(f: Int => Unit) = {
        while(..) { .. f.apply(i) .. }
    }
}

Virtual call:
- Run-time type of f defines which code to run

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

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

Virtual call:
- Run-time type of f defines which code to run
- VM statistics: what types for f get here?

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

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

Virtual call:
• Run-time type of f defines which code to run
• VM statistics: what types for f get here?
• JIT: skip lookup (with guard) if monomorphic

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

"The Inlining Problem" – coined by Cliff Click ¹

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

See also: shipilev.net/blog/2015/black-magic-method-dispatch
Megamorphic Callsites

"The Inlining Problem" – coined by Cliff Click

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

- `f.apply` is hot, but megamorphic → not inlined by VM

See also: [shipilev.net/blog/2015/black-magic-method-dispatch](shipilev.net/blog/2015/black-magic-method-dispatch)
Megamorphic Callsites

"The Inlining Problem" – coined by Cliff Click \(^1\)

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

- `f.apply` is hot, but megamorphic → not inlined by VM
- Fix: inline `foreach` → copy of `f.apply` is monomorphic

\(^1\) www.azulsystems.com/blog/cliff/2011-04-04-fixing-the-inlining-problem

See also: shipilev.net/blog/2015/black-magic-method-dispatch
Megamorphic Callsites

"The Inlining Problem" – coined by Cliff Click

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

- `f.apply` is hot, but megamorphic → not inlined by VM
- Fix: inline `foreach` → copy of `f.apply` is monomorphic
- Call to `foreach` typically not hot → not inlined by VM


See also: [shipilev.net/blog/2015/black-magic-method-dispatch](http://shipilev.net/blog/2015/black-magic-method-dispatch)
Megamorphic Callsites

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

github.com/lrytz/benchmarks
Captured Local

```
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.ele += x
  }
}
```
Captured Local

```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
```

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

Slow
• Why? Not obvious..
Inlining

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

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

Inline foreach and function body
val r = IntRef(0)
val f = new anonfun(r)
(1 to 10000) foreach f

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

Inline foreach and function body

Still slow (same as before)!
  • Why? IntRef
  • Escape analysis fails..
Help Out the JVM

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

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

Eliminate the closure allocation
Help Out the JVM

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.
Eliminate the IntRef?

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

Local var instead of IntRef

```scala
var r = 0
var x = 0
while (x < 10000) {
    r += x
}
```
Eliminate the IntRef?

```
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.
En Graphe

![Bar chart comparing time for different Scala constructs: `foreach`, `while + IntRef (captured)`, `while + IntRef (no closure)`, and `while + var`. The chart indicates a performance improvement of approximately 4x for `while + IntRef (captured)`.](image-url)
Summary

• Scala compiler can fix some known performance issues
  → New optimizer provides better abstractions to implement heuristics

• Need to be prudent and benchmark-driven
Optimizer Sub-Agenda

• Why do we need a compile-time optimizer?

• Features and roadmap

• Inherent limitations

• Distant future: a whole-program optimizer?
Features

• Some local optimizations in 2.11.6
  ➞ Dead code elimination, simplify jumps
  ➞ More to come. Goal: generate clean code

• Inliner in 2.11.7 (work is almost done)

• Thereafter
  ➞ Eliminate allocations: closures, tuples, boxes
  ➞ Testing, benchmarking, tuning heuristics
Inliner

- Transformation from bytecode to bytecode
- Clean call graph representation
  - Future-proof: experiment with heuristics
- Reliable and configurable error reporting
- Well tested
  - Community build
  - "Insane" mode: all-you-can-inline
Collaboration

• GitHub repo and issue tracker: scala-opt/scala
  ➔ Keep track of plans and tasks
  ➔ Plenty available: from "rewrite x efficiently" to "implement type analysis"
  ➔ File new issues for bugs in the optimizer

• Questions and discussions
  ➔ Compiler hacker's Gitter channel: scala/scala
Optimizer Sub-Agenda

• Why do we need a compile-time optimizer?
• Features and roadmap
• Inherent limitations
• Distant future: a whole-program optimizer?
Binary Compatibility

• Inlining from a library enforces a specific version
  ➔ All bets are off if the runtime classpath has a different version

• Problematic for library authors: forces specific versions for dependencies
Binary Compatibility

- Inlining from a library enforces a specific version
  - All bets are off if the runtime classpath has a different version
- Problematic for library authors: forces specific versions for dependencies
Binary Compatibility

• Inlining from a library enforces a specific version
  ➔ All bets are off if the runtime classpath has a different version

• Problematic for library authors: forces specific versions for dependencies

Conflict when depending on both someLib and otherLib
Binary Compatibility

- Library authors: don't inline from the classpath
  - Harsh limitation: `Range.foreach` stays slow
- Deployed applications: optimize freely
  - Ensure same classpath at runtime
  - Consider building dependencies from source
- Future: safe but restricted cross-library inlining?
Optimizer Sub-Agenda

- Why do we need a compile-time optimizer?
- Features and roadmap
- Inherent limitations
- Distant future: a whole-program optimizer?
Outlook: Global Optimizer

• Would solve the binary compatibility issues
  ➔ Libraries are compiled without optimizations
  ➔ Final program compilation optimizes everything

• Approach works successfully in Scala.js
Outlook: Global Optimizer

• More liberty under closed-world assumption
  ➔ Eliminate unused (public) code
  ➔ Global data flow analyses
  ➔ Assume closed type hierarchies

• Challenges
  ➔ Just classfiles? New IR (like Scala.js, TASTY)?
  ➔ Support run-time reflection?
Thank You!