Tales from Compiling to the JVM

Lukas Rytz, Scala Team @ Lightbend

Scala 2.12 in Bytecode

- Java-style encoding for lambdas
- Default methods for traits
- A new bytecode optimizer

Overview ⛅

- InvokeDynamic to compile lambdas
  - Indy under the hoods
  - Challenges: boxing, specialization, captures, ...
- Default Methods to compile traits
  - Supercalls and invokespecial
  - Performance considerations

InvokeDynamic (indy)

- Bootstrap method
    - Runs *once*, when indy is first executed
    - Arguments from the bytecode descriptor
- Target method
    - Invoked on each indy execution
    - Acts on the ordinary JVM stack

InvokeDynamic (indy)

```
invokedynamic name(argTps)resTp bsREf bsArgs
```

`MethodHandle` reference to bootstrap method

```
def myBootstrap(predefArgs, customArgs): CallSite
```

```
class CallSite {
  val/var target: MethodHandle // invoked method
}
```

Scala

Lightbend

Indy-Lambda

```
(s: String) => s.trim
```

```
def $anonfun(s: String) = s.trim
```

SAM name

SAM interface

```
invokedynamic apply()Ls/Function1;
  LambdaMetafactory.altMetafactory    // bootstrap
  (Lj/l/Object;)Lj/l/Object;          // SAM type
  A.$anonfun(Lj/l/String;)Lj/l/String // body meth
```

LambdaMetaFactory

- Synthesizes and loads a new class that implements the SAM interface
- Returns a `CallSite` with a target that creates a new instance
  - If nothing is captured, the `CallSite` target returns a singleton instance

LMF Boxing Adaptation

Erasure: `(Obect)String`

```
trait T[T] { def apply(x: T): String }

val f: T[Int] = (x: Int) => "x:" + x

<synth> def anonfun$f(x: Int) = "x:" + x
```

LMF supports such differences, adds an unboxing conversion

PR #4497

Boxing Scala vs Java

```
val a: Int = (null: Integer) // 0 in Scala
int a      = (Integer) null; // NPE in Java
```

```
trait T[T] { def apply(x: T): String }
val f: T[Int] = (x: Int) => "x:" + x

f.asInstanceOf[T[Any]].apply(null)
```

```
<synth> def anonfun$f$adapted(x: Object) =
  anonfun$f(unboxToInt(x))
```

Specialization

```
trait A[@spec(Int) T]  { def apply(x: T): Int }
class C extends A[Int] { def apply(x: Int) = x }
```

```
trait A {
  def apply(x: Object): Object
  def apply$mcI$sp(x: Int): String = apply(box(x))
}
class C extends A {
  def apply(x: Object) = apply$mcI$sp(unbox(x))
  def apply$mcI$sp(x: Int) = x
}
```

LMF 💔 Specialization

```
trait A[@spec(Int) T] { def apply(x: T): Int }

val f: T[Int] = x => x
```

Should not box

This is the SAM,
LMF will implement it

```
trait A {
  def apply(x: Object): Object
  def apply$mcI$sp(x: Int): String = apply(box(x))
}
```

Scala

Lightbend

Don't subvert `@spec`

- `FunctionN`: hand-written specializations where the specialized method is abstract
- User-defined SAM types: don't use LMF, create an anonymous class at compile-time

`$outer` for local classes

```
class A {
  def f = () => { class C; serialize(new C) }
}
```

```
class $anofun { // 2.11
  def apply() = { class C; serialize(new C) }
}

class A { // 2.12
  def $anonfun { class C; serialize(new C) }
}
```

`$outer` is `$anonfun`

`$outer` is A

A Final's Secret 💌

```
class A {
  class B
  final class C
}

scala> classOf[A#B].getDeclaredFields.toList
List(public final A A$B.$outer)

scala> classOf[A#C].getDeclaredFields.toList
List()

scala> (new a1.C:Any) match {case _:a2.C => "OK"}
OK
```

Fix `$outer` Capture 🎣

- Mark local classes with no subclasses `final`

- The existing logic eliminates the `$outer` field if it is not needed

PR #5099

More $outer Capture

```scala
class A {
  val f = () => { def local = 1; local }
}
```

- 2.11: `local` is lifted to the `$anonfun` class
- 2.12: `local` ends up in `A`, the closure needs to capture and store the outer A
  - Emit local methods static when possible

Scala

Lightbend

Lazy Val Init Lock 🔐

```
class A {
  def f = () => { lazy val x = 1; x }
}

// generates
def x(v: IntRef) = { if(!init) lzyCompute(v) .. }
def lzyCompute(v: IntRef) = this.synchronized{..}
```

- 2.11: methods generated in `$anonfun`. 2.12: in `A`
- Contention on the A instance, deadlocks

Local Lazies à la Dotty ⏺️

- Observation: local lazies are boxed anyway
- Synchronize initialization on the box itself

```
def f = () => { lazy val x = 1; x }

// generates
def x(v: LazyInt) =
  if (v.init) v.value else lzyCompute(v)

def lzyCompute(v: LazyInt) = v.synchronized{..}
```

Overview

- InvokeDynamic to compile lambdas
  - Indy under the hoods
  - Challenges: boxing, specialization, captures, ...
- Default Methods to compile traits
  - Supercalls and invokespecial
  - Performance considerations

Default Methods

- Looks like it could be simple:

```
trait T { def f = 1 }

interface T { default int f() { return 1; } }
```

- Challenges
  - Multiple inheritance / linearization
  - Super calls

Forwarders ↪️ 2.11

```
trait T { def f = 1 }
class C extends T

interface T {
  int f();
}
class T$class {
  public static int f(T $this) { return 1; }
}
class C implements T {
  public int f() { return T$class.f(this); }
}
```

Scala

Lightbend

Forwarders ↪️ 2.12

```
class A { def f = 1 }
trait T extends A { override def f = 2 }
class C extends T
```

T and A are unrelated

```
interface T { default int f() { return 2; } }

class C extends A implements T {
  public int f() { T.super.f(); }
}
```

invokespecial

Scala Lightbend

JUnit 4 💔 Default Methods

```
trait T { @Test def runMe() { .. } }
@RunWith(..) class C extends T

// Test C failed: No runnable methods
```

- `-Xmixin-force-forwarders:junit`
  - Enabled by default in RC1
- JUnit 5 will support default methods

Default Methods Perf 🏎

- JIT compiler does not fully optimize default methods
- Scala compiler: 15% slower without forwarders
  - Likely affects other Scala projects
  - No forwarders in RC1 – feedback welcome!
  - Try `-Xmixin-force-forwarders:true`

Invokespecial 💍

- Used for private methods, constructors, super calls
- Method lookup is dynamic!

```
class C extends B {.. invokespecial A.f ..}
```

  - If A is a superclass (transitive) of C, lookup starts at B, otherwise it starts at A
  - Method lookup in superclasses, then interfaces

PR #5369

Bug in 2.11

```scala
class A { def f = 1 }
class B extends A { override def f = 2 }
trait T extends A
class C extends B with T {
  def t = super[T].f // should be 1
}

// invokespecial A.f in class C
// Lookup for f starts in B (not A)

// 2.12: "error: cannot emit super call"
```

Invokespecial 👪 Parents

```
trait T { def f = 1 }
trait U extends T
class C extends U { def t = super.f }
```

invokespecial `T.f` is not allowed unless C implements T

```
trait T {
  default int f() { return 1; }
  static int f$($this: T) {
    $this.f();
  }
}
class C { def t = T.f$(this) }
```

invokespecial `T.f`

Wrapping Up 🎁

scala-library-2.11.8.jar	5.5M
scala-library-2.12.0-M3.jar	5.4M
scala-library-2.12.0-M4.jar	5.0M
scala-library-2.12.0-M5.jar	4.4M
scala-library-2.12.0-RC1.jar	4.3M

10 Years Against the Spec

```
object O { }

// Scala 2.5 (2007) - 2.11 (2016)
public final class O$ {
  public static final O$ MODULE$
  public static <clinit> {
    new O$()
  }
  private <init> {
    MODULE$ = this
  }
}
```

Illegal by spec, Java 9: can assign static final field only in `<clinit>`

Scala

Lightbend

Move to <clinit>?

```
class C { println(O.f) }
object O { new C(); def f = 1 }

public final class O$ {
  public static final O$ MODULE$
  public static <clinit> { new O$() }
  private <init> {
    MODULE$ = this
    new C().log
  }
}
```

Thank You!

Scala

Lightbend