Changes on the Surface

• Lambdas for SAM types: Java 8 interop
• Scaladoc: a new look
• REPL: tab-completion
• Minor changes in the standard library
Changes Under the Hood

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

• I: Changes on the surface
• II: Some internals of HotSpot
• III: The Scala 2.12 Optimizer
• IV: New Bytecode in Scala 2.12
  • InvokeDynamic for Lambdas
  • Default Methods for Traits
Lambdas for SAM Types

• SAM = "Single Abstract Method"

• Lambda syntax to create SAM type instances

```scala
scala> new Thread(() => println("hi")).run
hi
```

• Same as in Java 8, mostly useful for interop
Interop Example: Streams

```scala
scala> val myList = java.util.Arrays.asList("a1", "a2", "b1", "c2", "c1")
scala> myList.stream.
    filter(_.startsWith("c")).
    map(_.toUpperCase).
    sorted.
    foreach println
C1
C2
```
Scaladoc's New Look

http://www.scala-lang.org/api/2.12.0-RC2
REPL tab-completion

• Available in 2.11.8, 2.12.0

• Uses the presentation compiler (Scala IDE, ensime)

```
scala> List("a", "b").map(_.<TAB>
...
toDouble toLowerCase toUpperCase

scala> List("a", "b").map(_.<TAB>
scala> List("a", "b").map(_.toUpperCase<TAB>

def toUpperCase(): String
def toUpperCase(x$1: java.util.Locale): String
```
Right-Biased Either

```scala
def toInt(s: String): Either[MyError, Int] = 
   ...

def sum(a: String, b: String) = for {
   x <- toInt(a)
   y <- toInt(b)
   } yield x + y

sum: (a: String, b: String)Either[MyError,Int]

sum("1", "2")
res1: Either[MyError,Int] = Right(3)
```
Changes to the Library

• A mutable.TreeMap (sorted map)

• Deprecated JavaConversions: only explicit asScala / asJava using JavaConverters

• Various minor performance improvements
Agenda

• I: Changes on the surface
• II: Some internals of HotSpot
• III: The Scala 2.12 Optimizer
• IV: New Bytecode in Scala 2.12
  • InvokeDynamic for Lambdas
  • Default Methods for Traits
Java Bytecode Example

// def f(x: Int) = x + 1

public f(I)I
  ILOAD 1
  ICONST_1
  IADD
  IRETURN
val stack: Stack[Any]
val frameIdx: Stack[Int]

def intp(code: List[Instr]) = {
  code.head match {
    case IConst(n) => stack.push(n)
    case ILoad(n) => stack.push(stack(frameIdx.head + n))
    case IAdd => stack.push(stack.pop() + stack.pop())
    ...
  }
  intp(code.tail)
}

frameIndex.head →

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>x+1</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

this

Scala
Making it Fast

• Bytecode that is executed "many" times is compiled to native code (assembly)

• Two compilers in JDK 7+
  • C1 ("client" compiler in JDK 6): fast
  • C2 ("server"): advanced optimisations, slower
"Many" Executions

- Method invocation counter: 2k → C1, 15k → C2
  - C1 assembly is *instrumented*: update invocation counter, other metrics
- Loop counter: 60k iterations → C1, ... → C2
  - Assembly entrypoint at the loop start
  - Switch to the assembly code: "on-stack replacement" (OSR)
Optimizations

"JVM JIT compilation overview" by Vladimir Ivanov
http://www.slideshare.net/ZeroTurnaround/vladimir-ivanovjvmjitcompilationoverview-24613146

- compiler tactics
  - delayed compilation
  - tiered compilation
  - on-stack replacement
  - delayed reoptimization
  - program dependence graph rep.
  - static single assignment rep.
- proof-based techniques
  - exact type inference
  - memory value inference
  - memory value tracking
  - constant folding
  - reassociation
  - operator strength reduction
  - null check elimination
  - type test strength reduction
  - type test elimination
  - algebraic simplification
  - common subexpression elimination
  - integer range typing
- flow-sensitive rewrites
  - conditional constant propagation
  - dominating test detection
  - flow-carried type narrowing
  - dead code elimination
- language-specific techniques
  - class hierarchy analysis
  - devirtualization
  - symbolic constant propagation
  - autobox elimination
  - escape analysis
  - lock elision
  - lock fusion
  - de-reflection
- speculative (profile-based) techniques
  - optimistic nullness assertions
  - optimistic type assertions
  - optimistic type strengthening
  - optimistic array length strengthening
  - untaken branch pruning
  - optimistic N-morphic inlining
  - branch frequency prediction
  - call frequency prediction
- memory and placement transformation
  - expression hoisting
  - expression sinking
  - redundant store elimination
  - adjacent store fusion
  - card-mark elimination
  - merge-point splitting
- loop transformations
  - loop unrolling
  - loop peeling
  - safepoint elimination
  - iteration range splitting
  - range check elimination
  - loop vectorization
- global code shaping
  - inlining (graph integration)
  - global code motion
  - heat-based code layout
  - switch balancing
  - throw inlining
- control flow graph transformation
  - local code scheduling
  - local code bundling
  - delay slot filling
  - graph-coloring register allocation
  - linear scan register allocation
  - live range splitting
  - copy coalescing
  - constant splitting
  - copy removal
  - address mode matching
  - instruction peepholeing
  - DFA-based code generator
Inlining

- Inlining enables most other optimizations
- Duplicated code can be specialized

Heuristics decide what to inline:
- Small methods (35 bytes) are inlined
- "Hot" callsites are inlined (up to 325 bytes)
- Max depth of 9
Inlining Virtual Methods

• Java / Scala has virtual methods by default

• Many callsites are monomorphic
  • Class Hierarchy Analysis (CHA): a virtual method with no overrides can be inlined (C1, C2)
  • Profile-based inlining (C2): inline if the receiver at a callsite is always the same
  • Profiles collected by interpreter and C1 assembly
Speculative Inlining

• Assumptions can invalidate compiled code
  • A method gets an override when a new class is loaded
  • A new receiver type reaches a (previously monomorphic) callsite

• Deoptimization: the assembly is discarded, the interpreter takes over
Learn (a lot) More

- "JVM Mechanics", talk by Doug Hawkins (Azul)
  - Slides: http://www.slideshare.net/dougqh/jvm-mechanics-when-does-the
  - Video: https://www.youtube.com/watch?v=E9i9NJeXGmM
Agenda

• I: Changes on the surface
• II: Some internals of HotSpot
• III: The Scala 2.12 Optimizer
• IV: New Bytecode in Scala 2.12
  • InvokeDynamic for Lambdas
  • Default Methods for Traits
Megamorphic Callsites

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

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

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

```scala
(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
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
```

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

Slow
- Why? Not obvious...

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

Still slow (same as before)!
- Why? IntRef
- Escape analysis fails..
Closure Elimination

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

Eliminate the closure allocation

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

Fast! JVM escape analysis kicks in.
Box Elimination

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

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

Local var instead of IntRef

Same as before! JVM optimizes the IntRef just fine.
Bars

- foreach
- while + IntRef (captured)
- while + IntRef (no closure)
- while + var

Time

- 0
- 3.25
- 6.5
- 9.75
- 13

~ 4x
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

• I: Changes on the surface

• II: Some internals of HotSpot

• III: The Scala 2.12 Optimizer

• IV: New Bytecode in Scala 2.12
  • InvokeDynamic for Lambdas
  • Default Methods for Traits
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)

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

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

MethodHandle reference to bootstrap method
Indy-Lambda

(s: String) => s.trim

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

invokedynamic apply()Lscala/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

```
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.
Boxing 📦 Scala vs Java

```scala
// Scala
val a: Int = (null: Integer) // 0 in Scala

// Java
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> defanonfun$f$adapted(x: Object) = anonfun$f(unboxToInt(x))
```
Specialization

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

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

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

---

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))  
}
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
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
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

```scala
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{..}
```
Agenda

• I: Changes on the surface
• II: Some internals of HotSpot
• III: The Scala 2.12 Optimizer
• IV: New Bytecode in Scala 2.12
  • InvokeDynamic for Lambdas
  • Default Methods for Traits
Default Methods

• Looks like it could be simple:

```scala
trait T { def f = 1 }
interface T { default int f() { return 1; } }
```

• Challenges
  • Super calls
  • Multiple inheritance / linearization
  • Performance
Invokespecial 🎩

• Used for private methods, constructors, super calls

• Method lookup is dynamic!

```scala
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
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
```scala
trait T {
  def f = 1
}
```

### Trait U extends T
```scala
trait U extends T
```

### Class C extends U
```scala
class C extends U {
  def t = super.f
}
```

### Trait T
```scala
trait T {
  default int f() {
    return 1;
  }
  static int f$(this: T) {
    this.f();
  }
}
```

### Class C
```scala
class C {
  def t = T.f$(this)
}
```
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); }
}
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
JUnit 4 ❤ Default Methods

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

// Test C failed: No runnable methods

- `-Xmixin-force-forwarders:junit`
  - Enabled by default

- JUnit 5 will support default methods
Default Methods Perf

• Observation: using default methods degrades startup performance

• Compiling a simple HelloWorld.scala
  • Relying on default methods: 3.9s
  • With mixin forwarders: 2.9s

• Hot performance (sbt) is not affected
Mixin Forwarders in RC2

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

interface T {
  default public static int f$(T $this) {
    return $this.f();
  }
  default public int f() { return 1; }
}

class C implements T {
  public int f() { return T.f$(this); }
}
JVM and Default Methods

- Class Hierarchy Analysis is disabled for default methods
  - Prevents inlining in C1, likely other optimizations
- Class loading: populating class vtables with default methods is slow
  - Search through all ancestors
  - Mixin forwarders avoid this for classes; 60% speedup on `scala -version`
  - Still a hotspot in `generate_default_methods`, when loading interfaces
Thank You!