Scala 2.12 and Java 8
More fun together

Adriaan Moors

Typesafe

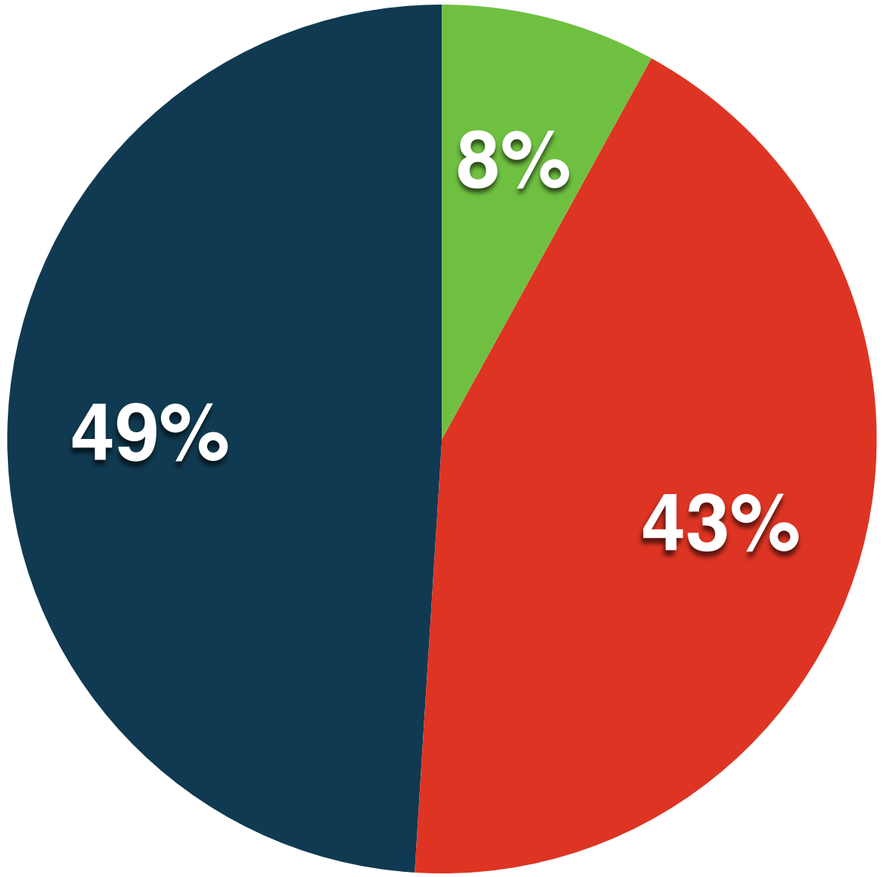
12 Nov 2014

Director's cut

Scala: practical FP & OO

- Unify functions & objects: more productive!
- Developed at EPFL for over a decade
- Vibrant, growing community!
- Typesafe stewards the project, fosters community
  - Reactive Platform implemented in Scala (has both Scala/Java8 API)

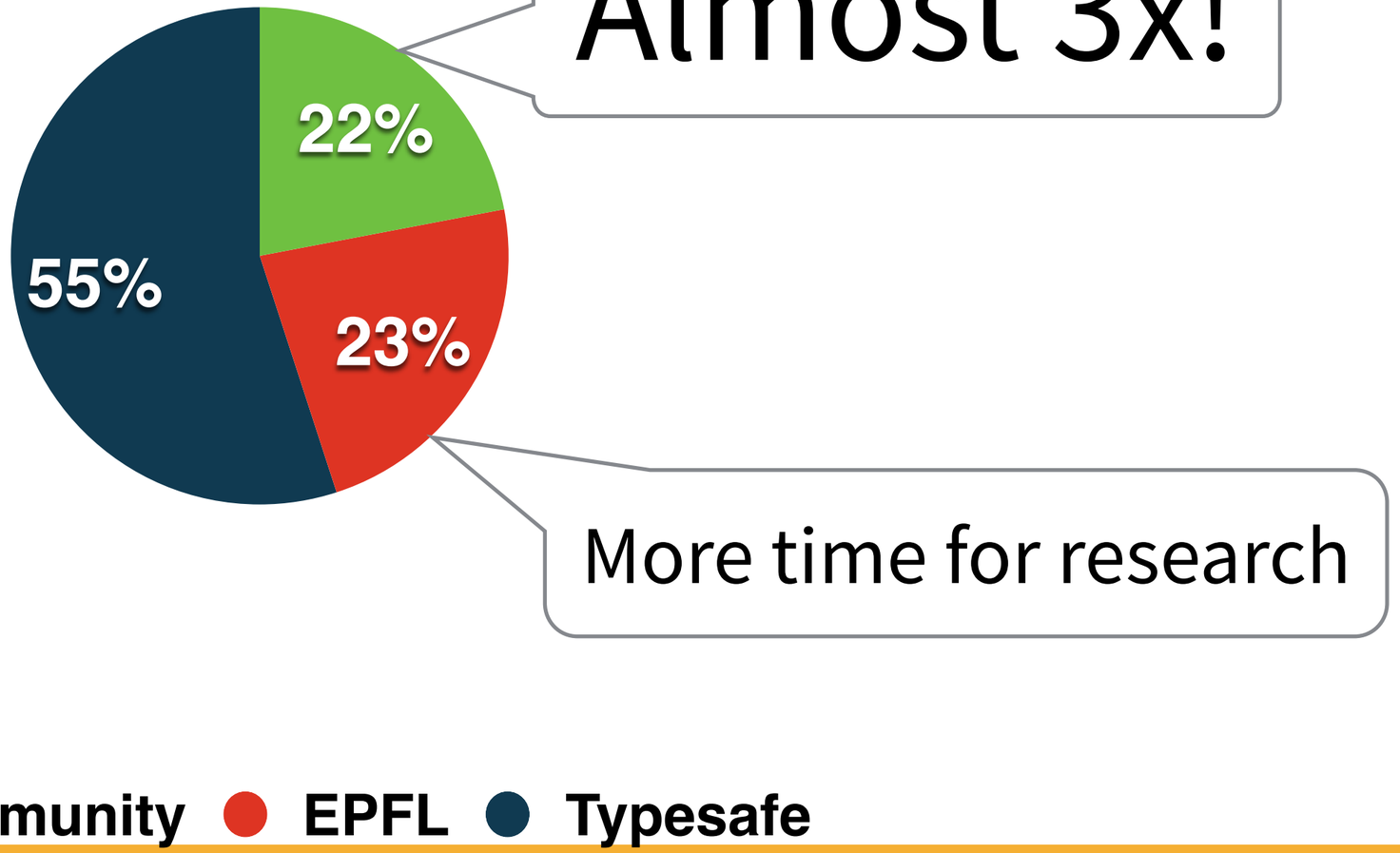
Scala 2.10 commits



>> absolute #:
macros, value classes,
implicit classes,
language imports,
new pattern matcher,
…

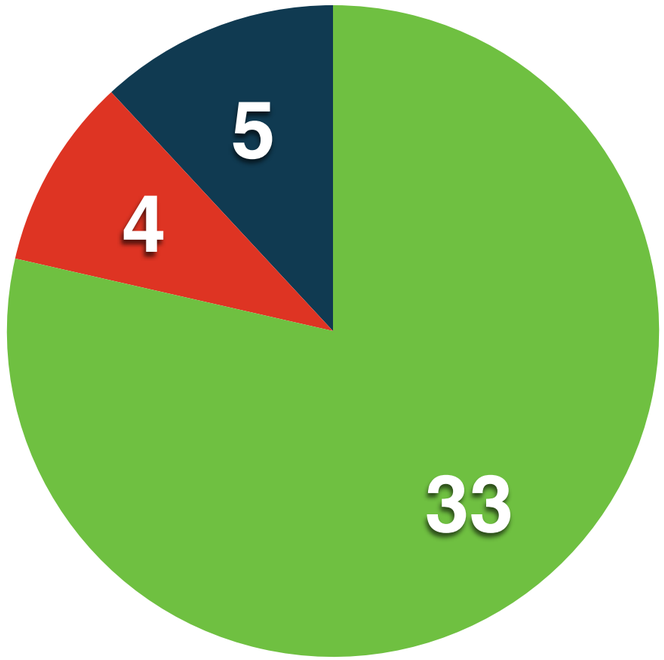
Scala 2.11 commits

<< absolute #:
stabilisation,
modularisation.



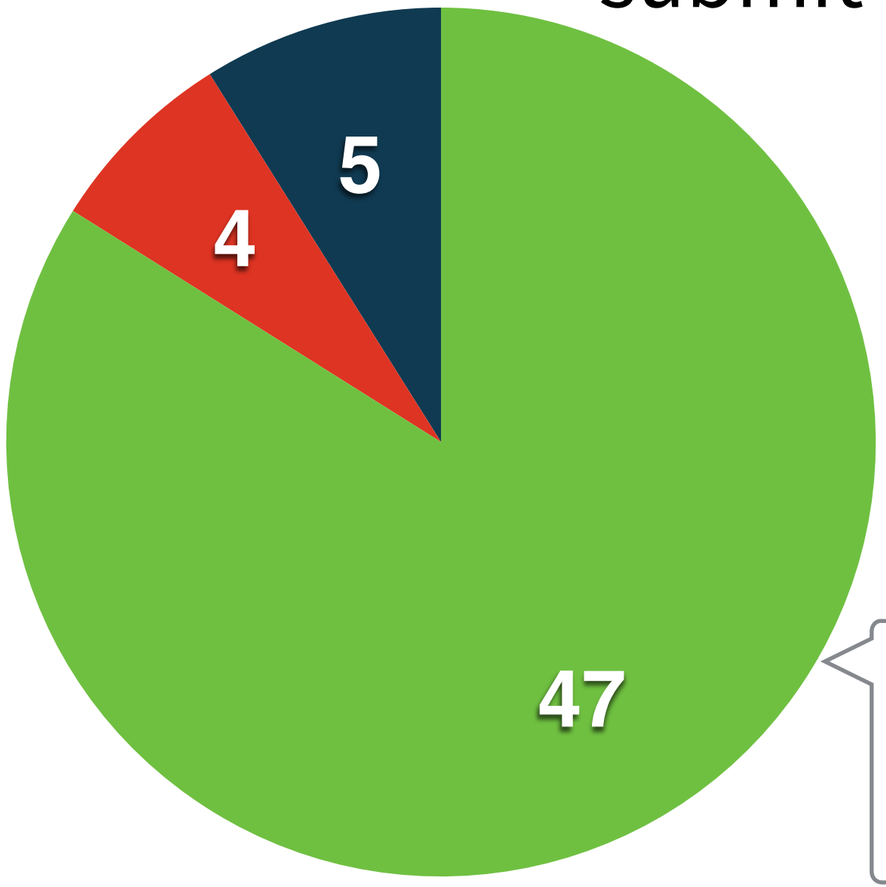
Scala 2.10 Team Size

core team = members
submit 90% of total commits



Scala 2.11 Team Size

core team = members
submit 90% of total commits



+ 42%

Numbers

	commits 2.10	commits 2.11	team (90%)	team (90%)	everyone	everyone
community	8%	**22%**	33	**47**	69	**104**
epfl	43%	**23%**	4	4	19	**13**
typesafe	49%	54%	5	5	22	22
all	4495	2582	42	56	110	139

reproduce: git shortlog -sn --no-merges 2.11.x --not 2.10.x

DEVOXX TM

Thank you!

Keep up the good work — no pressure.

2.12 Roadmap

scala-lang.org/news/2.12-roadmap

- Move to Java 8
  - Be good JVM citizen, enjoy new features!
  - Must target Java 8 for real benefit (`INDY`, `default`, fork-join).
- Legacy, long-term support
  - Scala 2.11 & 2.12 match features (where possible on JDK 6)
  - 2.12.0 (& last public 2.11.x) scheduled after Java 7 public EOL (2016)
  - Commercial support? Give us a call.

Talk Roadmap

- **Functional Programming** in Scala
- Using functional **Java 8 APIs** in **Scala** (SAM synthesis)
- Using functional **Scala APIs** in **Java 8** (SAMifying =>)
- Compiling Scala **closures**: **`invokedynamic`**
- Scala 2.12's Java 8 **back-end** (`default` methods for traits)

(Functional) Programming in Scala

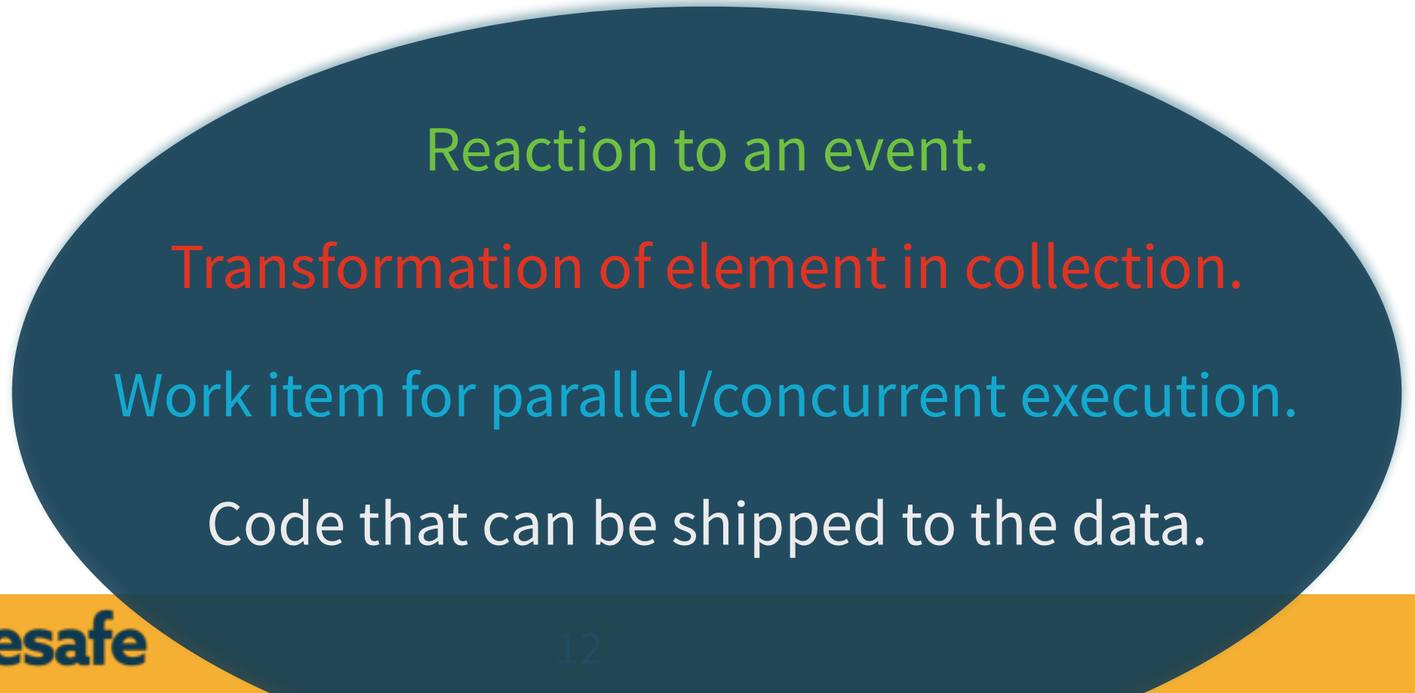
Divide into abstractions of the right size

&

conquer^W **compose**.

Functional Programming in Scala

Often, **functions** are the right size.



(Functional) Programming in Scala

Need a bigger abstraction?

`trait & object`

Definitions…

OO: programming with objects

closed set of methods, open set of types

FP: programming with functions

open set of function, closed set of types

orthogonal:pure/impure, lazy/strict,
static/dynamic typing, math/reality

FP ≈ OO in the small

object = bundle of functions

function ≈ object with one method

Scala = unifier

Functional Programming in Scala

Functions: **easy** to **understand**
knowing (only) their arguments
(if they are **pure**).

Functional Programming in Scala

Purity = *usually* good idea,
underpins **scaling**
& **fault isolation**.

Scala nudges you towards Purity

Focus on expressions.

Statements cause side-effects.

```
var x = 0
if (someCondition) x = 1
else x = -1

…
return x
```

```
val x =
  if (someCondition) 1
  else -1

…
x
```

Simplify: no need for ternary operator.

`var` / `val` same # letters

Scala is pragmatic

If mutation makes more sense

```
var procCount = 0
val f = { (x: Foo) => procCount += 1; x.process }
```

go for it!

var

CONTROVERSY?

Methods take objects
as arguments.

CONTROVERSY

Higher-order functions
compose function values.

Composing functions

- List: `List(1, 2, 3) foreach println`
- Option: abstract over null checks
- Validation: reason about failure
- Database: persists data too, optimize it
- Async: delay computation, need handle for it

flatMap that shit!

Types are high-tech plumbing

Types guide **safe composition**.

There's more to life than plumbing

Type checking kills common **run-time bugs.**

Focus testing on the non-trivial.

Plumbing should stay out of the way

Type inference minimizes cost of **`typing`**.

Exposed plumbing

Java's type inference
still limited
type boilerplate…

Ever wonder what keeps scalac busy?

Scala: powerful local type inference.
You only write **public type signatures**.

(*Or not* — but, I'd recommend it.)

So, what does it take to FP?

- function = value
  - compact syntax to define them
- expression-based – statements cause side-effects
  - immutable by default
- type system to make combinators safe
  - type inference to keep types out of the way
- hide the monads: for comprehensions, scala.async
- pattern matching!

Meet SAM

- Elegant retro-fit of function types onto OO VM & language
  - Function type = interface with one abstract method
  - (Can still have default methods)
- Before:

```
new Thread(new Runnable{def run() = ???})
```

- SAM:

```
new Thread(() => ???)
```

Using Java 8 Stream from Scala REPL

```
$ scala -Xexperimental # need 2.11.5-SNAPSHOT

scala> import java.util.Arrays, java.util.stream.Stream

// List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");
scala> val myList = Arrays.asList("a1", "a2", "b1", "c2", "c1")
myList: java.util.List[String] = [a1, a2, b1, c2, c1]
```

Scala code: gist.github.com/adriaanm/892d6063dd485d7dd221

Original Java 8 code: github.com/winterbe/java8-tutorial

Using Java 8 Stream from Scala REPL

```
/*
myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)      // s -> s.toUpperCase
    .sorted()
    .forEach(System.out::println); // s -> System.out.println(s)
*/
scala> myList.stream.filter(s => s.startsWith(“c”)).
         map(_.toUpperCase).sorted.forEach(println)
C1
C2
```

Original Java 8 code: github.com/winterbe/java8-tutorial

Using Java 8 Stream from Scala REPL

```
/*
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    });
*/
scala> Stream of ("d2", "a2", "b1", "b3", "c") filter {
         s => println(s"filter: $s"); true
       }
res1: java.util.stream.Stream[String] =
java.util.stream.ReferencePipeline$2@4648ce9
```

Original Java 8 code: github.com/winterbe/java8-tutorial

Single abstract method synthesis

```
scala> Stream of ("d2", "a2", "b1", "b3", "c") filter {
         s => println(s"filter: $s"); true
       }

// Behind the scenes, scalac generated:
scala> Stream of ("d2", "a2", "b1", "b3", "c") filter {
  def test$body(s: String): Boolean = {
    println(s"filter: $s"); true
  }
  new Predicate {
    def test(s: String): Boolean = test$body(s)
  }
}
```

Original Java 8 code: github.com/winterbe/java8-tutorial

But what about the types?

```
scala> val s = myList.stream.map(_.toUpperCase)
s: Stream[?0] = ReferencePipeline$3@55d8f6bb

// wait what!? (I told you this was experimental)
```

Original Java 8 code: github.com/winterbe/java8-tutorial

:power mode to the rescue!

```
scala> :power
** Power User mode enabled - BEEP WHIR GYVE **
** :phase has been set to 'typer'.           **
** scala.tools.nsc._ has been imported       **
** global._, definitions._ also imported     **
** Try  :help, :vals, power.<tab>            **

// Can now call into the compiler running behind the scenes
```

Original Java 8 code: github.com/winterbe/java8-tutorial

Mind your wildcards

```
…
s: Stream[?0] = ReferencePipeline$3@55d8f6bb

scala> typeOf[s.type].widen.typeArgs.head.typeSymbol.info
res14: $r.intp.global.Type =  >: String

// Java wildcards are pretty tricky…
// inference will improve by 2.12.0
```

Original Java 8 code: github.com/winterbe/java8-tutorial

Stream's map signature

Java

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

Scala

```
def map[R](mapper: T => R): Stream[R]
```

Stream's map signature

Java

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

Scala

```
def map[R](mapper: T `Function1` R): Stream[R]
```

Stream's map signature

Java

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

Scala

```
def map[R](mapper: Function1[T, R]): Stream[R]
```

Zooming in on Function

Java

```
Function<? super T, ? extends R>
```

Scala

```
Function1[T, R]
```

Variance: use-site v definition-site

Scalafied (use-site variance)

```
Function[_ >: T, _ <: R]
given trait Function[T, R]
```

=:=

Scala (definition-site variance)

```
Function1[T, R]
given trait Function1[-T, +R]
```

Using Java 8 `Optional` from Scala REPL

```
// legacy code:
class Outer(var nested: Nested = null)
class Nested(var inner: Inner)
class Inner(var foo: String)

/* Optional.of(new Outer())
   .flatMap(o -> Optional.ofNullable(o.nested))
   .flatMap(n -> Optional.ofNullable(n.inner))
   .flatMap(i -> Optional.ofNullable(i.foo))
   .ifPresent(System.out::println); */
scala> Optional.of(new Outer()).
         flatMap(o => Optional.ofNullable(o.nested)).
         flatMap(n => Optional.ofNullable(n.inner)).
         flatMap(i => Optional.ofNullable(i.foo)).
         ifPresent(println)
```

Original Java 8 code: github.com/winterbe/java8-tutorial

Idiomatic use of Scala's `Option`

```
// More idiomatically:
for { o   <- Option(new Outer())
      n   <- Option(o.nested)
      i   <- Option(n.inner)
      foo <- Option(i.foo) } println(foo)

// Desugars to:
 Option(new Outer()).foreach{
   (o: Outer) => Option(o.nested).foreach{
     (n: Nested) => Option(n.inner).foreach{
       (i: Inner) => Option(i.foo).foreach{
         (foo: String) => println(foo)        }}}}
```

Original Java 8 code: github.com/winterbe/java8-tutorial

Dress up Java's `Optional` like Scala's

```
import java.util.function.Consumer, java.util.Optional,
       Optional.{ofNullable => Option}

implicit class extForeach[T](o: Optional[T]) {
  def foreach(f: Consumer[_ >: T]) = o ifPresent f
}
```

Original Java 8 code: github.com/winterbe/java8-tutorial

Using Java 8 Optional from Scala REPL

```
// Re-interpreted using Optional.ofNullable, ifPresent
for { o   <- Option(new Outer())
      n   <- Option(o.nested)
      i   <- Option(n.inner)
      foo <- Option(i.foo) } println(foo)

// Now desugars to:
 Optional.ofNullable(new Outer()).ifPresent{
   (o: Outer) => Optional.ofNullable(o.nested).ifPresent{
     (n: Nested) => Optional.ofNullable(n.inner).ifPresent{
       (i: Inner) => Optional.ofNullable(i.foo).ifPresent{
         (foo: String) => println(foo)        }}}}
```

Original Java 8 code: github.com/winterbe/java8-tutorial

Getting friendly with Java 8

- In 2.12, `@FunctionalInterface scala.FunctionN`
- Can't do this in 2.11
  - > 1 `abstract` method, no `default` methods on Java 6
  - Next best thing: scala/scala-java8-compat

```
JFunction1<String, String> f1 = (String s) -> s;

// JFunction1<String, String> <: scala.Function1<String, String>:
scala.Function1<String, String> f2 = f1;
```

Getting friendly with Java 8

```
import static scala.compat.java8.JFunction.*;

// `func` = identity; steers type checker towards SAM synth
scala.Function1<String, String> f3 = func((String s) -> s);

// javac's type inference can infer the parameter type,
// based on the ascribed type of `f4`.
scala.Function1<String, String> f4 = func(s -> s)
```

Java 8 compatibility layer

```
package scala.compat.java8;

import scala.runtime.BoxedUnit;

public final class JFunction {
  public static <R> scala.Function0<R> func
     (JFunction0<R> f){ return f; }
  public static <T1, R> scala.Function1<T1, R> func
     (JFunction1<T1, R> f) { return f; }
  public static <T1> scala.Function1<T1, BoxedUnit> proc
     (JProcedure1<T1> p) { return p; }
```

Java 8 compatibility layer

```
package scala.compat.java8;

@FunctionalInterface
public interface JFunction1<T1, R> extends scala.Function1<T1, R> {
  @Override default <A> scala.Function1<T1, A>
      andThen(scala.Function1<R, A> g) {
        return scala.Function1$class.andThen(this, g);
      }
  @Override default <A> scala.Function1<A, R>
      compose(scala.Function1<A, T1> g) {
         return scala.Function1$class.compose(this, g);
      }
  …
}
```

scalac -Ydelambdafy originally by @jamesiry

- Java 8-style lambdas
  - Simulate on Java 6 (Scala 2.11 experiment)
  - Same bytecode as javac on Java 8 (2.12 default)
- Smaller anonymous function class files
  - No longer contains lambda body and required constant pool entries

Function literal expansion

```
class C(val c: Double) {
  def outer(x: Int) = (y: Int) => (x + y) * c
}
// expanded:
class C(val c: Double) {
  def outer(x: Int) = new samClass(x)
                // capture environment^^^
  private def body(y: Int, x: Int): Double = (x + y) * c

  class samClass(x: Int) extends (Int => Double) {
    def apply(y: Int) = body(y, x)
  }
}
```

Function literal expansion: INDY style

```
class C(val c: Double) {
  def outer(x: Int) = (y: Int) => (x + y) * c
}
// expanded:
class C(val c: Double) { // bootstrap: LambdaMetaFactory.meta
  def outer(x: Int) = (INDY
     LMF MT[JFunction1.apply] MH[body] // [static] sam + impl
     x) // [dynamic] capture environment

  private def body(x: Int, y: Int): Double = (x + y) * c
}
```

Delambdafy, meet INDY

matchmaker: @retronym

- Still lower body to method, but skip creating class statically;
- instead: `invokedynamic LambdaMetaFactory`
  - **bootstrap**: link call site based on static arguments
    - *once* per lambda *definition*
    - conceptually: create anonymous class (delegate to Java 8 RT)
  - **invocation**: capture environment (dynamic arguments)
    - create function value (instance of SAM type)
- Must wait for 2.12 to do this by default.

New back-end & optimizer

by @lrytz

- Reworking & integrating @magarciaEPFL's new back-end
  - Landing first in 2.11.x (behind flag)
  - Will be default in 2.12
- Uses ASM: optimizer deals with bytecode directly (no more IR)
- Improved inliner
  - Can disable inlining from classpath
  - Delambdafy simplifies closure elimination
- Eliminate more boxing (prim<->ref, closures, tuples)

Default methods for traits

Compile Scala traits to interfaces with default methods

Source compatibility more likely to imply binary compatibility

Promising, but early days (lrytz/traits-default-methods)

Default methods for traits: challenges

- Fields (encoded as getter/setter, field in class)
- Linearization & super (qualified super calls, super accessors)
  - super chain not known until traits mixed in to class
- Initialization (call `$init$` method at end of constructor)

Thanks! Questions!

See you at **Scaladays**!

SF: March 16-18

Amsterdam: June 8-10