Scala
Where It Came From
Where It is Going

Scala Days San Francisco

Martin Odersky

Scala
Where It Came From

Scala Days San Francisco

Martin Odersky

“Scala” is going nowhere

“Scala is a gateway drug to Haskell”

Recognizing this fact,
we should phase out the name “Scala”

Idea and Design: Sandro Stucky

Now Seriously...

Where It Came From

1980s	Modula-2, Oberon
1990-95	Functional Programming: λ calculus, Haskell, SML
1995-98	Pizza
1998-99	GJ, javac
2000-02	Functional Nets, Funnel

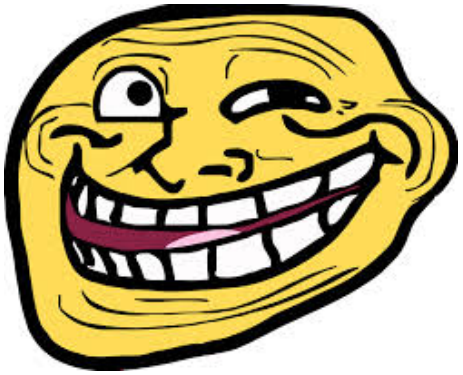
Motivation for Scala

- Grew out of Funnel
- Wanted to show that we can do a *practical* combination of OOP and FP.
- What got dropped:
  - Concurrency was relegated to libraries
  - No tight connection between language and core calculus (fragments were studied in the vObj paper and others.)
- What got added:
  - Native object and class model, Java interop, XML literals.

Why <XML>?

I wanted Scala to have a hipster syntax.

- Everybody uses [..] for arrays, so we use (..)
- Everybody uses <..> for types, so we use [..]
- But now we needed to find another use of <..>



What Makes Scala Scala?

Scala is

- functional
- object-oriented / modular
- statically typed
- strict
    - Closest predecessor: OCaml.
    - Differences: OCaml separates object and module system, Scala unifies them
    - OCaml uses Hindley/Milner, Scala subtyping + local type inference.

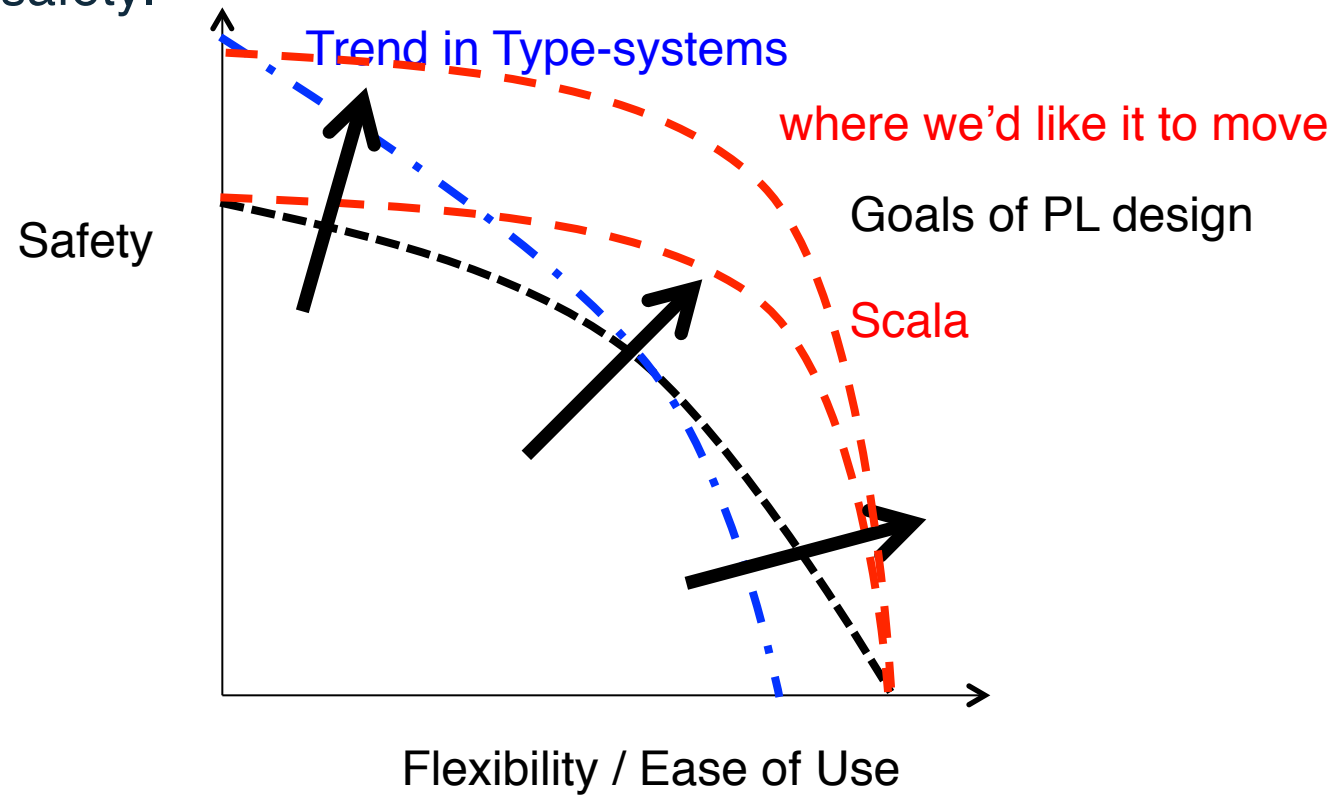
1st Invariant: A Scalable Language

- Instead of providing lots of features in the language, have the right abstractions so that they can be provided in libraries.

- This has worked quite well so far.
- It implicitly trusts programmers and library designers to “do the right thing”, or at least the community to sort things out.

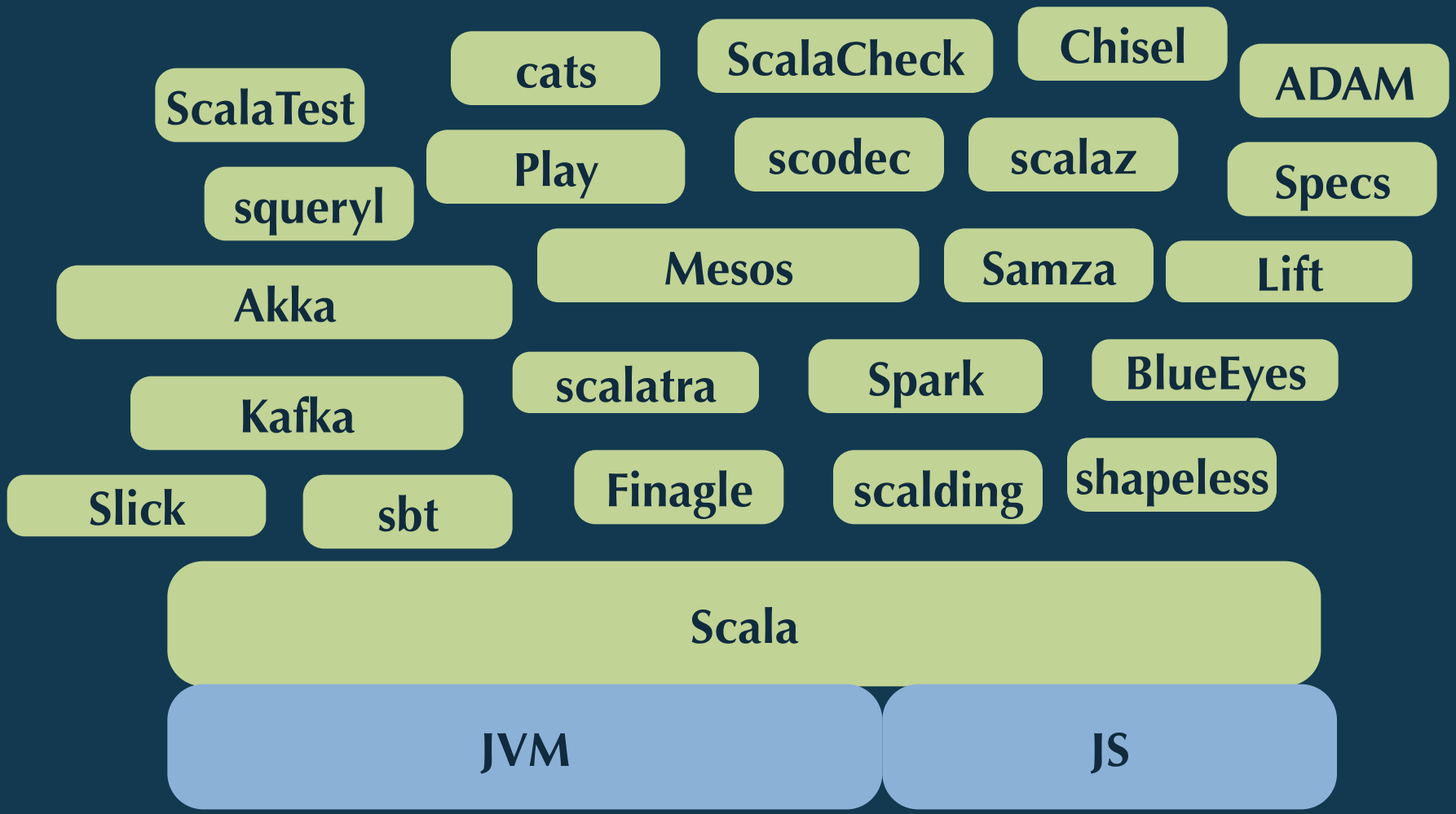
2nd Invariant: It's about the Types

- Scala's core is its type system.
- Most of the advanced types concepts are about flexibility, less so about safety.



The Present

An Emergent Ecosystem



New Environment: Scala.JS

Feb 5, 2015:

- Scala.JS 0.6 released
- No longer experimental!
- Fast
- Great interop with Javascript libraries

Why does Scala.JS work so well?

- Because of @srjd, and the great people who contribute.
- But also: It plays to the strengths of Scala
  - Libraries instead of primitives
  - Flexible type system
  - Geared for interoperating with a host language.

Tool Improvements

- Much faster incremental compiler, available in sbt and IDEs
- New IDEs
  - Eclipse IDE 4.0
  - IntelliJ 14.0
  - Ensime: make the Scala compiler available to help editing

ScalaIDE FOR ECLIPSE

Features Documentation Download News Team Roadmap

Version 4.0.0 Now Available!

Scala IDE provides advanced editing and debugging support for the development of pure Scala and mixed Scala-Java applications.

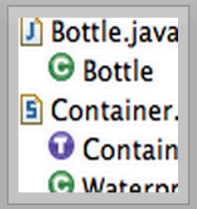
Now with a shiny Scala debugger, semantic highlight, more reliable JUnit test finder, an ecosystem of related plugins, and much more.

Download IDE or use the update sites

With:

- New debugger
- Faster builds
- Integrated ScalaDoc

Features



Support for Mixed Scala/Java Projects

Support for mixed Scala/Java projects and any combination of Scala/Java project dependencies, allowing straightforward references from Scala to Java and vice versa.

Editing

Twitter

Latest @ScalaIDE buzz:

Tweets Follow

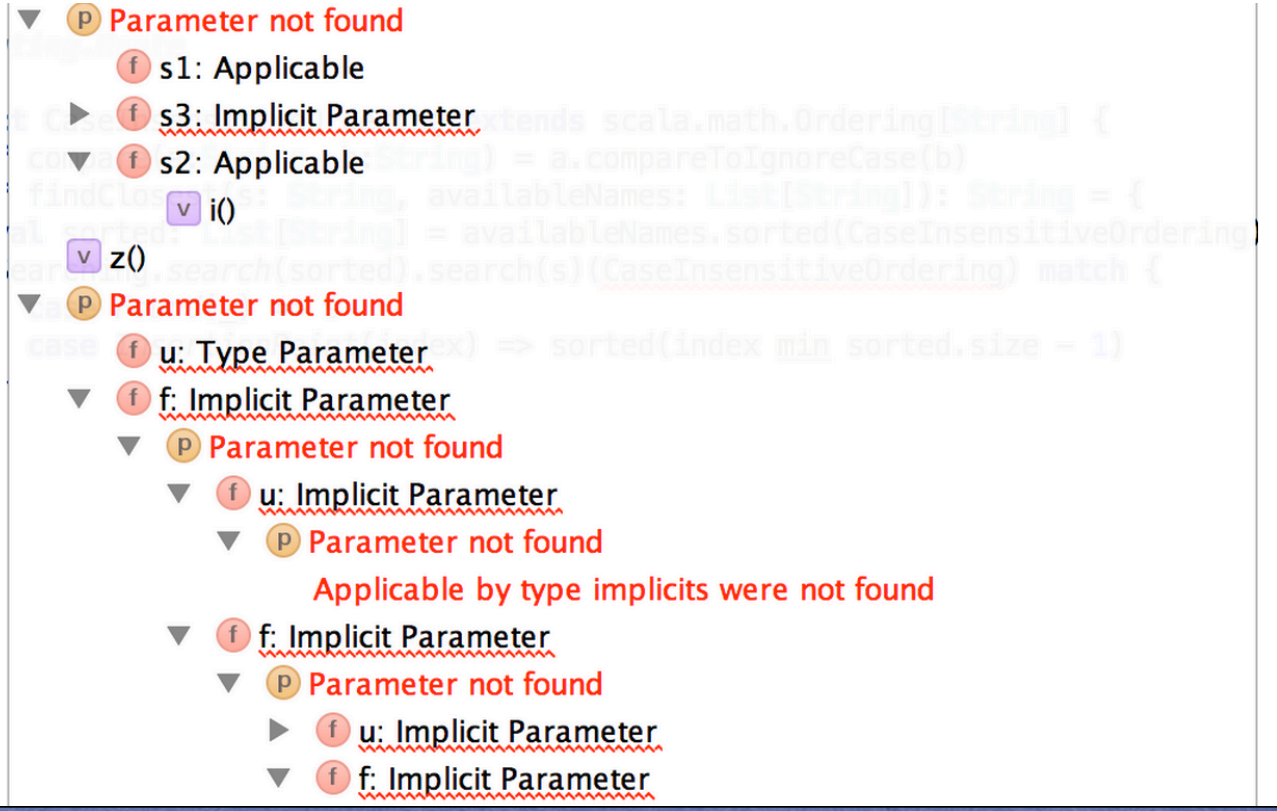
Scala IDE @ScalaIDE 6 Mar

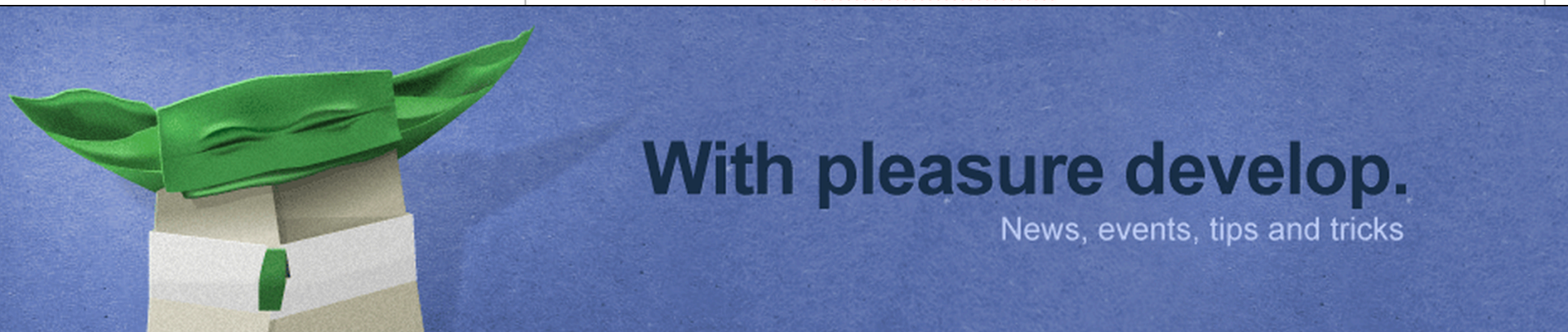
Scala IDE 4.0.0 on Scala 2.11.6, with Scala 2.10.5. Update your current install, or download the new version

Typesafe

IntelliJ 14.0 Scala plugin

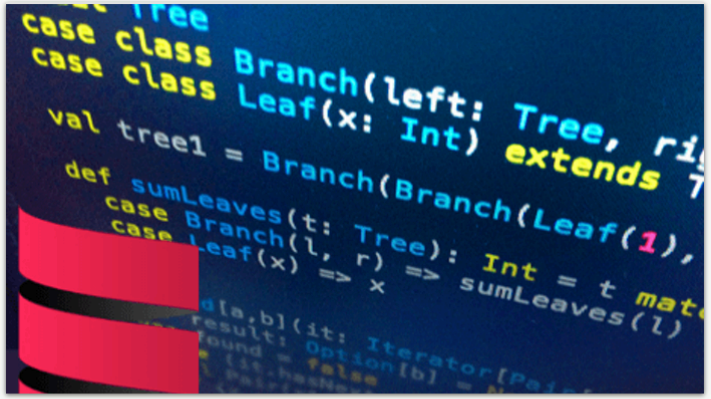
With a cool implicit tracker





Online Courses

coursera



Functional Programming Principles in Scala

Principles of Reactive Programming
April 13, 2015

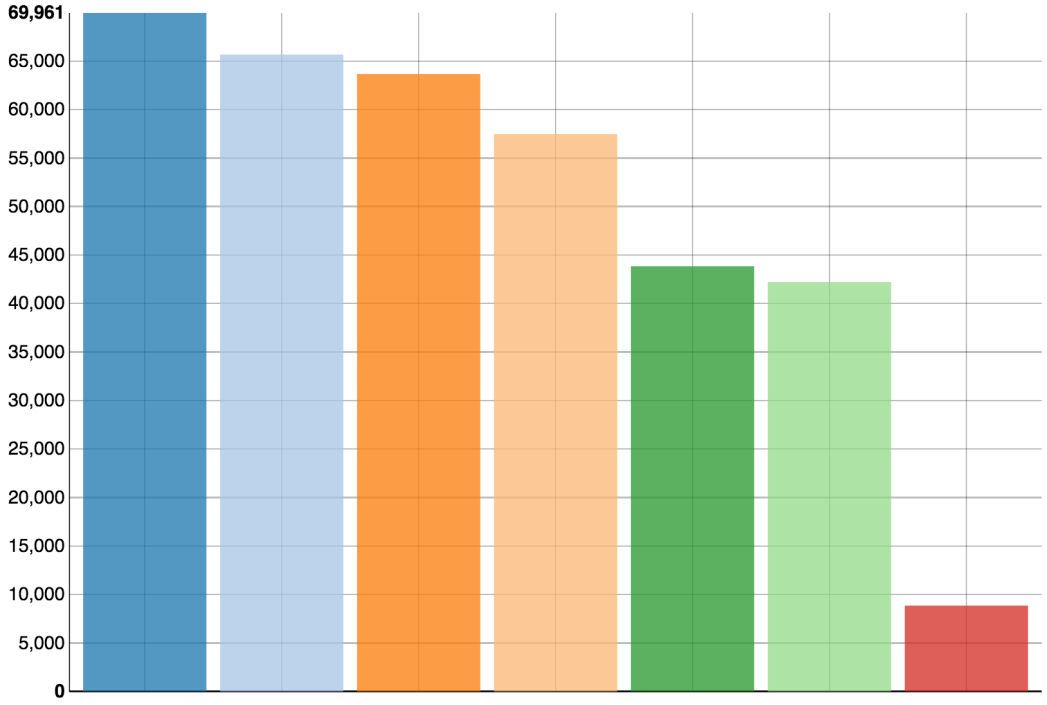
Session Stats

coursera

So far:

400’000 inscriptions

Success rate ~ 10%



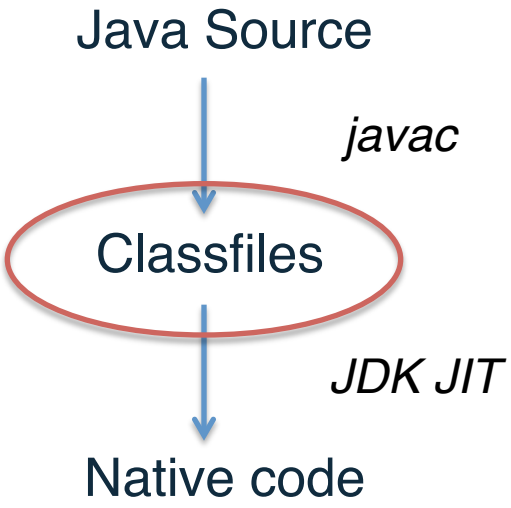
Where It Is Going?

Emergence of a platform

- Core libraries
- Specifications:
    - Futures
    - Reactive Streams
    - Spores
- Common vocabulary

→ Beginnings of a reactive platform, analogous to Java EE

JDK: The Core of the Java Platform



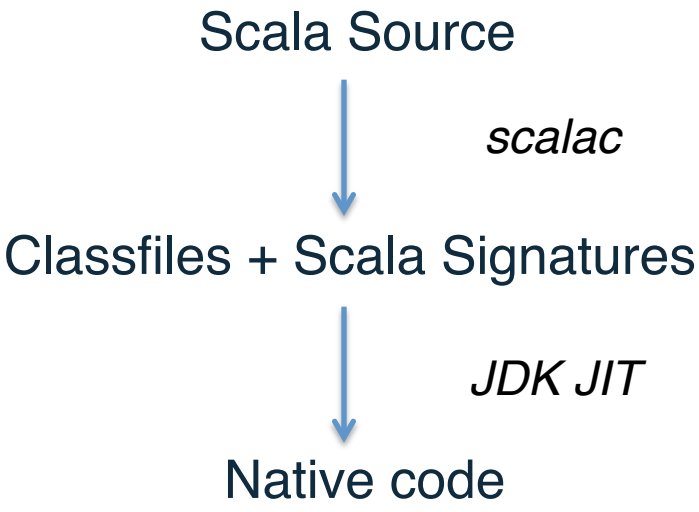
What Are Classfiles Good For?

- Portability across hardware
- Portability across OS/s
- Interoperability across versions
- Place for
  - optimizations,
    - analysis,
    - instrumentation

So what is the analogue of the JDK for Scala?

Scala piggybacks on the JDK

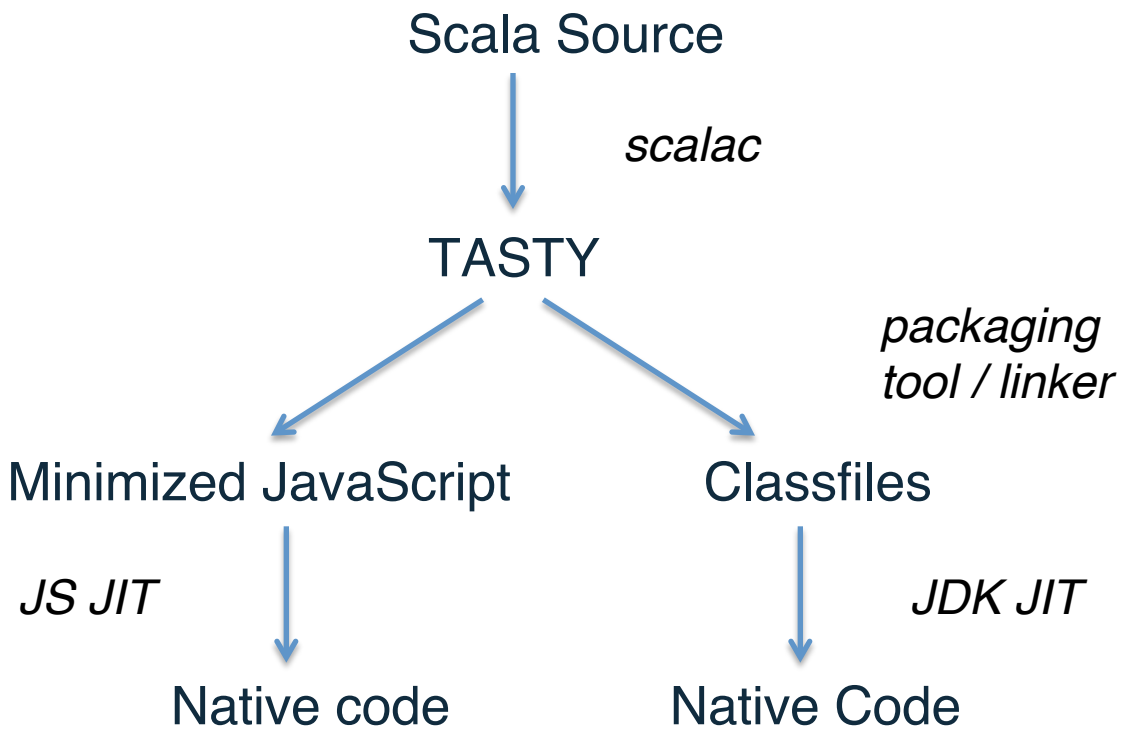
Picture so far:



Challenges for Scala

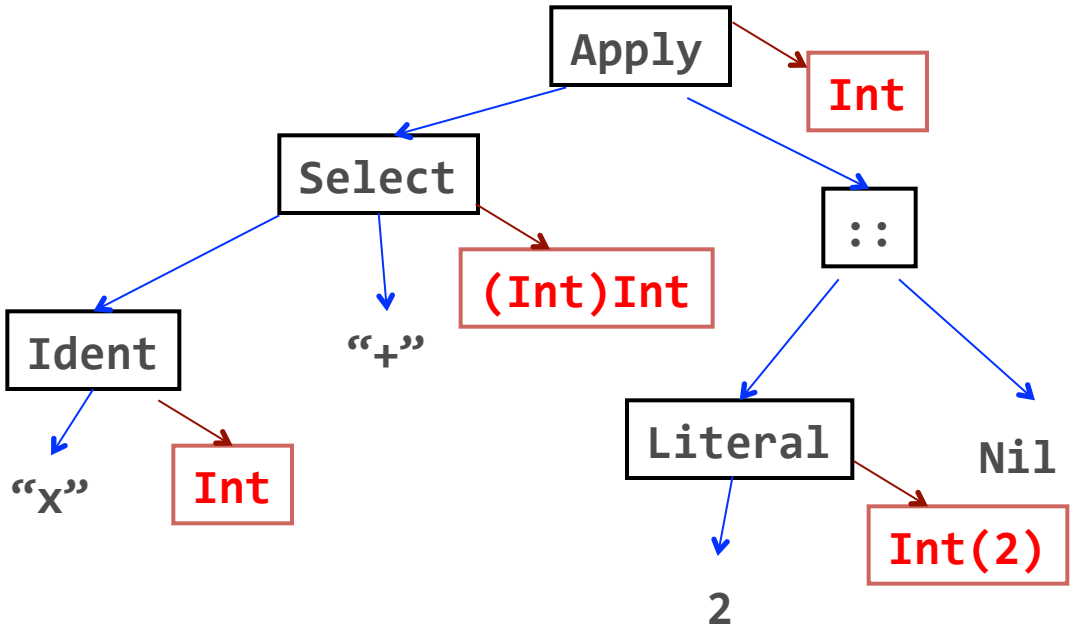
- Binary compatibility
  - *scalac* has way more transformations to do than *javac*.
  - Compilation schemes change
  - Many implementation techniques are non-local, require co-compilation of library and client. (e.g. trait composition).
- Having to pick a platform
  - Previously: platform is “The JDK.”
  - In the future: Which JDK? 7, 8, 9, 10? And what about JS?

A Scala-Specific Platform



The Core

- TASTY: Serialized Typed Abstract Syntax Trees
- E.g., here's a TAST for `x + 2`



Serialized TASTY File Format

A reference format for analysis + transformation of Scala code

high-level

complete

detailed.

TASTY Reference Manual

Version 0.05
11 Mar 2015
Martin Odersky

1 Preface

This reference manual describes the TASTY serialization format for typed syntax trees representing Scala programs. A motivation and a short summary of the format is found in the companion document A TASTY Alternative.

Notation:

We use BNF notation. Terminal symbols start with at least two consecutive upper case letters. Each terminal is represented as a single byte tag. Non-terminals are mixed case. Prefixes of the form *lower case letter**_ are for explanation of semantic content only, they can be dropped without changing the grammar.

2 Overall Layout and Vocabulary

```
File          = Header majorVersion_Nat minorVersion_Nat UUID
                nameTable_Length Name* Section*
```

A Tasty *file* consists of a header, a name table and a number of sections.

```
Section       = NameRef Length Bytes
Length        = Nat                    // length of rest of entry in bytes
```

Example:

`def plus2(x: Int) = x + 2` becomes

```
57:       DEFDEF(24) 19[plus2]
60:         PARAMS(9)
62:           PARAM(7) 20[x]
65:             TYPEREF 21[Int]
67:               TERMREF 22[scala@:]
69:                 SHARED 5
71:         TYPEREF 21[Int]
73:           SHARED 54
75:         APPLY(6)
77:           SELECT 25[$plus@scala.Int:scala.Int]
79:             TERMREFdirect 62
81:           INTconst 2
```

x + 2

Overall: TASTY trees take up ~25% of classfile size

What We Can Do With It

Applications:

- instrumentation
- optimization
- code analysis
- refactoring

Publish once, run everywhere.

Automated remapping to solve binary compatibility issues.

Language and Foundations

Connect the Dots

DOT: A calculus for

Papers in FOOL ‘12, OOPSLA ’14.

Work on developing a fully expressive machine-verified version is still onoping.

dotc: A compiler for a cleaned up version of Scala.

lampepfl/dotty on Github

DOT (in FOOL '12)

Syntax

x, y, z	Variable	$L ::=$	Type label
l	Value label	$\quad L_c$	class label
m	Method label	$\quad L_a$	abstract type label
$v ::=$	Value	$S, T, U, V, W ::=$	Type
$\quad x$	variable	$\quad p.L$	type selection
$t ::=$	Term	$\quad T\left\{z \Rightarrow \overline{D}\right\}$	refinement
$\quad v$	value	$\quad T \wedge T$	intersection type
$\quad \textbf{val}\ x = \textbf{new}\ c;\ t$	new instance	$\quad T \vee T$	union type
$\quad t.l$	field selection	$\quad \top$	top type
$\quad t.m(t)$	method invocation	$\quad \bot$	bottom type
$p ::=$	Path	$S_c, T_c ::=$	Concrete type
$\quad x$	variable	$\quad p.L_c \mid T_c\left\{z \Rightarrow \overline{D}\right\} \mid T_c \wedge T_c \mid \top$	
$\quad p.l$	selection	$D ::=$	Declaration
$c ::= T_c\left\{\overline{d}\right\}$	Constructor	$\quad L : S..U$	type declaration
$d ::=$	Initialization	$\quad l : T$	value declaration
$\quad l = v$	field initialization	$\quad m : S \to U$	method declaration
$\quad m(x) = t$	method initialization		
$s ::= \overline{x \mapsto c}$	Store	$\Gamma ::= \overline{x : T}$	Environment

Reduction $\boxed{t \mid s \ \to\ t' \mid s'}$

$$\frac{y \mapsto T_c\left\{\overline{l = v'}\ \overline{m(x) = t}\right\} \in s}{y.m_i(v) \mid s \ \to\ [v/x_i]t_i \mid s} \quad \text{(MSEL)}$$

$$\textbf{val}\ x = \textbf{new}\ c;\ t \mid s \ \to\ t \mid s, x \mapsto c \quad \text{(NEW)}$$

$$\frac{y \mapsto T_c\left\{\overline{l = v}\ \overline{m(x) = t}\right\} \in s}{y.l_i \mid s \ \to\ v_i \mid s} \quad \text{(SEL)}$$

$$\frac{t \mid s \ \to\ t' \mid s'}{e[t] \mid s \ \to\ e[t'] \mid s'} \quad \text{(CONTEXT)}$$

where evaluation context $e ::= [\,] \mid e.m(t) \mid v.m(e) \mid e.l$

dotc: Cleaning up Scala

~~XML Literals~~ → String interpolation

```
xml"""<A>this slide</A>"""
```

~~Procedure Syntax~~ → _

~~Early initializers~~ → Trait parameters

```
trait 2D(x: Double, y: Double)
```

More Simplifications

Existential types

~~`List[T] forSome { type T},`~~ `List[_]`

Higher-kinded types

`List`

→ Type with uninstantiated type members

`List`

Type Parameters as Syntactic Sugar

```
class List[T] { ... }
```

expands to

```
class List {
  type List$T
  private type T = List$T
}
```

General Higher-Kinded Types through Typed Lambdas

```
type Two[T] = (T, T)
```

expands to

```
type Two = Lambda {
  type hk$Arg
  type Apply = (hk$arg, hk$arg)
}
```

General Higher-Kinded Types through Typed Lambdas

```
Two[String]
```

expands to

```
Two {
  type hk$Arg = String
} # Apply
```

New Concepts

Type unions `(T&U)` and intersections `(T|U)`

- replace compound types (T with U)
- Eliminate potential of blow-up in least upper bound / greatest lower bound operations
- Make the type system cleaner and more regular (e.g. intersection, union are commutative).
- But pose new challenges for compilation. E.g.

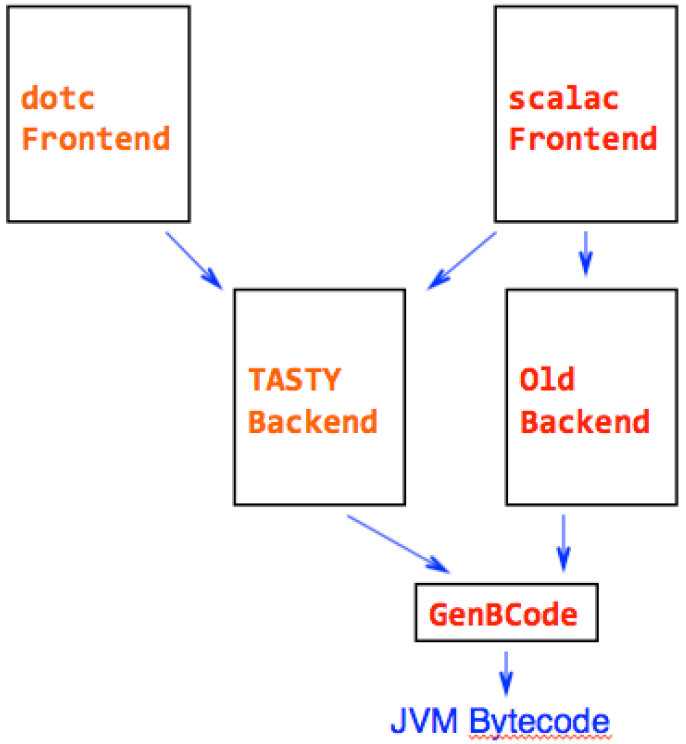
```
class A { def x = 1 }
class B { def x = 2 }

val ab: A | B = ???
ab.x          // which x gets called?
```

Status

Compiler close to completion

Should have an alpha release by ScalaDays Amsterdam

Plan to use TASTY for merging dotc and scalac.



Plans for Exploration

1. Implicits that compose

We already have implicit lambdas

```
implicit x => t          implicit transaction => body
```

What about if we also allow implicit function types?

```
implicit Transaction => Result
```

Then we can abstract over implicits:

```
type Transactional[R] = implicit Transaction => R
```

Types like these compose, e.g.

```
type TransactionalIO[R] = Transactional[IO[R]]
```

New Rule:

If the expected type of an expression *E* is an implicit function, *E* is automatically expanded to an implicit closure.

```
def f: Transactional[R] = body
```

expands to

```
def f: Transactional[R] =
  implicit _: Transaction[R] => body
```

2. Better Treatment of effects

So far, purity in Scala is by convention, not by coercion.

In that sense, Scala is not a pure functional language.

We'd like to explore "scalarly" ways to express effects of functions.

Effects can be quite varied, e.g.

- Mutation
- IO
- Exceptions
- Null-dereferencing, ...

Two essential properties:

- they are additive,
- they propagate along the call-graph.

“Though Shalt Use Monads for Effects”

Monads are cool

But for Scala I hope we find something even better.

- Monads don’t commute.
- Require monad transformers for composition.
- I tried to wrap my head around it, but then it exploded.

Idea: Use implicits to model effects as ca

to get back to this!

instead of this

```
def f: R throws Exc = ...
```

use this

```
def f(implicit t: CanThrow[Exc]): R = ...
```

or add this

```
type throws[R, Exc] =
  implicit CanThrow[Exc] => R
```

In Summary

Scala

- established functional programming in the mainstream,
- showed that a fusion with object-oriented programming is possible and useful,
- promoted the adoption of strong static typing,
- has lots of enthusiastic users, conference attendees included.

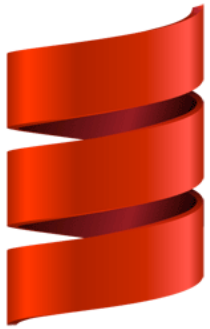
Despite it being 10 years out it has few close competitors.

Our Aims

- Make the platform more powerful
- Make the language simpler
- Work on foundations to get to the essence of Scala.

Let's continue to work together to achieve this.

Thank You!



Scala