Scala in 2016

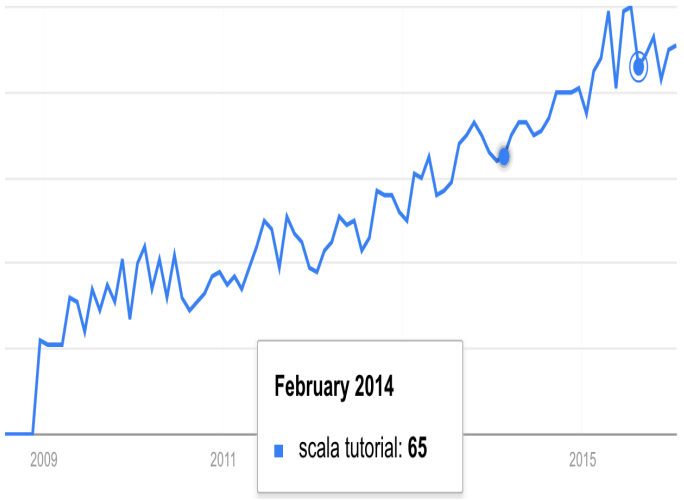
-

Martin Odersky

2015 was on the quiet side

- Maturing tools: 2.11.x, IDEs, sbt
- Steady growth

indeed.com jobs



google trends

In 2016, things will move again

- Scala 2.12 release
- Rethinking the Scala libraries
- New target platforms
- DOT and dotty

Scala 2.12

Optimized for Java 8

Uses Java 8's lambdas and default methods

for shorter code and faster execution speed.

Projected release date: mid 2016.

In case you are still on Java 6/7, Scala 2.11 will be around for a while.

Beyond 2.12

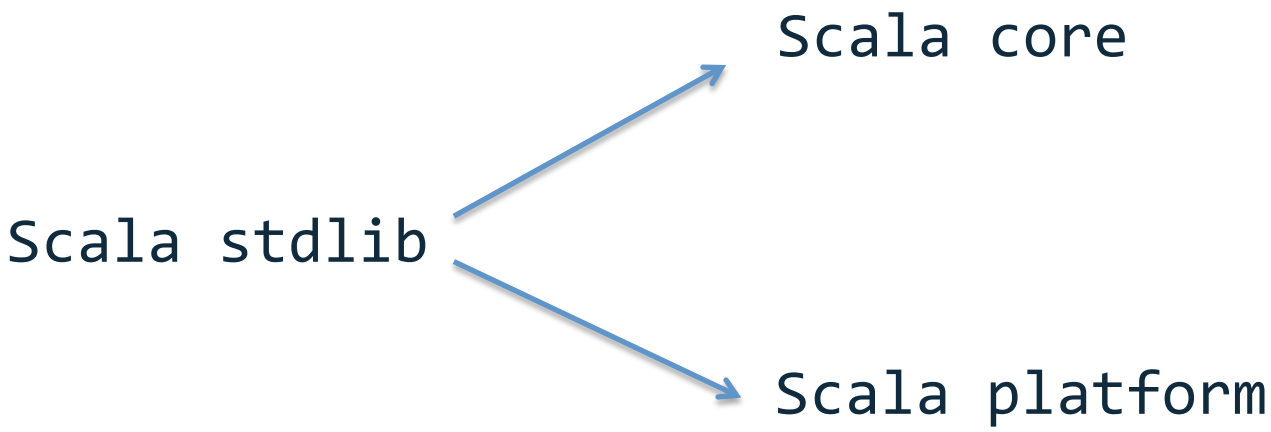
- Scala 2.13 will focus on the libraries

- Plans to revamp collections
  - Simpler to use
  - More in line with Spark usage
  - Better lazy collections (views)

Beyond 2.12

Scala 2.13 will focus on the libraries

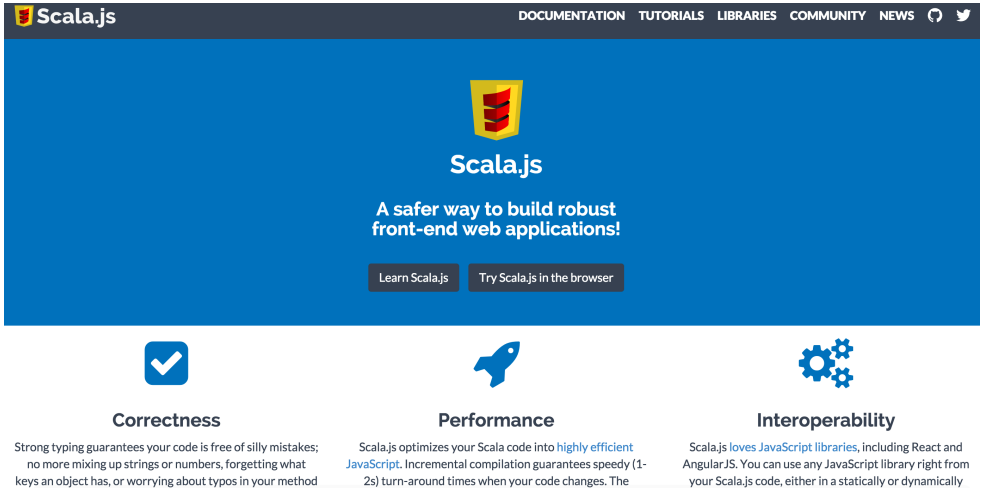
- Better modularization. One option would be a split:



- Your input and help is vital for this.

New Platforms

- Scala JS: 0.6.6 released
  - js.TupleN, a JS equivalent of Scala tuples
  - Support for JUnit
  - Faster linking
  - New website



- Denys Shabalin is working on a LLVM target.
  - Help and contributions welcome!

DOT

We finally have a proven foundation for Scala!

The DOT calculus talks about a minimal language subset, chosen so that

- we can make and prove formal statements about it
- we can encode much of the rest of the language in it.

This concludes an 8 year effort

It opens the door to do language work with much better confidence than before.

DOT Terms

- Translated to Scala notation, the language covered by DOT is:

Value	`v`	=	`(x: T) => t`	Function
			`new { x: T => d }`	`Object`
Definition	`d`	=	`def a = t`	Method definition
			`type A = T`	Type
Term	`t`	=	`v`	Value
			`x`	Variable
			$t_1(t_2)$	Application
			`t.a`	Selection
			{ val x = t_1; t_2 }	Local definition.

DOT Types

The Types covered by DOT are:

Type	T	=	`Any`	Top type
			`Nothing`	Bottom type
			`x.A`	Selection
			`(x: T1) => T2`	Function
			`{ def a: T }`	Method declaration
			`{ type T >: T1 <: T2 }`	Type declaration
			`T1 & T2`	Intersection
			`{ x => T }`	Recursion

Type Soundness

The following property was shown with a mechanized proof:

> *If a term `t` has type `T`,*
> *and evaluation of `t` terminates:*
>
> *the result will be a value `v` of type `T`.*

Why is This Important?

It gives us a technique to reason about correctness of other language features.

dotty

`dotty` is working name for our new Scala compiler.

- Builds on DOT in its internal data structures.
- Generics get expressed as type members.
- Supports an evolution of the Scala programming language.

- A first alpha release is expected this year.
  - Targeted at contributors and experimenters.

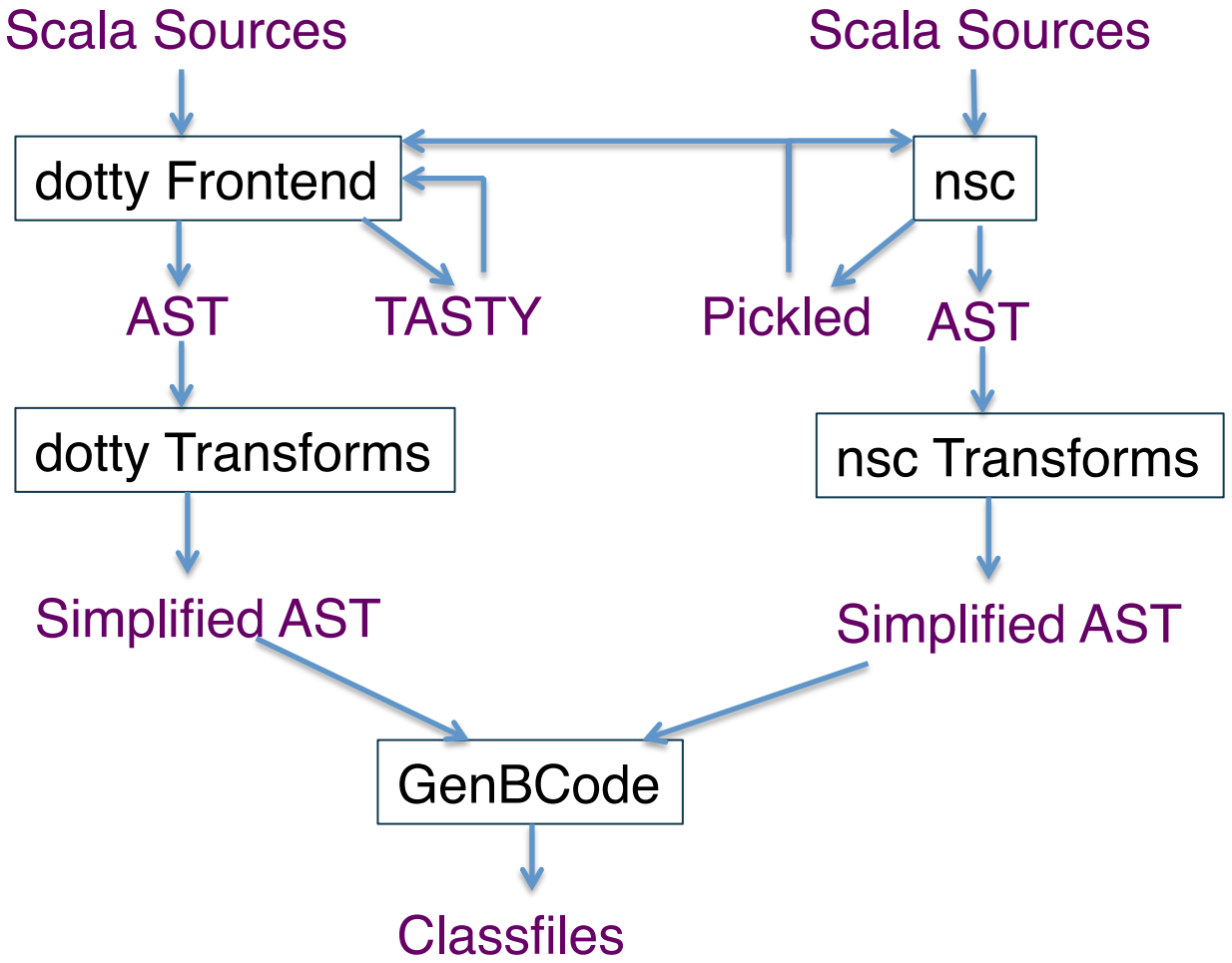
dotty – Technical Data

A bit more than half the size of the current Scala compiler, `nsc`.

- dotty: 45 KLoc
- nsc: 75 KLoc

- About twice the speed of `nsc`.
  - should improve significantly in the future.

dotty Architecture



Language evolution

Overall goal:

- Make language simpler to use
- Improve safety guarantees, reduce boilerplate.

Some new features supported by dotty:

- Trait parameters
- Intersection types
- Union types

Added: Trait Parameters

Can parameterize traits just like classes.

```
trait Logging(f: File) {
  f.open()
  onExit(f.close())
  def log(msg: String) = f.write(msg)
}

class C extends Logging(new File("log.data"))
```

Removed: Early Definitions

- The following is no longer supported:

```
trait Logging {
  val f: File
  f.open()
  onExit(f.close())
  def log(msg: String) = f.write(msg)
}

class C extends {
  val f = new File("log.data”)
} with Logging
```

Trait or Class?

Classes and traits now have largely the same capabilities.

Rule of thumb:

- When it's fully defined, make it a *class*
- When it's abstract, make it a *trait*

Abstract classes are retained mainly for Java interop and for optimization.

Added: Intersection Types

```
A & B       // Values that are both an A and a B
```

Intersection types replace compound types

```
A with B
```

What's the difference?

Difference between `A & B` and `A with B`

Consider:

```
trait A { def f: A }
trait B { def f: B }
val ab: A & B
val ba: B & A
```

Give the types of:

```
ab.f: ?
ba.f: ?
```

Difference between `A & B` and `A with B`

Consider:

```
trait A { def f: A }
trait B { def f: B }
val ab: A & B
val ba: B & A
```

Give the types of:

```
ab.f: A & B
ba.f: ?
```

Difference between `A & B` and `A with B`

Consider:

```
trait A { def f: A }
trait B { def f: B }
val ab: A & B
val ba: B & A
```

Give the types of:

```
ab.f: A & B
ba.f: A & B
```

Difference between `A & B` and `A with B`

Consider:

```
trait A { def f: A }
trait B { def f: B }
val ab: A & B
val ba: B & A
```

Give the types of:

```
ab.f: B & A
ba.f: B & A
```

Both work, since we have `A & B = B & A`

Difference between `A & B` and `A with B`

Consider:

```
trait A { def f: A }
trait B { def f: B }
val ab: A with B
val ba: B with A
```

Give the types of:

```
ab.f: ?
ba.f: ?
```

Difference between `A & B` and `A with B`

Consider:

```
trait A { def f: A }
trait B { def f: B }
val ab: A with B
val ba: B with A
```

Give the types of:

```
ab.f: B
ba.f: ?
```

Difference between `A & B` and `A with B`

Consider:

```
trait A { def f: A }
trait B { def f: B }
val ab: A with B
val ba: B with A
```

Give the types of:

```
ab.f: B
ba.f: A
```

Difference between `A & B` and `A with B`

Consider:

```
trait A { def f: A }
trait B { def f: B }
val ab: A with B
val ba: B with A
```

Give the types of:

```
ab.f: B
ba.f: A
```

Hence, A & B != B & A.
& is commutative, but **`with`** isn't.

Added: Union Types

Union types are the dual of intersection types.

```
A | B       // Values that are an A or a B
```

Example:

```
String | List[Int]
```

Use union types for ad-hoc open sums

ad-hoc: Can't plan ahead to define common supertrait

open: Arbitrary number of operands

A lightweight, efficient alternative to `Either`.

Eliminated: Exploding lubs

Scala's type inferencer often needs to compute the least upper bound (lub) of two or more types.

For instance, in an if:

```
scala> if (true) Vector(0) else List(1, 2, 3)
res0: scala.collection.immutable.Seq[Int] with
      scala.collection.AbstractSeq[Int] with
      Serializable
      = Vector(0)
```

Eliminated: Exploding lubs

But sometimes the least upper bound is very large:

```
scala> if (true) Vector(0) else Range(0, 10)
res0: scala.collection.immutable.IndexedSeq[Int] with scala.collection.AbstractSeq[Int] with S
erializable with scala.collection.CustomParallelizable[Int,scala.collection.parallel.immutable
.ParSeq[Int] with Serializable{def seq: scala.collection.immutable.IndexedSeq[Int] with scala.
collection.AbstractSeq[Int] with Serializable with scala.collection.CustomParallelizable[Int,s
cala.collection.parallel.immutable.ParSeq[Int] with Serializable]{def dropRight(n: Int): scala
.collection.immutable.IndexedSeq[Int] with scala.collection.AbstractSeq[Int] with Serializable
; def takeRight(n: Int): scala.collection.immutable.IndexedSeq[Int] with scala.collection.Abst
ractSeq[Int] with Serializable; def drop(n: Int): scala.collection.immutable.IndexedSeq[Int] w
ith scala.collection.AbstractSeq[Int] with Se...
```

Eliminated: Exploding lubs

Reformatted:

```
scala> if (true) Vector(0) else Range(0, 1)
res0: scala.collection.immutable.IndexedSeq[Int] with
      scala.collection.AbstractSeq[Int] with
      Serializable with
      scala.collection.CustomParallelizable[
        Int,
        scala.collection.parallel.immutable.ParSeq[Int] with
        Serializable{
          def seq: scala.collection.immutable.IndexedSeq[Int]
with
                   scala.collection.AbstractSeq[Int] with
                   Serializable with
                   scala.collection.CustomParallelizable[

          ....
```

Eliminated: Exploding lubs

Union types avoid exploding lubs because the least upper bound of `A` and `B` is simply `A | B`.

```
dotty> If (true) Vector(0) else Range(1)
res0: scala.collection.immutable.Vector[Int] |
      scala.collection.immutable.List[Int]
      = Vector(0)
```

Who’s working on all this?

- Scala is very much community driven.
- Your contribution counts!

Stewardship

We are about to create a new entity for helping organize open source work on Scala.

The Scala Center will act like a foundation.

It will be organized as an independent unit of EPFL.

Scala Center Missions

The Center has two missions:

1. Organize open source projects around Scala.
2. Organize and develop online teaching.

We need your help to do this!

If you want to help us, contact me after the talk.

Thank You!

Typesafe