Dotty => Scala3

What you need to know going into 2020

about me

- Working at Accenture for 11+ years
- First exposure to Scala via Spark almost 4 years ago
- Currently wrestling with Monads — far from an expert, but a passionate life-long learner…

Agenda

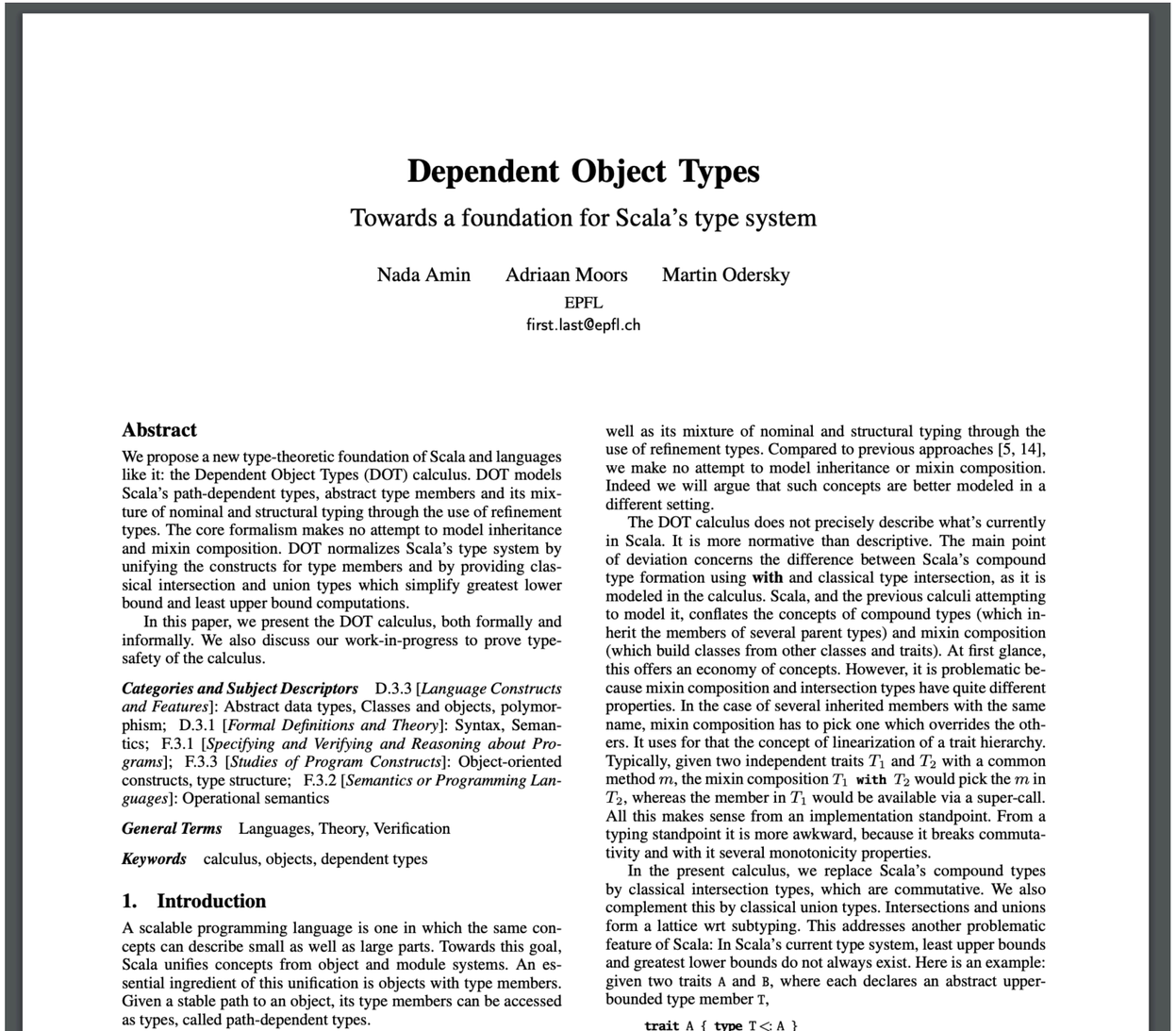
- A quick history of Dotty
- Highlight on new capabilities
- Notable changes (how your code will look different)
- Current IDE support
- Impacts on tooling and libraries
- Planning your own migration; cross-compiling to 2.x compat libraries
- Discussion and controversy: is this the new Python 3?

What is Dotty?

Dependent Object Types (DOT)

“DOT normalizes Scala’s type system by unifying the constructs for type members and by providing classical intersection and union types which simplify greatest lower bound and least upper bound computations.”

- Started 6+ years ago “as a research project to explore what a new Scala could look like”
- Intended to be Scala 3



Dependent Object Types

Towards a foundation for Scala’s type system

Nada Amin Adriaan Moors Martin Odersky
EPFL
first.last@epfl.ch

Abstract

We propose a new type-theoretic foundation of Scala and languages like it: the Dependent Object Types (DOT) calculus. DOT models Scala’s path-dependent types, abstract type members and its mixture of nominal and structural typing through the use of refinement types. The core formalism makes no attempt to model inheritance and mixin composition. DOT normalizes Scala’s type system by unifying the constructs for type members and by providing classical intersection and union types which simplify greatest lower bound and least upper bound computations.

In this paper, we present the DOT calculus, both formally and informally. We also discuss our work-in-progress to prove type-safety of the calculus.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Abstract data types, Classes and objects, polymorphism; D.3.1 [*Formal Definitions and Theory*]: Syntax, Semantics; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]; F.3.3 [*Studies of Program Constructs*]: Object-oriented constructs, type structure; F.3.2 [*Semantics or Programming Languages*]: Operational semantics

General Terms Languages, Theory, Verification

Keywords calculus, objects, dependent types

1. Introduction

A scalable programming language is one in which the same concepts can describe small as well as large parts. Towards this goal, Scala unifies concepts from object and module systems. An essential ingredient of this unification is objects with type members. Given a stable path to an object, its type members can be accessed as types, called path-dependent types.

well as its mixture of nominal and structural typing through the use of refinement types. Compared to previous approaches [5, 14], we make no attempt to model inheritance or mixin composition. Indeed we will argue that such concepts are better modeled in a different setting.

The DOT calculus does not precisely describe what’s currently in Scala. It is more normative than descriptive. The main point of deviation concerns the difference between Scala’s compound type formation using **with** and classical type intersection, as it is modeled in the calculus. Scala, and the previous calculi attempting to model it, conflates the concepts of compound types (which inherit the members of several parent types) and mixin composition (which build classes from other classes and traits). At first glance, this offers an economy of concepts. However, it is problematic because mixin composition and intersection types have quite different properties. In the case of several inherited members with the same name, mixin composition has to pick one which overrides the others. It uses for that the concept of linearization of a trait hierarchy. Typically, given two independent traits T_1 and T_2 with a common method m, the mixin composition T_1 `with` T_2 would pick the m in T_2, whereas the member in T_1 would be available via a super-call. All this makes sense from an implementation standpoint. From a typing standpoint it is more awkward, because it breaks commutativity and with it several monotonicity properties.

In the present calculus, we replace Scala’s compound types by classical intersection types, which are commutative. We also complement this by classical union types. Intersections and unions form a lattice wrt subtyping. This addresses another problematic feature of Scala: In Scala’s current type system, least upper bounds and greatest lower bounds do not always exist. Here is an example: given two traits `A` and `B`, where each declares an abstract upper-bounded type member `T`,

```
trait A { type T <: A }
```

What is Tasty

(Typed Abstract Syntax Trees)

- Tasty is the high-level interchange (serialization) format for Scala 3
- Represents the syntactic structure of programs and also contains complete information about types and positions
- The compiler uses it to support separate compilation
- The foundation for a new generation of reflection-based macros: it replaces the original def macros and the `scala.reflect` infrastructure.

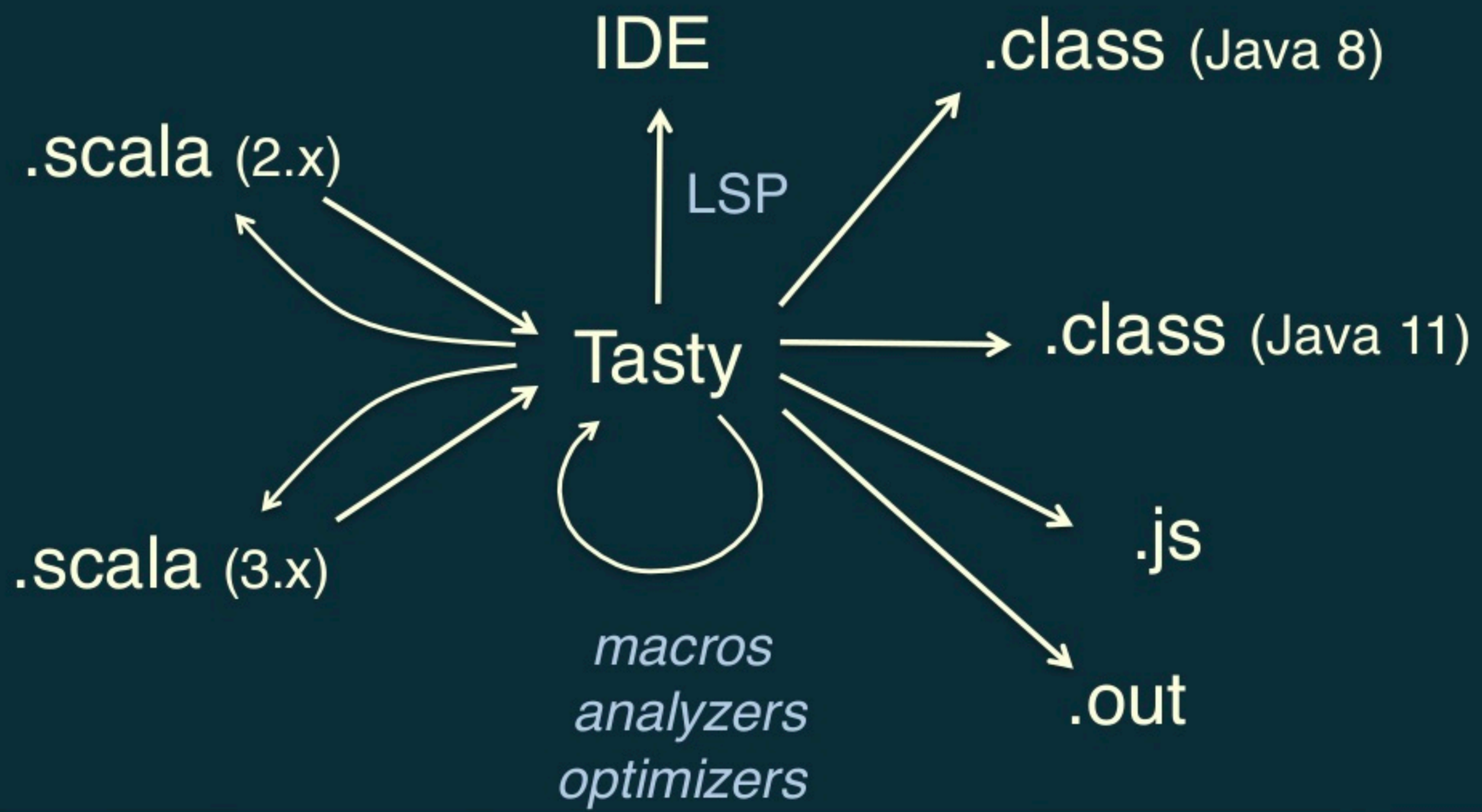
Important Use Cases:

- Language servers for IDEs (via LSP)
- Cross Binding -> Escape the curse of binary compatibility

Tasty Demo

- .tasty files are now co-located with .class files
- Are included in published JARs
- Can be explored with “dotc -print-tasty path-to/foo.tasty”
- Sizes are approximately same size as class files
- Seems to be sort of equivalent to old C object files

The Tasty Vision



Original Roadmap:

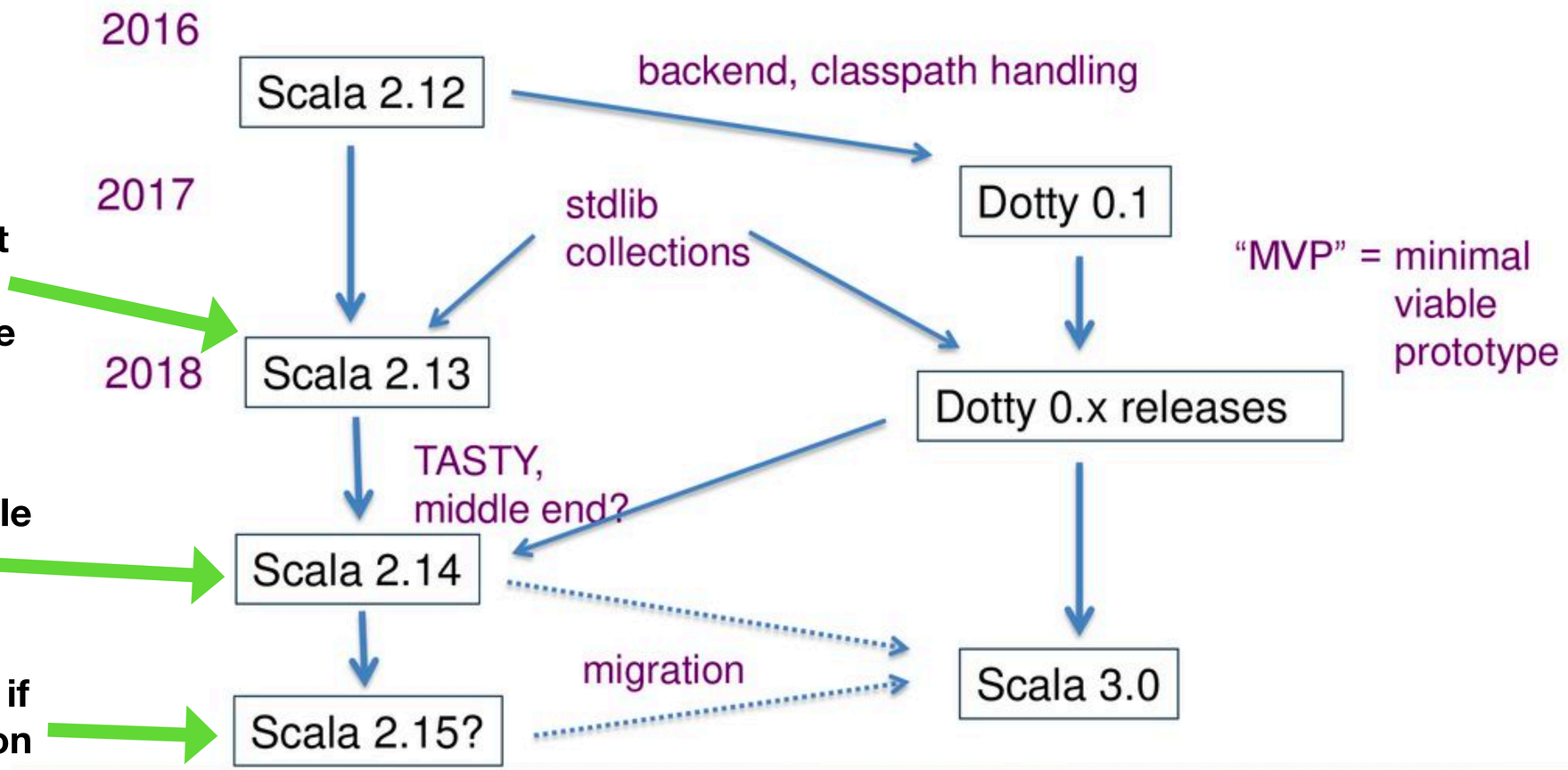
Goal of Scala 2.13 is only about stdlib changes, intended as a bridge for Dotty (which uses the same stdlib) to ensure compatibility.

Goal of Scala 2.14 was to compile to TASTY to enable binary compatibility.

Scala 2.15 would be considered if needed to complete the migration

Roadmap (from previous years)

This is open source work, depends on community's contributions.
→ Roadmap **is tentative, no promises:**

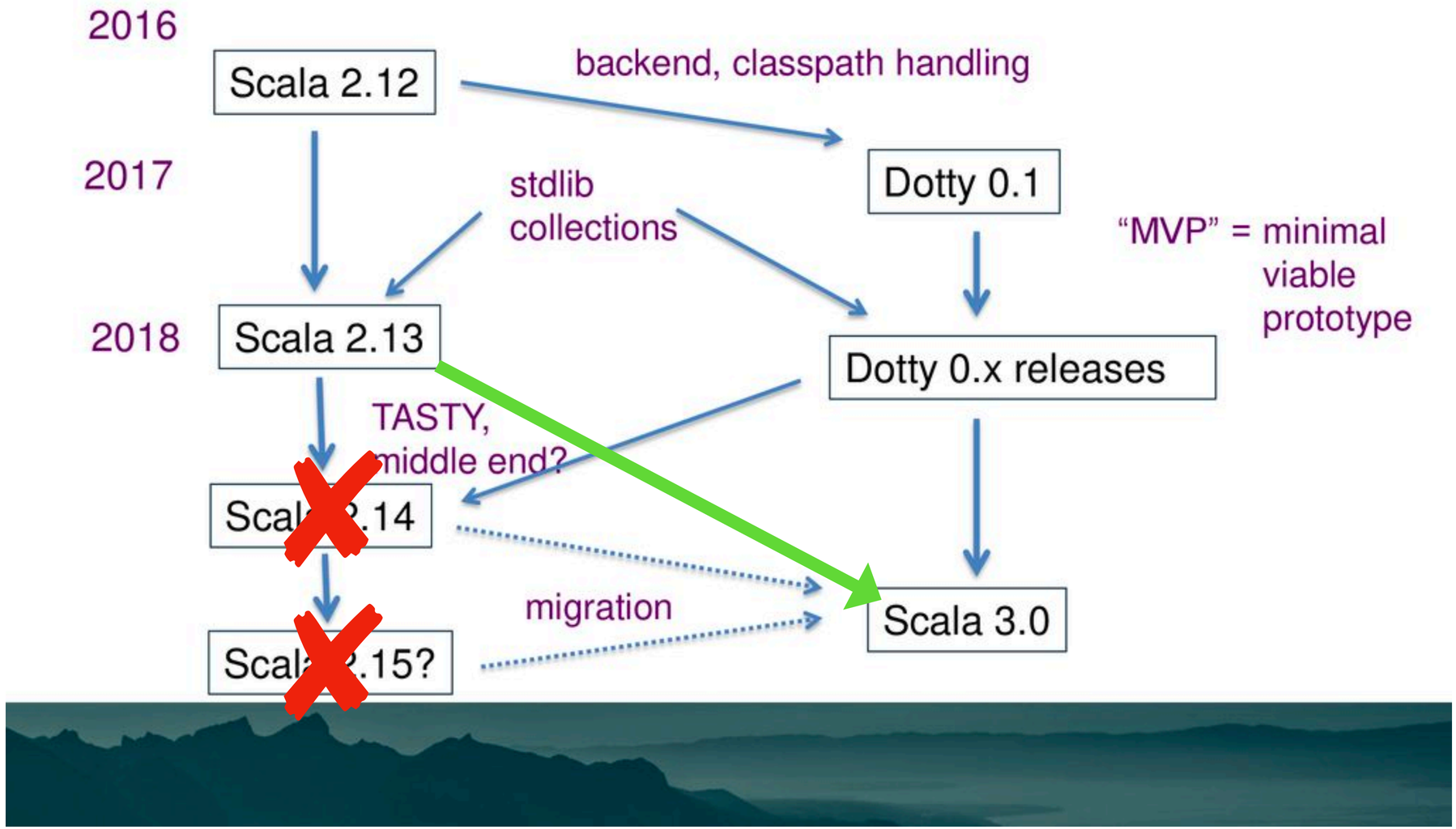


Borrowed from Adriaan Moors presentation

Notable Recent Events

- **18 December:** New roadmap announcement that Scala 2.14 is cancelled and 2.13 will be the last 2.x version. Goal: first RC by end of 2020
- **20 December:** Dotty 0.21.0-RC1 released and announced as Feature Complete (no new features, but syntax may evolve)

Roadmap


2016
Scala 2.12
backend, classpath handling
2017
stdlib
collections
Dotty 0.1
"MVP" = minimal
viable
prototype
2018
Scala 2.13
Dotty 0.x releases
TASTY,
middle end?
Scala 2.14
migration
Scala 3.0
Scala 2.15?

Cool New Features

Union Types

```
val user: Member | Email
val safe: String | Null
```

Intersection Types

```
trait A
trait B
val x: A & B
```

- Replaces “with” and represents the new model of multiple inheritance
- If A & B both define a member “foo” of different types, then A & B will have a member “foo” whose types is the intersection.

Real Enums!

```
enum Color {
  case Red, Green, Blue
}
```

```
enum Color(val rgb: Int) {
  case Red   extends Color(0xF0000)
  case Green extends Color(0x00FF00)
  case Blue  extends Color(0x0000FF)
}
```

```
scala> val red = Color.Red
val red: Color = Red
scala> red.ordinal
val res0: Int = 0
```

```
scala> Color.valueOf("Blue")
val res0: Color = Blue
scala> Color.values
val res1: Array[Color] = Array(Red, Green, Blue)
```

- You can add definitions/ members to an enum just like a class
- It’s also possible to add an explicit companion object

Enums Powerful Enough for Algebraic Data Types!

```
enum Option[+T] {
  case Some(x: T)
  case None
}
```

```
enum Option[+T] {
  case Some(x: T) extends Option[T]
  case None       extends Option[Nothing]
}
```

Extension Methods

Reducing the need for implicit classes

```
implicit class StringOps(s: String)
{
  def addHi: String = s"Hi, $s"
}

“Bob".addHi

val res1: String = Hi, Bob
```

```
def (s: String) addHello: String =
    s"Hello, $s"

“Bob".addHello

val res1: String = Hello, Bob
```

Trait parameters

```
trait Animal(val species: String)

trait Voice(val sound: String)

case class Pet(val name: String) extends Animal("canine") with Voice("Woof")

val myDog = Pet("Nutmeg")

scala> s"This is ${myDog.name} who is a ${myDog.species} and goes ${myDog.sound}"
val res1: String = This is Nutmeg who is a canine and goes Woof
```

Generic Tuples (22)

```
(a, b, c) == (a, (b, (c, ())))
```

Type Lambdas

Lets you express a higher-minded type directly without a type definition

Two simple examples from *Functional Programming for Mortals*:

Kind Projection:

```
trait Foo[F[?]]
object FooEitherString extends Foo[Either[String,_]]
object FooEitherString extends Foo[[X] =>> Either[String,X]]
```

No more Id type alias for things like mocking

```
trait Echoable[F[?]] {
   def read: F[String]
   def write(msg: F[String]): F[Unit]
}

object Mock extends Echoable[_] // or Echoable[[X] =>> X]
{  def read: String = "hello"
   def write(msg: String): Unit = ()
}
```

New “Quiet” (Indentation) Syntax

```scala
package example

trait Ord[T]
  def compare(x: T, y: T): Int
  def (x: T) < (y: T) = compare(x, y) < 0
  def (x: T) > (y: T) = compare(x, y) > 0

given Ord[Int]
  def compare(x: Int, y: Int) =
    if (x < y) -1 else if (x > y) +1 else 0

given [T]: (ord: Ord[T]) => Ord[List[T]]
  def compare(xs: List[T], ys: List[T]): Int = (xs, ys) match
    case (Nil, Nil) => 0
    case (Nil, _) => -1
    case (_, Nil) => +1
    case (x :: xs1, y :: ys1) =>
      val fst = ord.compare(x, y)
      if (fst != 0) fst else compare(xs1, ys1)
```

How your code will look different

(Not much)

Partial List of Changes

Package Objects	Deprecated (Still around for Scala 3.0, but dropped after that.) The reason is a lot more things can be written now at the top level.
Symbol Literals	Removed, but symbol ‘Goober can still be represented as Symbol(“Goober”). Symbol class will be deprecated in the future.
Wildcard Arguments in Types	F[_] should be rewritten as F[?], so that F[_] will become shorthand for type lambda [X] =>> C[X]. Both supported in Scala 3.0 unless -strict setting is in place.
Implicit Conversions	Requires an import scala.language.implicitConversions, intended to discourage abuses of the feature.

Implicits

Implicits have a lot of challenges. We felt there should be a concerted effort to get out the good parts of implicits and remove the traps and the pitfalls as much as possible. And part of the answer for that—not all of it but part of the answer for that is syntax."

Martin Odersky - A Tour of Scala 3 from Scala Days 2019

*Being very powerful, implicits are easily over-used and mis-used… **Conditional implicit values** are a cornerstone for expressing type classes, whereas most applications of **implicit conversions** have turned out to be of dubious value. The problem is that many newcomers to the language start with defining implicit conversions since they are easy to understand and seem powerful and convenient.*

Dotty Documentation - Contextual Abstractions Overview

implicit => given

- Terminology is freshly minted. (July 2019 slides had “delegate” instead) Also note “implicitly” => “summon”
- Effort made to disambiguate *implicit instances*, *implicit parameters* and *implicit conversions*
- Some simplification (less need to put in companion objects; instances can be anonymous)
- Fewer restrictions for implicit parameters (doesn’t have to be last curried group, multiple clauses allowed, etc.) based on previous pain points

Impacts on Tools and Libraries (small sample)

scalafix	Strategic tool for code migration of Scala 2 to Scala 3 conventions, but I don’t see any examples so might be in-progress.
simulacrum	Depends heavily on deprecated version of Scala macros. No migration path.
cats	Heavy dependency on simulacrum, but simulacrum-scalefix project is experimenting on migrating this. Overall, support seems active.
zio	Tests compilation on nightly Dotty builds. (Haven’t tried any examples.)
scalatest	ScalaTest 3.1.0 only works with OLD Dotty 0.17 (Released back in July. Current version is 0.21) Dotty g8 templates use JUnit.

Planning your Migration

- Scala 3 and Scala 2 share the same standard library.
- With some small tweaks it is possible to cross-build code for both Scala 2 and 3. We will provide a guide defining the shared language subset that can be compiled under both versions.
- The Scala 3 compiler has a `-language:Scala2` option that lets it compile most Scala 2 code and at the same time highlights necessary rewritings as migration warnings.
- “The compiler can perform many of the rewritings automatically using a -rewrite option”

Planning your Migration cont.

If your build contains dependencies that have only been published for Scala 2.x, you may be able to get them to work on Dotty by replacing:

```
libraryDependencies += "a" %% "b" % "c"
```

with

```
libraryDependencies += ("a" %% "b" % "c").withDottyCompat(scalaVersion.value)
```

“This will have no effect when compiling with Scala 2.x, but when compiling with Dotty this will change the cross-version to a Scala 2.x one. This works because Dotty is currently retro-compatible with Scala 2.x.”

Is this Python 3 (or Perl 6)?

Probably