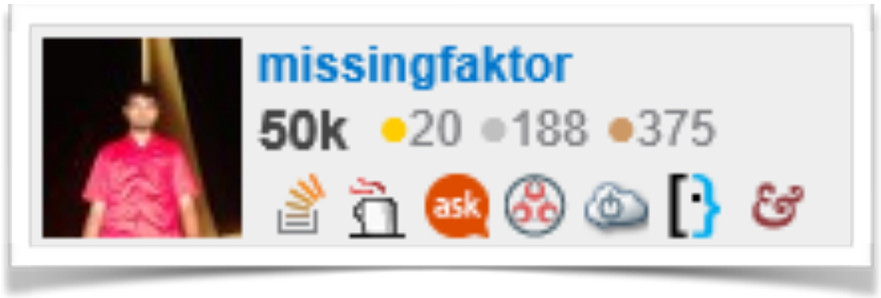
Object-Functional Programming

A beautiful unification or a kitchen sink?

Rahul Goma Phulore (@missingfaktor)
Functional Conf, Bengaluru
2014

A bit about me



Scala

ThoughtWorks®

Situation today

FP is superior to OO

OO is a natural way of thinking

Hybrid incurs too much complexity

Object oriented programming is a passé

Hybrid is the way

Situation today

Let the feather-ruffling begin!

Paradigms

par·a·digm

/ˈperəˌdīm/

noun

A **framework containing** the basic **assumptions**, **ways of thinking**, and **methodology** that are commonly **accepted by** members of a scientific **community**. Such a cognitive framework shared by members of any discipline or group.

Paradigms

Teaching Programming Languages in a Post-Linnaean Age

Shriram Krishnamurthi

SIGPLAN Workshop on Undergraduate Programming Language Curricula, 2008

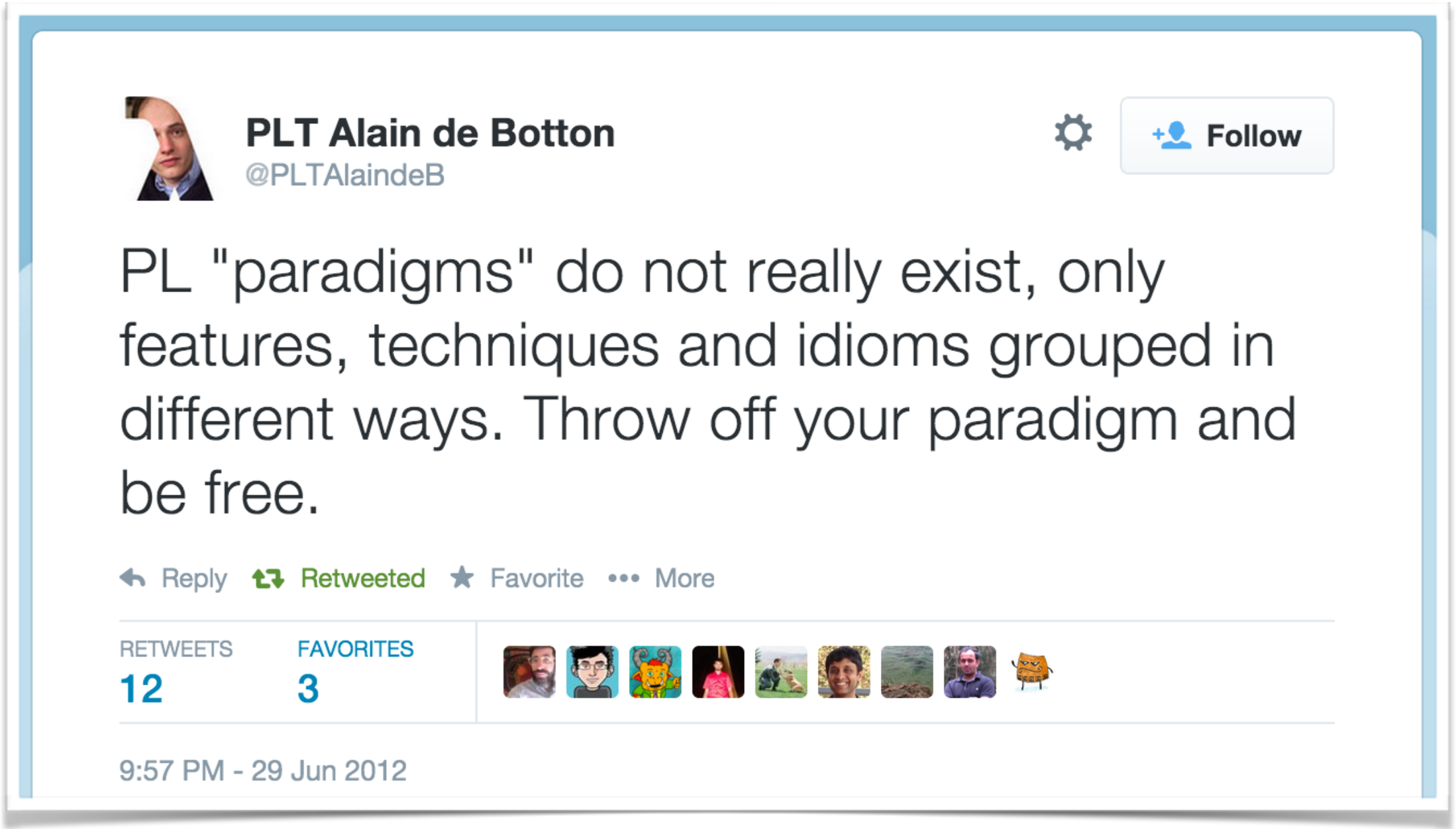
Abstract

Programming language ‘‘paradigms’’ are a moribund and tedious legacy of a bygone age. Modern language designers pay them no respect, so why do our courses slavishly adhere to them? This paper argues that we should abandon this method of teaching languages, offers an alternative, reconciles an important split in programming language education, and describes a textbook that explores these matters.

Paradigms

Labels such as “OO” and “functional” have so many conflicting accepted interpretations that they are almost totally devoid of meaning.

Paradigms


PLT Alain de Botton
@PLTAlaindeB
Follow
PL "paradigms" do not really exist, only features, techniques and idioms grouped in different ways. Throw off your paradigm and be free.
Reply
Retweeted
Favorite
More
RETWEETS
12
FAVORITES
3
9:57 PM - 29 Jun 2012

Paradigms

- Paradigms are computer science equivalent of tribalism.
- Differences often socio-political/cultural, than technical.
- **A lot of room for cross-pollination!**

Proposition

- If we forget paradigms and admit interesting and useful ideas, it will lead us to better abstractions and better programming languages.

So what are today's two most popular "paradigms"?

Object-Oriented Programming

- Objects
  - First-class modules
  - Self-recursion (Yay, μ!)
    - Subtyping
- Traits/classes

Functional Programming

- First-class functions
  - Higher order functions
  - Function composition
- Immutability

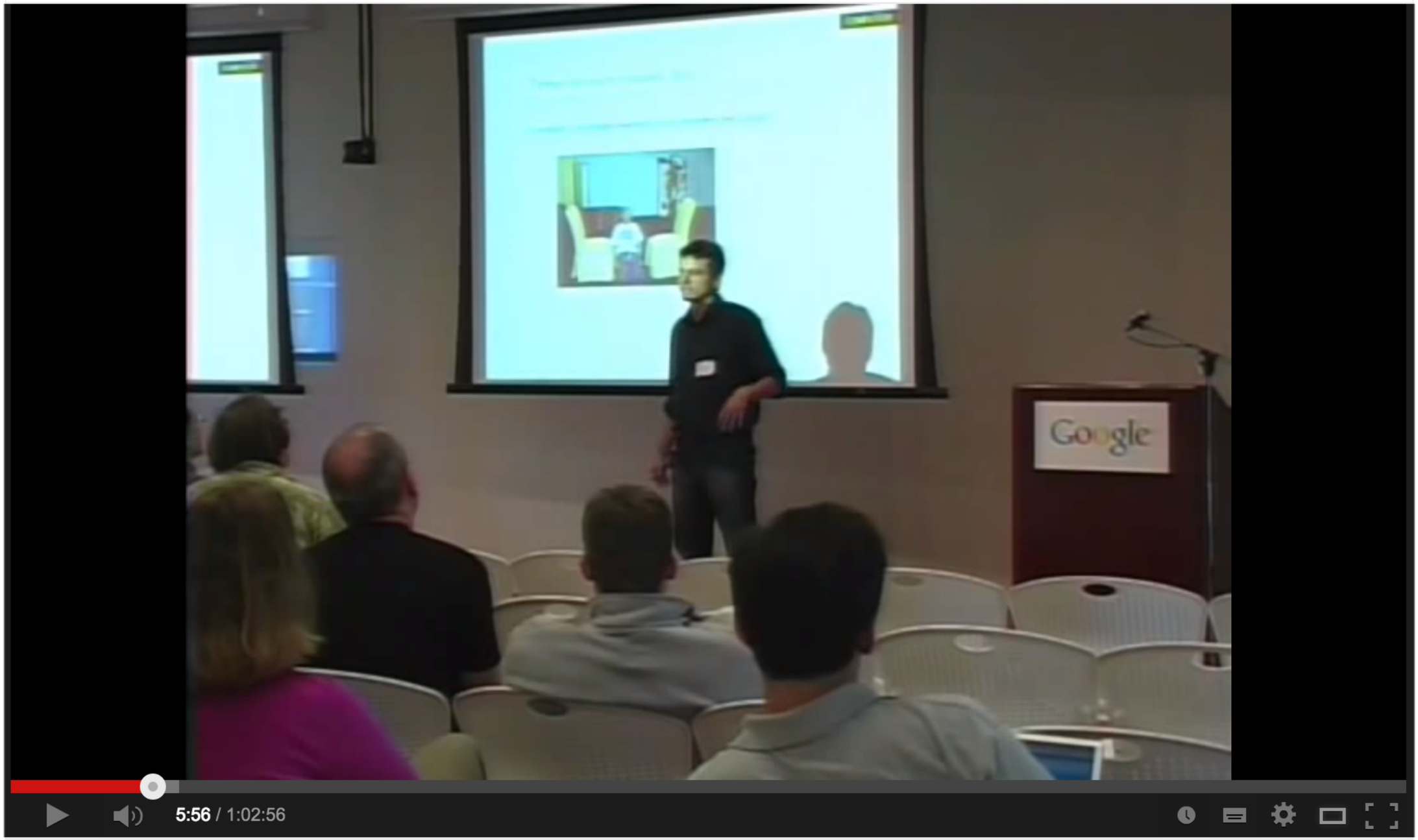
<insert>Typed</insert> Functional Programming

- First-class functions
  - Higher order functions
  - Function composition
- Immutability
- Algebraic data types
- Type-classes
- Equational reasoning

Scala!

“The Scala Experiment”

Google Tech Talk, 2006



Unification!

Yann, elf motivated
@theatrus

Follow

If a grand unified theory of programming language existed, its implementation would be called Scala

Reply Retweet Favorite More

Or...?

Bryan O'Sullivan
@bos31337

Follow

If you like programming languages and food, here's the culinary equivalent of Scala.

Reply Retweet Favorite ••• More

RETWEETS 1,122 FAVORITES 565

12:36 PM - 21 Jan 2014

Scala

- A few orthogonal features
  - Traits (and classes)
  - Objects
  - Implicits

#1 Functions as objects

- First-class functions.
- ...that are still objects!
- Function{N} traits.

#1 Functions as objects

```
val f = (x: Int, y: Int) => x + y

// desugars to

val f = new Function2[Int, Int, Int] {
  def apply(x: Int, y: Int) = x + y
}
```

#1 Functions as objects

```
(Int, Int) => Int

// desugars to

Function2[Int, Int, Int]
```

#1 Functions as objects

- Data structures as functions.

```
trait Seq[+A] extends (Int => A)
trait Set[-A] extends (A => Boolean)
trait Map[-K, +V] extends (K => V)
```

#1 Functions as objects

- Your own function types.

```
trait Parser[+A]
  extends (Input => ParseResult[A])
```

```
trait LabeledFunction[-A, +B]
  extends (A => B) {

  def label: String
}
```

#2 Records and classes

- **case classes** ≈ immutable records in other functional languages.
- but still classes.

#2 Records and classes

- Auto generated methods:
  - equals
  - hashCode
  - toString
  - copy
  - apply (in companion)
  - unapply (in companion)

#2 Records and classes

```haskell
-- Record update in Haskell

data A t = A
  { a :: t
  , b :: Int } deriving (Eq, Show)

a1 = A 1 2
a2 = a1 { a = "someString" }
```

#2 Records and classes

```
// Record update in Scala

case class A[T](a: T, b: Int) {
   // Compiler generated method.
   // def copy[T'](a: T' = this.a, b: Int = this.b): A[T']
   //   = new A[T'](a, b)
}

val a1: A[Int] = A(1, 2)
val a2: A[String] = a1.copy(a = "someString")
```

#3 Algebraic Data Types

- Sum types, product types, and more.

```haskell
-- Haskell

data Option a = Some a
              | None
```

#3 Algebraic Data Types

- Simply a sealed class hierarchy!

```
// Scala

sealed trait Option[+A]
case class Some[+A](value: A) extends Option[A]
case object None extends Option[Nothing]
```

#3 Algebraic Data Types

- unapply:

```
case class Foo(x: Int, y: Int)

// Generated
object Foo {
  def unapply(value: Foo): Option[(Int, Int)] = {
    Some(value.x, value.y)
  }
}
```

#3 Algebraic Data Types

- Namespace your data constructors.

```haskell
-- Haskell

data ColorChoice = Custom Color | Default
backgroundColor = Custom Red
```

```scala
// Scala
sealed trait ColorChoice

object ColorChoice {
  case class Custom(c: Color) extends ColorChoice
  case object Default extends ColorChoice
}

val backgroundColor = ColorChoice.Custom(Color.Red)
```

#3 Algebraic Data Types

- Extract common behavior in mixins.

```
abstract class Enum[E : Manifest] {
  def all: Seq[E]

  final def fromString(string: String): Try[E] = {
    all
      .find(_.toString == string)
      .toTry(new IllegalNameException(string))
  }
}
```

#3 Algebraic Data Types

- Extract common behavior in mixins.

```
sealed trait Directive

object Directive extends Enum[Directive] {
  case object Zob extends Directive
  case object Drob extends Directive

  lazy val all: Seq[Directive] = Seq(Zob, Drob)
}
```

#4 Pattern matching

- Magic in most languages. (Including Haskell.)
- In Scala, most aspects of pattern matching are not magic, and customizable by user.

#4 Pattern matching

- Contract:

```
// match
unapply: A => Boolean
```

```
// match and extract
unapply: A => Option[B]

// match and extract many values
unapplySeq: A => Option[Seq[B]]
```

#4 Pattern matching

- Your “non-ADT” objects can have pattern matching. Preserve encapsulation. (The feature is called “extractors”.)
- Regex a good example:

```
val ModuleId = "mdlc:(.*):(.*)".r

"mdlc:spam:egg" match {
  case ModuleId(a, b) => (a, b)
}
```

#4 Pattern matching

- Advanced use case: Composing patterns.

```
val Positive = Matcher[Int](_ > 0)
val Even = Matcher[Int](_ % 2 == 0)

val EvenPositive = Even and Positive

6 match {
  case EvenPositive() => true
  case _ => false
}
```

#4 Pattern matching

- Advanced use cases: Composing patterns.

```
case class Matcher[I](cond: I => Boolean) {
  def unapply(i: I) = cond(i)

  def and(that: Matcher[I]) = Matcher { in =>
    this.cond(in) && that.cond(i)
  }
}
```

#4 Pattern matching

- Even pattern matching blocks are first-class values!

```
val block1: PartialFunction[String, Int] = {
  case "move" => 1
  case "shift" => 2
}

val block2: PartialFunction[String, Int] = {
  case "drop" => 3
}
```

#4 Pattern matching

- You can store them in variables, return from a function, compose them etc.

```
val composedBlock = block1 orElse block2

composedBlock(“drop") // gives 3

composedBlock.isDefinedAt(“drop") // gives true
```

#4 Pattern matching

- I like them so much, I ported them to Clojure!

README.md — Rahul - Updated README as per new “branding”. — 6 months ago

project.clj — Sumit - First attempt with speclj — 2 months ago

README.md

match-block

Pattern matching blocks as values.

Turning abstractions into first-class values gives us an ability to abstract over and compose them,

#5 Type-classes

- Ingredients for type-classes:
  - traits
  - objects
  - implicits
- Started out as “poor man’s type-classes”, evolved into something much greater.

#5 Type-classes

- Implicits! The most misunderstood part of Scala.

#5 Type-classes

Channing Walton
@channingwalton

Following

Much confusion around #scala implicits. Sometimes I wonder if the word ‘implicit’ is part of the problem. It connotes something magical.

Guildford, Surrey

Reply Retweet Favorite ••• More

#5 Type-classes

- Better terms:
  - Evidence
  - Witness

#5 Type-classes

`ev: Foo`	Passing context
`ev: A => B`	Implicit conversion (generally frowned upon)
`ev: A <:< B`	"Generalized" type constraints
`ev: A <~< B`	Generalized type constraints, but better
`ev: T[A]`	Type classes
`ev: T[A, B]`	Multiparameter type classes, functional dependencies, and even some dependent typing stuff!

#5 Type-classes

Facts — implicit vals.

Rules — implicit defs taking implicit vals as parameters.

#5 Type-classes

- Haskell
  - type-classes and instances are second-class citizens.
  - Global, and non-modular. (Some would argue this is a Good Thing ™, and enables better reasoning.)
  - Abstracting over them requires special extensions like constraint kinds.
  - More advanced use cases need more extensions – multi-parameter type-classes, functional dependencies.

#5 Type-classes

- Scala
  - Type-classes and instances are first-class citizens.
  - Not global. Can be put into mixins, namespaced, imported etc.
  - You can abstract over them using regular language features.
  - “Advanced use cases” that I mentioned before also get taken care of.

#5 Type-classes

- Implicit calculus

A

The Implicit Calculus: A New Foundation for Generic Programming

BRUNO C. D. S. OLIVEIRA, The University of Hong Kong
TOM SCHRIJVERS, Ghent University
WONTAE CHOI, Seoul National University
WONCHAN LEE, Seoul National University
KWANGKEUN YI, Seoul National University
PHILIP WADLER, University of Edinburgh

Generic programming (GP) is an increasingly important trend in programming languages. Well-known GP mechanisms, such as type classes and the C++0x concepts proposal, usually combine two features: 1) a special type of interfaces; and 2) *implicit instantiation* of implementations of those interfaces.

Scala *implicits* are a GP language mechanism, inspired by type classes, that break with the tradition of coupling implicit instantiation with a special type of interface. Instead, implicits provide only implicit instantiation, which is generalized to work for *any types*. Scala implicits turn out to be quite powerful and useful to address many limitations that show up in other GP mechanisms.

...and much much more!

- ML-like modules. In some ways, better!
- Virtual classes (Cake pattern)

...and much much more!

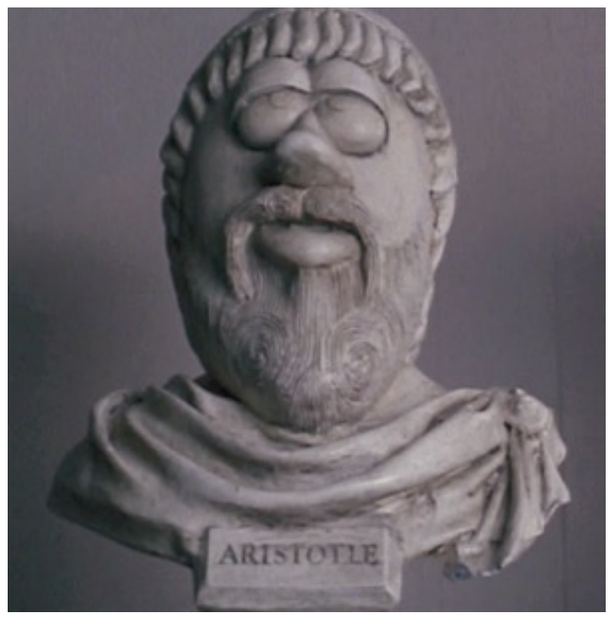
- Type families
- Datatype-generic programming (Shapeless)
- Type-level programming

In summary

- There is method to this apparent madness!
- Unification can give you a simpler mental framework to work with.

In summary

The whole is greater and simpler than the sum of its parts.



PLT Aristotle

But...

One man's unification is another man's conflation!

The other side of the coin

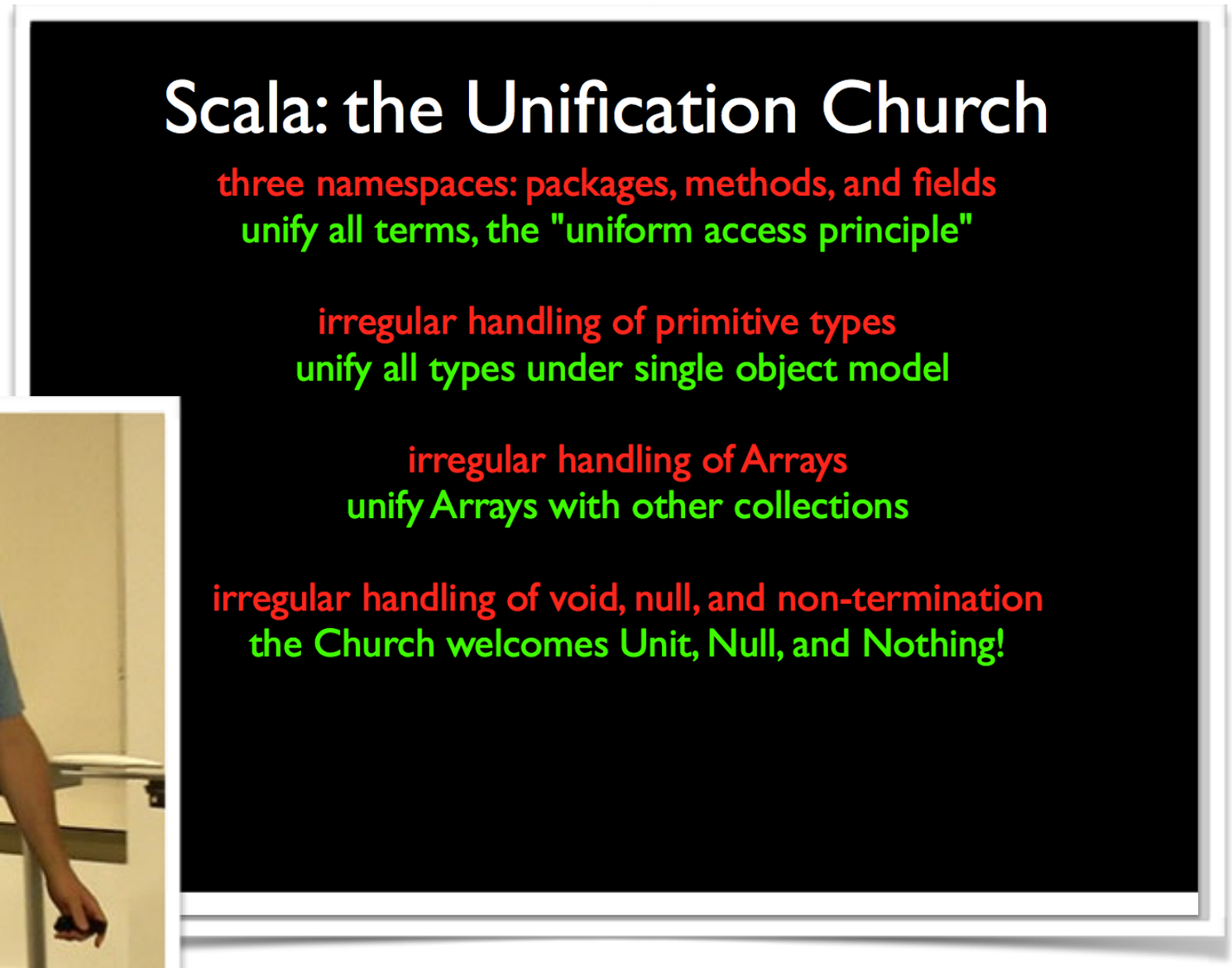
- A certain impedance mismatch between some ideas. And when it shows, it **hurts**!
  e.g. GADTs in Scala.

The other side of the coin

- Too much rope. Room for awkward metaphor mixing.
- Idioms and styles are largely matter of convention.
- There's no free lunch. There are always trade-offs.

The other side of the coin

- Hear it from Paul Philips



Takeaways

¬ ∀ north carolina
@jcoglan

Follow

Software is overrun with absolutist "movements" and sorely lacking in nuanced context-aware analysis.

Reply Retweeted Favorited More

RETWEETS	FAVORITES
14	14

6:08 PM - 23 Apr 2014

- Keep an open mind.
- Stay a humble learner.
- Forget paradigms. Embrace ideas.

Thank You!

Credits

- Programming languages community responsible for this amazingly diverse and exciting landscape.
- People who helped me with the talk with their valuable suggestions (Rahul Kavale, Rhushikesh Apte, Mushtaq Ahmed, and others. Thanks!)