Typeclass-driven Polymorphism in Scala

Ben Hutchison

LambdaJam, Brisbane 2013

realestate.com.au Australia's No.1 property site
Supported by my employer
careers.realestate.com.au

“Classic” map

- Collection method **map(f)** transforms a collection by applying f to each element

List(1, 2, 3).map(toString) == List(“1”, “2”, “3”)

- Scala 2.7 collections included a map method with the “classic” signature

```
trait scala.Iterable[A] { ..
  def map [B](f : (A) => B) : Iterable[B]
}
```

2.7 Collections API problems

- Subclasses had to repeatedly override map to return a more specific collection type

```
trait scala.Seq[A] { ..
    override def map [B](f : (A) => B) : Seq[B] = {…}}
```

- Not just map
  - Approx 30 other generic methods had to be repeatedly overridden in each sub-trait

```
trait scala.Seq[A] { ..
    override def filter (p : (A) => Boolean) : Seq[A] = {…}}
```

A solution?

- The repetition of methods can be solved by passing a higher-kinded Collection type to a common super definition

```
trait TraversableLike[+Elem, +Coll[+X]] {
    def map[NewElem](f: Elem => NewElem): Coll[NewElem]
    def filter(p: Elem => Boolean): Coll[Elem]
}
trait scala.Seq[A] extends TraversableLike[A, Seq]
```

New requirements..

- The designers of the Collections redesign also wished to tackle tricky use cases like:

Map(“a” -> 1, “b” -> 2) map {case (x, y) => (y, x)}

== Map(1 -> “a”, 2 -> “b”)

Map(“a” -> 1, “b” -> 2) map {case (x, y) => y}

== List(1, 2)

New requirements

- Another example: Bitsets
  - “Bitsets are sets of non-negative integers which are represented as variable-size arrays of bits packed into 64-bit words. The memory footprint of a bitset is determined by the largest number stored in it.”

```
BitSet(1, 2, 3).map(_ * 2) == BitSet(2, 4, 6)

BitSet(1, 2, 3).map(_.toFloat) == Set(2.0, 4.0, 6.0)
```

2.8 Collections re-design (2010)

- Code-duplication greatly reduced
- New requirements met
- Great paper title

LIPIcs Leibniz International Proceedings in Informatics

Fighting Bit Rot with Types (Experience Report: Scala Collections)

M. Odersky[1], A. Moors[2*]

[1] EPFL, Switzerland
martin.odersky@epfl.ch

[2] K.U.Leuven, Belgium
adriaan.moors@cs.kuleuven.be

ABSTRACT. We report on our experiences in redesigning Scala's collection libraries, focussing on the role that type systems play in keeping software architectures coherent over time. Type systems can make software architecture more explicit but, if they are too weak, can also cause code duplication. We show that code duplication can be avoided using two of Scala's type constructions: higher-kinded types and implicit parameters and conversions.

1 Introduction

Bit rot is a persistent problem in most long-running software projects. As software systems evolve, they gain in bulk but lose in coherence and clarity of design. Consequently, maintenance costs increase and adaptations and fixes become more complicated. At some point, it's better to redesign the system from scratch (often this is not done and software systems are left to be limping along because the risk of a redesign is deemed to high).

At first glance it seems paradoxical that bits should rot. After all, computer programs differ from other engineering artefacts in that they do not deteriorate in a physical sense.

map() & friends in Scala 2.8


```
trait TraversableLike[+A, +Repr] {
  def map[B, That](f: (A) ⇒ B)(implicit bf: CanBuildFrom[Repr, B, That]): That
  def filter(p: (A) ⇒ Boolean): Repr
}
trait Seq[+A] extends TraversableLike[A, Seq[A]]
```

- Separate *implementation traits* (ending in -Like) where method implementations defined
  - parameterized on the representation type Repr

map() & friends in Scala 2.8


```
trait TraversableLike[+A, +Repr] {
  def map[B, That](f: (A) ⇒ B)(implicit bf: CanBuildFrom[Repr, B, That]): That
  def filter(p: (A) ⇒ Boolean): Repr
}
trait Seq[+A] extends TraversableLike[A, Seq[A]]
```

- map is passed a 3-parameter CanBuildFrom typeclass
  - Source type **Repr**
  - New element type **B**
  - Result type **That**

map() & friends in Scala 2.8


```
trait TraversableLike[+A, +Repr] {
  def map[B, That](f: (A) ⇒ B)(implicit bf: CanBuildFrom[Repr, B, That]): That
  def filter(p: (A) ⇒ Boolean): Repr
}
trait Seq[+A] extends TraversableLike[A, Seq[A]]
```

- Paraphrasing of what CanBuildFrom[Repr, B, That] means:
  - "*when mapping from a **Repr** to a new element type **B**, the new collection shall have type **That***"
- Type-inference guided by the result of implicit parameter lookup!

map() & friends in Scala 2.8


```
trait TraversableLike[+A, +Repr] {
  def map[B, That](f: (A) ⇒ B)(implicit bf: CanBuildFrom[Repr, B, That]): That
  def filter(p: (A) ⇒ Boolean): Repr
}
trait Seq[+A] extends TraversableLike[A, Seq[A]]
```

- Note filter() does not accept a CanBuildFrom typeclass
  - Never changes the element type, only the number of elements

CanBuildFrom


```
trait CanBuildFrom[-From, -Elem, +To] {
  def apply(from: From): Builder[Elem, To]
  def apply(): Builder[Elem, To]
}
```

- A factory for *Builders, m*utable object that accumulates elements, then builds a new collection with them

```
object Seq {
  implicit def canBuildFrom[A]: CanBuildFrom[Seq[_], A, Seq[A]] = {..}
}
```

CanBuildFrom instances

- Default typeclass instances are defined for each collection type
  - Live in *companion objects* (global provider scope)

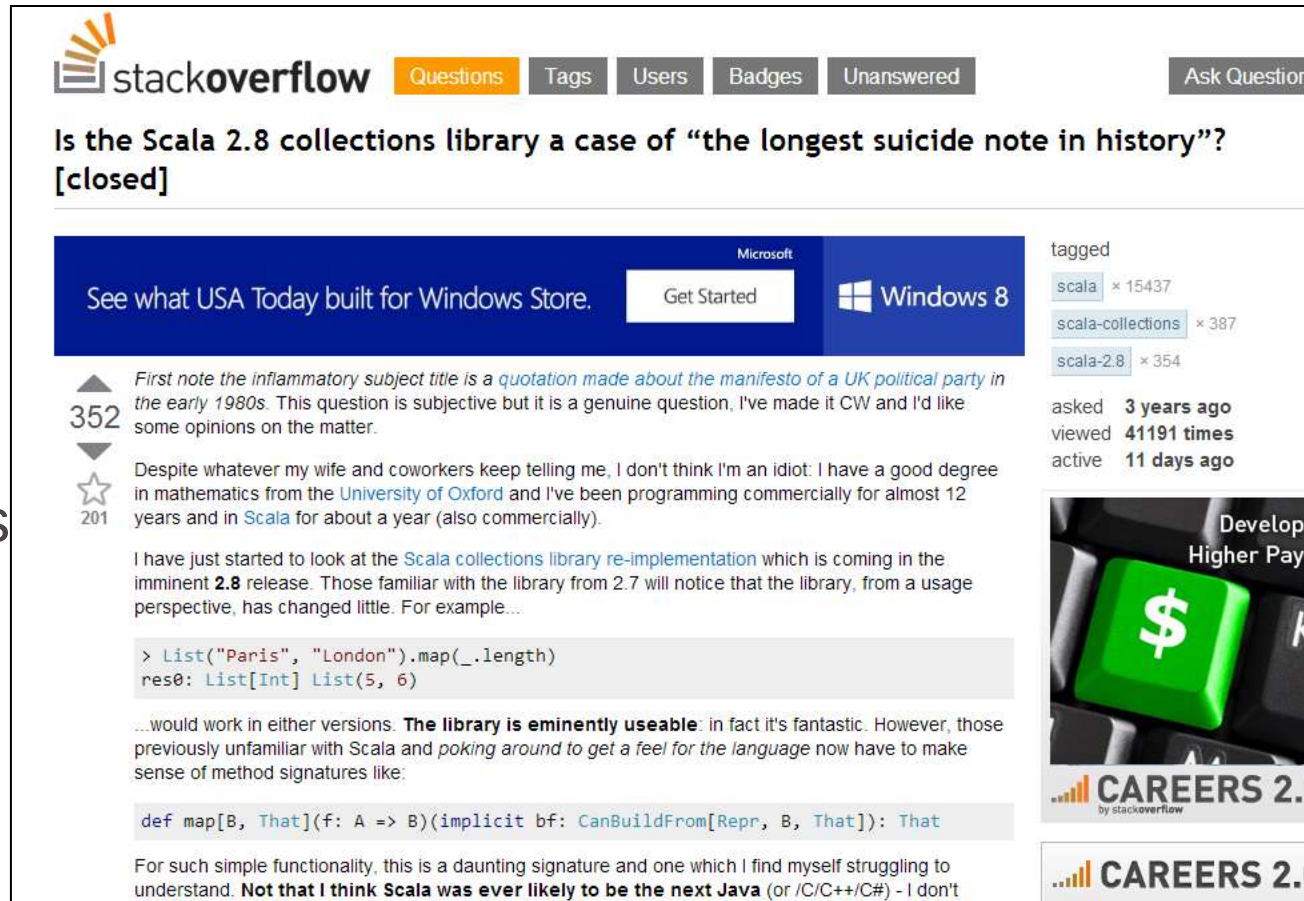
```
object Set {
  implicit def canBuildFrom[A]: CanBuildFrom[Set[_], A, Set[A]] = {..}
}
```

- Defaults can be overridden by *more specific* typeclasses
  - Definition of *specificity* partial ordering is 2 dense pages in Scala spec (Section 6.26.3 Overload Resolution)
  - Basically follows intuition

```
object BitSet {
  implicit def canBuildFrom: CanBuildFrom[BitSet, Int, BitSet]= {..}
}
```

After Scala 2.8

- Scala 2.8 Collections API was the first time typeclass-driven polymorphism used in Scala
  - Mechanism “worked” but was unfamiliar to Scala devs
  - Extra complexity in type signatures unwelcome



stackoverflow Questions Tags Users Badges Unanswered Ask Question

Is the Scala 2.8 collections library a case of “the longest suicide note in history”? [closed]

352

201

First note the inflammatory subject title is a quotation made about the manifesto of a UK political party in the early 1980s. This question is subjective but it is a genuine question, I've made it CW and I'd like some opinions on the matter.

Despite whatever my wife and coworkers keep telling me, I don't think I'm an idiot: I have a good degree in mathematics from the University of Oxford and I've been programming commercially for almost 12 years and in Scala for about a year (also commercially).

I have just started to look at the Scala collections library re-implementation which is coming in the imminent **2.8** release. Those familiar with the library from 2.7 will notice that the library, from a usage perspective, has changed little. For example...

```
> List("Paris", "London").map(_.length)
res0: List[Int] List(5, 6)
```

...would work in either versions. **The library is eminently useable**: in fact it's fantastic. However, those previously unfamiliar with Scala and *poking around to get a feel for the language* now have to make sense of method signatures like:

```
def map[B, That](f: A => B)(implicit bf: CanBuildFrom[Repr, B, That]): That
```

For such simple functionality, this is a daunting signature and one which I find myself struggling to understand. **Not that I think Scala was ever likely to be the next Java** (or /C/C++/C#) - I don't

tagged
scala × 15437
scala-collections × 387
scala-2.8 × 354

asked **3 years ago**
viewed **41191 times**
active **11 days ago**

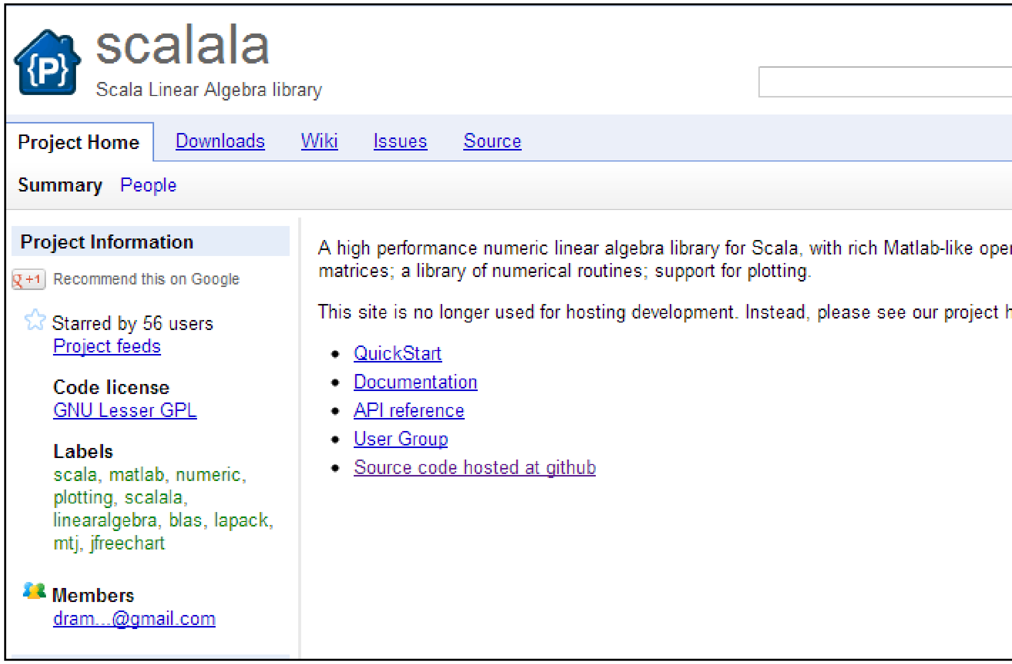
After Scala 2.8

- To my knowledge, Stanford's Daniel Ramage was the first person to apply the technique outside Collections

```
def :*[B,That](b : B)(implicit op : BinaryOp[This,B,OpMul,That]) : That = op(repr,b)

scala> SparseArray(1,0,2) :* Array(2.0,2.0,2.0)
res3: SparseArray[Double](2.0,0.0,4.0) **

scala> SparseArray(1,0,2) :+ Array(2,2,2)
res4: Array[Int](3,2,4) **
```



After Scala 2.8

- “Functional Dependencies in Scala”
  - July 2011 blog by Scala researcher/engineer Miles Sabin
  - Recognized that typeclass-driven polymorphism technique was equivalent to *Functional Dependencies* in Haskell

Home Scala Consultancy Scala Training News Blog About Us Contact Us

Chuusai

Scala Consultancy & Training

Functional Dependencies in Scala

Posted by Miles Sabin on 16th Jul 2011

Functional dependencies are a near-standard extension to Haskell (present in GHC and elsewhere) which allow constraints on the type parameters of type classes to be expressed and then enforced by the compiler. They allow the programmer to state that some of the type parameters of a multi-parameter type class are completely determined by the others. One particularly common application of this is allowing the result type of a function to be a parameter but nevertheless determined by the type(s) of it’s argument(s) — the examples on the Haskell wiki page that I linked to above illustrate that extremely well.

Does this have an equivalent in Scala? We certainly have the equivalent problem — whenever

What is "Functionally Dependent"?

- The technique relies on a 3+ parameter typeclass where the last parameter is uniquely determined by the first two
  - I.e. it is "functionally dependent" upon them

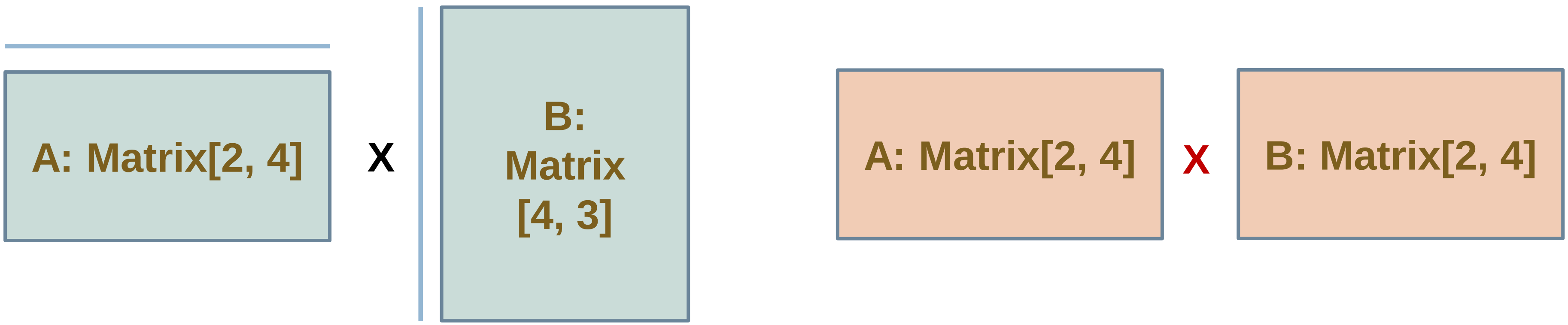
def map[B, That](f: (A) ⇒ B)(implicit bf: CanBuildFrom[Repr, B, That]): That

- Working in tandem with type-inference, this creates a *type-level function*
  - **CanBuildFrom[A, B, C]** is **A ⊗ B => C**

- The static type of **map()** is *computed* by a type level function that runs during compilation

An Example: Shaped Matrices

- Matrix multiplication is only defined when the columns in the matrix left match the rows in the right matrix



1. Encode a matrix's shape into it's type
2. Typeclass-based polymorphic product operator
3. Type-safe, shaped matrix multiplication!

https://github.com/benhutchison/TypeclassDrivenPolymorphism.git

Thank you