

Types Working For You

Richard Dallaway, @d6y

 underscore.io

Modern type system
with lots of power



Julio Capote

@capotej

scala career timeline:

year 1: dope, gonna write some terse code

year 2: hmm, maybe i shouldnt use every feature

year 3: java 8 looks nice

RETWEETS

114

LIKES

175



3:47 PM - 4 Feb 2016



Alessandro Zoffoli

@AL333Z

"[...] We write down the type signature for an operation we want, then "follow the types" to an implementation."

[#fpinscala](#) [#scala](#) [#fp](#)

RETWEET

LIKES

1

4

10:22 AM - 15 Jan 2016

Two Themes

Straightforward Scala

Types Working for Us

Progression

Part 1 Straightforward Scala

Part 2 Functional Programming

Part 3 Typelevel Programming

– Part 1 –

Straightforward Scala

The only problem was we had no idea what the code was doing at first.

<http://jimplush.com/talk/>

The only problem was we had no idea what the code was doing at first.

We came across a strange symbol we hadn't seen in our projects before

<http://jimplush.com/talk/>

The only problem was we had no idea what the code was doing at first.

We came across a strange symbol we hadn't seen in our projects before

The spaceship operator `<|*|>`

The only problem was we had no idea what the code was doing at first.

We came across a strange symbol we hadn't seen in our projects before

The spaceship operator `<|*|>`

Someone said out loud "what the hell is that?"

<http://jimplush.com/talk/>

“It’s about having a maintainable code base where you can have people cross projects easily and get new hires up to speed rapidly”

Power!

Protect the team from it

and

Get the benefit of it

What can we do?

1. Expressions, types, & values
2. Objects and classes
3. Algebraic data types
4. Structural recursion
5. Sequencing computation
6. Type classes

1. Expressions, types, & values
2. Objects and classes
3. Algebraic data types
4. Structural recursion
5. Sequencing computation
6. Type classes

Algebraic data types

Structural recursion

Algebraic data types

data into code

Structural recursion

transformation

Model data with logical
ors and logical *ands*

A website visitor is:

- anonymous; *or*
- logged in

A logged in user has:

- an ID; *and*
- facts we know about them

Two Patterns

and (product types)
or (sum types)

Sum and product together
make algebraic data types

Structure of the code
follows the structure of
the data

A website visitor is:

- anonymous; *or*
- logged in


```
sealed trait Visitor
```

```
case class Anonymous()  
  extends Visitor
```

```
case class User()  
  extends Visitor
```

A logged in user has:

- an ID; *and*
- facts we know about them

An anonymous has:

- an ID

```
sealed trait Visitor
```

```
case class Anonymous()  
  extends Visitor
```

```
case class User()  
  extends Visitor
```

```
sealed trait Visitor
```

```
case class Anonymous(id: Id)  
  extends Visitor
```

```
case class User(id: Id, facts: Set[Fact])  
  extends Visitor
```

Structural recursion

```
def serveAd(v: Visitor): Advert = ???
```

Structure of the code
follows the structure of
the data

```
def serveAd(v: Visitor): Advert = ???
```



```
def serveAd(v: Visitor): Advert =  
  v match {  
    case User(_, info) =>  
    case Anonymous(id) =>  
  }
```

```
def serveAd(v: Visitor): Advert =  
  v match {  
    case User(_, info) => relevantAd(info)  
    case Anonymous(id) => adRotation(id)  
  }
```

```
def serveAd(v: Visitor): Advert =  
  v match {  
    case User(_, info) => relevantAd(info)  
    case Anonymous(id) => adRotation(id)  
  }
```

Structure

ADT & Structural Recursion

Straightforward part of Scala.

Clear, productive, occurs frequently.

Be opinionated in what you use.

Structure helps us.

– Part 2 –

Help from FP Ideas

Combining lists

Concatenating strings

Union of sets

Combining things in a loop

Chaining logical operations

Adding numbers

Building up a JavaScript expression

Showing errors in a UI

...

A **combine** function and
an **empty** value

Addition

Empty	Combine
0	+

Set

Empty	Combine
Set.empty	union

For any T

Empty	Combine
A zero for T	A way to combine two Ts and give me back a T

A **combine** function and
an **empty** value

Monoid

A **combine** function and
an **empty** value
...and **laws**



0:36 / 2:25



Monoids civil war - Doctor Who - The Ark - BBC



BBCClassicDoctorWho

 **Subscribe**

47K

6,333

The boss asks...

What's the total visits to the web site?

```
def report(vs: List[Int]): Int = ???
```

For any T

Empty	Combine
A zero for T	A way to combine two Ts and give me back a T

For any T

```
trait Monoid[T] {  
  def empty: T  
  def combine(x: T, y: T): T  
}
```



```
val addition = new Monoid[Int] {  
    def empty = 0  
    def combine(x: Int, y: Int) = x+y  
}
```

fold

```
def fold(vs: List[Int]): Int =  
  vs match {  
    case Nil => 0  
    case v :: rest => v + fold(rest)  
  }  
  
fold(List(1,2,3))  
// 6
```

```
fold(1, 2, 3)
```

fold(1, 2, 3)

1

fold(1,2,3)

1 + fold(2,3)

`fold(1,2,3)`

`1 + fold(2,3)`

`2`

fold(1,2,3)

1 + fold(2,3)

2 + fold(3)

fold(1,2,3)

1 + fold(2,3)

2 + fold(3)

3 + fold()

0

0 + 3 + 2 + 1 = 6

```
def fold(vs: List[Int]): Int =  
  vs match {  
    case Nil => 0  
    case v :: rest => v + fold(rest)  
  }  
  
fold(List(1,2,3))  
// 6
```

```
def fold(vs: List[Int], m: Monoid[Int]): Int =  
  vs match {  
    case Nil => 0  
    case v :: rest => v + fold(rest)  
  }  
  
fold(List(1,2,3), addition)  
// 6
```

```
def fold(vs: List[Int], m: Monoid[Int]): Int =  
  vs match {  
    case Nil => m.empty  
    case v :: rest => m.combine(v, fold(rest, m))  
  }  
  
fold(List(1,2,3), addition)  
// 6
```

```
def fold[T](vs: List[T], m: Monoid[T]): T =  
  vs match {  
    case Nil => m.empty  
    case v :: rest => m.combine(v, fold(rest, m))  
  }  
  
fold(List(1,2,3), addition)  
// 6
```

Split on cases,
inspect values you have

```
def fold[T](vs: List[T], m: Monoid[T]): T =  
  vs match {  
    case Nil => ???  
    case v :: rest => ???  
  }  
  
fold(List(1,2,3), addition)  
// 6
```

```
def fold[T](vs: List[T], m: Monoid[T]): T =  
  vs match {  
    case Nil => m.empty  
    case v :: rest => ???  
  }  
  
fold(List(1,2,3), addition)  
// 6
```


But back to Monoids...

The boss asks...

What's the total visits to the web site?

```
def report(vs: List[Int]): Int =  
  fold(vs, addition)
```

Benefits

Composition

Flexibility

Problem Solving

The boss asks...

How many distinct visitors?

```
def report(vs: List[Visitor]): Int = ???
```

Set

Empty	Combine
Set.empty	union

The boss says...

Argh!

The servers are OutOfMemory

HyperLogLog

Empty	Combine
<code>new HLL()</code>	<code>HLL.plus</code>

Armon Dadgar (Papers We Love, 2015)
“Bloom Filters and HyperLogLog”

The boss asks...

Who are the really keen
visitors to the site?

Count-Min Sketch

Empty	Combine
<code>new CMS()</code>	<code>CMS.plus</code>

Laura Bledaite (Scala eXchange 2015)

“Count-Min Sketch in Real Data Applications”

We can safely run
a **parallel** version
of **fold**

Laws

$$a + 0 = a$$

$$(a + b) + c = a + (b + c)$$

Identity & Associativity

a combine empty = a

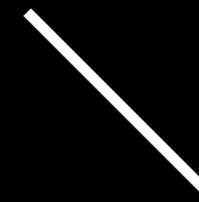
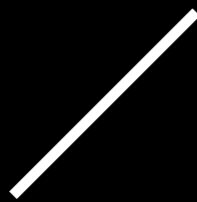
(a combine b) combine c
= a combine (b combine c)

a combine b



combine

combine



c

d

e

f

```
1 sealed trait Visitor {  
2   def id: Id  
3 }  
4  
5 final case class Anonymous(id: Id) extends Visitor  
6 final case class User(id: Id, email: Email) extends Visitor  
7  
8
```

```
users.scala — /Users/richard/Desktop/QCon-London-2016/code
users.scala
1 sealed trait Visitor {
2   def id: Id
3 }
4
5 final case class Anonymous(id: Id) extends Visitor
6 final case class User(id: Id, email: Email) extends Visitor
7
8
Ensim Error: 10 Warnings: 0
File Project No Issues src/main/scala/users.scala 7:1 LF UTF-8 Scala
```

Errors: 10 Warnings: 0

Its a monoid



I know this

...so we fold

Summary

Types and laws give us flexibility & help lead us to solutions.

They help us every day.

– Part 3 –

A Taste of Typelevel

```
csv(  
  List("Date", "Metric"),  
  List(  
    List("Mon", "Low"),  
    List("Tue", "High") )  
)
```

Date	Metric
Mon	Low
Tue	High

```
csv(  
  List("Date"),  
  List(  
    List("Mon", "Low"),  
    List("Tue", "High") )  
)
```

Date	
Mon	Low
Tue	High

How can we prevent that error
happening again?

```
def csv(  
  hdrs: List[String],  
  rows: List[List[String]]  
): String = ???
```



```
import shapeless._
import syntax.sized._

def csv[N <: Nat](
  hdrs: List[String],
  rows: List[List[String]]
): String = ???
```

```
import shapeless._
import syntax.sized._

def csv[N <: Nat](
  hdrs: Sized[List[String], N],
  rows: List[Sized[List[String], N]]
): String = ???
```

```
csv(  
  Sized("Date"),  
  List(  
    Sized("Mon", "Low"),  
    Sized("Tue", "High") )  
)
```

Sized[List, 1]

csv(

Sized("Date"),

List(

Sized("Mon", "Low"),

Sized("Tue", "High"))

)

Sized[List, 2]

How?

`Sized("Date")` constructs
`Sized[Nat]`

`Nat` implements numbers as
types

```
sealed trait Nat
trait Succ[P <: Nat] extends Nat
trait Zero extends Nat
```

Zero 0

Succ[Zero] 1

Succ[Succ[Zero]] 2

Succ[Succ[Succ[Zero]]] 3

```
sealed trait Nat
trait Succ[P <: Nat] extends Nat
trait Zero extends Nat
```



```
sealed trait Nat
trait Succ[P <: Nat] extends Nat
trait Zero extends Nat

type One = Succ[Zero]
type Two = Succ[One]
```

```
sealed trait Nat
trait Succ[P <: Nat] extends Nat
trait Zero extends Nat

type One = Succ[Zero]
type Two = Succ[One]

implicitly[Succ[Zero] == One]
```

```
sealed trait Nat
trait Succ[P <: Nat] extends Nat
trait Zero extends Nat
```

```
type One = Succ[Zero]
type Two = Succ[One]
```

```
implicitly[Succ[Zero] == One]
implicitly[Succ[One] == Succ[Succ[Zero]]]
```

```
sealed trait Nat
trait Succ[P <: Nat] extends Nat
trait Zero extends Nat
```

```
type One = Succ[Zero]
type Two = Succ[One]
```

```
implicitly[Succ[Zero] == Two]
```

error:

Cannot prove that Succ[Zero] == Two.

Merging Fields

```
case class User(  
  id      : Long,  
  name    : String,  
  email   : Option[String])
```

```
val user = User(  
  123L,  
  "Bruce Wayne",  
  Some("bruce@example.org"))
```

```
PATCH /user/123
```

```
{  
  "name" : "Batman"  
}
```

```
case class User(  
  id      : Long,  
  name    : String,  
  email   : Option[String])
```

```
case class Update(  
  name    : Option[String],  
  email   : Option[Option[String]])
```



```
val user = User(  
  123L,  
  "Bruce Wayne",  
  Some("bruce@example.org"))
```

```
val update = Update(  
  Some("Batman"),  
  None)
```

How do we get to...

```
User(  
  123L,  
  "Batman",  
  Some("bruce@example.org"))
```

Bulletin

<https://github.com/davegurnell/bulletin>

How?

User	String	Option[String]	...
Update	Option[String]	Option[Option[String]]	...

How?

User	String	Option[String]	...
Update	Option[String]	Option[Option[String]]	...
	Head		

How?

User	String	Option[String]	...
Update	Option[String]	Option[Option[String]]	...
	Head	The Rest...	

How?

Type constraints

Implicit methods

HLists

Labelled generic

Macros

...

```
val user = User(
  123L,
  "Bruce Wayne",
  Some("bruce@example.org"))

val update = Update(
  Some("Batman"),
  None)

import bulletin._

val updated = user.merge(update)

// User(
//   123L,
//   "Batman",
//   Some("bruce@example.org"))
```

```
val user = User(  
  123L,  
  "Bruce Wayne",  
  Some("bruce@example.org"))
```

```
val update = Update(  
  Some("Batman"),  
  None)
```

```
import bulletin._  
val updated = user.merge(update)
```

```
// User(  
//   123L,  
//   "Batman",  
//   Some("bruce@example.org"))
```


Summary

The compiler can help (maybe more than you thought).

Reduce boilerplate code.

Using Power Tools

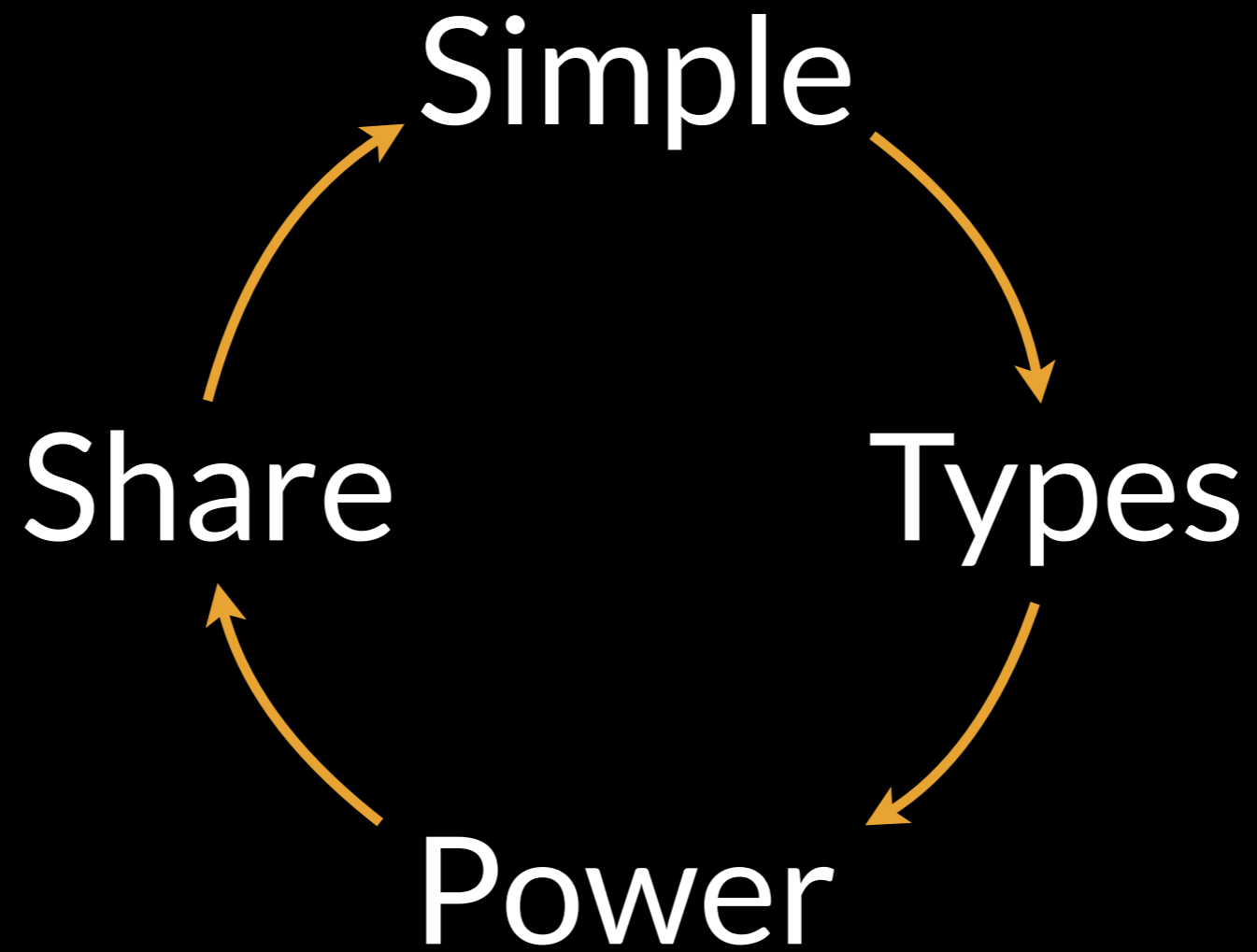
Can go one of two ways...

Using Power Tools

Can go one of two ways...

What the hell
is that?

It's a monoid!
I know this



2008

‘The name Scala stands for
“scalable language.”

The language is so named
because it was designed
to grow with the demands of its
users.’

What have we seen?

Some straightforward parts of Scala

– Clear, maintainable, helpful

Encoding ideas in types

– flexibility, leads us to solutions

Let the compiler do it

– when it make sense for your demands

Summary

Scala scaling with your needs

–be opinionated in what you use, more when needed

Types working for us, not stopping us

–functional programming, share what you learn

Thanks!

Richard Dallaway, @d6y

 underscore.io

Thanks!

Richard Dallaway, @d6y

Amanda Laucher

Wesley Reisz

Noel Welsh

Dave Gurnell

Miles Sabin

Jono Ferguson

Julio Capote

Alessandro Zoffoli

 underscore.io