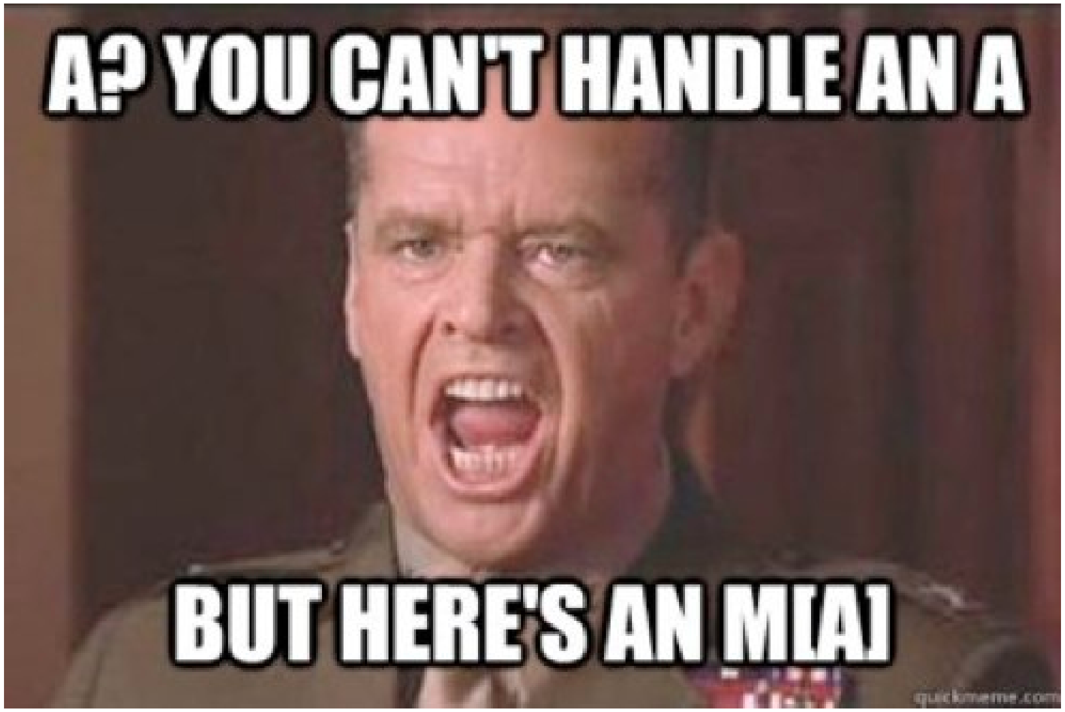
scalaz "For the Rest of Us"

Adam Rosien
adam@rosien.net

@arosien #ScalaIO

Scalaio



`scalaz` has a (undeserved?) reputation as being, well, kind of crazy.

So this talk is specifically *not* about:

- Functors, Monads, or Applicative Functors
- Category theory
- Other really cool stuff you should learn about (eventually)

This talk *is* about **every-day situations** where `scalaz` can:

- *Reduce* syntactical noise
- Provide useful types that solve *many classes* of problems
- *Add* type-safety with minimal "extra work"

Getting Started

In `build.sbt`:

```
libraryDependencies +=
  "org.scalaz" %% "scalaz-core" % "7.0.4"
```

Then:

```
import scalaz._
import Scalaz._

// profit
```

Memoization

Memoization

The goal: cache the result of an expensive computation.

```
def expensive(foo: Foo): Bar = ...

val f: Foo

expensive(f) // $$$
expensive(f) // $$$
...
```

Memoization

Typically you might use a `mutable.Map` to cache results:

```
val cache = collection.mutable.Map[Foo, Bar]()

cache.getOrElseUpdate(f, expensive(f)) // $$$
cache.getOrElseUpdate(f, expensive(f)) // 1¢
```

Memoization

You can try to make it look like a regular function, avoiding the `getOrElseUpdate()` call:

```
val cache: Foo => Bar =
  collection.mutable.Map[Foo, Bar]()
    .withDefault(expensive _)

cache(f) // $$$ (miss & NO fill)
cache(f) // $$$ (miss & NO fill)
```

But it doesn't actually cache.

Memoization

In `scalaz`:

```
def expensive(foo: Foo): Bar = ...

// Memo[Foo, Bar]
val memo = Memo.immutableHashMapMemo {
  foo: Foo => expensive(foo)
}

val f: Foo

memo(f) // $$$ (cache miss & fill)
memo(f) // 1¢  (cache hit)
```

Memoization

Many memoization strategies:

```
Memo.immutableHashMapMemo[K, V]

Memo.mutableHashMapMemo[K, V]

// remove + gc unreferenced entries
Memo.weakHashMapMemo[K, V]

// fixed size, K = Int
Memo.arrayMemo[V](size: Int)
```

Memoization

`scalaz` memoization:

- Pros: uniform types for memoizer and function
- Cons: less low-level control

Style

Style

Remove the need for temporary variables:

```
val f: A => B
val g: B => C

// using temps:
val a: A = ...
val b = f(a)
val c = g(b)

// or via composition, which is a bit ugly:
val c = g(f(a))

// "unix-pipey", aka, the Thrush combinator
val c = a |> f |> g
```

Style

When you just can't stand all that (keyboard) typing:

```
val p: Boolean

// ternary-operator-ish
p ? "yes" | "no" // if (p) "yes" else "no"

val o: Option[String]

o | "meh"        // o.getOrElse("meh")
```

Style

More legible (and more type-safe):

```
// scala
Some("foo")  // Some[String]
None         // None.type

// scalaz
"foo".some   // Option[String]
none         // Option[Nothing], oops!
none[String] // Option[String]
```

Style

Pros: less noise, more expressive, more type-safe

Cons: you have to know these operators

Domain Validation

Domain Validation

This isn't good:

```
case class SSN(
  first3: Int,
  second2: Int,
  third4: Int)

SSN(123, 123, 1234)
//       ^^^ noo!
```

Domain Validation

This shouldn't be possible:

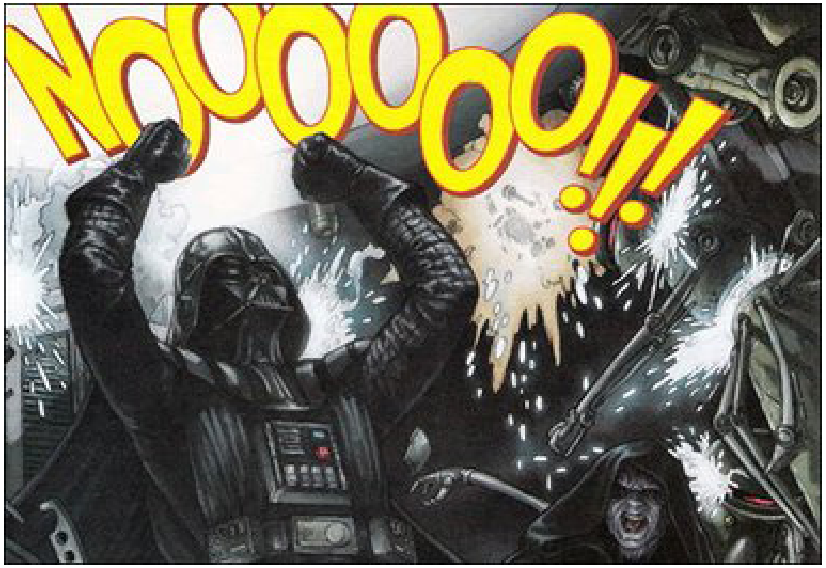
```
case class Version(major: Int, minor: Int)

Version(1, -1)
//         ^^ noooo!
```

Domain Validation

Meh:

```
case class Dependency(
  organization: String,
  artifactId: String,
  version: Version)

Dependency("zerb", "", Version(1, 2))
//                 ^^ nooooo!
```

Domain Validation



Domain Validation

The problem is that the types as-is aren't really accurate. `String`s and `Int`s are being used too broadly. We really want "`Int`s greater than zero", "`String`s that match a pattern", etc.

You can do the checks in the constructor:

```
case class Version(major: Int, minor: Int) {
  require(
    major >= 0,
    "major must be >= 0: %s".format(major))
  require(
    minor >= 0,
    "minor must be >= 0: %s".format(minor))
}
```

Domain Validation

But this has downsides:

- Validation failures happen as **late** as possible.
- You only get one failure, but **more than one violation** may be happening.
- You have to catch exceptions, which is just **tedious**.

Domain Validation

Errors in Scala can be represented in many ways:

```
Option[A] :=
  Some[A](a: A)    | None

Either[A, B] :=
  Right[B](b: B)   | Left[A](a: A)

Try[A] :=
  Success[A](a: A) | Failure[A](ex: Throwable)
```

Domain Validation

Errors in Scala can be represented in many ways:

```
Option[A] :=
  Some[A](a: A)    | None

Either[A, B] :=
  Right[B](b: B)   | Left[A](a: A)

Try[A] :=
  Success[A](a: A) | Failure[A](ex: Throwable)

Validation[E, A] :=
  Success[A](a: A) | Failure[E](e: E)
```

Domain Validation

```
val major: Int = ...
val minor: Int = ...
val version: ??? =
  Version.validate(major, minor)

version | Version(1, 0) // provide default

// handle failure and success
version.fold(
  (fail: ???)        => ...,
  (success: Version) => ...)
```

Domain Validation

Using `scalaz`, a `Validation` can either be a **`Success`** or **`Failure`**:

```
Version.validate(1, 2)
// Success(Version(1, 2))
// ^^^^^^^

Version.validate(1, -1)
// Failure(NonEmptyList("digit must be >= 0"))
// ^^^^^^^    ^
//            ^
//            to be defined
```

Domain Validation

Model the `>= 0` constraint:

```
case class Version(
  major: Int, // >= 0
  minor: Int) // >= 0

object Version {
  def validDigit(digit: Int):
    Validation[String, Int] =
      (digit >= 0) ?
        digit.success[String] |
        "digit must be >= 0".fail
  ...
}
```

Domain Validation

Combine constraints:

```
object Version {
  def validDigit(digit: Int):
    Validation[String, Int] = ...

  // WAT?
  def validate(major: Int, minor: Int):
    ValidationNel[String, Version] =
    (validDigit(major).toValidationNel |@|
     validDigit(minor).toValidationNel) {
      Version(_, _)
    }
}
```

Domain Validation

Let's break down `validDigit(major).toValidationNel`:

```
validDigit(major)
// Validation[String, Int]

validDigit(major).toValidationNel
// Validation[NonEmptyList[String], Int]

// "Nel" = NonEmptyList
type ValidationNel[E, A] =
  Validation[NonEmptyList[E], A]
```

Domain Validation

```
val maj = validDigit(major).toValidationNel
val min = validDigit(minor).toValidationNel
// Both ValidationNel[String, Int]
//                            ^^^

val mkVersion = Version(_, _)
// (Int, Int) => Version

val version = (maj |@| min) { mkVersion }
// ValidationNel[String, Version]
//                       ^^^^^^^
```

Domain Validation

The general form of combining `ValidationNel`:

```
(ValidationNel[E, A] |@|
 ValidationNel[E, B]) {
  (A, B) => C
} // ValidationNel[E, C]

(ValidationNel[E, A] |@|
 ValidationNel[E, B] |@|
 ValidationNel[E, C]) {
  (A, B, C) => D
} // ValidationNel[E, D]

// etc.
```

Talk to Lars about the new way to do this via `HList`.

Domain Validation

The "rules":

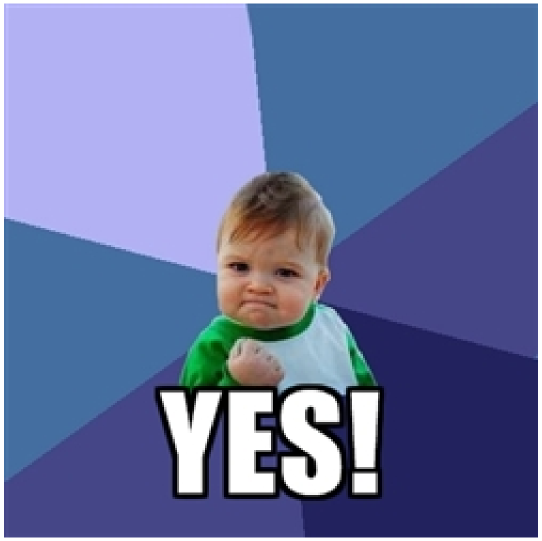
```
Success |@| Success // Success
Success |@| Failure // Failure
Failure |@| Success // Failure
Failure |@| Failure // Failure
// and accumulate fail values!

// Accumulate?
NonEmptyList("foo") |+| NonEmptyList("bar")
// NonEmptyList("foo", "bar")

// |+| "appends" things according to "the rules"
```

Domain Validation

An improvement?

- Pro: `Validation` is **more appropriate** than `Try` or `Either`/`Left`/`Right`.
- Pro: Each rule is **just a function** producing a `Validation`.
- Pro: Rules can be **composed** together into new validations, of differing types.
- Pro: Composed rules **accumulate** all the errors vs. failing fast.
- Con: `toValidationNel`, `|@|`, etc., is **incomprehensible** if you're not familiar.

Overall:



Operations on "deep" data structures

Operations on "deep" data structures

Let's say you have some nested structure like a tree:

```
// the data
case class Foo(name: String, factor: Int)

// a node of the tree
case class FooNode(
  value: Foo,
  children: Seq[FooNode] = Seq())
```

Operations on "deep" data structures

Make a tree of `Foo`'s:

```
val tree =
  FooNode(
    Foo("root", 11),
    Seq(
      FooNode(Foo("child1", 1)),
      FooNode(Foo("child2", 2)))) // <-- * 4
```

Task: Create a new tree where the *second child's* `factor` is multiplied by 4.

Operations on "deep" data structures

Let's try all at once:

```
val secondTimes4: FooNode => FooNode =
  node => node.copy(children = {
    val second = node.children(1)
    node.children.updated(
      1,
      second.copy(
        value = second.value.copy(
          factor = second.value.factor * 4)))
  })
```

Eww: temporary variables, `x.copy(field = f(x.field))` boilerplate, deep nesting for every level.

Operations on "deep" data structures

Wouldn't it be better to have one thing to address something in a `Foo` or `FooNode`, and just combine them?

```
val second = // second node child
val value  = // value of a node
val factor = // factor field of Foo

val secondFactor = second ??? value ??? factor

secondFactor(tree) // get 2

secondFactor.set(8,     tree) // set 8 there
secondFactor.mod(_ * 4, tree) // modify there
```

Operations on "deep" data structures

```
val second: Lens[FooNode, FooNode] =
  Lens.lensu(
    (node, c2) => node.copy(
      children = node.children.updated(1, c2)),
    _.children(1))

val value: Lens[FooNode, Foo] =
  Lens.lensu(
    (node, value) => node.copy(value = value),
    _.value)

val factor: Lens[Foo, Int] =
  Lens.lensu(
    (foo, fac) => foo.copy(factor = fac),
    _.factor)
```

Operations on "deep" data structures

The `Lens` type encapsulates "getters" and "setters" on another type.

```
Lens.lensu[Thing, View](
  set: (Thing, View) => Thing,
  get: Thing => View)

val thing: Thing
val lens:  Lens[Thing, View] = ...

val view: View = lens(thing) // apply = get

// "set" a view
val thing2: Thing = lens.set(view, thing)
```

Operations on "deep" data structures

Lenses *compose*:

```
val secondFactor: Lens[FooNode, Int] =
  second andThen value andThen factor     /*
  ^              ^             ^
  Lens[FooNode, FooNode]       ^
                 ^             ^
                 Lens[FooNode, Foo]
                               ^
                               Lens[Foo, Int]
                               */
```

Operations on "deep" data structures

```
/* FooNode(
     Foo("root", 11),
     Seq(
       FooNode(Foo("child1", 1)),
       FooNode(Foo("child2", 2))))
                             ^
                             ^
                             ^  */
secondFactor(tree)        // 2
```

Operations on "deep" data structures

```
/* FooNode(
     Foo("root", 11),
     Seq(
       FooNode(Foo("child1", 1)),
       FooNode(Foo("child2", 2))))
                             ^
                             ^  */
secondFactor.mod(       _ * 4, tree)
/* FooNode(                  ^
     Foo("root", 11),        ^
     Seq(                    ^
       FooNode(Foo("child1", ^)),
       FooNode(Foo("child2", 8))))
 */
```

Operations on "deep" data structures

`scalaz` for "deep" access:

- Pros: composable so you can "go deeper" **for free**.
- Cons: Need to be manually created. (But there is an experimental compiler plugin to autogenerate them for all case classes!)

Thanks!

scalaz "For the Rest of Us"

Adam Rosien
`adam@rosien.net`

`@arosien #ScalaIO`

Thank the `scalaz` authors: runarorama, retronym, tmorris, larsh and lots others.

Credits, sources and references:

- This presentation: arosien/scala-io-2013-scalaz-talk
- `scalaz` homepage: https://github.com/scalaz/scalaz
- Eugene Yakota, Learning Scalaz