Scalaz: Functional Programming in Scala

Rúnar Bjarnason

http://scalaz.org

Who am I?

Rúnar Óli Bjarnason

@runarorama

Senior Software Engineer, Capital IQ, Boston

Contributor to Scalaz since 2008

Author, "Functional Programming in Scala" (Manning, 2012)

What is Scalaz?

A library for pure functional programming in Scala

- Purely functional data types
- Type classes
- Pimped-out standard library
- Effect tracking
- Concurrency abstractions

Functional Programming

Functional Programming is Programming with Functions

What is a Function?

A function `f : A => B` relates every value of type `A` to exactly one value of type `B`. This is all a function is allowed to do. No *side-effects*!

What's a Side-Effect?

- Reading/writing files
- Re-assigning variables
- Setting fields on objects
- Mutating data structures
- Throwing an exception

Anything that violates *Referential Transparency*

Referential Transparency

An expression `e` is *referentially transparent* if every occurrence of `e` can be replaced with its value without affecting the observable result of the program.

A function `f` is *pure* if the expression `f(x)` is referentially transparent for all referentially transparent `x`.

A Side-Effect Appeals to a Lie

```
// Not a function of type Int => Int
def foo(x: Int): Int = {
  launchTheMissile
  x + 1
}
```

Functional Programming is: being honest about types.

Why?

- *Modularity*: Elements of a program can be separated, repurposed, and recombined.
- *Compositionality*: Understand the components, and the rules of combination, and you understand the whole.

In Scala, FP is a Discipline

Scala does not enforce referential transparency. It's up to you to keep your types honest.

Scalaz: Getting Started

```
import scalaz._
import Scalaz._
```

Scalaz: Getting Started

Where is the code?

- `A => Identity[A]`
- `M[A] => MA[M[_], A]`
- `M[A, B] => MAB[M[_,_], A, B]`
- Wrappers: `OptionW[A]`, `ListW[A]`, etc.

Example: Type-safe Equality

Old and busted:

```
def isInMap(k: String,
            v: String,
            m: Map[String, String]): Boolean =
  m.get(k) == v
```

Example: Type-safe Equality

New hotness:

```
scala> def isInMap(k: String,
     |              v: String,
     |              m: Map[String, String]): Boolean =
     |   m.get(k) === v
<console>:17: error: type mismatch;
 found   : String
 required: Option[String]
         m.get(k) === v
                      ^
```

Example: Type-safe Equality

A type class:

```
trait Equal[A] {
  def equal(a1: A, a2: A): Boolean
}
```

`Equal[T]` witnesses that `T` can be compared for equality.

Example: Type-safe Equality

An equality for strings

```
implicit val stringEq: Equal[String] = equal(_ == _)
```

Example: Type-safe Equality

An equality for options

```
implicit def optionEq[A:Equal]: Equal[Option[A]] =
  equal { (a1, a2) =>
    (a1 |@| a2)(_ === _) | (a1.isEmpty && a2.isEmpty)
  }
```

Example: Type-safe Equality

New hotness:

```
def isInMap[K,V:Equal](k: K, v: V, m: Map[K,V]): Boolean =
  m.get(k) === Some(v)
```

Example: Type-safe Equality

Benefits:

- Type safety
- Compositionality
- Modularity

Monoid Type Class

```scala
trait Semigroup[M] {
  def append(a: M, b: M): M
}
```

append(a, append(b, c)) = append(append(a, b), c)

```scala
trait Monoid[M] extends Semigroup[M] {
  val zero: M
}
```

- append(a, zero) = a
- append(zero, a) = a

Examples of Monoids

- `Int` with + and `0`
- `Int` with `*` and `1`
- `Boolean` with || and `false`
- `A => A` with `compose` and `identity`
- `List[A]` with ++ and `Nil`
- `String` with + and `""`

Scalaz has *a lot* of `Monoid` instances.

Using a Monoid

```
scala> 1 |+| 2
res0: Int = 3

scala> mzero[Int]
res1: Int = 0
```

Monoids are Type Safe

```
scala> 1 + "3"
res9: String = 13

scala> 1 |+| "3"
<console>:14: error: type mismatch;
 found   : String("3")
 required: Int
              1 |+| "3"
                    ^
```

Monoids Compose

(Monoid[A], Monoid[B]) => Monoid[(A, B)]

```
scala> (1, "foo") |+| (3, "bar")
res12: (Int, String) = (4,foobar)

scala> mzero[(Int, String)]
res13: (Int, String) = (0,"")
```

Monoids Compose

`Monoid[B]` => `Monoid[A => B]`

```
f |+| g = (x => f(x) |+| g(x))

mzero[A => B] = (x => mzero[B])
```

Monoids Compose

```
Monoid[A] => Monoid[Option[A]]
```

```
scala> some("abc") |+| some("def")
res2: Option[String] = Some(abcdef)

scala> mzero[Option[String]]
res3: Option[String] = None
```

Monoids Compose

`Monoid[V]` => `Monoid[Map[K,V]]`

```
scala> Map("a" -> 2, "b" -> 1) |+| Map("a" -> 3, "c" -> 4)
res14: Map[String,Int] = Map(a -> 5, c -> 4, b -> 1)
```

Monoids add Modularity

The same code can be re-used for all monoids:

```
def sum[A:Monoid](as: List[A]): A =
  as.foldLeft(mzero)(_ |+| _)
```

Foldable

If there exist implicit `Foldable[M]` and `Monoid[A]`, then

- M[A].sum: A
- M[B].foldMap(B => A): A

Foldable

Example:

```
scala> List(1,2,3).asMA.sum
res15: Int = 6

scala> List(some(1), some(4), none).asMA.sum
res16: Option[Int] = Some(5)

scala> some(2) foldMap (_ + 1)
res17: Int = 3

scala> List(3,4,5) foldMap multiplication
res18: scalaz.IntMultiplication = 60
```

Validation

Purely functional error handling

```
sealed trait Validation[+E, +A]
case class Success[+E, +A](a: A) extends Validation[E, A]
case class Fail[+E, +A](e: E) extends Validation[E, A]
```

Validation

Creating successes and failures

```
scala> 10.success
res24: scalaz.Validation[Nothing,Int] = Success(10)

scala> "oops".fail
res25: scalaz.Validation[String,Nothing] = Failure(oops)
```

Validation

```
def checkEmail(e: String): Validation[String, String] =
  if (validEmail(e))
    email.success
  else
    "Invalid email address".fail
```

Validation

Validations compose with `map` and `flatMap`.

```
def validateWebForm(name: String,
                    email: String,
                    phone: String)
: Validation[String, WebForm] =
  for {
    e <- checkEmail(email)
    p <- checkPhone(phone)
  } yield WebForm(name, e, p)
```

Validation

If the failure type is a `Monoid`, we can accumulate failures.

```
def validateWebForm(name: String,
                    email: String,
                    phone: String)
: Validation[String, WebForm] =
  (checkEmail(email) |@| checkPhone(phone)) {
    WebForm(name, _, _)
  }
```

Validation

Scalaz provides `ValidationNEL` where the failure is a list.

```
type ValidationNEL[E, A] = Validation[NonEmptyList[E], A]
```

Validation

```scala
def validateWebForm(name: String,
                    email: String,
                    phone: String)
: ValidationNEL[String, WebForm] =
  (checkEmail(email).liftFailNel |@|
   checkPhone(phone).liftFailNel) {
    WebForm(name, _, _)
  }
```

Validation

A list of validations can be turned into a validation of a list.

```
scala> type MyValidation[A] = Validation[String, A]
defined type alias StringValidation

scala> val x: List[MyValidation[Int]] =
     |   List(1.success, 2.success)
x: List[MyValidation[Int]] = List(Success(1), Success(2))

scala> x.sequence
res9: MyValidation[List[Int]] = Success(List(1, 2))
```

Applicative Functors

Any `M[A]` can be composed with `|@|` if there exists `Applicative[M]` in implicit scope.

```
scala> (some(30) |@| some(10) |@| some(2)) { (x, y, z) =>
     |   if (x > 20) y else z }
res10: Option[Int] = Some(10)

scala> (List("foo","bar") |@| List("baz","qux")) { _ |+| _ }
res11: List[String] = List(foobaz, fooqux, barbaz, barqux)

scala> (((_: Int) + 1) |@| ((_: Int) * 2)) { _ |+| _ }
res12: Int => Int = <function1>

scala> res12(4)
res13: Int = 13
```

Applicative Functors

Any `F[G[A]]` can be inverted to `G[F[A]]` if there exists `Applicative[G]` and `Traverse[F]`.

```
scala> List(some(1), some(2), some(3)).sequence
res14: Option[List[Int]] = Some(List(1, 2, 3))

scala> res14.sequence
res15: List[Option[Int]] = List(Some(1), Some(2), Some(3))
```

Applicative Functors

Any `F[G[A]]` can be inverted to `G[F[A]]` if there exists `Applicative[G]` and `Traverse[F]`

```
scala> type IntFn[A] = Int => A
defined type alias IntFn

scala> val fs: List[IntFn[Int]] = List(_ + 1, _ * 2)
fs: List[Int => Int] = List(<function1>, <function1>)

scala> val f = fs.sequence
f: Int => List[Int] = <function1>

scala> f(10)
res16: List[Int] = List(11, 20)
```

Applicative Functors

```
trait Applicative[M[_]] {
  // (a |@| f)(_(_))
  def apply[A, B](a: M[A], f: M[A => B]): M[B]

  def pure[A](a: A): M[A]
}

trait Traverse[T[_]] {
  def traverse[M[_]:Applicative, A, B](
    a: T[A],
    f: A => M[B]): M[T[B]]
}

x.sequence = traverse(x, a => a)
```

State

```
f: (A, S) => (B, S)
g: (B, S) => (C, S)

f andThen g : (A, S) => (C, S)
```

State

```
f: A => S => (B, S)
g: B => S => (C, S)
```

How to compose these?

State

```
type State[S, A] = S => (A, S)

f: A => State[S, B]
g: B => State[S, C]

((a: A) => f(a) flatMap g): A => State[S, C]
```

State

A generic zipWithIndex:

```
type IntState[A] = State[Int, A]

def indexed[M[_]:Traverse, A](m: M[A]) =
  m.traverse[IntState, (A, Int)](a => for {
    x <- init
    _ <- modify((_:Int) + 1)
  } yield (a, x)) ! 0
```

Thank You!

More information:

- http://scalaz.org
- #scalaz on Freenode
- http://groups.google.com/scalaz

Come talk to me at any time today.