Applicative Functor

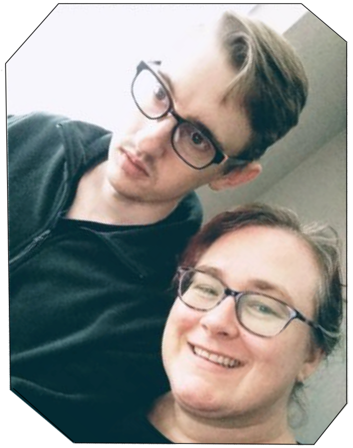
Part 3

learn how to use Applicative Functors with Scalaz
through the work of

Sam Halliday
@fommil

Debasish Ghosh
@debasishg

John A De Goes
@jdegoes



Julie Moronuki
Chris Martin
@chris_martin
@argumatronic

slides by @philip_schwarz

slideshare https://www.slideshare.net/pjschwarz

@philip_schwarz

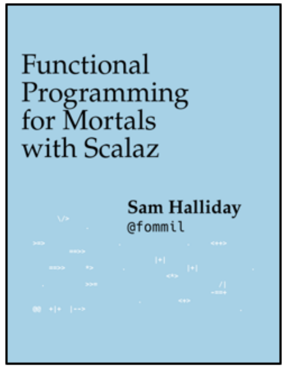
In **Part 2**, we translated the username/password validation program from **Haskell** into **Scala** and to do that we coded our own **Scala Applicative typeclass** providing **<*>** and ***>** functions.

Let's now look at how we can avoid coding **Applicative** ourselves by using **Scalaz**.

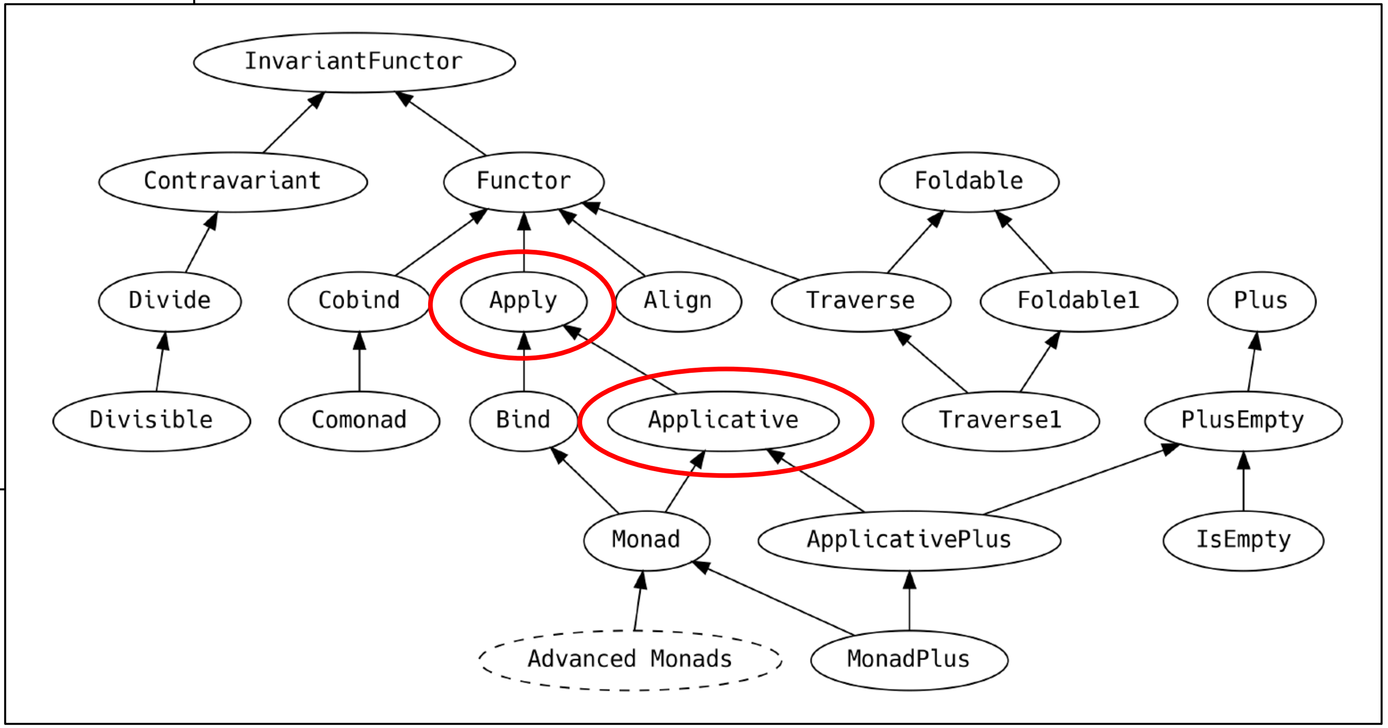
As Sam Halliday explains in his great book, **Functional Programming for Mortals with Scalaz**, in **Scalaz** there is an **Applicative typeclass** and it extends an **Apply typeclass** which in turn extends the **Functor typeclass**.

Functional Programming for Mortals with Scalaz

the book by **Sam Halliday**



@fommil



5.6.1 Apply

Apply extends Functor by adding a method named ap which is similar to map in that it applies a function to values. However, with ap, the function is in the same context as the values.

```
@typeclass trait Apply[F[_]] extends Functor[F] {
  @op("<*>") def ap[A, B](fa: =>F[A])(f: =>F[A => B]): F[B]
...
```

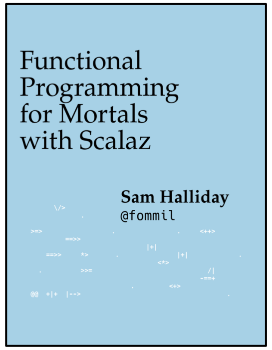
<*> is the **Advanced** TIE Fighter, as flown by Darth Vader. Appropriate since it looks like an angry parent. Or a sad Pikachu.

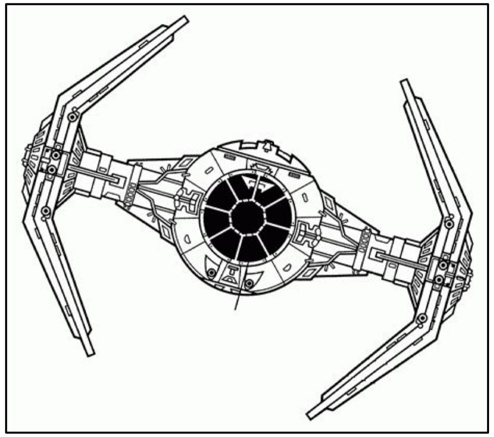
It is worth taking a moment to consider what that means for a simple data structure like Option[A], having the following implementation of ap

```
implicit def option[A]: Apply[Option[A]] = new Apply[Option[A]] {
  override def ap[A, B](fa: => Option[A])(f: => Option[A => B]) = f match {
    case Some(ff) => fa.map(ff)
    case None => None
  }
  ...
}
```

To implement ap, we must first extract the function ff: A => B from f: Option[A => B], then we can map over fa. The extraction of the function from the context is the important power that Apply brings, allowing multiple functions to be combined inside the context.

Sam Halliday @fommil





Star Wars **TIE Fighter**

Let's try out the (automatically available) **Apply** instance for **Option**

```
import scalaz._, Scalaz._

val inc: Int => Int = _ + 1

assert( (2.some <*> inc.some) == 3.some )
assert( (  None <*> inc.some) == None    )
assert( (2.some <*> None)     == None    )
assert( (  None <*> None)     == None    )
```

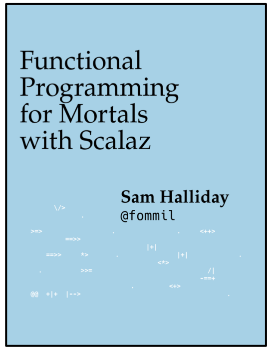
Returning to **Apply**, we find **applyX** boilerplate that allows us to **combine parallel functions and then map over their combined output**:

```
@typeclass trait Apply[F[_]] extends Functor[F] {
  ...
  def apply2[A,B,C](fa: =>F[A], fb: =>F[B])(f: (A, B) => C): F[C] = ...
  def apply3[A,B,C,D](fa: =>F[A],fb: =>F[B],fc: =>F[C])(f: (A,B,C) =>D): F[D] =
  ...
  ...
  def apply12[...]
```

Read apply2 as a contract promising: "if you give me an F of A and an F of B, with a way of combining A and B into a C, then I can give you an F of C". There are many uses for this contract and the two most important are:

- **constructing some typeclasses for a product type C from its constituents A and B**
- **performing effects in parallel**, like the drone and google algebras we created in Chapter 3, **and then combining their results**.

Sam Halliday @fommil



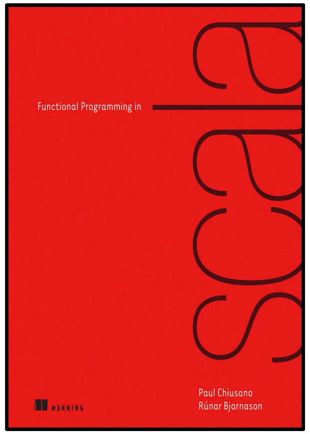
See next slide for a reminder of examples of this use that we have already seen.

We'll look at this second use later on.

Remember in **Part 1**, when we saw **FP in Scala** explaining that **Applicative** can be formulated either in terms of **unit** and **map2** or in terms of **unit** and **apply** (see right hand side) ? In **Scalaz apply** is called **ap** (see previous slide), and **map2** is called **apply2**. Similarly for **map3**, **map4**, etc, which in **Scalaz** are called **apply3**, **apply4**, etc (see also next slide).

The names **map2**, **map3**, etc make sense if you compare the signatures with **map**'s signature and think of **map** as being **map1**.

BTW, there is a typo in the signature of **map2** in **FP in Scala**: it should end in F[C] rather than F[A] !



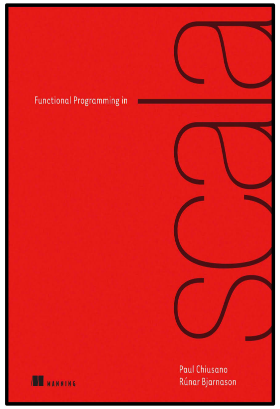
EXERCISE 12.2

Hard: The name *applicative* comes from the fact that we can formulate the Applicative interface using an alternate set of primitives, unit and the function apply, rather than unit and map2. Show that this formulation is equivalent in expressiveness by defining map2 and map in terms of unit and apply. Also establish that apply can be implemented in terms of map2 and unit.

```
trait Applicative[F[_]] extends Functor[F] {

  def apply[A,B](fab: F[A => B])(fa: F[A]): F[B]
  def unit[A](a: => A): F[A]

  def map[A,B](fa: F[A])(f: A => B): F[B]
  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A,B) => C): F[A]
}
```

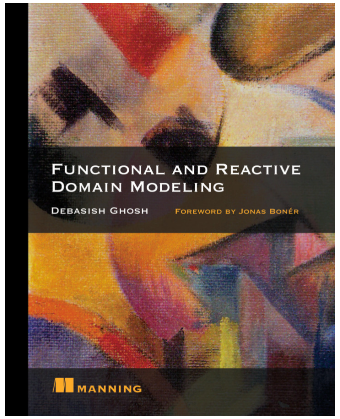
Define in terms of map2 and unit.

Define in terms of apply and unit.



Another reminder that in **FP in Scala**, **Scalaz**'s **apply2**, **apply3**, **apply4**, etc are called **map2**, **map3**, **map4**, etc.

Scalaz's **apply2**, **apply3**, **apply4**, etc are called the same in **Functional and Reactive Domain Modeling**



This slide also reminds us of examples that we have already seen of one of the two most important uses that **Sam Halliday** attributes to functions **apply2**, **apply3**, etc, i.e. '**constructing some typeclasses for a product type C from its constituents A and B**'

The `apply` method is useful for implementing `map3`, `map4`, and so on, and the pattern is straightforward. Implement `map3` and `map4` using only `unit`, `apply`, and the `curried` method available on functions.

```scala
def map3[A,B,C,D](fa: F[A],fb: F[B],fc: F[C])
                 (f: (A, B, C) => D): F[D]
def map4[A,B,C,D,E](fa: F[A],fb: F[B],fc: F[C],fd: F[D])
                   (f: (A, B, C, D) => E): F[E]
```

…

consider a web form that requires a name, a birth date, and a phone number:

```scala
case class WebForm(name: String, birthdate: Date, phoneNumber: String)
```

…to validate an entire web form, we can simply lift the `WebForm` constructor with `map3`:

```scala
def validWebForm(name: String,
                 birthdate: String,
                 phone: String): Validation[String, WebForm] =
  map3(
    validName(name),
    validBirthdate(birthdate),
    validPhone(phone))(
    WebForm(_,_,_))
```

If any or all of the functions produce `Failure`, the whole `validWebForm` method will return all of those failures combined.

After you invoke all three of the validation functions, you get three instances of `V[_]`. If all of them indicate a successful validation, you extract the validated arguments and pass them to a function, `f`, that constructs the final validated object. In case of failure, you report errors to the client. This gives the following contract for the workflow—let's call it `apply3`

```scala
def apply3[V[_], A, B, C, D](va: V[A], vb: V[B], vc: V[C])
  (f: (A, B, C) => D)
  : V[D]
```

The input contexts

The processing function

Validated output object in the same context

You don't yet have the implementation of `apply3`. But assuming you have one, let's see how the validation code evolves out of this algebra and plugs into the smart constructor for creating a `SavingsAccount`:

```scala
def savingsAccount(no: String, name: String, rate: BigDecimal,
  openDate: Option[Date], closeDate: Option[Date],
  balance: Balance): V[Account] = {

  apply3(
    validateAccountNo(no),
    validateOpenCloseDate(openDate, closeDate),
    validateRate(rate)
  ) { (n, d, r) =>
      SavingsAccount(n, name, r, d._1, d._2, balance)
  }
}
```

The three contexts of validation

The function that extracts information from the contexts and constructs a valid SavingsAccount

validateOpenCloseDate returns a Tuple2, and you access the two members using d._1 and d._2.

Indeed, **Apply** is so useful that it has **special syntax**:

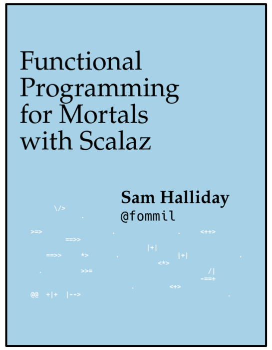
```
implicit class ApplyOps[F[_]: Apply, A](self: F[A]) {
  def *>[B](fb: F[B]): F[B] = Apply[F].apply2(self,fb)((_,b) => b)
  def <*[B](fb: F[B]): F[A] = Apply[F].apply2(self,fb)((a,_) => a)
  def |@|[B](fb: F[B]): ApplicativeBuilder[F, A, B] = ...
}
...
```

The syntax <* and *> (left bird and right bird) offer a convenient way to ignore the output from one of two parallel effects.

Right now we are interested in *>. We'll look at |@| a bit later.

@philip_schwarz

Sam Halliday @fommil

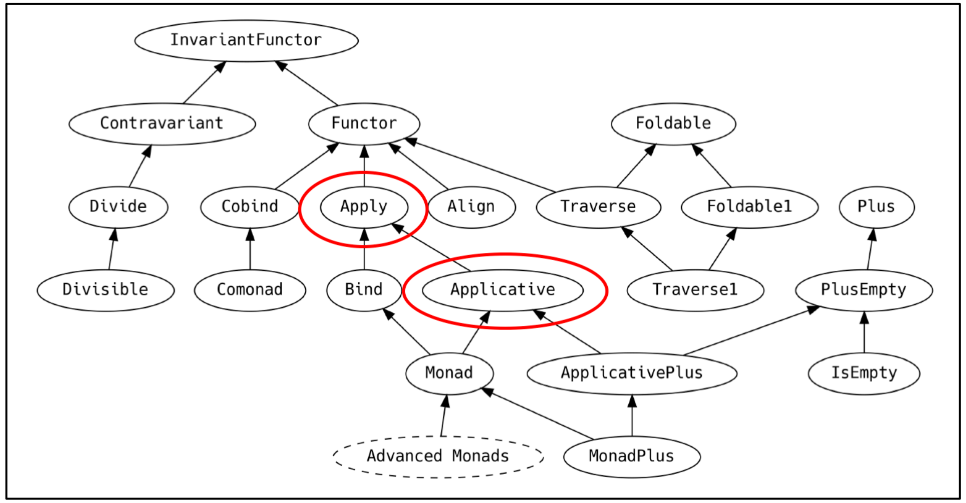


5.7 Applicative and Monad

From a functionality point of view, Applicative is Apply with a pure method and **Monad** extends **Applicative** with **Bind**.

```
@typeclass trait Applicative[F[_]] extends Apply[F] {
  def point[A](a: =>A): F[A]
  def pure[A](a: =>A): F[A] = point(a)
}

@typeclass trait Monad[F[_]] extends Applicative[F] with Bind[F]
```

In many ways, **Applicative** and **Monad** are the culmination of everything we've seen in this chapter. **pure** (or **point** as it is more commonly known for data structures) **allows us to create effects or data structures from values**.
Instances of **Applicative** must meet some laws, effectively asserting that all the methods are consistent: …



So **Scalaz** can automatically supply **Applicative** instances which provide the **<*>** and ***>** operators that we need for the **Scala** username and password validation program.

@philip_schwarz

Remember how in **Part 2** we looked at the behaviour of **<*>** and ***>** in **Haskell**'s **Either Applicative**?

```
*Main> Right((+) 1) <*> Right(2)
Right 3

*Main> Right((+) 1) <*> Left("bang")
Left "bang"

*Main> Left("boom") <*> Right(2)
Left "boom"

*Main> Left("boom") <*> Left("bang")
Left "boom"
```

```
*Main> Right(2) *> Right(3)
Right 3

*Main> Left("boom") *> Right(3)
Left "boom"

*Main> Right(2) *> Left("bang")
Left "bang"

*Main> Left("boom") *> Left("bang")
Left "boom"
```

Let's do the same using the **Scalaz Apply** instance for **Scalaz**'s **Disjunction** (**Either**)

```
scala> import scalaz._, Scalaz._
import scalaz._
import Scalaz._

scala> val inc: Int => Int = _ + 1
inc: Int => Int = $$Lambda$4315/304552448@725936c2
```

```
scala> 2.right[String]  <*> inc.right
res1: String \/ Int = \/-(3)

scala> 2.right[String]  <*> "bang".left
res2: String \/ Nothing = -\/(bang)

scala> "boom".left[Int] <*> inc.right
res3: String \/ Int = -\/(boom)

scala> "boom".left[Int] <*> "bang".left
res4: String \/ Nothing = -\/(bang)
```

```
scala> 2.right[String]  *> 3.right
res5: String \/ Int = \/-(3)

scala> "boom".left[Int] *> 3.right
res6: String \/ Int = -\/(boom)

scala> 2.right[String]  *> "bang".left
res7: String \/ Nothing = -\/(bang)

scala> "boom".left[Int] *> "bang".left
res8: String \/ Nothing = -\/(boom)
```

No surprises here: we see exactly the same behaviour using **Scalaz** as we see using **Haskell**.

```
assert( (2.right[String]  <*> inc.right   ) == 3.right.    )
assert( (2.right[String]  <*> "bang".left) == "bang".left)
assert( ("boom".left[Int] <*> inc.right   ) == "boom".left)
assert( ("boom".left[Int] <*> "bang".left) == "bang".left)
```

```
assert( (2.right[String]  *> 3.right     ) == 3.right.         )
assert( ("boom".left[Int] *> 3.right     ) == "boom".left       )
assert( (2.right[String]  *> "bang".left) == "bang".left.      )
assert( ("boom".left[Int] *> "bang".left) == "boom".left[Int])
```

So **Scalaz** provides the **Applicative typeclass**. What else do we need in our username and password validation program?

We need the **Validation** abstraction plus an **Applicative** instance for **Validation** instances whose error type is a **Semigroup**.

And sure enough, **Scalaz** provides that.

```
data Validation err a = Failure err | Success a

instance Semigroup err => Applicative (Validation err)
```

```
/**
 * Represents either:
 *  - Success(a), or
 *  - Failure(e).
 *
 * Isomorphic to scala.Either and scalaz.\/. The motivation for a Validation is to provide the
instance
 * Applicative[[a]Validation[E, a]] that accumulate failures through a scalaz.Semigroup[E].
 *
 ...
 ...
 *
 * @tparam E The type of the Failure
 * @tparam A The type of the Success
 */
sealed abstract class Validation[E, A] extends Product with Serializable {
```

@philip_schwarz

Remember how in Part 2 we looked at the behaviour of <*> and *> in Haskell's Validation Applicative, with Success being a number and Failure being a string?

```
*Main> Success((+) 1) <*> Success(2)
Success 3

*Main> Success((+) 1) <*> Failure("bang")
Failure "bang"

*Main> Failure("boom") <*> Success(2)
Failure "boom"

*Main> Failure("boom") <*> Failure("bang")
Failure "boombang"
```

```
*Main> Success(2) *> Success(3)
Success 3

*Main> Failure("boom") *> Success(3)
Failure "boom"

*Main> Success(2) *> Failure("bang")
Failure "bang"

*Main> Failure("boom") *> Failure("bang")
Failure "boombang"
```

Let's try out the **Scalaz** Applicative for Validation[String, Int]

Scalaz can automatically make available the Applicative because it can automatically make available a Semigroup instance for our error type, which is String.

```
scala> 2.success[String] <*> inc.success
res0: scalaz.Validation[String,Int] = Success(3)

scala> "boom".failure[Int] <*> inc.success[String]
res1: scalaz.Validation[String,Int] = Failure(boom)

scala> 2.success[String] <*> "bang".failure[Int=>Int]
res1: scalaz.Validation[String,Int] = Failure(bang)

scala> "boom".failure[Int] <*> "bang".failure[Int=>Int]
res2: scalaz.Validation[String,Int] = Failure(bangboom)
```

```
scala> 2.success[String] *> inc.success
res3: scalaz.Validation[String,Int => Int] = Success($$Lambda$4315/304552448@725936c2)

scala> "boom".failure[Int] *> inc.success[String]
res4: scalaz.Validation[String,Int => Int] = Failure(boom)

scala> 2.success[String] *> "bang".failure[Int=>Int]
res5: scalaz.Validation[String,Int => Int] = Failure(bang)

scala> "boom".failure[Int] *> "bang".failure[Int=>Int]
res6: scalaz.Validation[String,Int => Int] = Failure(boombang)
```

The only surprise here is the different results we get when call <*> with two failures. In Haskell we get "boombang" but in Scala we get "bangboom"

```
assert( (2.success[String]   <*> inc.success                ) == 3.success)
assert( ("boom".failure[Int] <*> inc.success[String]        ) == "boom".failure)
assert( (2.success[String]   <*> "bang".failure[Int=>Int]) == "bang".failure)
assert( ("boom".failure[Int] <*> "bang".failure[Int=>Int]) == "bangboom".failure)

assert( (2.success[String]   *> inc.success                ) == inc.success)
assert( ("boom".failure[Int] *> inc.success[String]        ) == "boom".failure)
assert( (2.success[String]   *> "bang".failure[Int=>Int] ) == "bang".failure)
assert( ("boom".failure[Int] *> "bang".failure[Int=>Int] ) == "boombang".failure)
```

OK, so we looked at **Validation**[String, Int] in **Scalaz**, but the error in our validation program, rather than being a String, is an **Error** type containing a list of strings.

```haskell
newtype Error = Error [String]

Validation Error Username
```

```scala
case class Error(error:List[String])

Validation[Error, Username]
```

We could do away with the **Error** type and just use `List[String]` as an error. Let's try out **Validation**`[List[String], Int]`

```scala
assert( (2.success[List[String]]   <*> inc.success                 ) == 3.success)
assert( (List("boom").failure[Int] <*> inc.success                 ) == List("boom").failure)
assert( (2.success[List[String]]   <*> List("bang").failure[Int=>Int] ) == List("bang").failure)
assert( (List("boom").failure[Int] <*> List("bang").failure[Int=>Int] ) == List("bang","boom").failure)

assert( (2.success[List[String]]    *> inc.success                 ) == inc.success)
assert( (List("boom").failure[Int]  *> inc.success.                ) == List("boom").failure)
assert( (2.success[List[String]]    *> List("bang").failure[Int=>Int] ) == List("bang").failure)
assert( (List("boom").failure[Int]  *> List("bang").failure[Int=>Int] ) == List("boom","bang").failure)
```

Again, **Scalaz** can automatically make available the **Applicative** because it can automatically make available a **Semigroup** instance for our error type, which is `List[String]`.

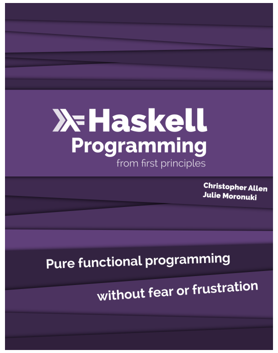
But strictly speaking, `List[String]`, is not a perfect fit for our error type because while an empty list is a `List[String]`, our list of error messages can never be empty: if we have an error situation then the list will contain at least one error. What we are looking for is the notion of **non-empty list**. Remember how in **Part 2** we saw that **Haskell** has this notion?

NonEmpty, a useful datatype

One useful datatype that can't have a **Monoid** instance but does have a **Semigroup** instance is the **NonEmpty** list type. It is a list datatype that can never be an empty list…

We can't write a **Monoid** for **NonEmpty** because it has no **identity** value by design! There is no empty list to serve as an **identity** for any operation over a **NonEmpty** list, yet there is still a **binary associative operation**: two **NonEmpty** lists can still be **concatenated**.

A type with a canonical **binary associative operation** but no **identity** value is a natural fit for **Semigroup**.



@bitemyapp
@argumatronic

Scalaz also has the notion of a **non-empty list**. It is called **NonEmptyList** and there is a mention of it in the Scaladoc for **Validation**, in the highlighted section below (which I omitted earlier)

```
/**
 * Represents either:
 *  - Success(a), or
 *  - Failure(e).
 *
 * Isomorphic to scala.Either and scalaz.\/. The motivation for a Validation is to provide the instance
 * Applicative[[a]Validation[E, a]] that accumulate failures through a scalaz.Semigroup[E].
 *
 * [[scalaz.NonEmptyList]] is commonly chosen as a type constructor for the type E. As a convenience,
 * an alias scalaz.ValidationNel[E] is provided as a shorthand for scalaz.Validation[NonEmptyList[E]],
 * and a method Validation#toValidationNel converts Validation[E] to ValidationNel[E].
 ...
 ...
 *
 * @tparam E The type of the Failure
 * @tparam A The type of the Success
 */
sealed abstract class Validation[E, A] extends Product with Serializable {
```

And here are the **Scalaz** docs for **NonEmptyList** and **ValidationNel**

```
/** A singly-linked list that is guaranteed to be non-empty. */
final class NonEmptyList[A] private[scalaz](val head: A, val tail: IList[A]) {
```

```
/**
 * An [[scalaz.Validation]] with a [[scalaz.NonEmptyList]] as the failure type.
 *
 * Useful for accumulating errors through the corresponding [[scalaz.Applicative]] instance.
 */
type ValidationNel[E, +X] = Validation[NonEmptyList[E], X]
```

@philip_schwarz

Tanks to the ValidationNel alias, we can succinctly define our error as ValidationNel[String, Int] rather than as Validation[NonEmptyList[String], Int].

Let's see an example of ValidationNel[String, Int] in action

```
scala> 2.successNel[String] <*> inc.successNel
res0: scalaz.ValidationNel[String,Int] = Success(3)

scala> List("boom").failureNel[Int] <*> inc.successNel
res1: scalaz.ValidationNel[List[String],Int] = Failure(NonEmpty[List(boom)])

scala> 2.successNel[String] <*> "bang".failureNel[Int=>Int]
res2: scalaz.ValidationNel[String,Int] = Failure(NonEmpty[bang])

scala> "boom".failureNel[Int] <*> "bang".failureNel[Int=>Int]
res3: scalaz.ValidationNel[String,Int] = Failure(NonEmpty[bang,boom])
```

```
scala> 2.successNel[String] *> inc.successNel
res4: scalaz.ValidationNel[String,Int => Int] = Success($$Lambda$4315/304552448@725936c2)

scala> List("boom").failureNel[Int] *> inc.successNel
res5: scalaz.ValidationNel[List[String],Int => Int] = Failure(NonEmpty[List(boom)])

scala> 2.successNel[String] *> "bang".failureNel[Int=>Int]
res6: scalaz.ValidationNel[String,Int => Int] = Failure(NonEmpty[bang])

scala> "boom".failureNel[Int] *> "bang".failureNel[Int=>Int]
res7: scalaz.ValidationNel[String,Int => Int] = Failure(NonEmpty[boom,bang])
```

The only surprise here is the different ordering of error messages that we get when we use <*> and *> with two failures

```
assert( (2.successNel[String]         <*> inc.successNel                ) == 3.successNel                          )
assert( (List("boom").failureNel[Int] <*> inc.successNel                ) == List("boom").failureNel               )
assert( (2.successNel[String]         <*> "bang".failureNel[Int=>Int]   ) == "bang".failureNel                     )
assert( ("boom".failureNel[Int]       <*> "bang".failureNel[Int=>Int]   ) == Failure(NonEmptyList("bang","boom")))

assert( (2.successNel[String]          *> inc.successNel                ) == inc.successNel                        )
assert( (List("boom").failureNel[Int]  *> inc.successNel                ) == List("boom").failureNel               )
assert( (2.successNel[String]          *> "bang".failureNel[Int=>Int]   ) == "bang".failureNel                     )
assert( ("boom".failureNel[Int]        *> "bang".failureNel[Int=>Int]   ) == Failure(NonEmptyList("boom","bang")))
```

In **Part 2** we got the **Scala** validation program to do **I/O** using the **Cats IO Monad**.

What shall we do now that we are using **Scalaz**?

In June 2018, the then upcoming **Scalaz 8 effect system**, containing an **IO Monad**, was pulled out of **Scalaz 8** and made into a standalone project called **ZIO** (see http://degoes.net/articles/zio-solo).

Now that we are using **Scalaz**, let's do **I/O** using ZIO rather than **Cats**.

Here is how the **IO** Data Type is introduced on the **ZIO** site

zio.dev/docs/datatypes/datatypes_io

Overview Data Types

IO

A value of type `IO[E, A]` describes an effect that may fail with an `E`, run forever, or produce a single `A`.

`IO` values are immutable, and all `IO` functions produce new `IO` values, enabling `IO` to be reasoned about and used like any ordinary Scala immutable data structure.

`IO` values do not actually *do* anything; they are just values that *model* or *describe* effectful interactions.

`IO` can be *interpreted* by the ZIO runtime system into effectful interactions with the external world. Ideally, this occurs at a single time, in your application's `main` function. The `App` class provides this functionality automatically.

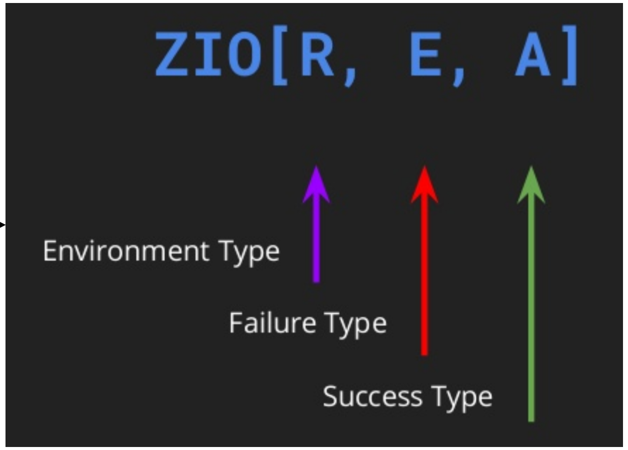
`IO[E,A]` however, is just an alias for `ZIO[Any,E,A]`

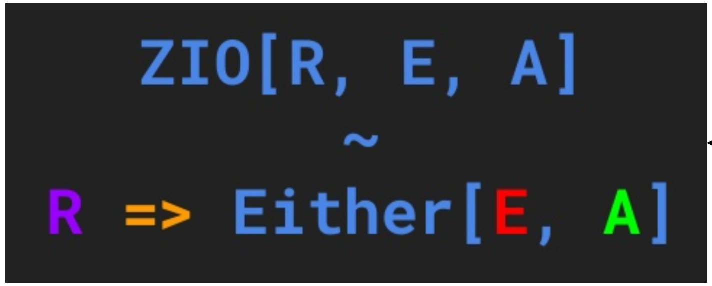
```
type IO[+E, +A] = ZIO[Any, E, A]
```

where **ZIO** is defined as follows:

@jdegoes
John A De Goes

```
/**
 * A ZIO[R, E, A] ("Zee-Oh of Are Eeh Aye") is an immutable data structure
 * that models an effectful program. The effect requires an environment R,
 * and the effect may fail with an error E or produce a single A.
 *
 ...
 */
sealed trait ZIO[-R, +E, +A] extends Serializable { self =>
 ...
```





For our validation program, the environment will be the **console**, the failure type will be **IOException**, and the Success type will be **Unit**.

`ZIO[Console, IOException, Unit]`

Requires `Console` Might Fail with `IOException` Might Succeed with `Unit` zio.dev/docs/getting_started.html

Your application can extend `App`, which provides a complete runtime system and allows you to write your whole program using ZIO:

```scala
import zio.App
import zio.console._

object MyApp extends App {

  def run(args: List[String]) =
    myAppLogic.fold(_ => 1, _ => 0)

  val myAppLogic =
    for {
      _    <- putStrLn("Hello! What is your name?")
      name <- getStrLn
      _    <- putStrLn(s"Hello, ${name}, welcome to ZIO!")
    } yield ()
}
```

`run` should return a ZIO value which has all its errors handled, which, in ZIO parlance, is an unexceptional ZIO value.

One way to do that, is to invoke `fold` over a ZIO value, to get an unexceptional ZIO value. That requires two handler functions: `eh: E => B` and `ah: A => B`. If myAppLogic fails, `eh` will be used to get from `e: E` to `b: B`; if it succeeds, `ah` will be used to get from `a: A` to `b: B`.

`myAppLogic`, as folded above, produces an unexceptional ZIO value, with `B` being `Int`. If `myAppLogic` fails, there will be a 1; if it succeeds, there will be a 0.

Here is ZIO's hello world program.

We'll use the same approach in the validation program.

The type of `myAppLogic` is `ZIO[Console, IOException, Unit]`, where `Console` provides `putStrLn` and `getStrLn`.

`ZIO[Console, IOException, Unit]`

Requires `Console`

Might Fail with `IOException`

Might Succeed with `Unit`

@philip_schwarz

OK, so we are finally ready to see a new version of the Scala validation program using **Scalaz** and ZIO.

In the next four slides we'll look at how the new version differs from the existing one.

Then in the subsequent slides, we'll look at the new version side by side with the Haskell version.

Using Scalaz instead of plain Scala

Because **Scalaz** provides all the abstractions we need, we can simply delete all the code on this slide!

```scala
trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A => B): F[B]
}

trait Semigroup[A] {
  def <>(lhs: A, rhs: A): A
}

implicit val errorSemigroup: Semigroup[Error] =
  new Semigroup[Error] {
    def <>(lhs: Error, rhs: Error): Error =
      Error(lhs.error ++ rhs.error)
  }

trait Applicative[F[_]] extends Functor[F] {
  def <*>[A,B](fab: F[A => B],fa: F[A]): F[B]
  def *>[A,B](fa: F[A],fb: F[B]): F[B]
  def unit[A](a: => A): F[A]
  def map[A,B](fa: F[A])(f: A => B): F[B] =
    <*>(unit(f),fa)
}

sealed trait Validation[+E, +A]
case class Failure[E](error: E) extends Validation[E, Nothing]
case class Success[A](a: A) extends Validation[Nothing, A]
```

```scala
def validationApplicative[E](implicit sg:Semigroup[E]):
  Applicative[λ[α => Validation[E,α]]] =

  new Applicative[λ[α => Validation[E,α]]] {

    def unit[A](a: => A) = Success(a)

    def <*>[A,B](fab: Validation[E,A => B], fa: Validation[E,A]): Validation[E,B] =
      (fab, fa) match {
        case (Success(ab), Success(a)) => Success(ab(a))
        case (Failure(err1), Failure(err2)) => Failure(sg.<>(err1,err2))
        case (Failure(err), _) => Failure(err)
        case (_, Failure(err)) => Failure(err)
      }

    def *>[A,B](fa: Validation[E,A], fb: Validation[E,B]): Validation[E,B] =
      (fa, fb) match {
        case (Failure(err1), Failure(err2)) => Failure(sg.<>(err1,err2))
        case _ => fb
      }
  }


val errorValidationApplicative = validationApplicative[Error]
import errorValidationApplicative._
```

Using plain Scala

The only changes are that **Error** is no longer needed and that rather than using our own **Validation**, we are using **Scalaz's ValidationNel** and creating instances of it by calling **failureNel** and **successNel**.

Using Scalaz

Using plain Scala

```scala
case class Username(username: String)   case class Password(password:String)

case class User(username: Username, password: Password)

case class Error(error:List[String])
```

```scala
def checkUsernameLength(username: String): Validation[Error, Username] =
  username.length > 15 match {
    case true  => Failure(Error(List("Your username cannot be " +
                                     "longer than 15 characters.")))
    case false => Success(Username(username))
  }
```

```scala
def checkPasswordLength(password: String): Validation[Error, Password] =
  password.length > 20 match {
    case true  => Failure(Error(List("Your password cannot be " +
                                     "longer than 20 characters.")))
    case false => Success(Password(password))
  }
```

```scala
def requireAlphaNum(password: String): Validation[Error, String] =
  password.forall(_.isLetterOrDigit) match {
    case false  => Failure(Error(List("Cannot contain white " +
                                      "space or special characters.")))
    case true   => Success(password)
  }
```

```scala
def cleanWhitespace(password:String): Validation[Error, String] =
  password.dropWhile(_.isWhitespace) match {
    case pwd if pwd.isEmpty => Failure(Error(List("Cannot be empty.")))
    case pwd                => Success(pwd)
  }
```

Using Scalaz

```scala
case class Username(username: String)   case class Password(password:String)

case class User(username: Username, password: Password)
```

```scala
def checkUsernameLength(username: String): ValidationNel[String, Username] =
  username.length > 15 match {
    case true =>   "Your username cannot be longer " +
                   "than 15 characters.".failureNel
    case false => Username(username).successNel
  }
```

```scala
def checkPasswordLength(password: String): ValidationNel[String, Password] =
  password.length > 20 match {
    case true  => "Your password cannot be longer "  +
                  "than 20 characters.".failureNel
    case false => Password(password).successNel
  }
```

```scala
def requireAlphaNum(password: String): ValidationNel[String, String] =
  password.forall(_.isLetterOrDigit) match {
    case false  => "Cannot contain white space "  +
                   "or special characters.".failureNel
    case true   => password.successNel
  }
```

```scala
def cleanWhitespace(password:String): ValidationNel[String, String] =
  password.dropWhile(_.isWhitespace) match {
    case pwd if pwd.isEmpty => "Cannot be empty.".failureNel
    case pwd                => pwd.successNel
  }
```

Using plain Scala

@philip_schwarz

Here the changes are not using **Error**, using **ValidationNel** instead of **Validation**, plus some small changes in how <*> and *> are invoked.

Using Scalaz

```scala
def validateUsername(username: Username): Validation[Error, Username] =
  username match {
    case Username(username) =>
       cleanWhitespace(username) match {
        case Failure(err)       => Failure(err)
        case Success(username2) =>
          *>(requireAlphaNum(username2),checkUsernameLength(username2))
      }
  }
```

```scala
def validatePassword(password: Password): Validation[Error, Password] =
  password match {
    case Password(pwd) =>
      cleanWhitespace(pwd) match {
        case Failure(err)  => Failure(err)
        case Success(pwd2) =>
          *>(requireAlphaNum(pwd2),checkPasswordLength(pwd2))
      }
  }
```

```scala
def makeUser(name:Username,password:Password):Validation[Error, User] =
  <*>(map(validateUsername(name))(User.curried),validatePassword(password))
```

```scala
def validateUsername(username: Username): ValidationNel[String, Username] =
  username match {
    case Username(username) =>
       cleanWhitespace(username) match {
        case Failure(err) => Failure(err)
        case Success(username2) =>
          requireAlphaNum(username2) *> checkUsernameLength(username2)
      }
  }
```

```scala
def validatePassword(password: Password): ValidationNel[String, Password] =
  password match {
    case Password(pwd) =>
      cleanWhitespace(pwd) match {
        case Failure(err) => Failure(err)
        case Success(pwd2) =>
          requireAlphaNum(pwd2) *> checkPasswordLength(pwd2)
      }
  }
```

```scala
def makeUser(name:Username,password:Password):ValidationNel[String, User] =
  validatePassword(password) <*> (validateUsername(name) map User.curried)
```

Not much at all to say about differences in the way **I/O** is done

Using Cats

Using ZIO

```scala
import cats.effect.IO

object MyApp extends App {

def getLine = IO { scala.io.StdIn.readLine }
def print(s: String): IO[Unit] = IO { scala.Predef.print(s) }

val main =
  for {
    _   <- print("Please enter a username.\n")
    usr <- getLine map Username
    _   <- print("Please enter a password.\n")
    pwd <- getLine map Password
    _   <- print(makeUser(usr,pwd).toString)
  } yield ()
}
```

```scala
import zio.App, zio.console.{getStrLn, putStrLn}

object MyApp extends App {

  def run(args: List[String]) =
    appLogic.fold(_ => 1, _ => 0)

  val appLogic =
    for {
      _   <- putStrLn("Please enter a username.\n")
      usr <- getStrLn map Username
      _   <- putStrLn("Please enter a password.\n")
      pwd <- getStrLn map Password
      _   <- putStrLn(makeUser(usr,pwd).toString)
    } yield ()
}
```

Now let's see the whole of the new version of the **Scala** program side by side with the **Haskell** program

@philip_schwarz

```haskell
newtype Username = Username String
  deriving Show
```

```haskell
newtype Password = Password String
  deriving Show
```

```haskell
data User = User Username Password
  deriving Show
```

```haskell
newtype Error = Error [String]
  deriving Show
```

```haskell
checkUsernameLength :: String -> Validation Error Username
checkUsernameLength username =
   case (length username > 15) of
      True  -> Failure (Error "Your username cannot be \
                              \longer than 15 characters.")
      False -> Success (Username username)
```

```haskell
checkPasswordLength :: String -> Validation Error Password
checkPasswordLength password =
   case (length password > 20) of
      True  -> Failure (Error "Your password cannot be \
                              \longer than 20 characters.")
      False -> Success (Password password)
```

```haskell
requireAlphaNum :: String -> Validation Error String
requireAlphaNum input =
   case (all isAlphaNum input) of
      False -> Failure "Your password cannot contain \
                       \white space or special characters."
      True  -> Success input
```

```haskell
cleanWhitespace :: String -> Validation Error String
cleanWhitespace "" = Failure (Error "Your password cannot be empty.")
cleanWhitespace (x : xs) =
   case (isSpace x) of
      True  -> cleanWhitespace xs
      False -> Success (x : xs)
```

Haskell

```scala
case class Username(username: String)   case class Password(password:String)

case class User(username: Username, password: Password)
```

```scala
def checkUsernameLength(username: String): ValidationNel[String, Username] =
  username.length > 15 match {
    case true =>    "Your username cannot be longer " +
                    "than 15 characters.".failureNel
    case false => Username(username).successNel
  }
```

```scala
def checkPasswordLength(password: String): ValidationNel[String, Password] =
  password.length > 20 match {
    case true  => "Your password cannot be longer "  +
                  "than 20 characters.".failureNel
    case false => Password(password).successNel
  }
```

```scala
def requireAlphaNum(password: String): ValidationNel[String, String] =
  password.forall(_.isLetterOrDigit) match {
    case false  => "Cannot contain white space "  +
                   "or special characters.".failureNel
    case true   => password.successNel
  }
```

```scala
def cleanWhitespace(password:String): ValidationNel[String, String] =
  password.dropWhile(_.isWhitespace) match {
    case pwd if pwd.isEmpty => "Cannot be empty.".failureNel
    case pwd                => pwd.successNel
  }
```

Scala

```haskell
validateUsername :: Username -> Validation Error Username
validateUsername (Username username) =
  case (cleanWhitespace username) of
    Failure err       -> Failure err
    Success username2 -> requireAlphaNum username2
                         *> checkUsernameLength username2
```

```scala
def validateUsername(username: Username): ValidationNel[String,Username] = username match
{
  case Username(username) =>
     cleanWhitespace(username) match {
      case Failure(err) => Failure(err)
      case Success(username2) =>
        requireAlphaNum(username2) *> checkUsernameLength(username2)
    }
}
```

```haskell
validatePassword :: Password -> Validation Error Password
validatePassword (Password password) =
  case (cleanWhitespace password) of
    Failure err       -> Failure err
    Success password2 -> requireAlphaNum password2
                         *> checkPasswordLength password2
```

```scala
def validatePassword(password: Password): ValidationNel[String, Password] = password match
{
  case Password(pwd) =>
    cleanWhitespace(pwd) match {
      case Failure(err) => Failure(err)
      case Success(pwd2) => requireAlphaNum(pwd2) *> checkPasswordLength(pwd2)
    }
}
```

```haskell
makeUser :: Username -> Password -> Validation Error User
makeUser name password =
User <$> validateUsername name
     <*> validatePassword password
```

Haskell

```scala
def makeUser(name: Username, password: Password): ValidationNel[String, User] =
  validatePassword(password) <*> (validateUsername(name) map User.curried)
```

Scala

```haskell
main :: IO ()
main =
  do
    putStr "Please enter a username.\n> "
    username <- Username <$> getLine
    putStr "Please enter a password.\n> "
    password <- Password <$> getLine
    print (makeUser username password)
```

```scala
object MyApp extends App {
  def run(args: List[String]) = appLogic.fold(_ => 1, _ => 0)

  val appLogic =    for {
      _   <- putStrLn("Please enter a username.\n")
      usr <- getStrLn map Username
      _   <- putStrLn("Please enter a password.\n")
      pwd <- getStrLn map Password
      _   <- putStrLn(makeUser(usr,pwd).toString)
    } yield ()
}
```

```scala
import zio.App
import zio.console.{
  getStrLn,
  putStrLn
}
```

As you can see, the **Scala** and **Haskell** programs now are very similar and pretty much the same size.

Now let's go back to **the second main use of the applyX functions** that we saw in **Scalaz**'s **Apply typeclass** on slide 2, i.e. **performing effects in parallel**.

@philip_schwarz

In **FP**, an **algebra** takes the place of an interface in Java… This is the layer where we define all **side-effecting interactions** of our system.

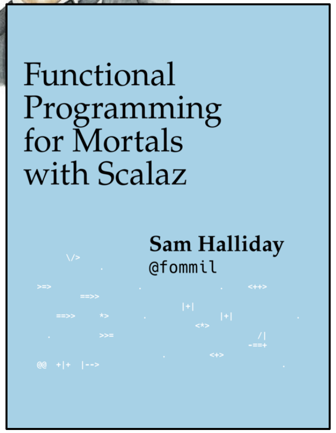
@fommil

```scala
package algebra

import scalaz.NonEmptyList

trait Drone[F[_]] {
  def getBacklog: F[Int]
  def getAgents: F[Int]
}

trait Machines[F[_]] {
  def getTime: F[Epoch]
  def getManaged: F[NonEmptyList[MachineNode]]
  def getAlive: F[Map[MachineNode, Epoch]]
  def start(node: MachineNode): F[MachineNode]
  def stop(node: MachineNode): F[MachineNode]
}
```

Algebra of **Drones** and **Machines**

There is **tight iteration** between writing the **business logic** and the **algebra**: it is a good level of abstraction to design a system.

business logic that defines the application's behaviour




```scala
package logic

import algebra._
import scalaz._
import Scalaz._

final case class WorldView(
  backlog: Int,
  agents: Int,
  managed: NonEmptyList[MachineNode],
  alive: Map[MachineNode, Epoch],
  pending: Map[MachineNode, Epoch],
  time: Epoch
)

trait DynAgents[F[_]] {
  def initial: F[WorldView]
  def update(old: WorldView): F[WorldView]
  def act(world: WorldView): F[WorldView]
}

final class DynAgentsModule[F[_]: Monad](D: Drone[F], M: Machines[F])
  extends DynAgents[F] {

  def initial: F[WorldView] = for {
    db <- D.getBacklog
    da <- D.getAgents
    mm <- M.getManaged
    ma <- M.getAlive
    mt <- M.getTime
  } yield WorldView(db, da, mm, ma, Map.empty, mt)

  def update(old: WorldView): F[WorldView] = ...
  def act(world: WorldView): F[WorldView] = ...
}
```

WorldView aggregates the return values of all the methods in the **algebras**, and adds a pending field.

Our **business logic** runs in an infinite loop (pseudocode)

```
state = initial()
while True:
  state = update(state)
  state = act(state)
```

A **module** to contain our main **business logic**. A **module** is pure and depends only on other **modules**, **algebras** and **pure functions**.

It indicates that we depend on **Drone** and **Machines**.

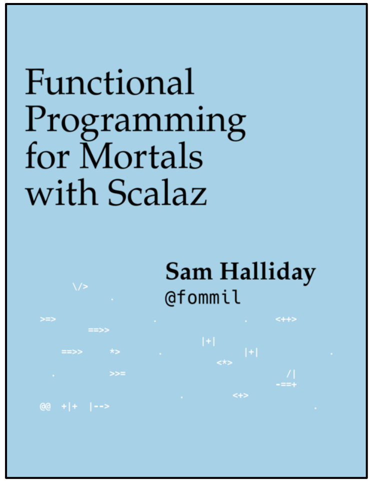
On the next slide we look in more detail at the **module** and its **initial** function.

```scala
final class DynAgentsModule[F[_]: Monad](D: Drone[F], M: Machines[F]) extends DynAgents[F] {

  def initial: F[WorldView] = for {
    db <- D.getBacklog
    da <- D.getAgents
    mm <- M.getManaged
    ma <- M.getAlive
    mt <- M.getTime
  } yield WorldView(db, da, mm, ma, Map.empty, mt)

  def update(old: WorldView): F[WorldView] = ...
  def act(world: WorldView): F[WorldView] = ...
}
```

The implicit `Monad[F]` means that `F` is **monadic**, allowing us to use **map**, **pure** and, of course, **flatMap** via for comprehensions.

We have access to the **algebra** of `Drone` and `Machines` as `D` and `M`, respectively.

Using a single capital letter name is a common naming convention for **monad** and **algebra** implementations.



`flatMap` (i.e. when we use the `<-` generator) **allows us to operate on a value that is computed at runtime**. **When we return an F[_] we are returning another program to be interpreted at runtime, that we can then `flatMap`**.

This is how we safely chain together sequential side-effecting code, whilst being able to provide a pure implementation for tests. **FP** could be described as **Extreme Mocking**.

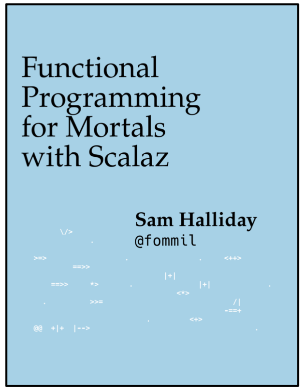
@fommil

The application that we have designed runs each of its **algebraic** methods **sequentially** (pseudocode)

```
state = initial()
while True:
  state = update(state)
  state = act(state)
```

But there are some obvious places where work can be performed in parallel. In our definition of initial we could ask for all the information we need at the same time instead of one query at a time.

As opposed to `flatMap` for sequential operations, Scalaz uses `Apply` syntax for parallel operations



Sam Halliday is about to show us how to do this switch from **sequential monadic `flatMap` operations** to **applicative parallel operations**, but he will be using the special **Apply syntax** that he has just mentioned.

Before he does that, in order to better understand the switch, we will first carry it out without using any **syntactic sugar**. Remember **Apply**'s **applyX** functions (**apply2**, **apply3**, etc) for applying a function **f** with **N** parameters to **N** argument values, each in a context **F**, and produce a result, also in a context **F** (see slide 4)?

Since **F** is a **Monad** and every **Monad** is also an **Applicative**, instead of obtaining the **N** arguments for function **f** by chaining the **F**s together sequentially using **flatMap**, we can have the **F**s computed in parallel and passed to **applyN**, which then obtains from them the arguments for function **f** and invokes the latter.

```
def apply5[A, B, C, D, E, R]
(
  fa: => F[A],
  fb: => F[B],
  fc: => F[C],
  fd: => F[D],
  fe: => F[E]
)(f: (A, B, C, D, E) => R): F[R]
```

```
final class DynAgentsModule[F[_]: Monad](D: Drone[F], M: Machines[F]) extends DynAgents[F] {
  def initial: F[WorldView] =
    for {
      db <- D.getBacklog
      da <- D.getAgents
      mm <- M.getManaged
      ma <- M.getAlive
      mt <- M.getTime
    } yield WorldView(db, da, mm, ma, Map.empty, mt)

  def update(old: WorldView): F[WorldView] = ...
  def act(world: WorldView): F[WorldView] = ...
}
```

SWITCH

```
def initial: F[WorldView] =
  Apply[F].apply5(
    D.getBacklog,
    D.getAgents,
    M.getManaged,
    M.getAlive,
    M.getTime
  ){ case (db, da, mm, ma, mt) => WorldView(db, da, mm, ma, Map.empty, mt) }
```

Now that we have seen how to use **applyX** to switch from **sequential** to **parallel** computation of **effects**, let's see how **Sam Halliday** did the same switch, but right from the start, used more convenient **Apply** **syntax**.

@fommil

In our definition of initial we could ask for all the information we need at the same time instead of one query at a time.

As opposed to **flatMap** for sequential operations, **scalaz** uses **Apply** syntax for **parallel operations**:

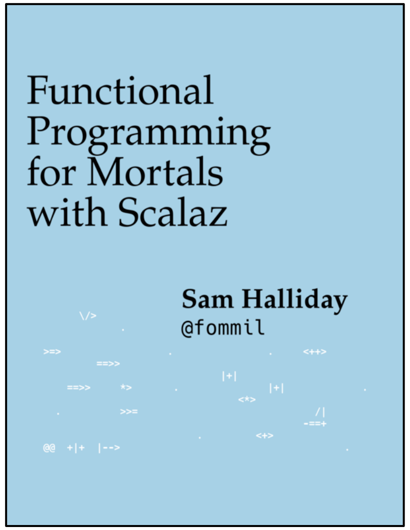
```
^^^^(D.getBacklog, D.getAgents, M.getManaged, M.getAlive, M.getTime)
```

which can also use infix notation:

```
(D.getBacklog |@| D.getAgents |@| M.getManaged |@| M.getAlive |@| M.getTime)
```

If each of the parallel operations returns a value in the same **monadic context**, we can apply a function to the results when they all return. Rewriting update to take advantage of this:

```
def initial: F[WorldView] =
  ^^^^(D.getBacklog, D.getAgents, M.getManaged, M.getAlive, M.getTime) {
    case (db, da, mm, ma, mt) => WorldView(db, da, mm, ma, Map.empty, mt)
  }
```



@philip_schwarz

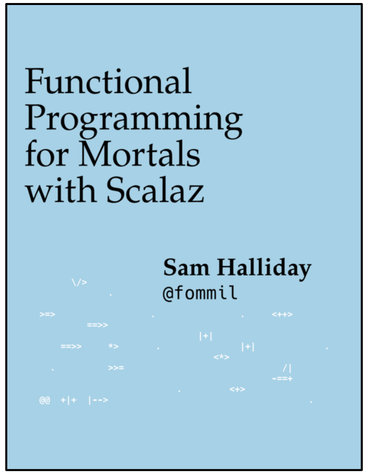
One place where the |@| syntax is mentioned is in **Scalaz's** ApplySyntax.scala

```
/** Wraps a value `self` and provides methods related to `Apply` */
final class ApplyOps[F[_],A] private[syntax](val self: F[A])(implicit val F: Apply[F]) extends Ops[F[A]] {
…
/**
 * DSL for constructing Applicative expressions.
 *
 * (f1 |@| f2 |@| ... |@| fn)((v1, v2, ... vn) => ...) is an alternative
 * to Apply[F].applyN(f1, f2, ..., fn)((v1, v2, ... vn) => ...)
 *
 * Warning: each call to |@| leads to an allocation of wrapper object. For performance sensitive code,
 *          consider using [[scalaz.Apply]]#applyN directly.
 */
final def |@|[B](fb: F[B]) = new ApplicativeBuilder[F, A, B] {
...
```

more on this warning in the next slide

@fommil

The |@| operator has many names. Some call it the *Cartesian Product Syntax*, others call it the *Cinnamon Bun*, the *Admiral Ackbar* or the *Macaulay Culkin*. We prefer to call it The *Scream operator*, after the Munch painting, because it is also the sound your CPU makes when it is parallelising All The Things.



|@|

The Scream. Edward Munch

Unfortunately, although the |@| syntax is clear, there is a problem in that a new Applicative-Builder object is allocated for each additional effect. If the work is **I/O-bound**, the memory allocation cost is insignificant. However, when performing **CPU-bound** work, use the alternative lifting with arity syntax, which does not produce any intermediate objects:

```
def ^[F[_]: Apply,A,B,C](fa: =>F[A],fb: =>F[B])(f: (A,B) =>C): F[C] = ...
def ^^[F[_]: Apply,A,B,C,D](fa: =>F[A],fb: =>F[B],fc: =>F[C])(f: (A,B,C) =>D): F[D] = ...
...
def ^^^^^^[F[_]: Apply, ...]
```

used like

```
^^^^(d.getBacklog, d.getAgents, m.getManaged, m.getAlive, m.getTime)
```

@philip_schwarz

Let's revisit the **makeUser** function in the **Scala** validation program and have a go at using **applyX**, |@| and ^.

using <*>

```
def makeUser(name: Username, password: Password): ValidationNel[String, User] =
  validatePassword(password) <*> (validateUsername(name) map User.curried)
```

```
type ValidationNelString[A] = ValidationNel[String,A]
```

using apply2

```
def makeUser(name: Username, password: Password): ValidationNel[String, User] =
  Apply[ValidationNelString].apply2(validateUsername(name), validatePassword(password))(User)
```

using |@|

```
def makeUser(name: Username, password: Password): ValidationNel[String, User] =
  ^(validateUsername(name), validatePassword(password))(User)
```

using ^

```
def makeUser(name: Username, password: Password): ValidationNel[String, User] =
  (validateUsername(name) |@| validatePassword(password))(User)
```

To conclude this slide deck, let's look at final validation example using **Scalaz** and the **|@|** syntax. The example is provided by **Debasish Ghosh** in his great book **Functional and Reactive Domain Modeling**.

Here's the basic structure of **Validation** in Scalaz:

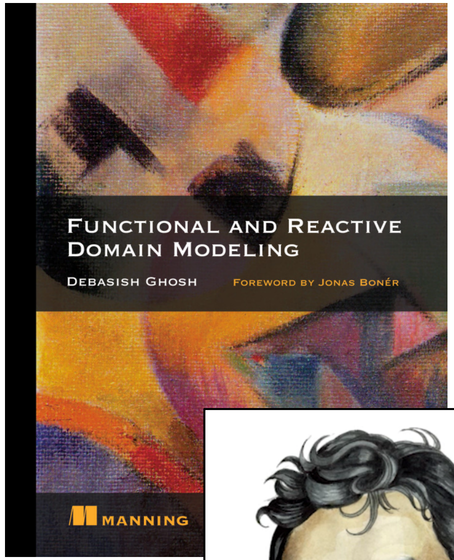
```
sealed abstract class Validation[+E, +A] { //.. }
final case class Success[A](a: A) extends Validation[Nothing, A]
final case class Failure[E](e: E) extends Validation[E, Nothing]
```

This looks awfully similar to `scala.Either[+A,+B]`, which also has two variants in **Left** and **Right**. In fact, `scalaz.Validation` is **isomorphic** to `scala.Either`[26]. In that case, **why have scalaz.Validation as a separate abstraction?** **Validation gives you the power to accumulate failures, which is a common requirement when designing domain models**.

A typical use case arises when you're validating a web-based form containing many fields and **you want to report all errors to the user at once. If you construct a Validation with a Failure type that has a Semigroup**[27], **the library provides an applicative functor for Validation, which can accumulate all errors that can come up in the course of your computation**. This also highlights an important motivation for using libraries such as **Scalaz**: You get to enjoy **more powerful functional abstractions on top of what the Scala standard library offers**. **Validation** is one of these **functional patterns** that make your code **more powerful**, **succinct**, and **free of boilerplates**.

In our discussion of **applicative functors** and the use case of validation of account attributes, we didn't talk about strategies of handling failure. But because in an **applicative effect** you get to execute all the validation functions independently, regardless of the outcome of any one of them, a useful strategy for error reporting is one that accumulates all errors and reports them at once to the user.

The question is, **should the error-handling strategy be part of the application code or can you abstract this in the pattern itself? The advantage of abstracting the error-handling strategy as part of the pattern is increased reusability and less boilerplate in application code, which are areas where FP shines**. And as I've said, **a beautiful combination of Applicative Functor and Semigroup patterns enables you to have this concern abstracted within the library itself**. When you start using this approach, you'll end up with **a library of fundamental patterns for composing code generically**. And types will play a big role in ensuring that **the abstractions are correct by construction**, and **implementation details don't leak into application-specific code**. You'll explore more of this in exercises 4.2 and 4.3 and in the other examples in the online code repository for this chapter.



Debasish Ghosh
@debasishg

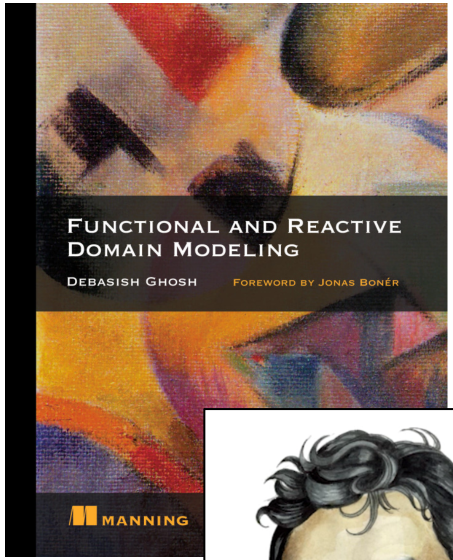
[27] A **semigroup** is a **monoid** without the **zero**.

EXERCISE 4.2 ACCUMULATING VALIDATION ERRORS (APPLICATIVELY)

Section 4.2.2 presented the Applicative Functor pattern and used it to model validations of domain entities. Consider the following function that takes a bunch of parameters and returns a fully-constructed, valid `Account` to the user:

```
def savingsAccount(
  no: String,
  name: String,
  rate: BigDecimal,
  openDate: Option[Date],
  closeDate: Option[Date],
  balance: Balance
): ValidationNel[String,Account] = { //..
```

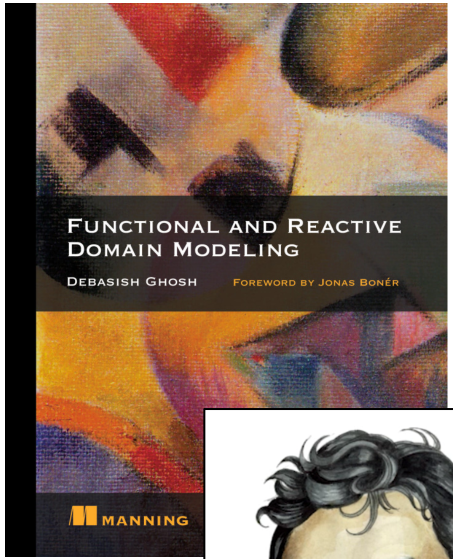
- The return type of the function is `scalaz.ValidationNel[String,Account]`, which is a shorthand for `Validation[NonEmptyList[String],Account]`. If all validations succeed, **the function returns `Success[Account]`, <u>or else it must return all the validation errors in</u> `Failure`**. This implies that **all validation functions need to run, regardless of the outcome of each of them**. This is the applicative effect.
- You need to implement the following validation rules:
  1. account numbers must have a minimum length of 10 characters,
  2. the rate of interest has to be positive, and
  3. the open date (default to today if not specified) must be before the close date.
- Hint: **Take a deep look at Scalaz's implementation of `Validation[E, A]`. Note how it provides an `Applicative` instance that supports accumulation of error messages through `Semigroup[E]`**. Note `Semigroup` is `Monoid` without a zero.



Debasish Ghosh
@debasishg

https://github.com/debasishg/frdomain/blob/master/src/main/scala/frdomain/ch4/patterns/Account.scala

```scala
/**
  * Uses Applicative instance of ValidationNEL which
  * accumulates errors using Semigroup
  */
object FailSlowApplicative {

  import scalaz._
  import syntax.apply._, syntax.std.option._, syntax.validation._

  private def validateAccountNo(no: String): ValidationNel[String, String] =
    if (no.isEmpty || no.size < 5)
        s"Account No has to be at least 5 characters long: found $no”.failureNel[String]
    else no.successNel[String]

  private def validateOpenCloseDate(od:Date,cd:Option[Date]): ValidationNel[String,String] = cd.map { c =>
      if (c before od)
        s"Close date [$c] cannot be earlier than open date [$od]”.failureNel[(Option[Date], Option[Date])]
      else (od.some, cd).successNel[String]
  }.getOrElse { (od.some, cd).successNel[String] }

  private def validateRate(rate: BigDecimal): ValidationNel[String,String] =
    if (rate <= BigDecimal(0))
      s"Interest rate $rate must be > 0”.failureNel[BigDecimal]
    else rate.successNel[String]
```



Debasish Ghosh

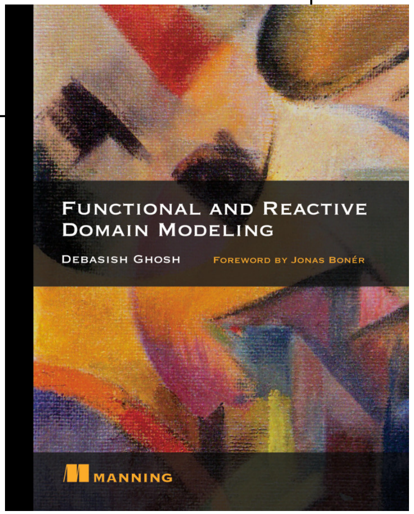
@debasishg

https://github.com/debasishg/frdomain/blob/master/src/main/scala/frdomain/ch4/patterns/Account.scala

```scala
final case class CheckingAccount(
  no: String, name: String,dateOfOpen: Option[Date], dateOfClose: Option[Date] = None, balance: Balance = Balance())
extends Account

final case class SavingsAccount(
  no:String, name:String, rateOfInterest:Amount, dateOfOpen:Option[Date], dateOfClose:Option[Date]=None, balance:Balance = Balance())
extends Account


def savingsAccount(no:String, name:String, rate:BigDecimal, openDate:Option[Date], closeDate: Option[Date], balance: Balance):
  Validation[NonEmptyList[String], Account] = {
  val od = openDate.getOrElse(today)
  (validateAccountNo(no) |@| validateOpenCloseDate(openDate.getOrElse(today), closeDate) |@| validateRate(rate)) { (n, d, r) =>
    SavingsAccount(n, name, r, d._1, d._2, balance)
    }
  }
}

def checkingAccount(no:String, name:String, openDate:Option[Date], closeDate: Option[Date], balance: Balance):
  Validation[NonEmptyList[String], Account] = {
  val od = openDate.getOrElse(today)
  (validateAccountNo(no) |@| validateOpenCloseDate(openDate.getOrElse(today), closeDate)) { (n, d) =>
    CheckingAccount(n, name, d._1, d._2, balance)
  }
}
```

Debasish Ghosh
@debasishg

https://github.com/debasishg/frdomain/blob/master/src/main/scala/frdomain/ch4/patterns/Account.scala



to be continued in Part IV