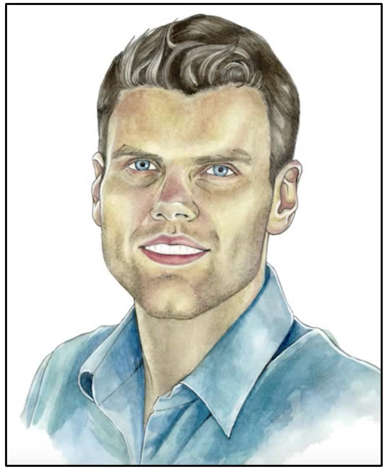
Applicative Functor

learn how to use an Applicative Functor to handle multiple independent effectful values through the work of

Sergei Winitzki

sergei-winitzki-11a6431



Runar Bjarnason

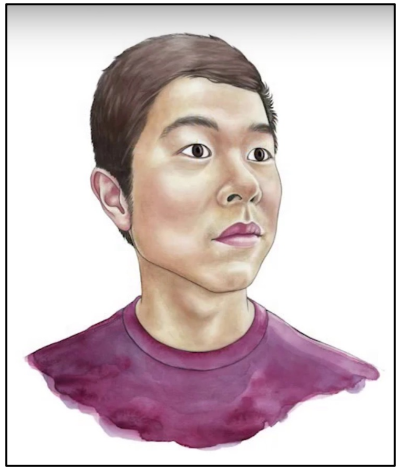
@runarorama

Paul Chiusano

@pchiusano

Debasish Ghosh

@debasishg



Adelbert Chang

@adelbertchang

slides by @philip_schwarz

Sergei Winitzki introduces the motivation for **Applicative Functors**

Motivation for applicative functors

- Monads are inconvenient for expressing *independent* effects

Monads perform effects *sequentially* even if effects are independent:

```
x ← Future { c1 }
y ← Future { c2 }
z ← Future { c3 }
```

```
Future { c1 }.flatMap { x ⇒
  Future { c2 }.flatMap { y ⇒
    Future { c3 }.map { z ⇒ ...}
} }
```

- We would like to parallelize independent computations
- We would like to accumulate *all* errors, rather than stop at the first one

Changing the order of monad's effects will (generally) change the result:

```
for {
  x ← List(1, 2)
  y ← List(10, 20)
} yield f(x, y)
// f(1, 10), f(1, 20), f(2, 10), f(2, 20)
```

```
for {
  y ← List(10, 20)
  x ← List(1, 2)
} yield f(x, y)
// f(1, 10), f(2, 10), f(1, 20), f(2, 20)
```

- We would like to express a computation where effects are unordered
  - This can be done using a method `map2`, *not* defined via `flatMap`: the desired type signature is $\text{map2} : F^A \times F^B \Rightarrow (A \times B \Rightarrow C) \Rightarrow F^C$
  - **Applicative functor** has `map2` and `pure` but is not necessarily a monad

Chapter 8: Applicative functors and profunctors
Part 1: Practical examples

Sergei Winitzki
Academy by the Bay

2018-06-25

Sergei Winitzki
sergei-winitzki-11a6431

Functional programming, chapter 8.
Applicative functors and profunctors.
Part 1: Practical examples YouTube

Defining `map2`, `map3`, etc.

Consider 1, 2, 3, ... commutative and independent "effects"

```
for { x1 ← c1
} yield f(x1)
```

c1.map(f)

this is a **flatMap**

this is a **map**

We want to replace that...

```
for { x1 ← c1
  x2 ← c2
} yield f(x1, x2)
```

REPLACE WITH

(c1, c2).map2(f)

with this kind of syntax perhaps, where we have **map2** which takes a tuple of parameters, it also takes a function **f** with two arguments and returns a container.

```
for { x ← c1
  x2 ← c2
  x3 ← c3
} yield f(x1, x2, x3)
```

REPLACE WITH

(c1, c2, c3).map3(f)

Here is a similar thing with 3 containers, where we have **map3**.

- Generalize from `map`, `map2`, `map3` to `mapN`:

$$\text{map}_1 : F^A \Rightarrow (A \Rightarrow Z) \Rightarrow F^Z$$

$$\text{map}_2 : F^A \times F^B \Rightarrow (A \times B \Rightarrow Z) \Rightarrow F^Z$$

$$\text{map}_3 : F^A \times F^B \times F^C \Rightarrow (A \times B \times C \Rightarrow Z) \Rightarrow F^Z$$

Sergei Winitzki
in sergei-winitzki-11a6431

Functional programming, chapter 8.
Applicative functors and profunctors.
Part 1: Practical examples YouTube

Sergei Winitzki defines map2 for Either[String, A]

Functional programming, chapter 8.
Applicative functors and profunctors.
Part 1: Practical examples YouTube

Sergei Winitzki
in sergei-winitzki-11a6431

```scala
class Chapter08_01_examplesSpec extends FlatSpec with Matchers {

  behavior of "examples"

  it should "implement map2 and map3 for Either" in {
    type Op[A] = Either[String, A]

    def safeDivide(x: Double, y: Double): Op[Double] = if (y == 0.0)
      Left(s"Error: dividing $x by 0\n")
    else Right(x / y)

    // Want to perform operations and collect all errors.
    def map2[A, B, Z](a: Op[A], b: Op[B])(f: (A, B) ⇒ Z): Op[Z] = (a, b) match {
      case (Left(s1), Left(s2)) ⇒ Left(s1 + s2) // Concatenate the two error
  messages.
      case (Left(s), Right(_)) ⇒ Left(s)
      case (Right(_), Left(s)) ⇒ Left(s)
      case (Right(x1), Right(x2)) ⇒ Right(f(x1, x2))
    }

    // We can now collect all error messages.
    map2(
      safeDivide(1, 0),
      safeDivide(2, 0)
    ) { (x, y) ⇒ x - y } shouldEqual Left("Error: dividing 1.0 by 0\nError:
  dividing 2.0 by 0\n")
```

In the case where we have two errors, we want to accumulate the two errors.

In these two cases there is not enough data to call f so we return an error.

In the case where we have two results, we can actually perform the computation that is requested, which is this function f.

Notice that **map2** does more than a **monadic for**, which **stops after the first error**: it can **accumulate all the errors**.

both errors are returned

```scala
// Note that this definition of `map2` is not equivalent to the monadic
definition:
(for {
  x ← safeDivide(1, 0)
  y ← safeDivide(2, 0)
} yield x - y) shouldEqual Left("Error: dividing 1.0 by 0\n")
```

only the first error is returned

Sergei Winitzki shows how to define **map3** using **map2**

```scala
  // Now let's define map3:
  def map3[A, B, C, Z](a: Op[A], b: Op[B], c: Op[C])(f: (A, B, C) ⇒ Z): Op[Z] = {
    // We would like to avoid listing 8 possible cases now.
    // Let's begin by applying map2() to (a, b).
    val opab: Op[(A, B)] = map2(a, b) { (x, y) ⇒ (x, y) } // Almost an identity function here...
    // Now we can use map2 again on opab and c:
    map2(opab, c) { case ((aa, bb), cc) ⇒ f(aa, bb, cc) }
  }
  // This is still awkward to generalize.

  map3(
    safeDivide(1, 0),
    safeDivide(2, 0),
    safeDivide(3, 1)
  ) { (x, y, z) ⇒ x - y } shouldEqual Left("Error: dividing 1.0 by 0\nError: dividing 2.0 by 0\n")

```

Sergei Winitzki
in sergei-winitzki-11a6431

Defining **Applicative** in terms of primitive combinators **map2** and **unit**

12.2 *The Applicative trait*

Applicative functors can be captured by a new interface, `Applicative`, in which `map2` and `unit` are primitives.

Listing 12.1 Creating the `Applicative` interface

```scala
trait Applicative[F[_]] extends Functor[F] {
  // primitive combinators
  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C]
  def unit[A](a: => A): F[A]

  // derived combinators
  def map[B](fa: F[A])(f: A => B): F[B] =
   map2(fa, unit(()))((a, _) => f(a))

  def traverse[A,B](as: List[A])(f: A => F[B]): F[List[B]]
   as.foldRight(unit(List[B]()))((a, fbs) => map2(f(a), fbs)(_ :: _))

}
```

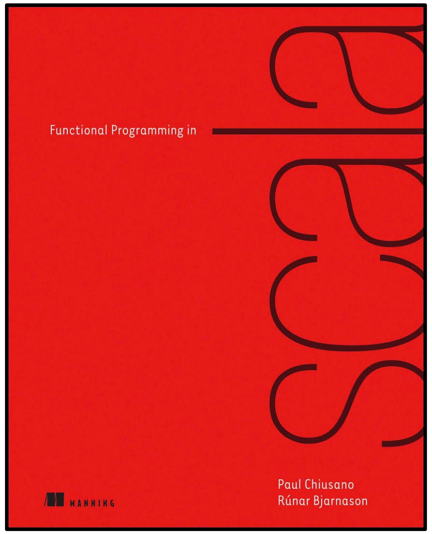
We can implement `map` in terms of `unit` and `map2`.

Recall `()` is the sole value of type `Unit`, so `unit(())` is calling `unit` with the dummy value `()`.

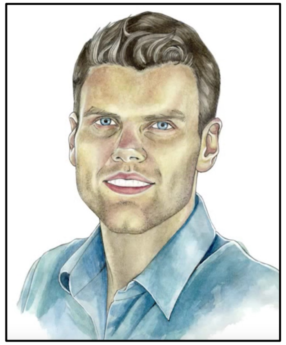
Definition of `traverse` is identical.

This establishes that *all applicatives are functors*. We implement `map` in terms of `map2` and `unit`, as we've done before for particular data types. The implementation is suggestive of laws for `Applicative` that we'll examine later, since we expect this implementation of `map` to preserve structure as dictated by the `Functor` laws.

Note that the implementation of `traverse` is unchanged. We can similarly move other combinators into `Applicative` that don't depend directly on `flatMap` or `join`.



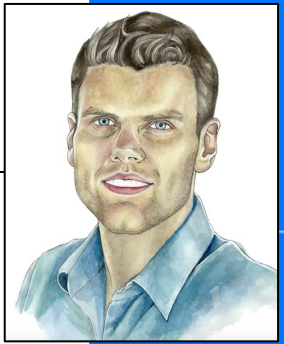
Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

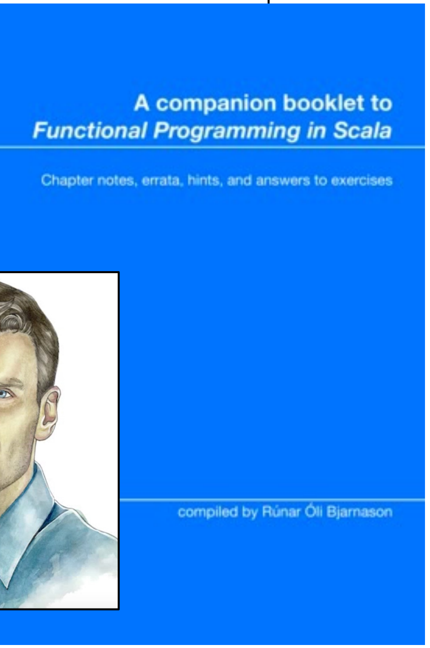


Applicative can also be defined using **apply** and **unit**

Answers to exercises

```scala
trait Applicative[F[_]] extends Functor[F] {
  // `map2` is implemented by first currying `f` so we get a function
  // of type `A => B => C`. This is a function that takes `A` and returns
  // another function of type `B => C`. So if we map `f.curried` over an
  // `F[A]`, we get `F[B => C]`. Passing that to `apply` along with the
  // `F[B]` will give us the desired `F[C]`.
  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C] =
    apply(map(fa)(f.curried), fb)

  // We simply use `map2` to lift a function into `F` so we can apply it
  // to both `fab` and `fa`. The function being lifted here is `_(_)`,
  // which is the same as the lambda notation `(f, x) => f(x)`. That is,
  // It's a function that takes two arguments:
  //   1. A function `f`
  //   2. An argument `x` to that function
  // and it simply applies `f` to `x`.
  def apply[A,B](fab: F[A => B])(fa: F[A]): F[B] =
    map2(fab, fa)(_(_))
  def unit[A](a: => A): F[A]

  def map[A,B](fa: F[A])(f: A => B): F[B] =
    apply(unit(f))(fa)
}
```





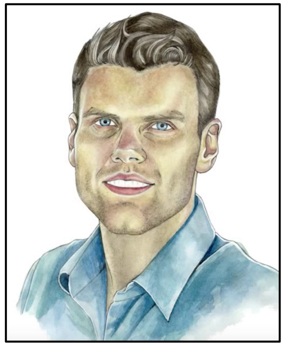
EXERCISE 12.2

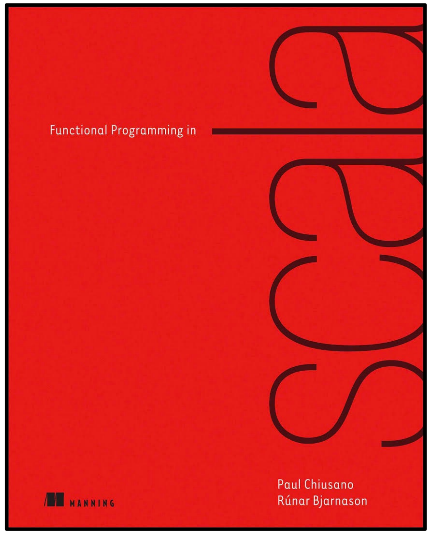
Hard: The name *applicative* comes from the fact that we can formulate the `Applicative` interface using an alternate set of primitives, `unit` and the function `apply`, rather than `unit` and `map2`. Show that this formulation is equivalent in expressiveness by defining `map2` and `map` in terms of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

```scala
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`.





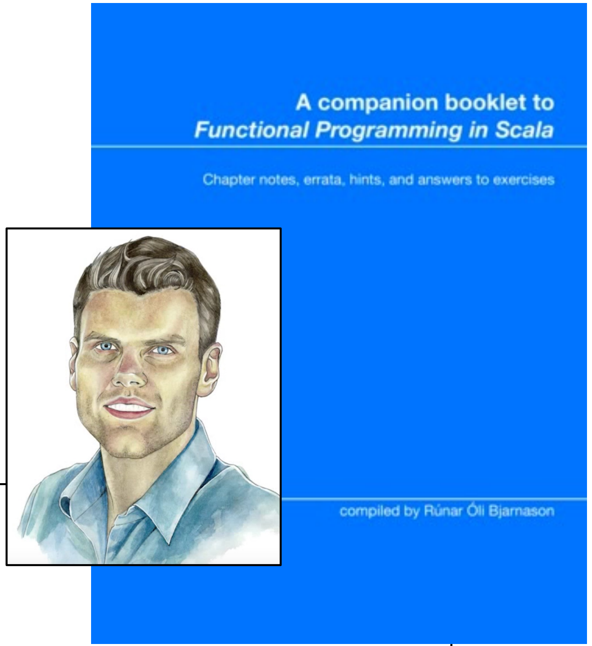
Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

Defining **map3** and **map4** using **unit**, **apply** and the **curried** method available on functions

EXERCISE 12.3

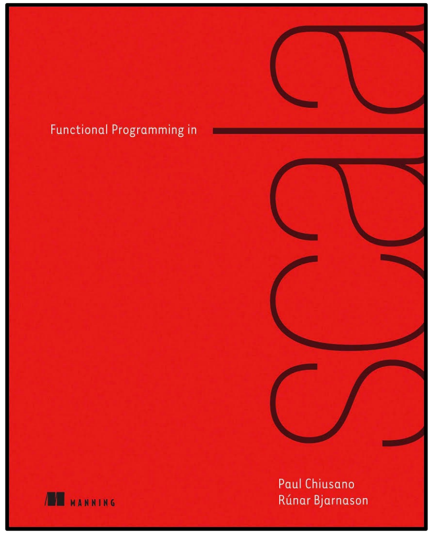
The apply method is useful for implementing map3, map4, and so on, and the pattern is straightforward. Implement map3 and map4 using only unit, apply, and the curried method available on functions.[1]

```
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]
```

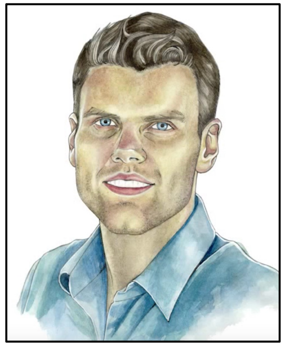


Exercise 12.03

```
/*
The pattern is simple. We just curry the function
we want to lift, pass the result to `unit`, and then `apply`
as many times as there are arguments.
Each call to `apply` is a partial application of the function
*/
def map3[A,B,C,D](fa: F[A],
                  fb: F[B],
                  fc: F[C])(f: (A, B, C) => D): F[D] =
  apply(apply(apply(unit(f.curried))(fa))(fb))(fc)

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]
  apply(apply(apply(apply(unit(f.curried))(fa))(fb))(fc))(fd)
```



Functional Programming in Scala



(by Paul Chiusano and Runar Bjarnason)

@pchiusano @runarorama

Validation - much like Either, except it can handle more than one error

Let's invent a new data type, `Validation`, that is much like `Either` except that it can explicitly handle more than one error:

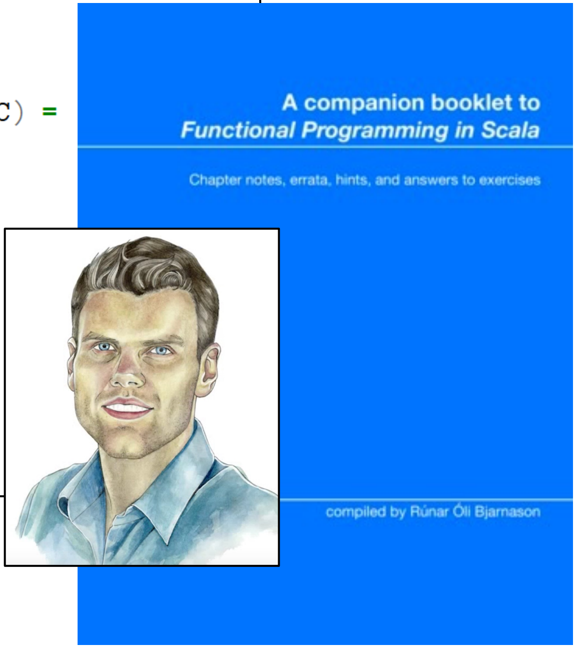
```
sealed trait Validation[+E, +A]

case class Failure[E](head: E, tail: Vector[E] = Vector())
  extends Validation[E, Nothing]

case class Success[A](a: A) extends Validation[Nothing, A]
```

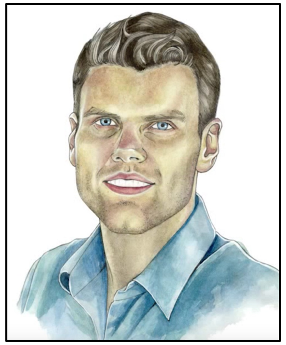
EXERCISE 12.6

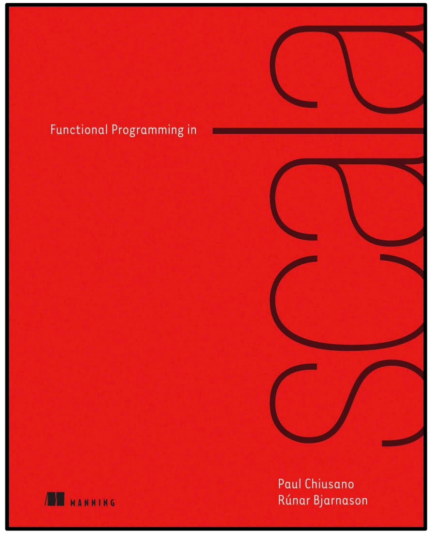
Write an `Applicative` instance for `Validation` that accumulates errors in `Failure`. Note that in the case of `Failure` there's always at least one error, stored in `head`. The rest of the errors accumulate in the `tail`.

Exercise 12.06

```
def validationApplicative[E]: Applicative[({type f[x] = Validation[E,x]})#f] =
  new Applicative[({type f[x] = Validation[E,x]})#f] {
    def unit[A](a: => A) = Success(a)
    override def map2[A,B,C](fa: Validation[E,A],
                             fb: Validation[E,B])(f: (A, B) => C) =
      (fa, fb) match {
        case (Success(a), Success(b)) => Success(f(a, b))
        case (Failure(h1, t1), Failure(h2, t2)) =>
          Failure(h1, t1 ++ Vector(h2) ++ t2)
        case (e@Failure(_, _), _) => e
        case (_, e@Failure(_, _)) => e
      }
  }
```







Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

Validation of independent values using map3

To continue the example, consider a web form that requires a name, a birth date, and a phone number:

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

This data will likely be collected from the user as strings, and we must make sure that the data meets a certain specification. If it doesn't, we must give a list of errors to the user indicating how to fix the problem. The specification might say that name can't be empty, that birthdate must be in the form "yyyy-MM-dd", and that phoneNumber must contain exactly 10 digits.

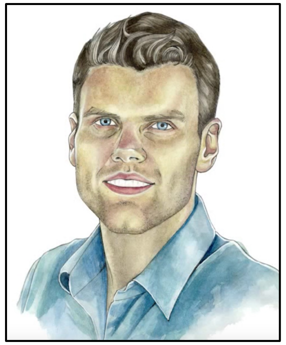
Listing 12.5 Validating user input in a web form

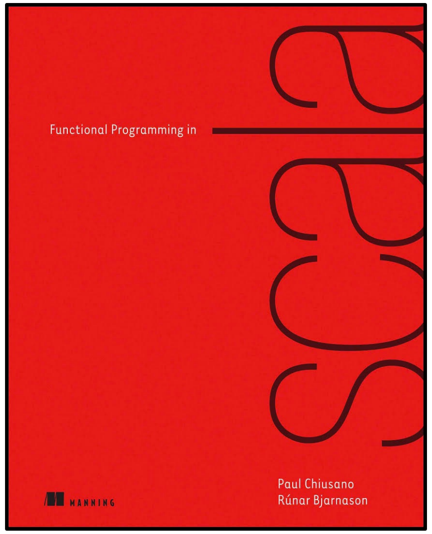
```
def validName(name: String): Validation[String, String] =
  if (name != "") Success(name)
  else Failure("Name cannot be empty")

def validBirthdate(birthdate: String): Validation[String, Date] =
  try {
    import java.text._
    Success((new SimpleDateFormat("yyyy-MM-dd")).parse(birthdate))
  } catch {
    Failure("Birthdate must be in the form yyyy-MM-dd")
  }

def validPhone(phoneNumber: String): Validation[String, String] =
  if (phoneNumber.matches("[0-9]{10}"))
      Success(phoneNumber)
  else Failure("Phone number must be 10 digits")
```

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

```
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.





Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

apply is also known as **ap** and **map2** is also known as **apply2**

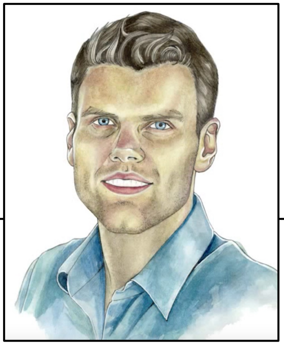
Answers to exercises

```scala
trait Applicative[F[_]] extends Functor[F] {
  // `map2` is implemented by first currying `f` so we get a function
  // of type `A => B => C`. This is a function that takes `A` and returns
  // another function of type `B => C`. So if we map `f.curried` over an
  // `F[A]`, we get `F[B => C]`. Passing that to `apply` along with the
  // `F[B]` will give us the desired `F[C]`.
  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C] =
    apply(map(fa)(f.curried), fb)

  // We simply use `map2` to lift a function into `F` so we can apply it
  // to both `fab` and `fa`. The function being lifted here is `_(_)`,
  // which is the same as the lambda notation `(f, x) => f(x)`. That is,
  // It's a function that takes two arguments:
  //   1. A function `f`
  //   2. An argument `x` to that function
  // and it simply applies `f` to `x`.
  def apply[A,B](fab: F[A => B])(fa: F[A]): F[B] =
    map2(fab, fa)(_(_))
  def unit[A](a: => A): F[A]

  def map[A,B](fa: F[A])(f: A => B): F[B] =
    apply(unit(f))(fa)
}
```

Listing 4.4 The Applicative Functor trait (simplified)

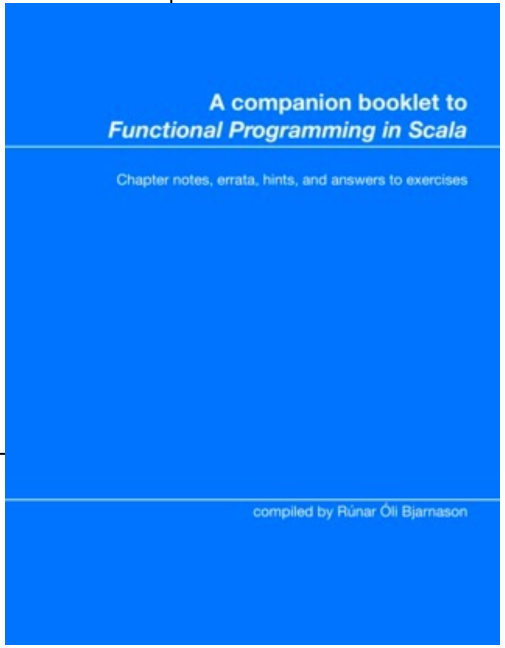
```scala
trait Applicative[F[_]] extends Functor[F] {
  def ap[A,B](fa: => F[A])(f: => F[A => B]): F[B]
  def apply2[A,B,C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C] =
    ap(fb)(map(fa)(f.curried))
  def lift2[A,B,C](f: (A, B) => C): (F[A], F[B]) => F[C] =
    apply2(_, _)(f)
  def unit[A](a: => A): F[A]
}
```

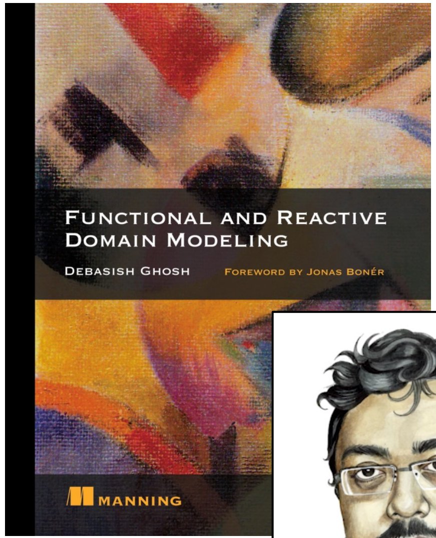
Primitive operations that implementing classes need to provide. You'll see a sample implementation shortly.



Runar Bjarnason)

@runarorama





Debasish Ghosh

@debasishg

Validation of independent values using apply3 (alternative name for map3)

```
type V[A] = Validation[String, A]

def validateAccountNo(no: String): V[String]
def validateOpenCloseDate(openDate: Option[Date], closeDate: Option[Date]):
     V[(Date, Option[Date])]
def validateRateOfInterest(rate: BigDecimal): V[BigDecimal]
```

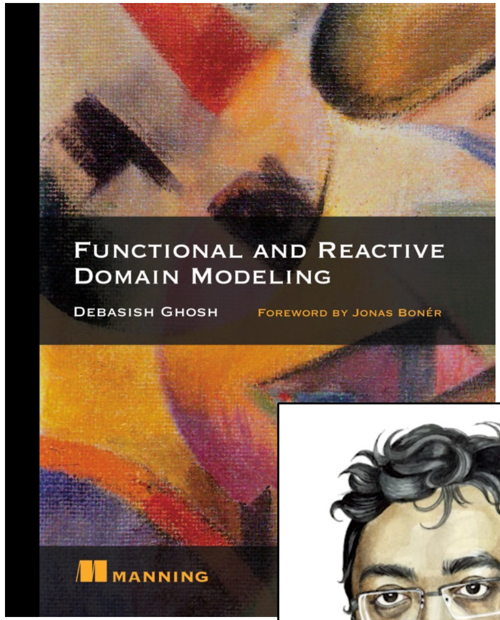
You know nothing about the implementation of `Validation` so far—just a placeholder for the type and how it fits into the algebra of the validation functions. After you invoke all three of the validation functions, you get three instances of `V[_]`. If all of them indicate a successful validation, you 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`[13]

```
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



Debasish Ghosh

@debasishg

Validation of independent values using apply3 (alternative name for map3)

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:

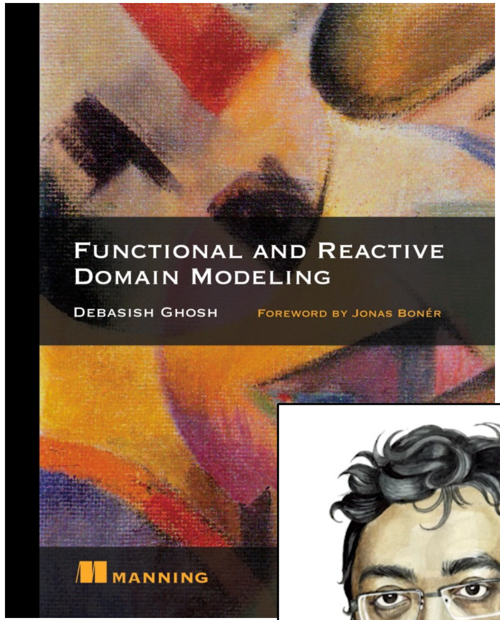
```
def savingsAccount(no: String, name: String, rate: BigDecimal,
  openDate: Option[Date], closeDate: Option[Date],
  balance: Balance): V[Account] = {

  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.

apply3 nicely fits this use case. But you need to generalize the entire workflow into an abstraction that can have a broader application. After all, apply3 or lift3 aren't APIs specific to validating a bunch of fields. Where will apply3 or lift3 live? Right now you've kept them in a global namespace and parameterized them on the context type constructor V[_]. Let's put them into a module and unearth the pattern inside.



Debasish Ghosh

@debasishg

Validation of independent values using **apply3** (alternative name for **map3**)

Listing 4.4 The Applicative Functor trait (simplified)

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

  def ap[A,B](fa: => F[A])(f: => F[A => B]): F[B]

  def apply2[A,B,C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C] =
    ap(fb)(map(fa)(f.curried))

  def lift2[A,B,C](f: (A, B) => C): (F[A], F[B]) => F[C] =
    apply2(_, _)(f)

  def unit[A](a: => A): F[A]
}
```

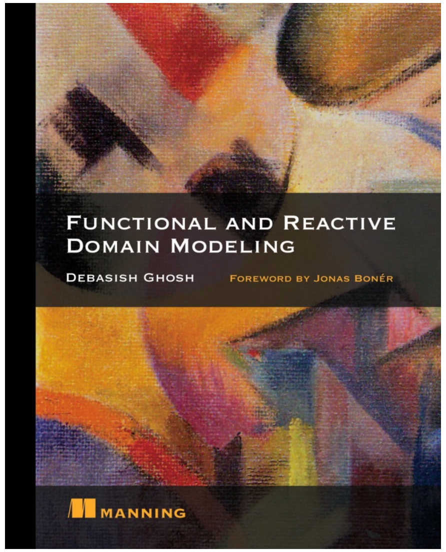
Primitive operations that implementing classes need to provide. You'll see a sample implementation shortly.

After you have the Applicative trait in place, provide an instance of Applicative for Validation.[14] And you have the implementation of savingsAccount with the validation logic implemented with applicative effects.

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

  // ..

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

You invoke apply3 from the Applicative instance of V.

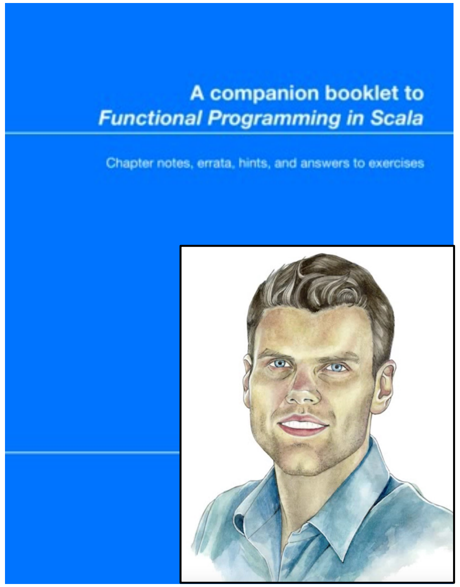
Debasish Ghosh
@debasishg



Exercise 12.03

```
/*
The pattern is simple. We just curry the function
we want to lift, pass the result to `unit`, and then `apply`
as many times as there are arguments.
Each call to `apply` is a partial application of the function
*/
def map3[A,B,C,D](fa: F[A],
                  fb: F[B],
                  fc: F[C])(f: (A, B, C) => D): F[D] =
  apply(apply(apply(unit(f.curried))(fa))(fb))(fc)

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]
  apply(apply(apply(apply(unit(f.curried))(fa))(fb))(fc))(fd)
```

Runar Bjarnason)
@runarorama



Working with effects

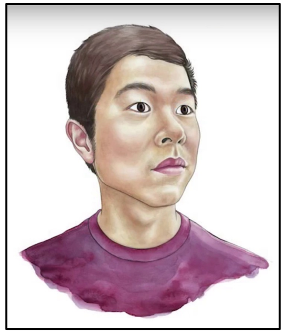
```scala
val x: Option[Int] = parseInt(...)

val y: Option[Int] = x match {
  case Some(int) => Some(int max 0)
  case None      => None
}
```

All these follow more or less the same pattern. We want to apply a **pure function** to an **effectful value** and we can't just do this using the normal mechanism because an **F** of **A** can't be treated as an **A**, if it could, this whole exercise would be futile. **The whole point is, F of A is separate from an A but we still want to be able to apply functions expecting an A to this value**. And so for each **F**, e.g. **Option**, **Either**, **State**, they have this notion of **getting inside this value and observing what that is, and in the case where we do have an A, apply the function and then we are also going to plumb some bits through**.

Working with effects

The Functor, Applicative, Monad talk YouTube

- Apply a **pure** function $f : A \to B$ to an **effectful** value F[A]
- "Inside" F[A] we apply the function to the value and propagate some extra bits through, giving F[B]
  - These "extra bits" tend to be the "something" effectful functions do



Adelbert Chang
@adelbertchang

Working with effects

```scala
val x: Either[String, Host] = getKey[Host]("host")

val y: Either[String, Endpoint] = x match {
  case Right(host) => Right(host / "api")
  case Left(err)   => Left(err)
}
```

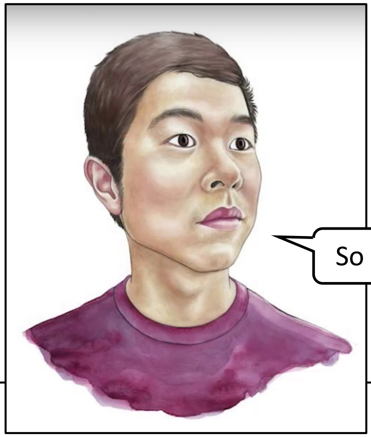
Functor

So for **Option** it is sort of **propagating the effect of not having a value**, for **Either**, **propagating the error**, and for State…
And so, **being the good programmers that we are, we see the same pattern occurring over and over again and we want to be able to talk about them generically**, so **we unify this pattern into this typeclass called Functor**.

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

Adelbert Chang

@adelbertchang



So here is an example for **Option**

```scala
new Functor[Option] {
  def map[A, B](fa: Option[A])(f: A => B): Option[B] =
    fa match {
      case Some(a) => Some(f(a))
      case None    => None
    }
}
```

Functor

In order to be a **Functor** you need to support this very generic **map** operation. In which given a **effectful value F** of **A** and a pure function **A** to **B**, I want you to somehow apply that function to the **effectful value**.

and here is an example for **Either**

```scala
new Functor[Either[E, ?]] {
  def map[A, B](fa: Either[E, A])(f: A => B): Either[E, B]
    fa match {
      case Right(a) => Right(f(a))
      case Left(e)  => Left(e)
    }
}
```

[1]? syntax is enabled by the kind-projector compiler plugin https://github.com/non/kind-projector

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

```scala
val i: Option[Int] = parseInt(...)
val bounded: Option[Int] = i.map(_ max 0)

val path: Either[String, Host] = getKey[Host]("host")
val api: Either[String, Endpoint] = path.map(_ / "api")
```

The Functor, Applicative, Monad talk YouTube

Working with a single effectful value

The Functor, Applicative, Monad talk YouTube

- What if we want to work with two or more effectful values?
- Apply a pure n-ary function to n effectful values
- Focus on tupling the values - (F[A], F[B]) => F[(A, B)]

Working with multiple effectful values

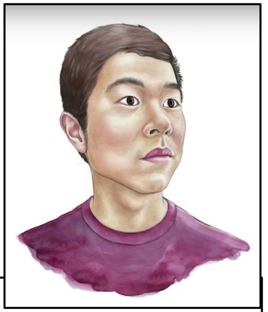
```scala
def pairEither[E, A, B]
  (ea: Either[E, A], eb: Either[E, B]): Either[E, (A, B)] =

  (ea, eb) match {
    case (Right(a), Right(b)) => Right((a, b))
    case (Left(e), _)         => Left(e)
    case (_, Left(e))         => Left(e)
  }
```

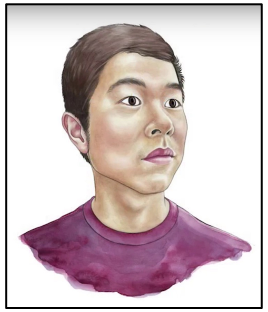
Working with multiple effectful values

```scala
def pairOption[A, B]
  (oa: Option[A], ob: Option[B]): Option[(A, B)] =

  (oa, ob) match {
    case (Some(a), Some(b)) => Some((a, b))
    case _                  => None
  }
```

Adelbert Chang

@adelbertchang





pairOption and **pairEither** follow more or less the same pattern. We have a notion of **looking inside two effectful values and then pairing them up together while remaining inside the effect**, and **this is what applicatives are all about**.

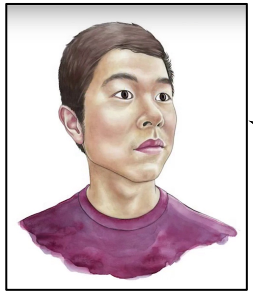
Applicative

Applicative

And of course we also want all applicatives to have this **map** operation, because **in order to work with N effectful values, we are going to zip them together, so we get a pair, and then we are going to map over it to destructure the pair and apply an N-ary function**.
And also, it **makes sense conceptually**, because if we claim to work with N **independent effectful values**, then I certainly should be able to work with a single **effectful value**.

```scala
trait Applicative[F[_]] extends Functor[F] {
  def zip[A, B](fa: F[A], fb: F[B]): F[(A, B)]
  def pure[A](a: A): F[A]

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

Applicative



So here are **Applicative** implementations for **Option** and **Either**

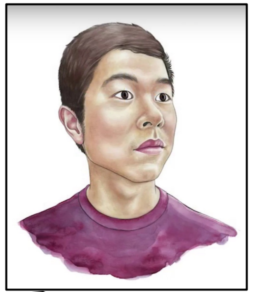
```
trait Applicative[F[_]] extends Functor[F] {
  def zip[A, B](fa: F[A], fb: F[B]): F[(A, B)]
  def pure[A](a: A): F[A]

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

```
new Applicative[Option] {
  def zip[A, B]
    (fa: Option[A], fb: Option[B]): Option[(A, B)] =

    (fa, fb) match {
      case (Some(a), Some(b)) => Some((a, b))
      case _                  => None
    }

  def pure[A](a: A): Option[A] = Some(a)

  ...
}
```

The Functor, Applicative, Monad talk

```
new Applicative[Either[E, ?]] {
  def zip[A, B]
    (fa: Either[E, A], fb: Either[E, B]): Either[E, (A, B)]

    (fa, fb) match {
      case (Right(a), Right(b)) => Right((a, b))
      case (Left(e), _)         => Left(e)
      case (_, Left(e))         => Left(e)
    }

  def pure[A](a: A): Either[E, A] = Right(a)

  ...
}
```

So here is some stuff that we couldn't do with **Functor** that we can do with **Applicative**. Let's say I parse two integers now, both of which may or may not fail, I want to **zip** them together and then **map** over it and then figure out which one of those integers is larger and this gives me back an **Option** of an integer.

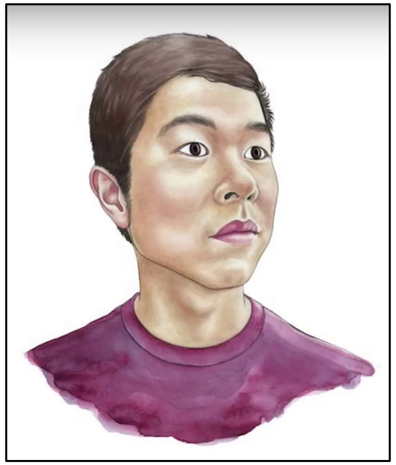


```
// Option[Int]
parseInt(...).zip(parseInt(...)).map {
  case (x, y) => x max y
}

// Either[Error, Endpoint]
getKey[Host]("host").zip(getKey[Port]("port")).map {
  case (host, p) => host :| p / "api"
}
```

I can parse two keys from a config, or I could parse 3 or 4 keys if I wanted to, and then **zip** them together and **map** over it and then get an endpoint, and that gets me **either an error or an endpoint** and **if any of those parses fail then I get the first error that I hit**.

Adelbert Chang
@adelbertchang

Applicative defined in terms of **zip** + **pure** or in terms of **ap** + **pure**



Adelbert Chang

@adelbertchang

Applicative

The Functor, Applicative, Monad talk YouTube

```scala
trait Applicative[F[_]] extends Functor[F] {
  def zip[A, B](fa: F[A], fb: F[B]): F[(A, B)]
  def pure[A](a: A): F[A]

  def map[A, B](fa: F[A])(f: A => B): F[B]

  def ap[A, B, C](ff: F[A => B])(fa: F[A]): F[B] =
    map(zip(ff, fa)) { case (f, a) => f(a) }
}
```

As a quick note, if you go to say **Cats** or **Scalaz** today, or **Haskell** even, and you look at **Applicative**, what you'll see is this **ap** formulation instead, so what I presented as **zip**, **map** and **pure**, we will typically see as **ap**, and **ap** sort of has a **weird type signature**, at least in **Scala**, **where you have a function inside of an F, and then you have an effectful value, and you want to apply the function to that value, all while remaining in F**, and **this has a nice theoretical story**, and sort of has **a nicer story in Haskell**, but in **Scala**, this sort of makes for an **awkward API**, and so **I like to introduce applicative in terms of zip and map for that reason, I think it makes for a better story, and I think zip is conceptually simpler, because you can sort of see that zip is about composing two values, in the easiest way possible, whereas ap sort of has a weird signature**.

That thing said, **ap is, for historical reasons, like the canonical representation of Applicative**, so if after this talk you go and look what **Applicative** is, you'll probably see **ap**. Just as a quick note, you can implement **ap** in terms of **map** and **zip**, like I have here. You can also go the other way, you can implement **zip** and **map** in terms of **ap**, and so, exercise left to the reader.