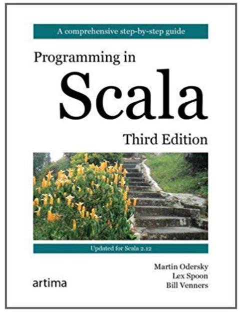
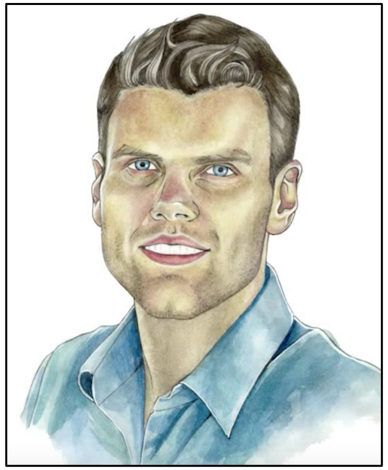
Sequence and Traverse

Part 2

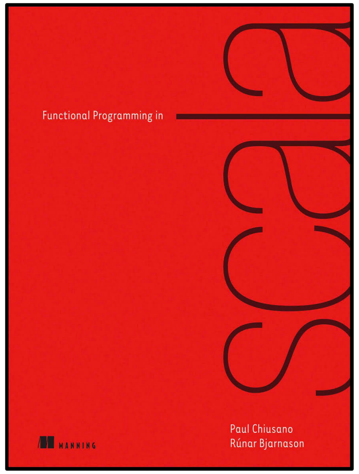
learn about the sequence and traverse functions
through the work of

Martin Odersky
@odersky





Runar Bjarnason
@runarorama

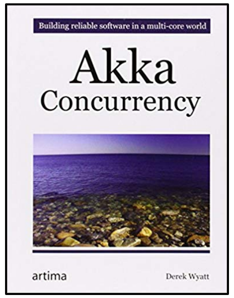


FP in Scala

Paul Chiusano
@pchiusano

Derek Wyatt
@derekwyatt

Adelbert Chang
@adelbertchang



slides by  @philip_schwarz

Let’s start by very quickly recapping a key idea we covered in part 1

@philip_schwarz

```scala
trait Monad[F[_]] extends Functor[F] {

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

  def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]

  def map[A,B](ma: F[A])(f: A => B): F[B] =
    flatMap(ma)(a => unit(f(a)))

  def map2[A,B,C](ma: F[A], mb: F[B])(f: (A, B) => C): F[C] =
    flatMap(ma)(a => map(mb)(b => f(a, b)))

  def sequence[A](lma: List[F[A]]): F[List[A]] =
    lma.foldRight(unit(List[A]()))((ma, mla) => map2(ma, mla)(_ :: _))

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

}
```

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

In part 1 of this presentation we saw how:

- **map2** can be implemented using **map** and **flatMap**
- **traverse** and **sequence** can be implemented using **unit** and **map2**
- since every **monad** has **unit**, **map** and **flatMap**, every **monad** also has **traverse** and **sequence**

e.g. if we define the **Option** monad and the **Either monad**, then they get **sequence** and **traverse** for free

```scala
val optionM = new Monad[Option] {

  def unit[A](a: => A): Option[A] = Some(a)

  def flatMap[A,B](ma:Option[A])(f:A => Option[B]):Option[B] =
    ma match {
      case Some(a) => f(a)
      case None    => None
  }
}

def parseIntsMaybe(a: List[String]): Option[List[Int]] =
  optionM.traverse(a)(i => Try(i.toInt).toOption)
```

```scala
type Validated[A] = Either[Throwable,A]

val eitherM = new Monad[Validated] {

  def unit[A](a: => A): Validated[A] = Right(a)

  def flatMap[A,B](ma:Validated[A])(f:A => Validated[B]):Validated[B] =
    ma match {
      case Left(l)  => Left(l)
      case Right(a) => f(a)
  }
}

def parseIntsValidated(a: List[String]): Either[Throwable,List[Int]] =
  eitherM.traverse(a)(i => Try(i.toInt).toEither)
```

@philip_schwarz

So in part 1 we coded up the **monad** trait and included in it a **sequence** method and a **traverse** method.

What about the **Scala** standard library? Does its notion of a **monad** include **sequence** and **traverse** methods?

Monads in Scala

Monads

Data structures with map and flatMap seem to be quite common.

In fact there's a name that describes this class of a data structures together with some algebraic laws that they should have.

They are called *monads*.

You could see a monad in Scala as a trait **M**, where **M** is the monad and **T** is the type parameter. It would have a flatMap method and a unit method

What is a Monad?

A monad M is a parametric type M[T] with two operations, flatMap and unit, that have to satisfy some laws.

```
trait M[T] {
  def flatMap[U](f: T => M[U]): M[U]
}

def unit[T](x: T): M[T]
```

In the literature, flatMap is more commonly called bind.

Examples of Monads

- List is a monad with unit(x) = List(x)
- Set is monad with unit(x) = Set(x)
- Option is a monad with unit(x) = Some(x)
- Generator is a monad with unit(x) = single(x)

flatMap is an operation on each of these types, whereas unit in Scala is different for each monad.

As you have seen, flatMap was available as an operation on each of these types, whereas the unit in **Scala** is different for each monad. Quite often it is just the name of the monad type applied to an argument, e.g. **List(x)**, but sometimes it is different.

In **Scala** we do not have a unit that we can call here, because in **Scala** every monad has a different expression that gives the unit value, therefore in **Scala** map is a primitive function that is also defined on every monad.

Monads and map

map can be defined for every monad as a combination of flatMap and unit:

```
m map f  ==  m flatMap (x => unit(f(x)))
         ==  m flatMap (f andThen unit)
```

To qualify as a monad, a type has to satisfy three laws that connect **flatmap** and **unit**.

Monad Laws

To qualify as a monad, a type has to satisfy three laws:

Associativity:

```
m flatMap f flatMap g  ==  m flatMap (x => f(x) flatMap g)
```

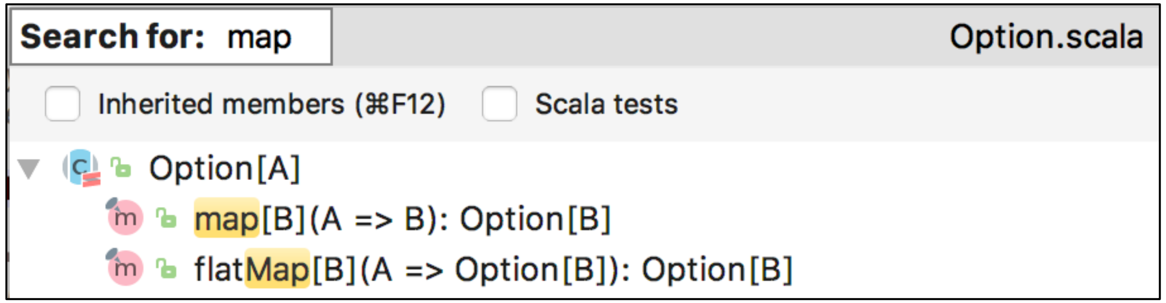
Left unit

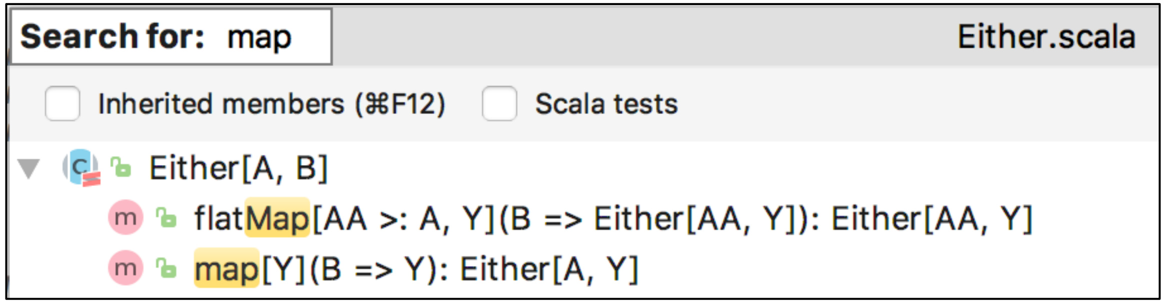
```
unit(x) flatMap f  ==  f(x)
```

Right unit

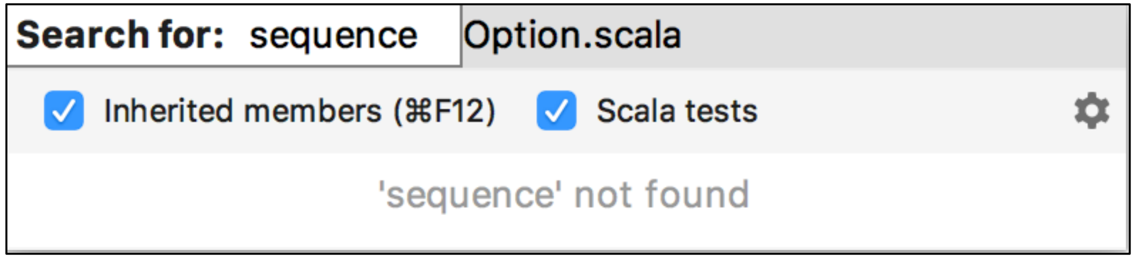
```
m flatMap unit  ==  m
```

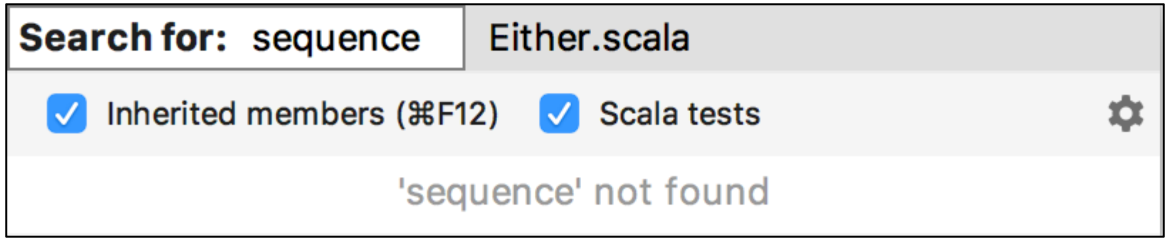
In Scala, Option and Either are monads, and yes, we can see that they have a flatMap function and a map function

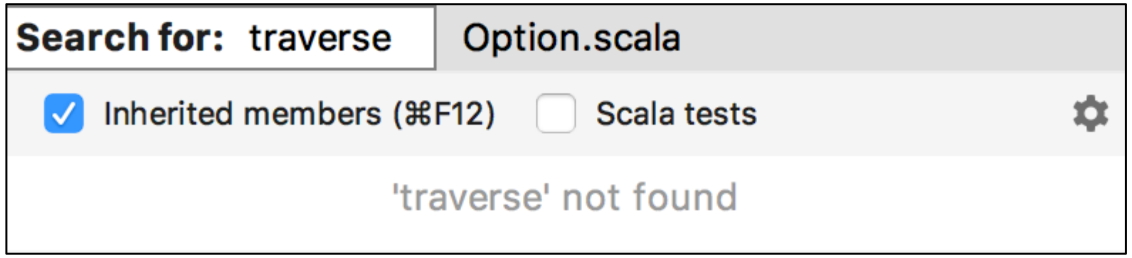


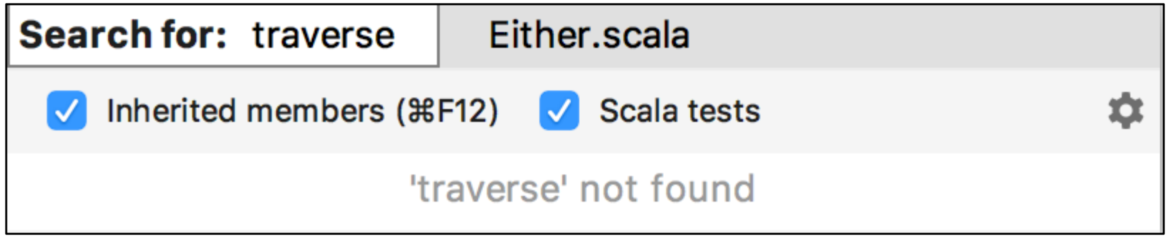


But do Option and Either have sequence and traverse functions? It doesn’t look like it:







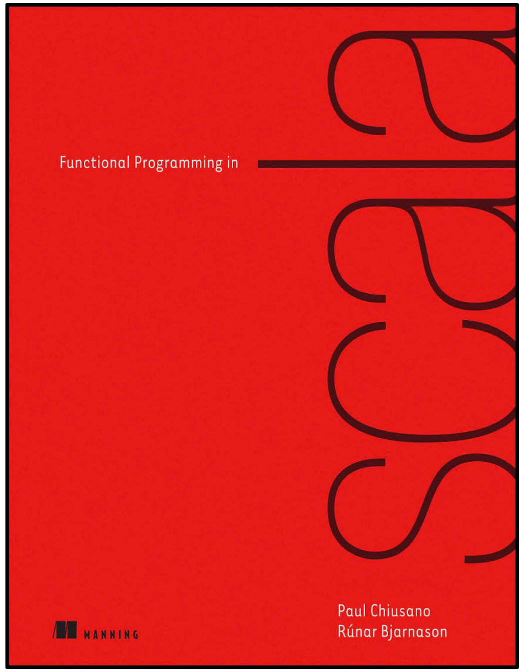


Option and Either in the standard library

As we mentioned earlier in this chapter, **both Option and Either exist in the Scala standard library** (`Option` API is at http://mng.bz/fiJ5; `Either` API is at http://mng.bz/106L), and **most of the functions we've defined here in this chapter exist for the standard library versions**.

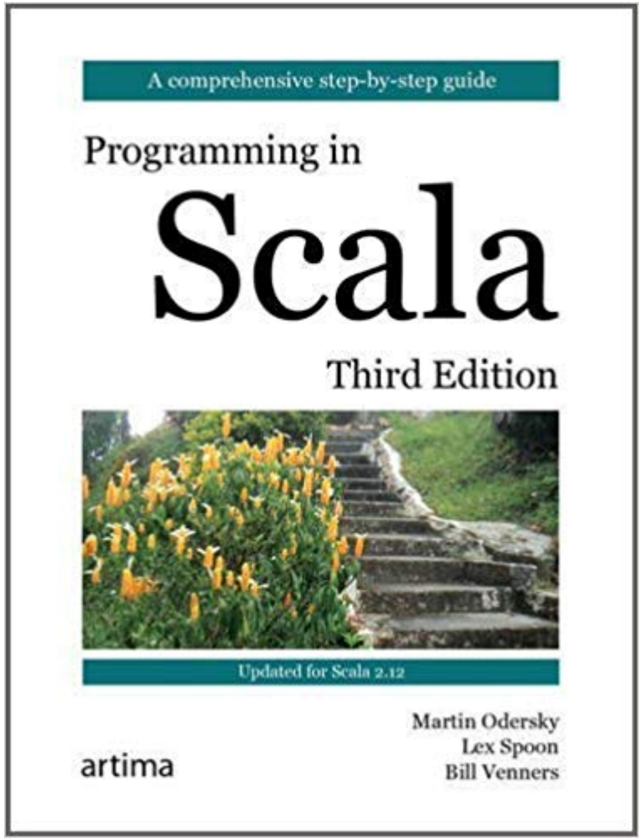
You're encouraged to read through the API for **Option** and **Either** to understand the differences. **There are a few missing functions, though, notably sequence, traverse, and map2.**

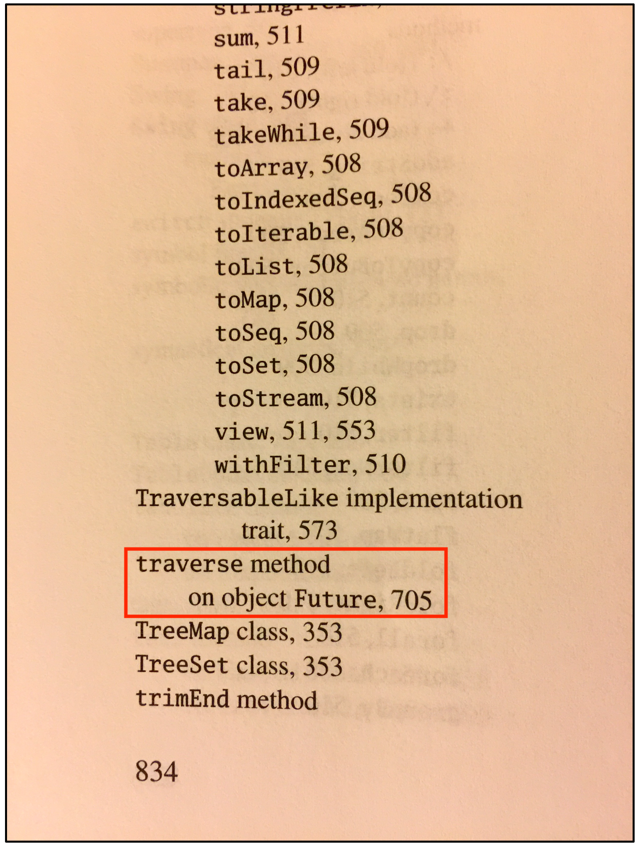
…



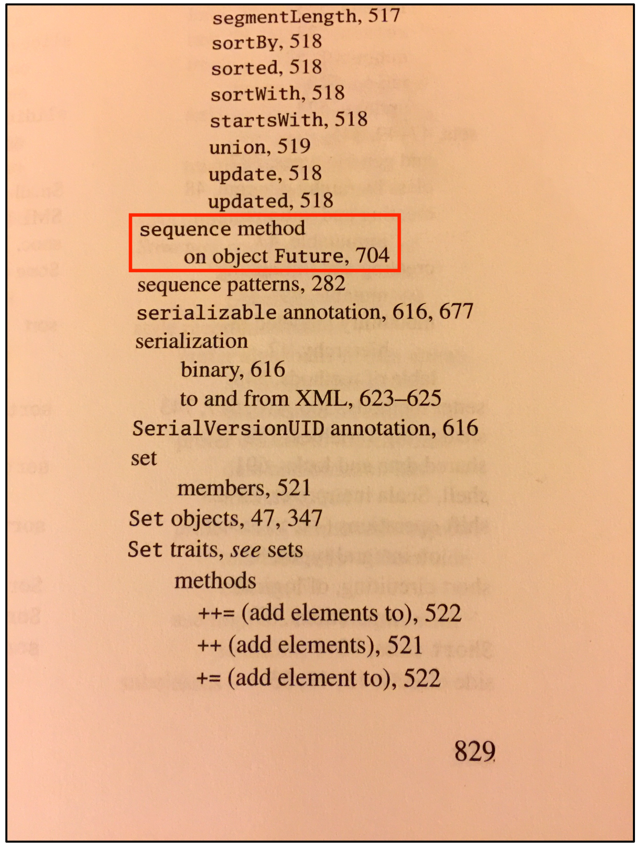
Functional Programming in Scala

Let's look for the **traverse** and **sequence** functions in the Index of Martin Odersky's **Scala** book:



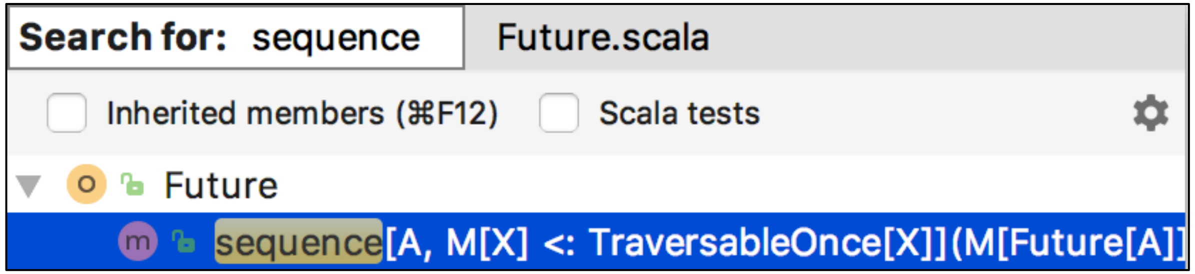


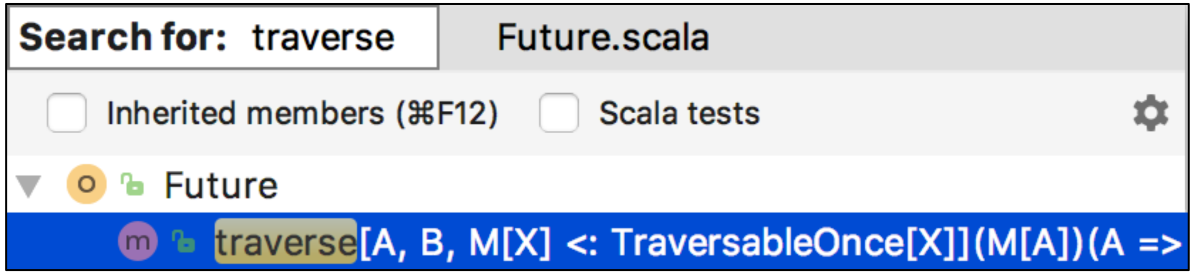
834



829

It looks like there are **sequence** and **traverse** functions in **Future**, but not in **Option** and **Either**.





scala.concurrent

Future

`object Future`

Future companion object.

Source Future.scala

Linear Supertypes

sequence

scala.concurrent.Future.**traverse**

Asynchronously and **non-blockingly** transforms a `TraversableOnce[A]` into a `Future[TraversableOnce[B]]` using the provided function `A => Future[B]`.

This is **useful for performing a parallel map**. For example, to **apply a function to all items of a list in parallel**:

```
val myFutureList = Future.traverse(myList)(x => Future(myFunc(x)))
```

scala.concurrent.Future.**sequence**

Simple version of `Future.traverse`. **Asynchronously** and **non-blockingly** transforms a `TraversableOnce[Future[A]]` into a `Future[TraversableOnce[A]]`.

Useful for reducing many `Future`s into a single `Future`.

Value Members

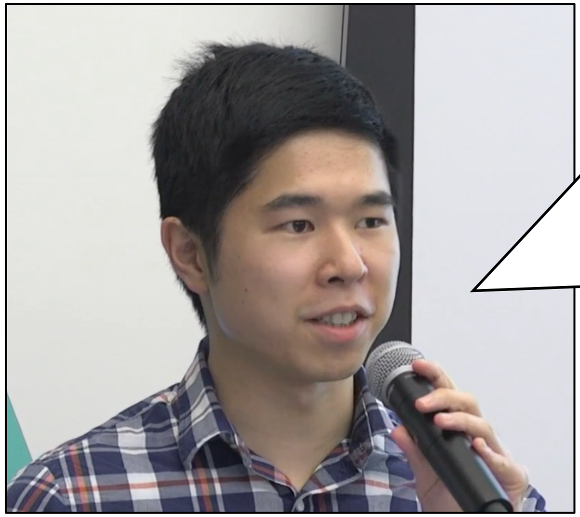
```
def sequence[A, M[X] <: TraversableOnce[X]](in: M[Future[A]])(implicit cbf:
    CanBuildFrom[M[Future[A]], A, M[A]], executor: ExecutionContext): Future[M[A]]
```

Simple version of `Future.traverse`.

```
def traverse[A, B, M[X] <: TraversableOnce[X]](in: M[A])(fn: (A) ⇒ Future[B])(implicit
    cbf: CanBuildFrom[M[A], B, M[B]], executor: ExecutionContext): Future[M[B]]
```

Asynchronously and non-blockingly transforms a `TraversableOnce[A]` into a `Future[TraversableOnce[B]]` using the provided function `A => Future[B]`.

People sometimes end up writing their own **traverse** functions for **Option**, **Either**, etc.



Adelbert Chang

@adelbertchang

The Functor, Applicative, Monad talk YouTube

I have been using Scala for 6/7 years now and by far the most common question that I have seen at work, that people ask me, and in chat rooms, etc, is **how do I do something like Future.traverse except instead of Future I want it with like Option or Either, and in the standard library you don't get that.**

The response ends up being, you can write your own little traverseOption, and then now you want to do it for Either and if you look at enough of these things, you notice **there is a common pattern here, you have one thing here that is sort of lifting of an empty list into some effect, and then we have this notion of combining two effectful values**.

The standard library does not have an abstraction that talks about these things so **you need to either write these yourself or reinvent applicative**, and … actually **write a generic traverse once and for all**.

note: no map2 method is being used here to implement traverse

```
def traverseOption[A, B]
  (as: List[A])(f: A => Option[B]): Option[List[B]] =

  as.foldRight(Option(List.empty[B])) { (a, bs) =>
    (f(a), bs) match {
      case (Some(h), Some(t)) => Some(h :: t)
      case _                  => None
    }
  }
```

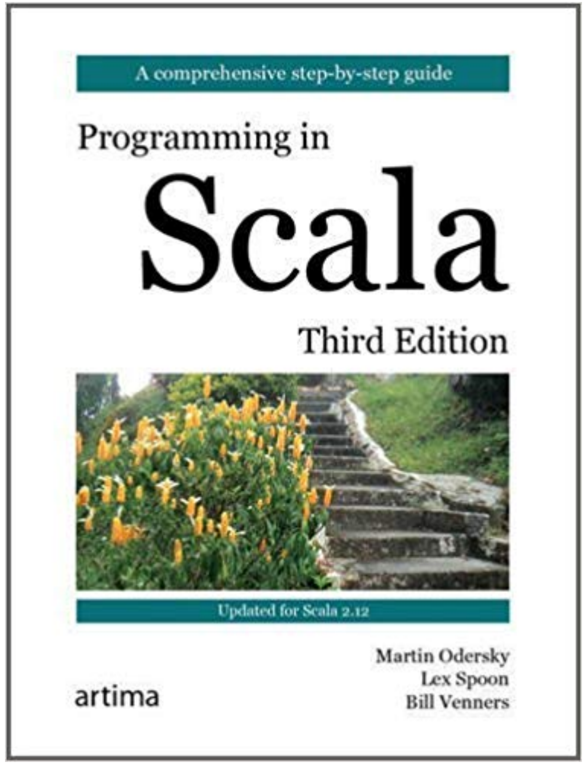
```
def traverseEither[E, A, B]
  (as: List[A])(f: A => Either[E, B]): Either[E, List[B]] =

  as.foldRight(Either.right(List.empty[B])) { (a, bs) =>
    (f(a), bs) match {
      case (Right(h), Right(t)) => Right(h :: t)
      case (Left(e), _)         => Left(e)
      case (_, Left(e))         => Left(e)
    }
  }
```

scala.concurrent.Future's sequence and traverse

```
scala> val fortyTwo = Future { 21 + 21 }
fortyTwo: scala.concurrent.Future[Int] = Future(Success(42))

scala> val fortySix = Future { 23 + 23 }
fortySix: scala.concurrent.Future[Int] = Future(<not completed>)

scala> val futureNums = List(fortyTwo, fortySix)
futureNums: List[scala.concurrent.Future[Int]] = List(Future(Success(42)), Future(Success(46)))
```

The **`Future.sequence`** method transforms a **`TraversableOnce`** collection of futures into a future **`TraversableOnce`** of values. For instance, in the following example, **`sequence`** is used to transform a `List[Future[Int]]` to a `Future[List[Int]]`:

```
scala> val futureList = Future.sequence(futureNums)
futureList: scala.concurrent.Future[List[Int]] = Future(<not completed>)

scala> futureList.value
res50: Option[scala.util.Try[List[Int]]] = Some(Success(List(42, 46)))
```



The **`Future.traverse`** method will change a **`TraversableOnce`** of any element type into a **`TraversableOnce`** of futures and "**`sequence`**" that into a future **`TraversableOnce`** of values. For example, here a `List[Int]` is transformed into a `Future[List[Int]]` by **`Future.traverse`**:

```
scala> val traversed = Future.traverse(List(1, 2, 3)) { i => Future(i) }
traversed: scala.concurrent.Future[List[Int]] = Future(<not completed>)

scala> traversed.value
res51: Option[scala.util.Try[List[Int]]] = Some(Success(List(1, 2, 3)))
```

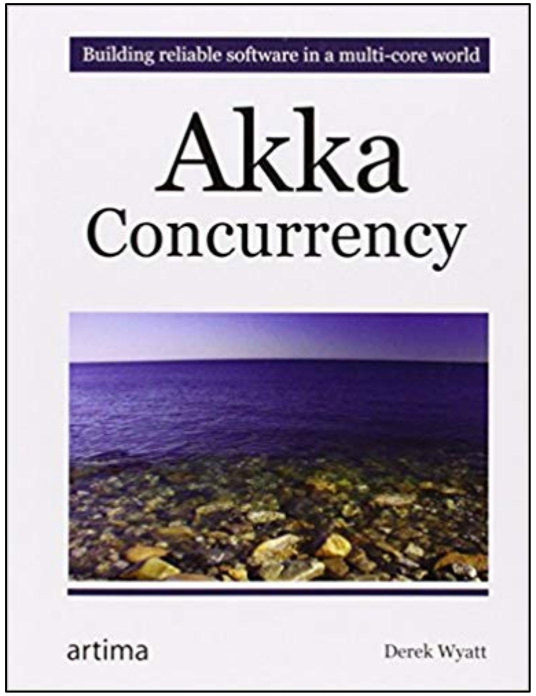
Example of using scala.concurrent.Future.sequence

Future.sequence

Do you remember *way* back to Chapter 4 when we described **the complexity involved with using actors to multiply a whackload of matrices together**? The core of the problem revolved around the non-determinacy of responses from actors that were performing the multiplications. **We had to remember who got what in order to maintain the right order of the responses. In a word, it was *icky***.

Futures provide a *much* better solution to this problem. Multiplying the matrices together isn't all that big of a deal—you just gotta do the math—**the bookkeeping and maintenance of trying to parallelize the computation is the real pain, and futures have a free solution to it**.

Let's set the stage for multiplying matrices with a **mock** `Matrix` class that multiplies things together and produces a resulting `Some[Matrix]` if things work and a `None` if they don't.[4] First, our `Matrix` class:

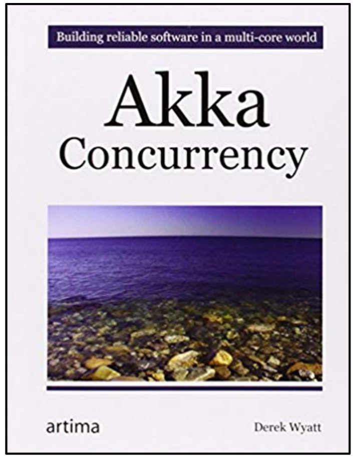



```
case class Matrix(rows: Int, columns: Int) {
  // "Multiply" the two matrices, ensuring that
  // the dimensions line up
  def mult(other: Matrix): Option[Matrix] = {
    if (columns == other.rows) {
      // multiply them...
      Some(Matrix(rows, other.columns))
    } else {
      None
    }
  }
}
```

```
scala> Matrix(3,5) mult Matrix(5,7)
res55: Option[Matrix] = Some(Matrix(3,7))

scala> Matrix(3,5) mult Matrix(4,7)
res56: Option[Matrix] = None
```

Next, we define a function that will multiply a sequence of matrices together using **foldLeft**:

```
def matrixMult(matrices: Seq[Matrix]): Option[Matrix] = {
  matrices.tail.foldLeft(Option(matrices.head)) { (acc, m) =>
    acc flatMap { a => a mult m }
  }
}
```




```
scala> val matricesAllLinedUp = List( Matrix(3,4), Matrix(4,5), Matrix(5,6), Matrix(6,7) )
matricesAllLinedUp: List[Matrix] = List(Matrix(3,4), Matrix(4,5), Matrix(5,6), Matrix(6,7))

scala> matrixMult(matricesAllLinedUp)
res63: Option[Matrix] = Some(Matrix(3,7))

scala> val matricesNotAllLiningUp = List( Matrix(3,4), Matrix(5,5), Matrix(5,6), Matrix(6,7) )
matricesNotAllLiningUp: List[Matrix] = List(Matrix(3,4), Matrix(5,5), Matrix(5,6), Matrix(6,7))

scala> matrixMult(matricesNotAllLiningUp)
res64: Option[Matrix] = None
```

Finally, we'll create **matrices**, a bunch of matrices that we'll multiply together:

```
…
val matrices = …
```

OK, the set up is finished. We have enough of a strawman in place that we can start multiplying them together in parallel. Next, we need to **break up the** `matrices` **sequence into groups—groups of five hundred** should do. **We'll then transform that sequence of groups into a sequence of** `futures` **that are performing the multiplications**.

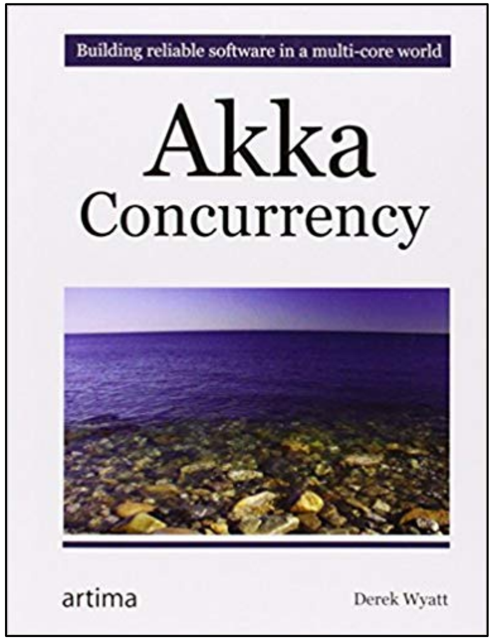
```
val futures = matrices.grouped(500).map { ms =>
  Future(matrixMult(ms))
}.toSeq
```

The `futures` value now contains **forty futures** that are all multiplying their sequence of matrices. If the `ExecutionContext` they're running on happens to have forty available threads, they'll all go in parallel; if not, they'll be executed in bits and pieces, but hopefully saturating your CPUs as much as possible.[5]

Once they're finished, we will have **forty resulting matrices** (or, if there was a dimensional problem somewhere, we'll have one or more **None** values). **We still need to multiply those forty together into one final matrix. This is the part that was so damn tricky with the actor-based solution. Questions arise**:

1. How do we know when they're all done?
2. In what order did the responses return?
3. How do we ensure we're multiplying the final forty in the right order?

None of those questions have any meaning anymore, now that we're using futures to do all of the bookkeeping for us. **We just have to transform the sequence of futures into a *single* future that holds a sequence of *results***. When we have that single future, we can easily transform the results using the `matrixMult` function we used to multiply all of the intermediates.



```scala
val multResultFuture = Future.sequence(futures) map { r =>
  matrixMult(r.flatten)
}
```

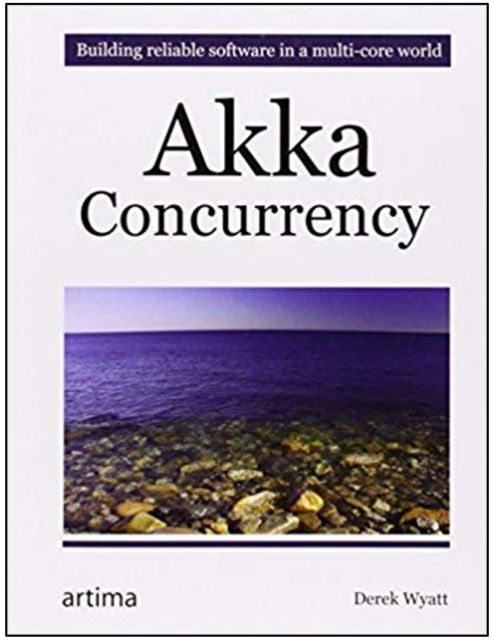
The only difference here is that the intermediate matrices aren't of type `Matrix` but of type `Option[Matrix]`, so we need to **`flatten`** them before sending them to **`matrixMult`**. However, that has nothing to do with the concurrency.

Now, `multResultFuture` holds a future `Option[Matrix]`, whose value we can grab in our usual way:

```scala
val finished = Await.result(multResultFuture, 1.second)
```

And we're done! You understand, of course, that **there was more code in that example to simply set up the problem and evaluate it than there was to actually perform the "complicated" parallelization, right?** If we did that with actors, the reverse would most certainly have been true!

If you need to scrape a bit of your brain off the floor and shove it back in through your ear due to the fact that your mind was just blown, I'll completely understand.



@derekwyatt

derekwyatt

Remember how the behaviour of **Option.sequence** (or **Either.sequence**) is affected by the presence of one or more **None** values (**Left** values) in the list being **sequenced**?

```
assert( sequence(List(Some(1),Some(2),Some(3))) == Some(List(1, 2, 3)) )
assert( sequence(List(Some(1),None,Some(3))) == None )
assert( sequence(List(None,None,None)) == None )
```

```
assert( sequence(List(Right(1),Right(2),Right(3))) == Right(List(1, 2, 3)) )
assert( sequence(List(Right(1),Left(-2),Right(3))) == Left(-2) )
assert( sequence(List(Left(0),Left(-1),Left(-2))) == Left(0) )
```

Does **Future.sequence** behave in a similar way? Let's see in the next slide.

@philip_schwarz

```scala
val futures: List[Future[Int]] = List( Future(2/2), Future(2/1) )

val futureList: Future[List[Int]] = Future.sequence(futures)

Await.ready(futureList, Duration.Inf)

assert(   futures.toString == "List(Future(Success(1)), Future(Success(2)))")
assert(futureList.toString == "Future(Success(List(1, 2)))")
```

if **all the futures in a list succeed**, then the future obtained by sequencing the list also **succeeds** and its value is a list.

behaviour of Future.sequence when one or more Futures fail

```scala
val futures: List[Future[Int]] = List(Future(2/2), Future(2/1), Future(2/0))

val futureList: Future[List[Int]] = Future.sequence(futures)

Await.ready(futureList, Duration.Inf)

val expectedFutures = List(
  "Future(Success(1))",
  "Future(Success(2))",
  "Future(Failure(java.lang.ArithmeticException: / by zero))")

assert(   futures.toString == expectedFutures.toString)
assert(futureList.toString == "Future(Failure(java.lang.ArithmeticException: / by zero))")
```

if **one of the futures in a list fails** due to some **exception**, then the future obtained by sequencing the list also **fails** due to that same **exception**

@philip_schwarz

```scala
val futures: List[Future[Int]] = List( Future(Nil(5)), Future(1/0), Future("".toInt) )

val futureList: Future[List[Int]] = Future.sequence(futures)

Await.ready(futureList, Duration.Inf)

val expectedFutures = List(
  "Future(Failure(java.lang.IndexOutOfBoundsException: 5))",
  "Future(Failure(java.lang.ArithmeticException: / by zero))",
  "Future(Failure(java.lang.NumberFormatException: For input string: \"\"))" )

assert(   futures.toString == expectedFutures.toString)
assert(futureList.toString == "Future(Failure(java.lang.IndexOutOfBoundsException: 5))")
```

if **multiple futures in a list fail** due to **exceptions**, then the future obtained by sequencing the list **fails** due to the same **exception** that caused the first of the multiple futures to **fail**.

Example – calling web services to create a List[Future[String]] and then seqencing it into a Future[List[String]]

```scala
private val postCodesServiceUrl: String = "http://api.postcodes.io/postcodes/<postCode>"
private val worldClockWebServiceUrl: String = "http://worldclockapi.com/api/json/gmt/now"
private val darkSkyWebServiceUrl: String = "https://api.darksky.net/forecast/0137524efeb07e2938ed5b3d200e92c2/<lat>,<long>"
private val internetChuckNorrisDatabaseUrl: String = "https://api.icndb.com/jokes/random"
```

```
# e.g. http://localhost:9000/info?postCode=tw181ql
GET     /info     controllers.DateTimeWeatherJokeController.info(postCode: String)
```

```scala
def info(postCode: String) = Action.async {

  val futureLatAndLong: Future[(String,String)] = getLatAndLongFromPostCodeService(postCode)
  val futureDateAndTime: Future[String] = getDateAndTimeFromWorldClockService
  val futureRandomJoke: Future[String] = getRandomJokeFromChuckNorrisDatabaseService

  val futureWeather: Future[String] = for {
    (lat,long) <- futureLatAndLong
    weather <- getWeatherFactsFromDarkSkyService(lat, long)
  } yield weather

  val listOfFutureItems: List[Future[String]] = List(futureDateAndTime, futureWeather, futureRandomJoke)
  val futureListOfItems: Future[List[String]] = Future.sequence(listOfFutureItems)

  val futurePageContent = futureListOfItems.map {
    _.mkString("\n") }.recover { case error: Throwable => s"The following error was encountered: ${error.getMessage}" }

  futurePageContent map { Ok(_) }
}
```

list of futures

Future.sequence

future of list

localhost:9000/info?postCode=tw181ql

```
Date and Time: 2018-12-16T12:33+00:00
Weather: partly cloudy with a temperature of 48.57 fahrenheit
Random Chuck Norris Joke: Once Chuck Norris signed a cheque and the bank bounced.
```

All the futures in List(futureDateAndTime, futureWeather, futureRandomJoke) are successful, so sequencing the list results in a future list whose values we display.

Example of what happens when things go wrong: erroneous host

```scala
private val worldClockWebServiceUrl: String = "http://worldclockapiz.com/api/json/gmt/now"
```

```scala
private def getDateAndTimeFromWorldClockService: Future[String] =
  for {
    response: JsValue <- getDataFromWebService(worldClockWebServiceUrl)
    dateAndTime: String = getCurrentDateTimeFrom(response)
  } yield dateAndTime


private def getDataFromWebService(url: String): Future[JsValue] =
  wsClient.url(url).get.map { response =>
    response.status match {
      case 200 => response.body[JsValue]
      case code => throw new RuntimeException(s"${url} responded with ${code}")
    }
  }
```

The future returned by **getDataFromWebService** fails due to an exception.

This failed future is the first one in List(futureDateAndTime, futureWeather, futureRandomJoke).

Sequencing the list results in a failed future, so we inform the user by showing them the message of the exception.

localhost:9000/info?postCode=tw181ql

The following error was encountered: worldclockapiz.com: nodename nor servname provided, or not known

Example of what happens when things go wrong: erroneous API

```scala
private val worldClockWebServiceUrl: String = "http://worldclockapi.com/api/jsonz/gmt/now"
```

```scala
private def getDateAndTimeFromWorldClockService: Future[String] =
  for {
    response: JsValue <- getDataFromWebService(worldClockWebServiceUrl)
    dateAndTime: String = getCurrentDateTimeFrom(response)
  } yield dateAndTime


private def getDataFromWebService(url: String): Future[JsValue] =
  wsClient.url(url).get.map { response =>
    response.status match {
      case 200 => response.body[JsValue]
      case code => throw new RuntimeException(s"${url} responded with ${code}")
    }
  }
```

The future returned by **getDataFromWebService** is successful, but its status is 404, so we throw an exception, which causes the first future in List(futureDateAndTime, futureWeather, futureRandomJoke), to fail.

Sequencing the list results in a failed future and so we inform the user by showing them the message of the exception.

localhost:9000/info?postCode=tw181ql

The following error was encountered: http://worldclockapi.com/api/jsonz/gmt/now responded with 404

Example of what happens when things go wrong: looking for nonexistent field in response

```scala
private def getDateAndTimeFromWorldClockService: Future[String] =
  for {
    response: JsValue <- getDataFromWebService(worldClockWebServiceUrl)
    dateAndTime: String = getCurrentDateTimeFrom(response)
  } yield dateAndTime

private def getCurrentDateTimeFrom(jsonValue: JsValue): String = {
  val dateAndTime = validateString(jsonValue \ "currentDateTimez")
  s"Date and Time: $dateAndTime"
}

private def validateString(result: JsLookupResult): String =
  result.validate[String] match {
    case successfulParsingResult: JsSuccess[String] =>
      successfulParsingResult.get
    case erroneousParsingResult: JsError =>
      s"Error accessing field:${erroneousParsingResult.toString}"
  }
```

We fail to extract dateAndTime from the response of **worldClockWebServiceUrl**, but all the futures in List(futureDateAndTime, futureWeather, futureRandomJoke) are successful, because we replace the missing dateAndTime with an error message.

Sequencing the list of futures results in a future list and when we display the values in the list, the first one contains the error message.

localhost:9000/info?postCode=tw181ql

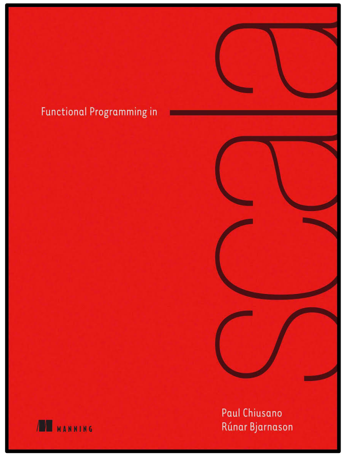
```
Date and Time: Error accessing field:JsError(List((,List(JsonValidationError(List('currentDateTimez' is undefined on object: {
Weather: partly cloudy with a temperature of 48.51 fahrenheit
Random Chuck Norris Joke: Chuck Norris is the only human being to display the Heisenberg uncertainty principle - you can never
```

12.1 Generalizing monads

By now we've seen various operations, like **sequence** and **traverse**, implemented many times for different **monads**, and in the last chapter we generalized the implementations to work for any **monad** F
…
Here, the implementation of **traverse** is using **map2** and **unit**, and we've seen that **map2** can be implemented in terms of **flatMap**:
…

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

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

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

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

  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C] =
    flatMap(fa)(a => map(fb)(b => f(a, b)))

  def sequence[A](lfa: List[F[A]]): F[List[A]] =
    traverse(lfa)(fa => fa)

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

What you may not have noticed is that a large number of the useful combinators on **Monad** can be defined using only **unit** and **map2**. The **traverse** combinator is one example—it doesn't call **flatMap** directly and is therefore agnostic to whether **map2** is primitive or derived. Furthermore, for many data types, **map2** can be implemented directly, without using **flatMap**.



FP in Scala

All this suggests a variation on **Monad**—the **Monad** interface has **flatMap** and **unit** as primitives, and derives **map2**, but **we can obtain a different abstraction by letting unit and map2 be the primitives**. We'll see that this new abstraction, called an **applicative functor**, is less powerful than a **monad**, but we'll also see that limitations come with benefits.

Defining Applicative in terms of primitive combinators map2 and unit

12.2 The Applicative trait

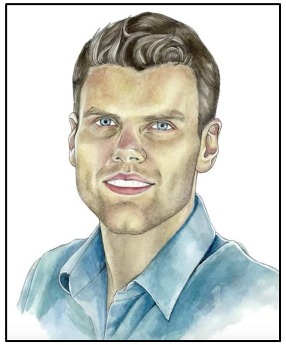
Applicative functors can be captured by a new interface, `Applicative`, in which `map2` and `unit` are primitives.

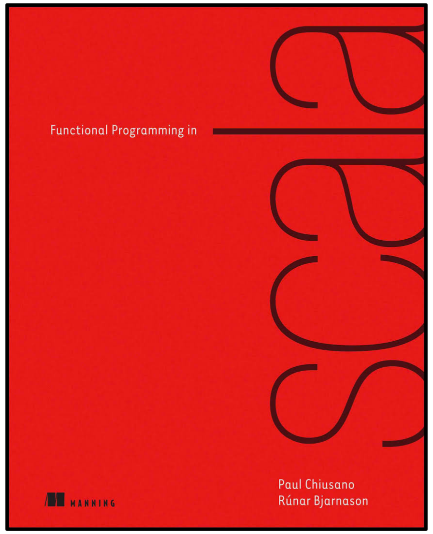
```
trait Applicative[F[_]] extends Functor[F] {

  // 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`.

Definition of `traverse` is identical.

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





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

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.

12.6 Traversable functors

We discovered **applicative functors** by noticing that our **traverse** and **sequence** functions (and several other operations) didn't depend directly on **flatMap**. **We can spot another abstraction by generalizing traverse and sequence once again**. Look again at the signatures of **traverse** and **sequence**:

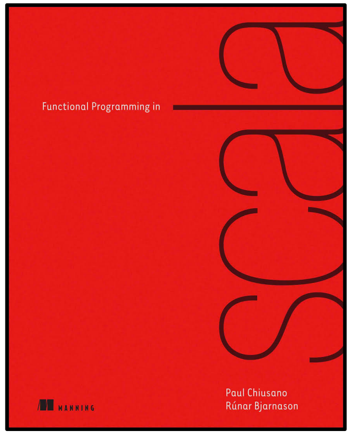
```
def traverse[F[_],A,B](as: List[A])(f: A => F[B]): F[List[B]]
def sequence[F[_],A](fas: List[F[A]]): F[List[A]]
```

Any time you see a concrete type constructor like `List` showing up in an abstract interface like `Applicative`, you may want to ask the question, **"What happens if I abstract over this type constructor?"** Recall from chapter 10 that **a number of data types other than `List` are `Foldable`. Are there data types other than `List` that are traversable? Of course!**

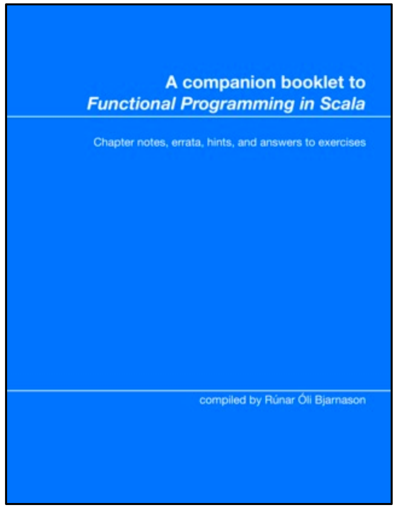
EXERCISE 10.12

On the `Applicative` trait, implement `sequence` over a `Map` rather than a `List`:

```
def sequenceMap[K,V](ofa: Map[K,F[V]]): F[Map[K,V]]
```



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



by Runar Bjarnason @runarorama

```
def sequenceMap[K,V](ofa: Map[K,F[V]]): F[Map[K,V]] =
  (ofa foldLeft unit(Map.empty[K,V])) {
    case (acc, (k, fv)) =>
      map2(acc, fv)((m, v) => m + (k -> v))
  }
```

An example of using **sequenceMap**, is not included. See the next slide for one.

@philip_schwarz

Example
using the **sequenceMap** function of an `Applicative[Option]`

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

  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C]
  def unit[A](a: => A): F[A]

  def map[A,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, mbs) => map2(f(a), mbs)(_ :: _))

  def sequence[A](lfa: List[F[A]]): F[List[A]] =
    lfa.foldRight(unit(List[A]()))((fa, lfa) => map2(fa, lfa)(_ :: _))

  def sequenceMap[K,V](ofa: Map[K,F[V]]): F[Map[K,V]] =
    (ofa foldLeft unit(Map.empty[K,V])) {
      case (acc, (k, fv)) =>
        map2(acc, fv)((m, v) => m + (k -> v))
    }
}
```

sequenceMap is defined using **Map.foldLeft**:

`foldLeft[B](z: B)(op: (B, A) => B): B`

Here are two simple, contrived examples of using **Map.foldLeft**. In the first one we reduce a **Map** to the **sum of its values**. In the second one we reduce it to the **concatenation of its keys**.

```
assert( Map("1" -> 1, "2" -> 2).foldLeft(0){ case (acc, (k, v)) => acc + v } == 3 )
assert( Map("1" -> 1, "2" -> 2).foldLeft(""){ case (acc, (k, v)) => acc + k } == "12" )
```

```
val optionApplicative = new Applicative[Option] {

  def map2[A, B, C](fa: Option[A], fb: Option[B])(f: (A, B) => C): Option[C] =
    (fa, fb) match {
      case (Some(a), Some(b)) => Some(f(a,b))
      case _ => None
    }

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

And here is an example creating an **optionApplicative** and using its **sequenceMap** function to turn a `Map[String,Option[Int]]` into an `Option[Map[String,Int]]`.

Note how just like when **sequencing** a **List** of **Option**, if there are any **None** values then the result of the **sequencing** is **None**.

@philip_schwarz

```
assert( optionApplicative.sequenceMap(Map("1" -> Some(1), "2" -> Some(2))) == Some(Map("1" -> 1, "2" -> 2)) )
assert( optionApplicative.sequenceMap(Map("1" -> Some(1), "x" -> None)) == None )
```

But traversable data types are too numerous for us to write specialized `sequence` and `traverse` methods for each of them. **What we need is a new interface**. We'll call it `Traverse`:[6]

```
trait Traverse[F[_]] {

  def traverse[M[_]:Applicative,A,B](fa: F[A])(f: A => M[B]): M[F[B]]
    sequence(map(fa)(f))

  def sequence[M[_]:Applicative,A](fma: F[M[A]]): M[F[A]] =
    traverse(fma)(ma => ma)
}
```

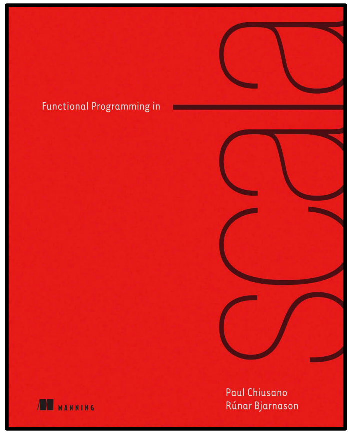
[6] The name `Traversable` is already taken by an unrelated trait in the Scala standard library.

The interesting operation here is `sequence`. Look at its signature closely. It takes `F[G[A]]` and swaps the order of `F` and `G`, so long as `G` is an applicative functor. Now, this is a rather abstract, algebraic notion. We'll get to what it all means in a minute, but first, let's look at a few instances of `Traverse`.

EXERCISE 12.13

Write `Traverse` instances for `List`, `Option`, and `Tree`.

```
case class Tree[+A](head: A, tail: List[Tree[A]])
```



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

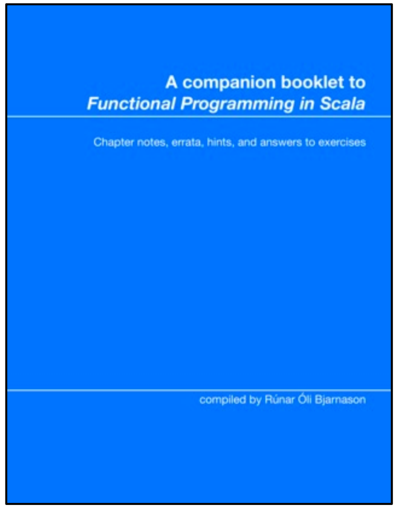
If you are confused by the fact that Traverse's traverse function uses a map function that is not defined anywhere, then you are not alone, but don't worry: we'll find out more about Traverse's map function very soon.

In the meantime, think of that traverse method as not having a body, i.e. being abstract. I reckon the body is there to point out that if Traverse did have a map function then it could be used to implement traverse.

In the next slide we'll see examples of Traverse instances that implement traverse without using such a map function.

Answer to **Exercise 12.13**: Write **Traverse** instances for **List**, **Option**, and **Tree**.

```scala
val listTraverse = new Traverse[List] {
  override def traverse[M[_],A,B](as: List[A])(f: A => M[B])(implicit M: Applicative[M]): M[List[B]] =
    as.foldRight(M.unit(List[B]()))((a, fbs) => M.map2(f(a), fbs)(_ :: _))
}

val optionTraverse = new Traverse[Option] {
  override def traverse[M[_],A,B](oa: Option[A])(f: A => M[B])(implicit M: Applicative[M]): M[Option[B]] =
    oa match {
      case Some(a) => M.map(f(a))(Some(_))
      case None    => M.unit(None)
    }
}

case class Tree[+A](head: A, tail: List[Tree[A]])

val treeTraverse = new Traverse[Tree] {
  override def traverse[M[_],A,B](ta: Tree[A])(f: A => M[B])(implicit M: Applicative[M]): M[Tree[B]] =
    M.map2(f(ta.head), listTraverse.traverse(ta.tail)(a => traverse(a)(f)))(Tree(_, _))
}
```



by Runar Bjarnason @runarorama

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

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

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

  def map[A,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,mbs) => map2(f(a),mbs)(_::_))

  def sequence[A](lfa: List[F[A]]): F[List[A]] =
    traverse(lfa)(fa => fa)
}
```

```scala
trait Traverse[F[_]] {

  def traverse[M[_]:Applicative,A,B](fa: F[A])(f: A => M[B]): M[F[B]]

  def sequence[M[_]:Applicative,A](fma: F[M[A]]): M[F[A]] =
    traverse(fma)(ma => ma)
}
```

```scala
trait Functor[F[_]] {

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

}
```

In the next four slides we are goint to try out the three traversable instances we have just seen: `listTraverse`, `optionTraverse` and `treeTraverse`.

@philip_schwarz

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

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

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

  def map[A,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,mbs) => map2(f(a),mbs)(_::_))

  def sequence[A](lfa: List[F[A]]): F[List[A]] =
    traverse(lfa)(fa => fa)
}
```

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

Sample usage of a Traverse[List] with an Applicative[Option]

Going from List[Option] to Option[List]

```scala
implicit val optionApplicative = new Applicative[Option] {

  def map2[A, B, C](fa: Option[A], fb: Option[B])(f: (A, B) => C): Option[C] =
    (fa, fb) match {
      case (Some(a), Some(b)) => Some(f(a,b))
      case _ => None
    }

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

```scala
trait Traverse[F[_]] {
  def traverse[M[_]:Applicative,A,B](fa: F[A])(f: A => M[B]): M[F[B]]
  def sequence[M[_]:Applicative,A](fma: F[M[A]]): M[F[A]] =
    traverse(fma)(ma => ma)
}
```

```scala
import scala.util.{Try,Success,Failure}
val parseInt:String=>Option[Int] = (s:String) => Try(s.toInt) match {
  case Success(n) => Option(n)
  case Failure(_) => None
 }
```

```scala
val listTraverse = new Traverse[List] {
  override def traverse[M[_],A,B](as: List[A])(f: A => M[B])(implicit M: Applicative[M]): M[List[B]] =
    as.foldRight(M.unit(List[B]()))((a, fbs) => M.map2(f(a), fbs)(_ :: _))
}
```

The function we use to traverse the list is the optionApplicative's own unit function, which just lifts its argument into an Option, so the result is always an Option of the original list.

```scala
assert( listTraverse.sequence(List(Option("a"), Option("b"), Option("c"))) == Option(List("a", "b", "c")) )

assert( listTraverse.traverse(List("a", "b", "c"))(optionApplicative.unit(_)) == Option(List("a", "b", "c")) )

assert( listTraverse.sequence(List(Option("1"), Option("2"), Option("3"))) == Option(List("1", "2", "3")) )

assert( listTraverse.traverse(List("1", "2", "3"))(parseInt) == Option(List(1, 2, 3)) )

assert( listTraverse.sequence(List(Option("1"), None, Option("3"))) == None )

assert( listTraverse.traverse(List("1", "x", "3"))(parseInt) == None )
```

If all the strings in the list can be parsed into integers then the result of traversing the list is an Option of a list of the parsed integers.
If any of the strings in the list cannot be parsed into an integer then the result of traversing is None.

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

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

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

  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,mbs) => map2(f(a),mbs)(_::_))

  def sequence[A](lfa: List[F[A]]): F[List[A]] =
    traverse(lfa)(fa => fa)
}
```

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

Sample usage of a Traverse[Option] with an Applicative[List]

Going from Option[List] to List[Option]

```scala
implicit val listApplicative = new Applicative[List] {
 def map2[A, B, C](fa: List[A], fb: List[B])(f: (A, B) => C): List[C] =
    for {
      a <- fa
      b <- fb
    } yield f(a,b)

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

```scala
trait Traverse[F[_]] {
  def traverse[M[_]:Applicative,A,B](fa: F[A])(f: A => M[B]): M[F[B]]
  def sequence[M[_]:Applicative,A](fma: F[M[A]]): M[F[A]] =
    traverse(fma)(ma => ma)
}
```

```scala
assert( listApplicative.map2(List(1,2,3),List(3,2,1))(_ + _) == List(4,3,2,5,4,3,6,5,4) )
assert( listApplicative.map2(List(1,2,3),List(3,2))(_ + _) == List(4,3,5,4,6,5) )
assert( listApplicative.map2(List(1,2,3),List())(_ + _) == List() )
```

Sample behaviour of the map2 method of this instance of Applicative[List].

```scala
val optionTraverse = new Traverse[Option] {
  override def traverse[M[_],A,B](oa: Option[A])(f: A => M[B])(implicit M: Applicative[M]): M[Option[B]] =
    oa match {
      case Some(a) => M.map(f(a))(Some(_))
      case None    => M.unit(None)
    }
}
```

```scala
val toChars : String => List[Char] = s => s.toList

assert( optionTraverse.traverse(Option("123"))(toChars) == List(Some('1'),Some('2'), Some('3')) )
assert( optionTraverse.sequence(Some(List('1','2','3'))) == List(Some('1'),Some('2'), Some('3')) )
```

The map2 function of this instance of Applicative[List] works well with this instance of Traverse[Option]. It causes optionTraverse's traverse to apply 'Some' to every list element.

See next slide for what happens when we use a different instance of Applicative[List].

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

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

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

  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,mbs) => map2(f(a),mbs)(_::_))

  def sequence[A](lfa: List[F[A]]): F[List[A]] =
    traverse(lfa)(fa => fa)
}
```

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

Sample usage of a Traverse[Option] with an Applicative[List]

Going from Option[List] to List[Option]

```scala
implicit val listApplicative = new Applicative[List] {
  def map2[A, B, C](fa: List[A], fb: List[B])(f: (A, B) => C): List[C] =
    (fa, fb) match {
      case (Nil,_) => Nil
      case (_,Nil) => Nil
      case (ha::ta,hb::tb) => f(ha,hb) :: map2(ta,tb)(f)
    }

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

```scala
trait Traverse[F[_]] {
  def traverse[M[_]:Applicative,A,B](fa: F[A])(f: A => M[B]): M[F[B]]
  def sequence[M[_]:Applicative,A](fma: F[M[A]]): M[F[A]] =
    traverse(fma)(ma => ma)
}
```

This slide differs from the previous one in that we use a different instance of Applicative[List], which results in the sequence and traverse functions of optionTraverse behaving differently

```scala
assert( listApplicative.map2(List(1,2,3),List(3,2,1))(_ + _) == List(4,4,4) )
assert( listApplicative.map2(List(1,2,3),List(3,2))(_ + _) == List(4,4) )
assert( listApplicative.map2(List(1,2,3),List())(_ + _) == List() )
```

Sample behaviour of the map2 method of this instance of Applicative[List].

```scala
val optionTraverse = new Traverse[Option] {
  override def traverse[M[_],A,B](oa: Option[A])(f: A => M[B])(implicit M: Applicative[M]): M[Option[B]] =
    oa match {
      case Some(a) => M.map(f(a))(Some(_))
      case None    => M.unit(None)
    }
}
```

```scala
val toChars : String => List[Char] = s => s.toList

assert( optionTraverse.traverse(Option("123"))(toChars) == List(Some('1')) )
assert( optionTraverse.sequence(Some(List('1','2','3'))) == List(Some('1')) )
```

The map2 function of this instance of Applicative[List] does not work well with this instance of Traverse[Option]. It causes optionTraverse's traverse to apply 'Some' to only the first element of the list.

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

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

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

  def map[A,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,mbs) => map2(f(a),mbs)(_::_))

  def sequence[A](lfa: List[F[A]]): F[List[A]] =
    traverse(lfa)(fa => fa)
}
```

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

Sample usage of a Traverse[Tree] with an Applicative[Option].

Going from Tree[Option] to Option[Tree]

```scala
implicit val optionApplicative = new Applicative[Option] {

  def map2[A, B, C](fa: Option[A], fb: Option[B])(f: (A, B) => C): Option[C] =
    (fa, fb) match {
      case (Some(a), Some(b)) => Some(f(a,b))
      case _ => None
    }

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

```scala
trait Traverse[F[_]] {
  def traverse[M[_]:Applicative,A,B](fa: F[A])(f: A => M[B]): M[F[B]]
  def sequence[M[_]:Applicative,A](fma: F[M[A]]): M[F[A]] =
    traverse(fma)(ma => ma)
}
```

```scala
import scala.util.{Try,Success,Failure}
val parseInt:String=>Option[Int] = (s:String) => Try(s.toInt) match {
  case Success(n) => Option(n)
  case Failure(_) => None
}
```

```scala
val treeTraverse = new Traverse[Tree] {
  override def traverse[M[_],A,B](ta: Tree[A])(f: A => M[B])(implicit M: Applicative[M]): M[Tree[B]] =
    M.map2(f(ta.head), listTraverse.traverse(ta.tail)(a => traverse(a)(f)))(Tree(_, _))
}
val listTraverse = new Traverse[List] {
  override def traverse[M[_],A,B](as: List[A])(f: A => M[B])(implicit M: Applicative[M]): M[List[B]] =
    as.foldRight(M.unit(List[B]()))((a, fbs) => M.map2(f(a), fbs)(_ :: _))
}
```

Note that treeTraverse uses listTraverse!

```scala
assert(treeTraverse.traverse(Tree("1", List( Tree("2", Nil), Tree("3", Nil))))(parseInt) == Some(Tree(1, List( Tree(2, Nil), Tree(3, Nil)))))

assert(treeTraverse.sequence(Tree(Option(1),List(Tree(Option(2),Nil),Tree(Option(3),Nil)))) == Option(Tree(1,List(Tree(2,Nil),Tree(3,Nil)))))

assert(treeTraverse.traverse(Tree("1", List( Tree("x", Nil), Tree("3", Nil))))(parseInt) == None)

assert(treeTraverse.sequence(Tree(Option(1), List( Tree(None, Nil), Tree(Option(3), Nil)))) == None)
```

To be continued in part III