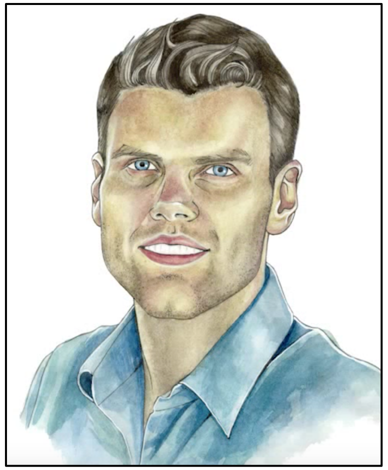
Sequence and Traverse

Part 1

learn about the sequence and traverse functions
through the work of



Runar Bjarnason
@runarorama

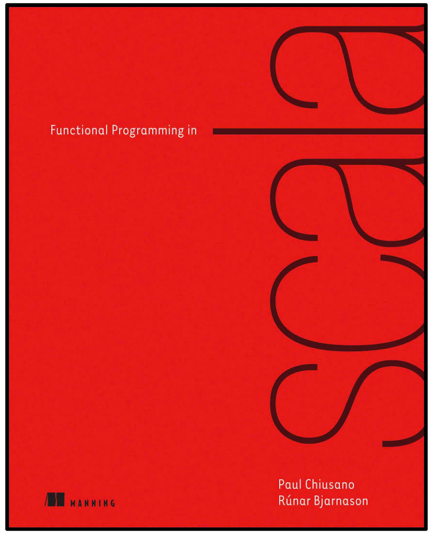
Paul Chiusano
@pchiusano

slides by

@philip_schwarz

There turns out to be a startling number of operations that can be defined in the most general possible way in terms of `sequence` and/or `traverse`

@runarorama



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

@pchiusano

Rúnar
@runarorama
Following

There are two basic answers to "how do I" questions in Scala. One is "don't do that". The other is "traverse".

8:07 am - 17 May 2017

One of my favourite topics for motivating usage of something like the **Cats** library is this small thing called **Traverse**.

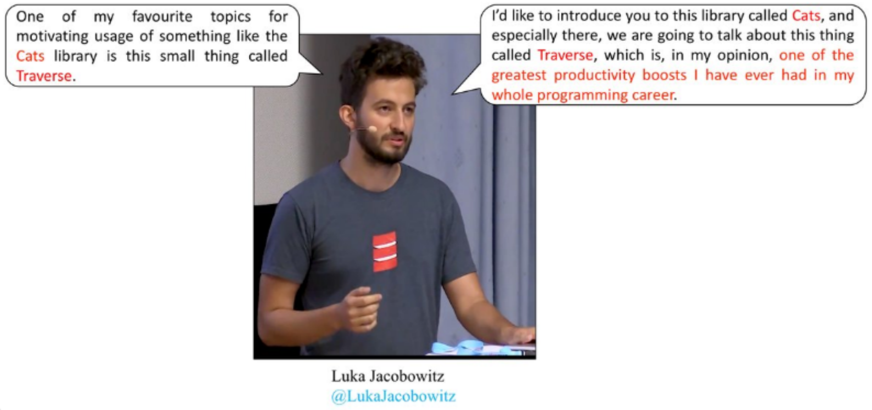
I'd like to introduce you to this library called **Cats**, and especially there, we are going to talk about this thing called **Traverse**, which is, in my opinion, **one of the greatest productivity boosts I have ever had in my whole programming career**.

Luka Jacobowitz
@LukaJacobowitz

YouTube Oh, all the things you'll traverse

Philip Schwarz @philip_schwarz · Sep 28

current status: watching "Oh, all the things you'll traverse" by @LukaJacobowitz
youtube.com/watch?v=GhLqTZ… youtube.com/watch?v=yEYPf4…



1 2 19

Jacob Wang
@jatcwang

Follow

Replying to @philip_schwarz @LukaJacobowitz

The answer to all the questions in life is not 42. It's Traverse.

3:03 AM - 29 Sep 2018

5 Retweets **17** Likes

5 17

John Ⓐ De Goes
@jdegoes

Following

It's very common to loop over a collection, perform an action for each element, and collect the results:

```
var results = []
for (var i = 0; i < col.length; i++) {
  results.push(performTask(col[i]))
}
```

In FP, we use `traverse` to achieve the same result:

```
col.traverse(performTask)
```

8:13 am - 10 Oct 2018

Introducing the sequence function

```
def sequence[A](a: List[Option[A]]): Option[List[A]]
```

Combines a list of **Options** into one **Option** containing a list of all the **Some** values in the original list. If the original list contains **None** even once, the result of the function is **None**; otherwise the result is **Some** with a list of all the values.

if the list is empty then the result is **Some** empty list

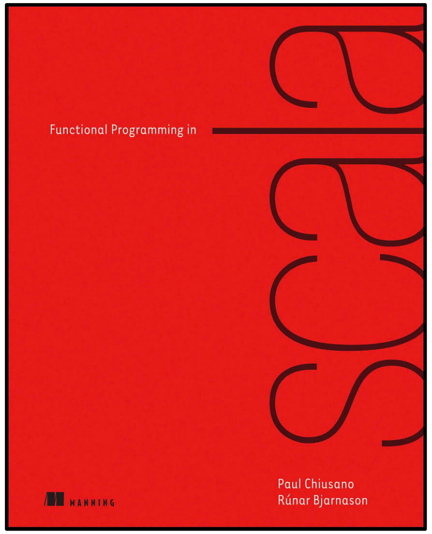
```
assert(    sequence(Nil) == Some(List()) )
assert( sequence(List()) == Some(List()) )
```

if the list contains all **Some** values then the result is **Some** list

```
assert(                    sequence(List(Some(1))) == Some(List(1))       )
assert(            sequence(List(Some(1),Some(2))) == Some(List(1, 2))    )
assert( sequence(List(Some(1),Some(2),Some(3))) == Some(List(1, 2, 3)) )
```

if the list contains any **None** value then the result is **None**

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



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

Implementing the sequence function

Here's an explicit recursive version:

```scala
def sequence[A](a: List[Option[A]]): Option[List[A]] = a match {
  case Nil => Some(Nil)
  case h :: t => h flatMap (hh => sequence(t) map (hh :: _))
}
```

It can also be implemented using **foldRight** and **map2**

```scala
def sequence[A](a: List[Option[A]]): Option[List[A]] =
  a.foldRight[Option[List[A]]](Some(Nil))((h,t) => map2(h,t)(_ :: _))
```

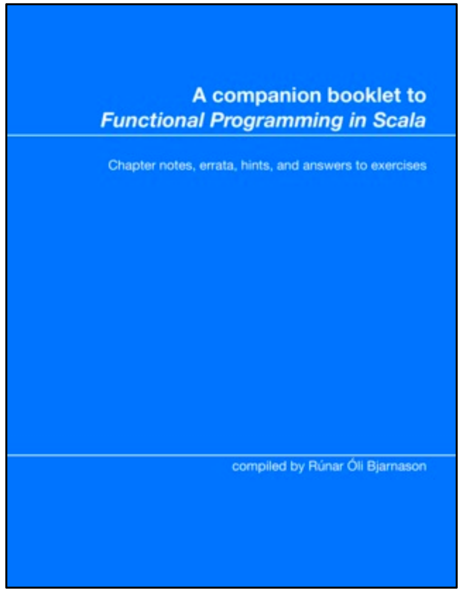
map2 being defined using **flatMap** and **map** (for example)

```scala
def map2[A,B,C](oa: Option[A], ob: Option[B])(f: (A, B) => C): Option[C] =
  oa flatMap (a => ob map (b => f(a, b)))
```

or what is equivalent, using a **for comprehension**

```scala
def map2[A,B,C](oa: Option[A], ob: Option[B])(f: (A, B) => C): Option[C] =
  for {
    a <- oa
    b <- ob
  } yield f(a, b)
```

```scala
assert(             map2(Some(3),Some(5))(_ + _) == Some(8) )
assert( map2(Some(3),Option.empty[Int])(_ + _) == None    )
assert( map2(Option.empty[Int],Some(5))(_ + _) == None    )
```



by Runar Bjarnason
@runarorama

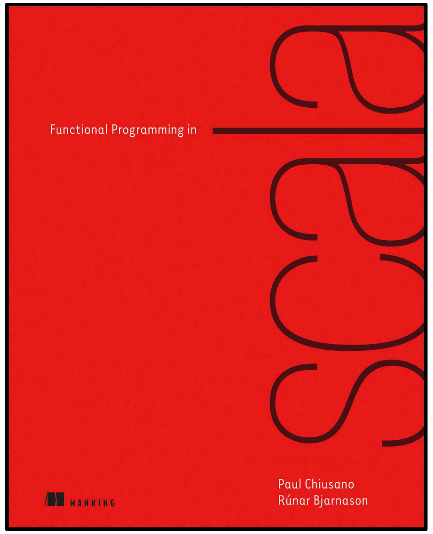
A simple example of using the sequence function

Sometimes we'll want to map over a list using a function that might fail, returning None if applying it to any element of the list returns None.

For example, what if we have a whole list of String values that we wish to parse to Option[Int]? In that case, we can simply sequence the results of the map.

```
import scala.util.Try

def parseIntegers(a: List[String]): Option[List[Int]] =
  sequence(a map (i => Try(i.toInt).toOption))
```

```
assert( parseIntegers(List("1", "2", "3"))   == Some(List(1, 2, 3) )
assert( parseIntegers(List("1", "x", "3"))   == None               )
assert( parseIntegers(List("1", "x", "1.2")) == None               )
```



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

Introducing the traverse function

```scala
def parseIntegers(a: List[String]): Option[List[Int]] =
  sequence(a map (i => Try(i.toInt).toOption))
```

Wanting to **sequence** the results of a **map** this way is **a common enough occurrence to warrant a new generic function**, **traverse**

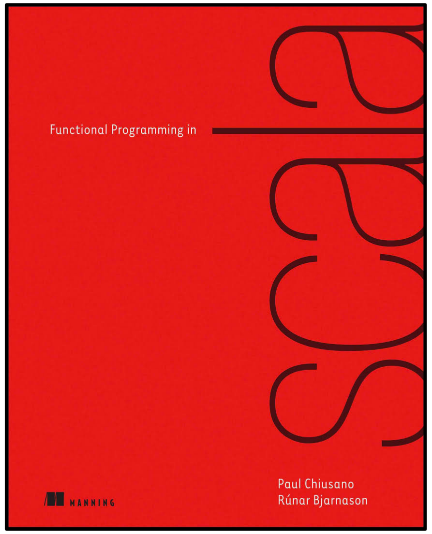
```scala
def traverse[A, B](a: List[A])(f: A => Option[B]): Option[List[B]] = ???
```

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

It is straightforward to implement traverse using **map** and **sequence**

```scala
def traverse[A, B](a: List[A])(f: A => Option[B]): Option[List[B]] =
  sequence(a map f)
```

But this is **inefficient**, since it **traverses the list twice**, first to convert each `String` to an `Option[Int]`, and a second pass to combine these `Option[Int]` values into an `Option[List[Int]]`.



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

Better implementations of the traverse function

Here's an explicit recursive implementation using **map2**:

```
def traverse[A, B](a: List[A])(f: A => Option[B]): Option[List[B]] =
  a match {
    case Nil => Some(Nil)
    case h::t => map2(f(h), traverse(t)(f))(_ :: _)
  }
```

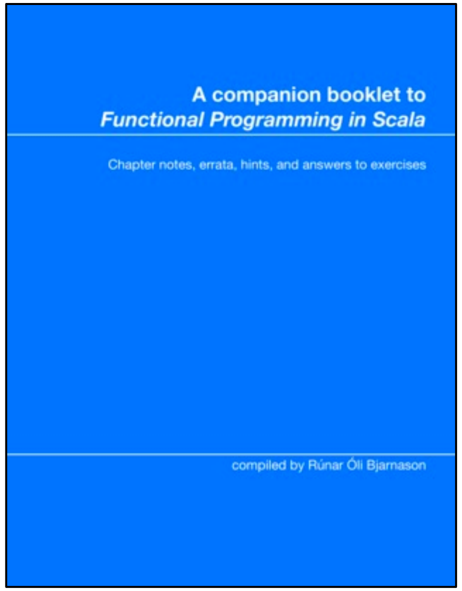
And here is a non-recursive implementation using **foldRight** and **map2**

```
def traverse[A, B](a: List[A])(f: A => Option[B]): Option[List[B]] =
  a.foldRight[Option[List[B]]](Some(Nil))((h,t) => map2(f(h),t)(_ :: _))
```

Just in case it helps, let's compare the implementation of **traverse** with that of **sequence**

```
def sequence[A](a: List[Option[A]]): Option[List[A]] =
  a.foldRight[Option[List[A]]](Some(Nil))((h,t) => map2(h,t)(_ :: _))
```

```
def traverse[A, B](a: List[A])(f: A => Option[B]): Option[List[B]] =
  a.foldRight[Option[List[B]]](Some(Nil))((h,t) => map2(f(h),t)(_ :: _))
```



by Runar Bjarnason
@runarorama

The close relationship between sequence and traverse

Just like it is possible to define **traverse** in terms of **sequence**

```
def traverse[A, B](a: List[A])(f: A => Option[B]): Option[List[B]] =
  sequence(a map f)
```

While this implementation of traverse is inefficient, it is still useful to think of **traverse** as first **map** and then **sequence**.

It is possible to define **sequence** in terms of **traverse**

```
def sequence[A](a: List[Option[A]]): Option[List[A]] =
  traverse(a)(x => x)
```

@philip_schwarz

Actually, it turns out that there is a similar relationship between **monadic combinators** **flatMap** and **flatten** (see next two slides).

Recall two of the three minimal sets of **combinators** that can be used to define a **monad**.
One set includes **flatMap** and the other includes **join** (in **Scala** this is also known as **flatten**)

@philip_schwarz

flatmap + unit

```scala
trait Monad[F[_]] {
  def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]
  def unit[A](a: => A): F[A]
}
```

The **flatMap** function takes an F[A] and a function from A to F[B] and returns an F[B]

```scala
def flatMap[A,B](ma: F[A])(f: A ⇒ F[B]): F[B]
```

What it does is apply to each A element of ma a function f producing an F[B], but instead of returning the resulting F[F[B]], it flattens it and returns an F[B].

map + **join** + unit

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

trait Monad[F[_]] extends Functor[F] {
  def join[A](mma: F[F[A]]): F[A]
  def unit[A](a: => A): F[A]
}
```

The **join** function takes an F[F[A]] and returns an F[A]

```scala
def join[A](mma: F[F[A]]): F[A]
```

What it does is “remove a layer” of F.

It turns out that **flatMap** can be defined in terms of **map** and **flatten**

```
def flatMap[A,B](ma: F[A])(f: A ⇒ F[B]): F[B] = flatten(ma map f)
```

and **flatten** can be defined in terms of **flatMap**

```
def flatten[A](mma: F[F[A]]): F[A] = flatMap(mma)(x ⇒ x)
```

So **flatMapping a function is just mapping the function first and then flattening the result**

and **flattening is just flatMapping the identity function** `x => x`

Now recall that **traverse** can be defined in terms of **map** and **sequence**:

```
def traverse[A, B](a: List[A])(f: A => Option[B]): Option[List[B]] = sequence(a map f)
```

and **sequence** can be defined in terms of **traverse**

```
def sequence[A](a: List[Option[A]]): Option[List[A]] = traverse(a)(x ⇒ x)
```

So here is the similarity

@philip_schwarz

flatMapping is mapping and then flattening - flattening is just flatMapping identity

traversing is mapping and then sequencing - sequencing is just traversing with identity

@philip_schwarz

But there is another similarity: **sequence** and **join** are both **natural transformations**.

Before we can look at why that is the case, let's quickly recap what a **natural transformation** is.

See the following for a less hurried introduction to natural transformations:

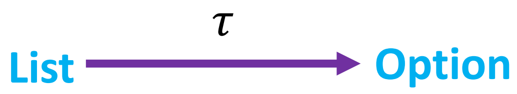
https://www.slideshare.net/pjschwarz/natural-transformations

slideshare @philip_schwarz

Concrete Scala Example: safeHead - natural transformation τ from List functor to Option functor

```scala
val length: String => Int = s => s.length

// a natural transformation
def safeHead[A]: List[A] => Option[A] = {
  case head::_ => Some(head)
  case Nil => None
}
```

natural transformation τ from List to Option



F[A] is type A lifted into context F
$f_{\uparrow F}$ is function f lifted into context F

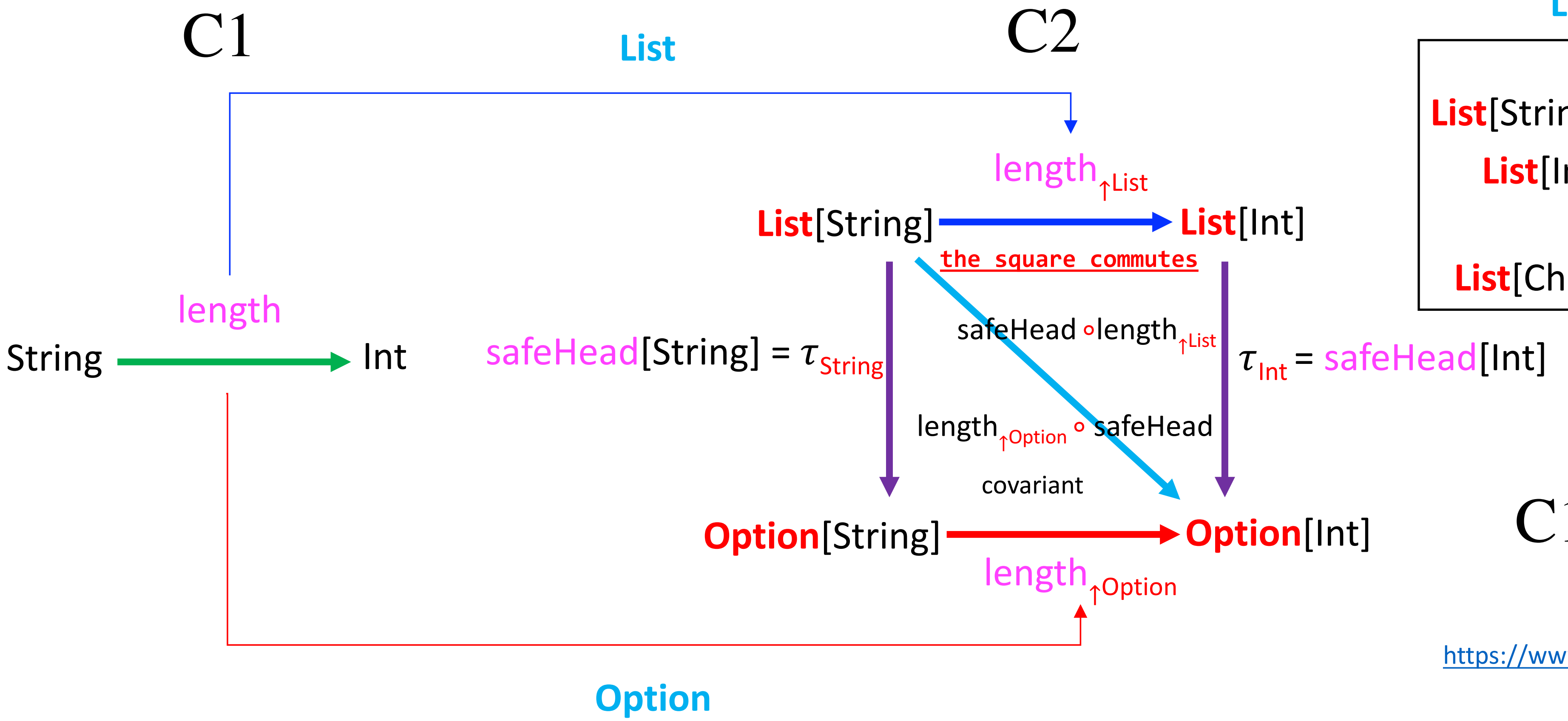
map lifts f into F
$f_{\uparrow F}$ is map f

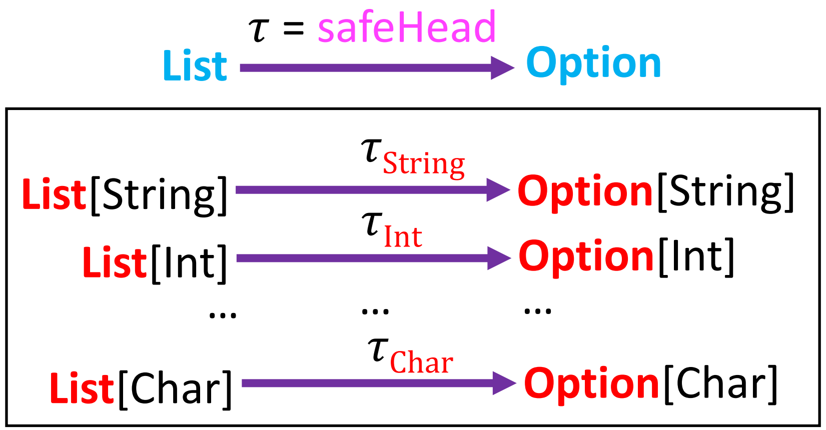
the square commutes

$$safeHead \circ length_{\uparrow List} = length_{\uparrow Option} \circ safeHead$$

$$safeHead \circ (map_{List}\ length) = (map_{Option}\ length) \circ safeHead$$

Naturality Condition





C1 = C2 = Category of Scala types and functions

https://www.slideshare.net/pjschwarz/natural-transformations

Concrete Scala Example: safeHead - natural transformation τ from List functor to Option functor

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

val listF = new Functor[List] {
  def map[A,B](f: A => B): List[A] => List[B] = {
    case head::tail => f(head)::map(f)(tail)
    case Nil => Nil
  }
}
```

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

List $\xrightarrow{\tau}$ Option

map_F lifts f into **F**
so $f_{\uparrow F}$ is map f

```scala
val length: String => Int = s => s.length

def safeHead[A]: List[A] => Option[A] = {
  case head::_ => Some(head)
  case Nil => None
}
```

Naturality Condition

the square commutes

$$safeHead \circ length_{\uparrow List} = length_{\uparrow Option} \circ safeHead$$

$$safeHead \circ (map_{List}\ length) = (map_{Option}\ length) \circ safeHead$$

```scala
val mapAndThenTransform: List[String] => Option[Int] = safeHead compose (listF map length)
val transformAndThenMap: List[String] => Option[Int] = (optionF map length) compose safeHead

assert(mapAndThenTransform(List("abc", "d", "ef")) == transformAndThenMap(List("abc", "d", "ef")))
assert(mapAndThenTransform(List("abc", "d", "ef")) == Some(3))
assert(transformAndThenMap(List("abc", "d", "ef")) == Some(3))

assert(mapAndThenTransform(List()) == transformAndThenMap(List()))
assert(mapAndThenTransform(List()) == None)
assert(transformAndThenMap(List()) == None)
```

@philip_schwarz

Having recapped what a **natural transformation** is, let’s see why **join** is a **natural transformation**.

In Category Theory **a Monad is a functor equipped with a pair of natural transformations** satisfying the laws of associativity and identity

Monads in Category Theory

In Category Theory, a Monad is a functor equipped with a pair of **natural transformations** satisfying the laws of associativity and identity.

What does this mean? If we restrict ourselves to the category of Scala types (with Scala types as the objects and functions as the arrows), we can state this in Scala terms.

A Functor is just a type constructor for which map can be implemented:

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

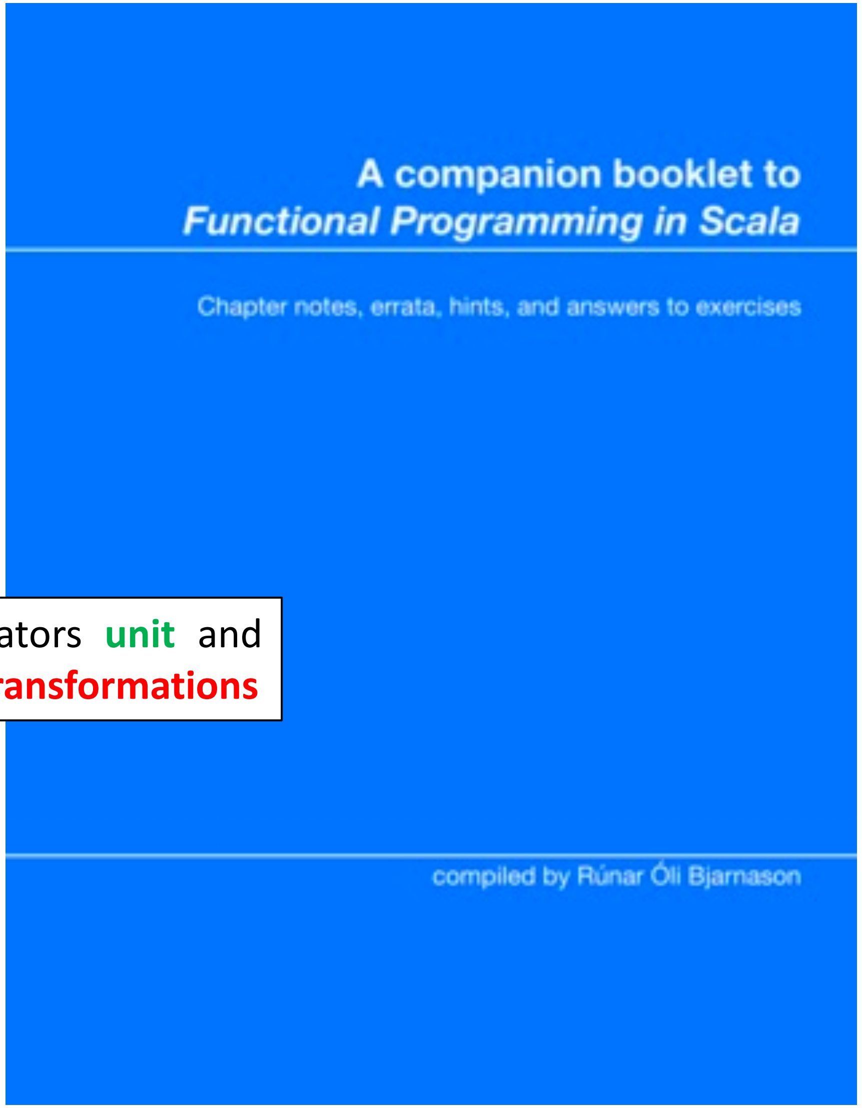
A **natural transformation** from a functor F to a functor G is just a polymorphic function:

```
trait Transform[F[_], G[_]] {
  def apply[A](fa: F[A]): G[A]
}
```

The **natural transformations** that form a monad for F are unit and join:

```
type Id[A] = A

def unit[F](implicit F: Monad[F]) = new Transform[Id, F] {
  def apply(a: A): F[A] = F.unit(a)
}

def join[F](implicit F: Monad[F]) = new Transform[({type f[x] = F[F[x]]})#f, F] {
  def apply(ffa: F[F[A]]): F[A] = F.join(ffa)
}
```

monadic combinators **unit** and **join** are **natural transformations**



(by Runar Bjarnason)

@runarorama

So is sequence a natural transformation, just like safeHead, unit and join?

@philip_schwarz

```
safeHead:             List[A] ⇒ Option[A]
unit:                       A ⇒ List[A]
join:           List[List[A]] ⇒ List[A]
sequence: List[Option[A]] ⇒ Option[List[A]]
```

Hmm, sequence differs from the other functions in that its input and output types are nested functors.

But wait, the input of join is also a nested functor.

Better check with the experts…

Asking the experts – exhibit number 1: for **sequence**, just like for **safeHead**, first **transforming** and then **mapping** is the same as first **mapping** and then **transforming**

```
scala> def safeHead[A]: List[A] => Option[A] = {
     |   case head :: _ => Some(head)
     |   case Nil => None
     | }
safeHead: [A]=> List[A] => Option[A]

scala> safeHead(List("a", "abc", "ab") map (_.length))
res11: Option[Int] = Some(1)

scala> safeHead(List("a", "abc", "ab")) map (_.length)
res12: Option[Int] = Some(1)

scala> def map2[A,B,C](a: Option[A], b: Option[B])(f: (A, B) => C): Option[C] =
     |   a flatMap (aa => b map (bb => f(aa, bb)))
map2: [A, B, C](a: Option[A], b: Option[B])(f: (A, B) => C)Option[C]

scala> def sequence[A](a: List[Option[A]]): Option[List[A]] =
     |   a.foldRight[Option[List[A]]](Some(Nil))((x,y) => map2(x,y)(_ :: _))
sequence: [A](a: List[Option[A]])Option[List[A]]

scala> sequence(List(Some("ab"),Some("abc"),Some("a"))) map (_ map (_.length))
res13: Option[List[Int]] = Some(List(2, 3, 1))

scala> sequence(List(Some("ab"),Some("abc"),Some("a")) map (_ map (_.length)))
res14: Option[List[Int]] = Some(List(2, 3, 1))

scala>
```

Asking the experts – exhibit number 2: why **sequence** seems to be a **natural transformation**, albeit a more complex one

It seems to me that **sequence** is a **natural transformation**, like **safeHead**, but in some higher-order sense. Is that right?

```
def safeHead[A]: List[A] => Option[A]
def sequence[A](a: List[Option[A]]): Option[List[A]]
```

- natural transformation **safeHead** maps the List Functor to the Option Functor
- I can either **first** apply **safeHead** to a list **and then** map the length function over the result
- Or I can **first** map the length function over the list **and then** apply **safeHead** to the result
- The overall result is the same
- In the first case, map is used to lift the length function into the List Functor
- In the second case, map is used to lift the length function into the Option Functor

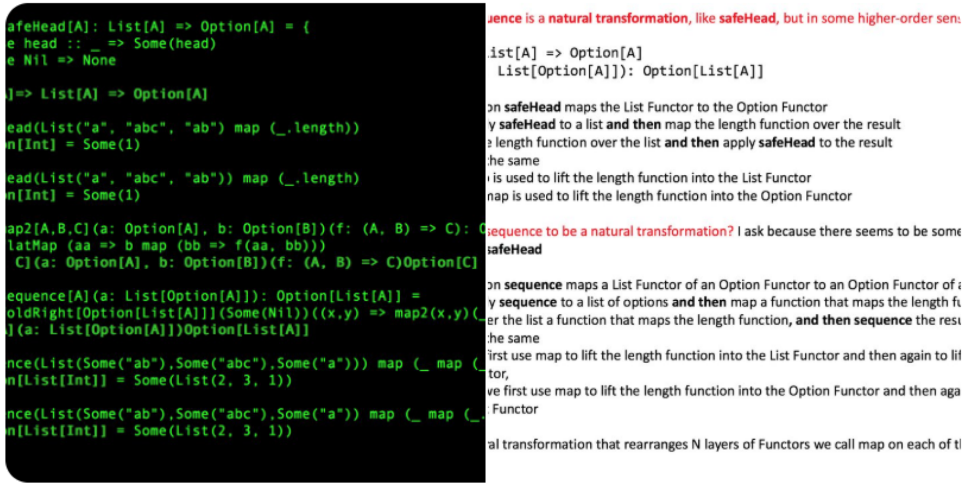
Is it correct to consider sequence to be a natural transformation? I ask because there seems to be something higher-order about **sequence** compared to **safeHead**

- natural transformation **sequence** maps a List Functor of an Option Functor to an Option Functor of a List Functor
- I can either **first** apply **sequence** to a list of options **and then** map a function that maps the length function
- Or I can **first** map over the list a function that maps the length function**, and then sequence** the result
- The overall result is the same
- In the first case, we first use map to lift the length function into the List Functor and then again to lift the resulting function into the Option Functor,
- In the second case, we first use map to lift the length function into the Option Functor and then again to lift the resulting function into the List Functor

It seems that for a natural transformation that rearranges N layers of Functors we call map on each of those layers before we apply a function.

Philip Schwarz @philip_schwarz · Oct 21

current status: @BartoszMilewski am I correct in thinking that sequence is a natural transformation, like safeHead, but in some higher-order sense?



1 1 3

Bartosz Milewski
@BartoszMilewski

Following

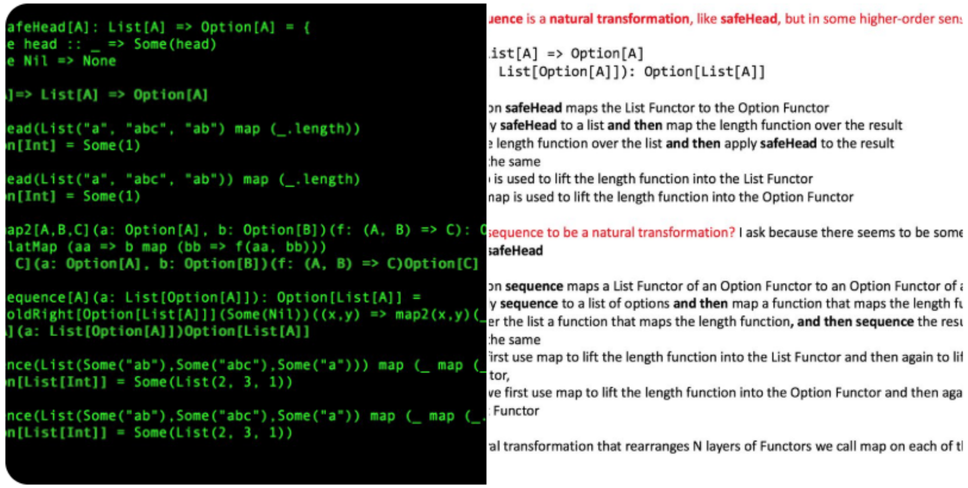
Replying to @philip_schwarz

Yes, it's a natural transformation. A composition of functors is a functor, so list of option is a functor, and so it option of list. You are defining a natural transformation between those two composite functors.

9:22 AM - 21 Oct 2018

Philip Schwarz @philip_schwarz · Oct 21

@runarorama am I correct in thinking that sequence is a natural transformation, like safeHead, but in some higher-order sense?



1

Rúnar
@runarorama

Following

Replying to @philip_schwarz

yes, in fact naturality of sequence is one of the Traversable laws

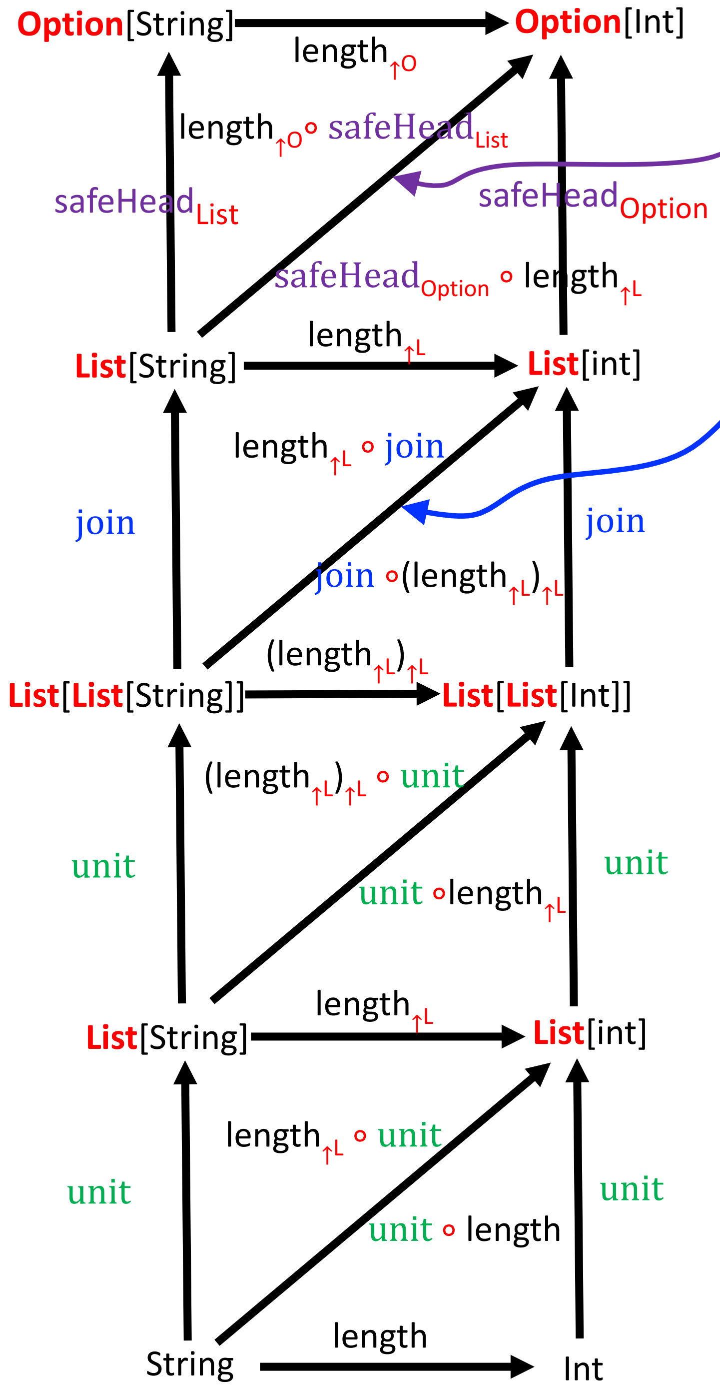
6:17 AM - 21 Oct 2018

@philip_schwarz

So yes, **sequence**, like **safeHead**, **unit** and **join**, is a **natural transformation**. They are all polymorphic functions from one **functor** to another.

```
safeHead:          List[A] ⇒ Option[A]
unit:                    A ⇒ List[A]
join:        List[List[A]] ⇒ List[A]
sequence: List[Option[A]] ⇒ Option[List[A]]
```

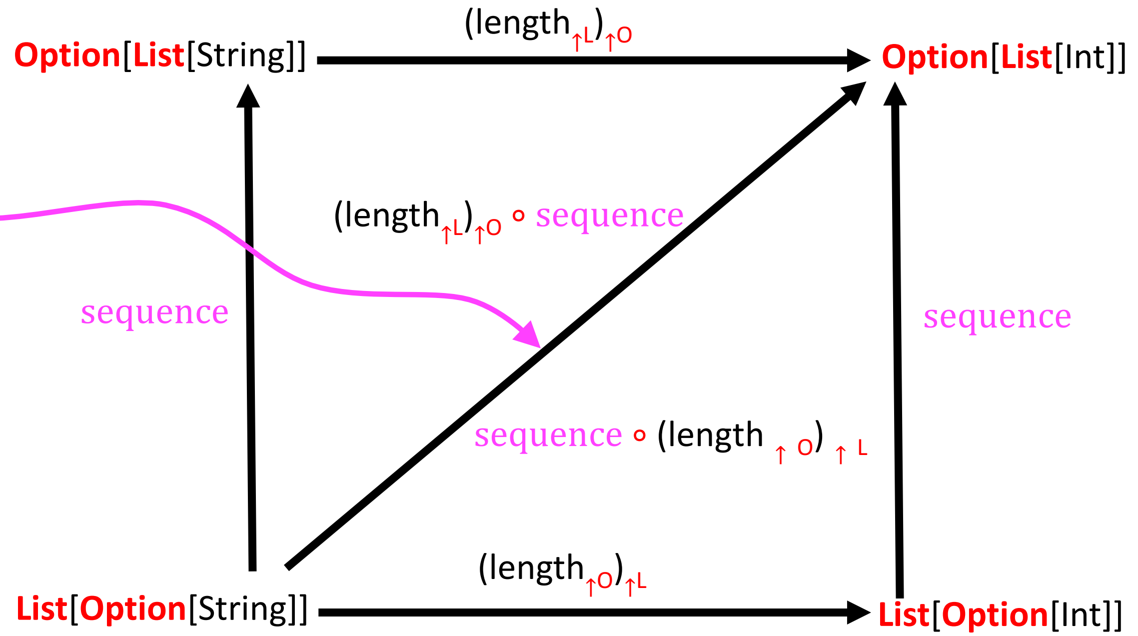
Let’s illustrate this further in the next slide using diagrams and some code.




```
val expected = Some(2)
// first map and then transform
assert(safeHead(List("ab","abc","a").map(_.length)) == expected)
// first transform and then map
assert((safeHead(List("ab","abc","a")).map(_.length)) == expected)
```

```
val expected = List(2,3,1)
// first map and then transform
assert(List(List("ab","abc"),Nil,List("a")).map(_.map(_.length)).flatten == expected)
// first transform and then map
assert(List(List("ab","abc"),Nil,List("a")).flatten.map(_.length) == expected)
```

```
val expected = Some(List(2,3,1))
// first map and then transform
assert(sequence(List(Some("ab"),Some("abc"),Some("a")).map.(_.map(_.length))) == expected)
// first transform and then map
assert(sequence(List(Some("ab"),Some("abc"),Some("a"))).map(_.map(_.length)) == expected)
```



natural transformations:
safeHead
unit aka pure aka η
join aka flatten aka μ
sequence

map lifts f into **F**
$f_{\uparrow L}$ is map f for **F=List**
$f_{\uparrow O}$ is map f for **F=Option**

From sequence for Option to sequence for Either

```
def sequence[A]  (a: List[Option[  A]]): Option[  List[A]]
```

⇩

```
def sequence[E,A](a: List[Either[E,A]]): Either[E,List[A]]
```

Combines a list of **Options** into one **Option** containing a list of all the **Some** values in the original list. If the original list contains **None** even once, the result of the function is **None**; otherwise the result is **Some** with a list of all the values.

⇩

Combines a list of **Eithers** into one **Either** containing a list of all the **Right** values in the original list. If the original list contains **Left** even once, the result of the function is **the first Left**; otherwise the result is **Right** with a list of all the values.

The sequence function for Either

```
def sequence[E,A](a: List[Either[E,A]]): Either[E,List[A]]
```

Combines a list of **Either**s into one **Either** containing a list of all the **Right** values in the original list. If the original list contains **Left** even once, the result of the function is the first **Left**; otherwise the result is **Right** with a list of all the values.

if the list is empty then the result is **Right** of empty list

```
assert(    sequence(Nil) == Right(List()) )
assert( sequence(List()) == Right(List()) )
```

if the list contains all **Right** values then the result is **Right** of a list

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

if the list contains any **Left** value then the result is the first **Left** value

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

From implementation of sequence for Option to implementation of sequence for Either

```
def sequence[A](a: List[Option[A]]): Option[List[A]] = a match {
  case Nil => Some(Nil)
  case h :: t => h flatMap (hh => sequence(t) map (hh :: _))
}
```

```
def sequence[E,A](a: List[Either[E,A]]): Either[E,List[A]] = a match {
    case Nil => Right(Nil)
    case h :: t => h flatMap (hh => sequence(t) map (hh :: _))
}
```

explicit recursive implementation

```
def sequence[A](a: List[Option[A]]): Option[List[A]] =
  a.foldRight[Option[List[A]]](Some(Nil))((h,t) => map2(h,t)(_ :: _))

def map2[A,B,C](a: Option[A], b: Option[B])(f: (A, B) => C): Option[C] =
  a flatMap (aa => b map (bb => f(aa, ab)))
```

```
def sequence[A,E](a: List[Either[E,A]]): Either[E,List[A]] =
  a.foldRight[Either[E,List[A]]](Right(Nil))((h,t) => map2(h,t)(_ :: _))

def map2[A,B,C,E](a: Either[E,A], b: Either[E,B])(f: (A, B) => C): Either[E,C] =
  a flatMap (aa => b map (bb => f(aa, bb)))
```

Implementation using **foldRight** and **map2**

From implementation of traverse for Option to implementation of traverse for Either

```
def traverse[A, B](a: List[A])(f: A => Option[B]): Option[List[B]] = a match {
  case Nil => Some(Nil)
  case h::t => map2(f(h), traverse(t)(f))(_ :: _)
}
```

⇩

```
def traverse[A,B,E](a: List[A])(f: A => Either[E, B]): Either[E,List[B]] = a match {
  case Nil => Right(Nil)
  case h::t => map2(f(h),traverse(t)(f))(_ :: _)
}
```

explicit recursive implementation

```
def traverse[A,B](a: List[A])(f: A => Option[B]): Option[List[B]] =
  a.foldRight[Option[List[B]]](Some(Nil))((h,t) => map2(f(h),t)(_ :: _))
```

⇩

```
def traverse[A,B,E](a: List[A])(f: A => Either[E,B]): Either[E,List[B]] =
  a.foldRight[Either[E,List[B]]](Right(Nil))((h, t) => map2(f(h),(t))(_ :: _))
```

Implementation using `foldRight` and `map2`

Simple example of using sequence function, revisited for Either

Sometimes we'll want to **map** over a list using a function that might fail, returning **Left[Throwable]** if applying it to any element of the list returns **Left[Throwable]**.

For example, what if we have a whole list of **String** values that we wish to parse to **Either[Throwable,Int]**? In that case, we can simply **sequence** the results of the **map**.

```
import scala.util.Try

def parseIntegers(a: List[String]): Either[Throwable,List[Int]] =
  sequence(a map (i => Try(i.toInt).toEither))
```

```
assert( parseIntegers(List("1", "2", "3"))   == Right(List(1, 2, 3) )
assert( parseIntegers(List("1", "x", "3"))   == Left(java.lang.NumberFormatException: For input string: "x") )
assert( parseIntegers(List("1", "x", "1.2")) == Left(java.lang.NumberFormatException: For input string: "x") )
```

The close relationship between sequence and traverse, revisited for Either

```
def traverse[A, B](a: List[A])(f: A => Option[B]): Option[List[B]] =
  sequence(a map f)
```

```
def traverse[A,B,E](a: List[A])(f: A => Either[E,B]): Either[E,List[B]] =
  sequence(a map f)
```

defining **traverse** in terms of **map** and **sequence**

```
def sequence[A](a: List[Option[A]]): Option[List[A]] =
  traverse(a)(x => x)
```

```
def sequence[A,E](a: List[Either[E,A]]): Either[E,List[A]] =
  traverse(a)(x => x)
```

defining **sequence** in terms of **traverse** and **identity**

The **sequence** and **traverse** functions we have seen so far (and the **map** and **map2** functions they depend on), are specialised for a particular type constructor, e.g. **Option** or **Either**. Can they be generalised so that they work on many more type constructors?

@philip_schwarz

```
def sequence[A](a: List[Option[A]]): Option[List[A]] =
  a.foldRight[Option[List[A]]](Some(Nil))((h,t) => map2(h,t)(_ :: _))

def traverse[A, B](a: List[A])(f: A => Option[B]): Option[List[B]] =
  a.foldRight[Option[List[B]]](Some(Nil))((h,t) => map2(f(h),t)(_ :: _))

def map2[A,B,C](oa: Option[A], ob: Option[B])(f: (A, B) => C): Option[C] =
  oa flatMap (a => ob map (b => f(a, b)))
```

E.g. in the above example, the **Option** specific items that **sequence**, **traverse** and **map2** depend on are **Option**'s **Some** constructor, **Option**'s **map** function, and **Option**'s **flatMap** function. In the case of **Either**, the dependencies are on **Either**'s **Right** constructor, **Either**'s **map** function and **Either**'s **flatMap** function.

Option and **Either** are **monads** and every **monad** has a **unit** function, a **map** function, and a **flatMap** function.

Since **Some** and **Right** are **Option** and **Either**'s **unit** functions and since **map2** can be implemented using **map** and **flatMap**, it follows that **sequence** and **traverse** can be implemented for every **monad**.

See the next slide for a reminder that every monad has **unit**, **map** and **flatMap** functions.

flatmap + unit

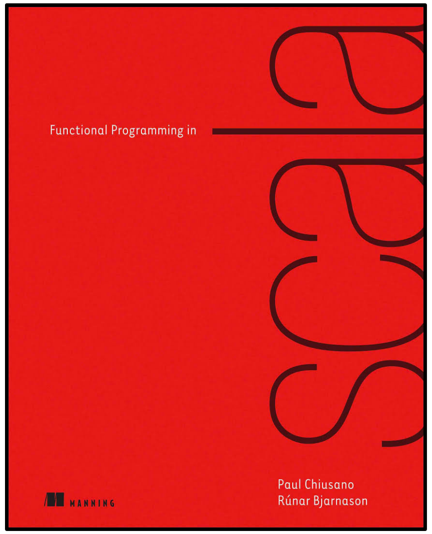
```
trait Monad[F[_]] {

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

  def join[A](mma: F[F[A]]): F[A] = flatMap(mma)(ma => ma)
  def map[A,B](m: F[A])(f: A => B): F[B] = flatMap(m)(a => unit(f(a)))
  def compose[A,B,C](f: A => F[B], g: B => F[C]): A => F[C] = a => flatMap(f(a))(g)
}
```

map + join + unit

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

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

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

  def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B] = join(map(ma)(f))
  def compose[A,B,C](f: A => F[B], g: B => F[C]): A => F[C] = a => flatMap(f(a))(g)
}
```

Kleisli composition + unit

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

  def compose[A,B,C](f: A => F[B], g: B => F[C]): A => F[C]
  def unit[A](a: => A): F[A]

  def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B] = compose((_:Unit) => ma, f)(())
  def map[A,B](m: F[A])(f: A => B): F[B] = flatMap(m)(a => unit(f(a)))
}
```



Functional Programming in Scala

The three different ways of defining a **monad**, and how a **monad** always has the following three functions: **unit**, **map**, **flatMap**

Generalising the signatures of map2, sequence and traverse so they work on a monad **F**

```
def sequence[A](a: List[Option[A]]): Option[List[A]]
```

⇩

```
def sequence[A](a: List[F[A]]): F[List[A]]
```

```
def traverse[A, B](a: List[A])(f: A => Option[B]): Option[List[B]]
```

⇩

```
def traverse[A,B](a: List[A])(f: A => F[B]): F[List[B]]
```

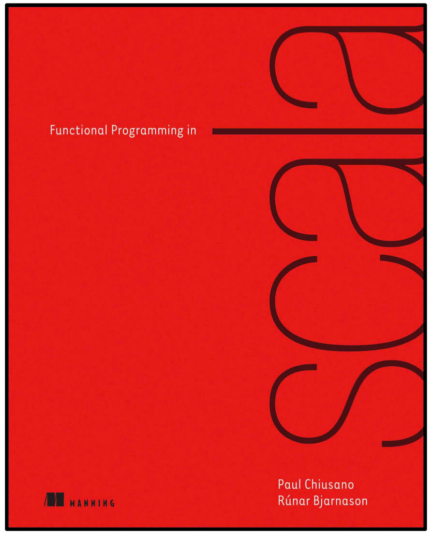
```
def map2[A,B,C](a: Option[A], b: Option[B])(f: (A, B) => C): Option[C]
```

⇩

```
def map2[A,B,C](a: F[A], b: F[B])(f: (A, B) => C): F[C]
```

How a monad can define sequence and traverse

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

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)(_ :: _))

}
```



Functional Programming in Scala

A simple example of using the traversable function of the Option monad

```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> parseIntsMaybe(List("1", "2", "3"))
res0: Option[List[Int]] = Some(List(1, 2, 3))

scala> parseIntsMaybe(List("1", "x", "3"))
res1: Option[List[Int]] = None

scala> parseIntsMaybe(List("1", "x", "y"))
res2: Option[List[Int]] = None
```

A simple example of using the traversable function of the Either monad

```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)
```

```
scala> parseIntsValidated(List("1", "2", "3"))
res0: Either[Throwable,List[Int]] = Right(List(1, 2, 3))


scala> parseIntsValidated(List("1", "x", "3"))
res1: Either[Throwable,List[Int]] = Left(java.lang.NumberFormatException: For input string: "x")


scala> parseIntsValidated(List("1", "x", "1.2"))
res2: Either[Throwable,List[Int]] = Left(java.lang.NumberFormatException: For input string: "x")
```

@philip_schwarz

But is it necessary to use a **monad** in order to define generic **sequence** and **traverse** methods? Find out in **part 2**.