Use Applicative

where applicable!

https://github.com/hermannhueck/use-applicative-where-applicable

Abstract

Most Scala developers are familiar with monadic precessing. Monads provide *flatMap* and hence for-comprehensions (syntactic sugar for *map* and *flatMap*).

Often we don't need Monads. Applicatives are sufficient in many cases.

In this talk I examine the differences between monadic and applicative processing and give some guide lines when to use which.

After a closer look to the Applicative trait I will contrast the gist of *Either* and *cats.data.Validated*.

I will also look at traversing and sequencing which harness Applicatives as well.

Examples are implemented with *Cats*.

Agenda

1. Monadic Processing
2. Aside: Curried functions
3. Aside: Effects
4. Applicative Processing
5. Comparing Monad with Applicative
6. The Applicative trait
7. Either vs. Validated
8. Traversals
9. Resources

1. Monadic Processing

See: *examples.MonadicProcessing*

The Problem:

How to compute values wrapped in a context?

Example context/effect: *Option*

```scala
def processInts(compute: (Int, Int, Int) => Int)
               (oi1: Option[Int], oi2: Option[Int], oi3: Option[Int])
               : Option[Int] = ???
```

The Problem:

How to compute values wrapped in a context?

Example context/effect: *Option*

```scala
def processInts(compute: (Int, Int, Int) => Int)
               (oi1: Option[Int], oi2: Option[Int], oi3: Option[Int])
               : Option[Int] = ???
```

```scala
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _

val result1 = processInts(sum3Ints, Some(1), Some(2), Some(3))
// result1: Option[Int] = Some(6)

val result2 = processInts(sum3Ints, Some(1), Some(2), None)
// result2: Option[Int] = None
```

The Standard Solution:

Monadic Processing with a for-comprehension

Example context/effect: *Option*

```
def processInts(compute: (Int, Int, Int) => Int)
               (oi1: Option[Int], oi2: Option[Int], oi3: Option[Int])
               : Option[Int] =
  for {
    i1 <- oi1
    i2 <- oi2
    i3 <- oi3
  } yield compute(i1, i2, i3)
```

The Standard Solution:

Monadic Processing with a for-comprehension

Example context/effect: *Option*

```scala
def processInts(compute: (Int, Int, Int) => Int)
               (oi1: Option[Int], oi2: Option[Int], oi3: Option[Int])
               : Option[Int] =
  for {
    i1 <- oi1
    i2 <- oi2
    i3 <- oi3
  } yield compute(i1, i2, i3)
```

```scala
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _

val result1 = processInts(sum3Ints, Some(1), Some(2), Some(3))
// result1: Option[Int] = Some(6)

val result2 = processInts(sum3Ints, Some(1), Some(2), None)
// result2: Option[Int] = None
```

Abstracting *Option* to *F[_]: Monad*

```
def processInts[F[_]: Monad](compute: (Int, Int, Int) => Int)
                            (fi1: F[Int], fi2: F[Int], fi3: F[Int]): F[Int] =
  for {
    i1 <- fi1
    i2 <- fi2
    i3 <- fi3
  } yield compute(i1, i2, i3)
```

Abstracting *Option* to *F[_]: Monad*

```
def processInts[F[_]: Monad](compute: (Int, Int, Int) => Int)
                            (fi1: F[Int], fi2: F[Int], fi3: F[Int]): F[Int] =
  for {
    i1 <- fi1
    i2 <- fi2
    i3 <- fi3
  } yield compute(i1, i2, i3)
```

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _

val result1 = processInts(sum3Ints, Some(1), Some(2), Some(3))
// result1: Option[Int] = Some(6)

val result2 = processInts(sum3Ints, Some(1), Some(2), None)
// result2: Option[Int] = None

val result3 = processInts(sum3Ints)(List(1, 2), List(10, 20), List(100, 200))
// result3: List[Int] = List(111, 211, 121, 221, 112, 212, 122, 222)

val result4 = processInts(sum3Ints)(Future(1), Future(2), Future(3))
Await.ready(result4, 1 second)
// result4 = scala.concurrent.Future[Int] = Future(Success(6))
```

Abstracting away the *Ints*

```
def processABC[F[_]: Monad, A, B, C, D](compute: (A, B, C) => D)
                                       (fa: F[A], fb: F[B], fc: F[C]): F[D] =
  for {
    a <- fa
    b <- fb
    c <- fc
  } yield compute(a, b, c)
```

Abstracting away the *Ints*

```
def processABC[F[_]: Monad, A, B, C, D](compute: (A, B, C) => D)
                                      (fa: F[A], fb: F[B], fc: F[C]): F[D] =
  for {
    a <- fa
    b <- fb
    c <- fc
  } yield compute(a, b, c)
```

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _

val result1 = processABC(sum3Ints, Some(1), Some(2), Some(3))
// result1: Option[Int] = Some(6)

val result2 = processABC(sum3Ints, Some(1), Some(2), None)
// result2: Option[Int] = None

val result3 = processABC(sum3Ints)(List(1, 2), List(10, 20), List(100, 200))
// result3: List[Int] = List(111, 211, 121, 221, 112, 212, 122, 222)

val result4 = processABC(sum3Ints)(Future(1), Future(2), Future(3))
Await.ready(result4, 1 second)
// result4 = scala.concurrent.Future[Int] = Future(Success(6))
```

Replacing the for-comprehension

(syntactic sugar) with *map / flatMap*

```
def processABC[F[_]: Monad, A, B, C, D](compute: (A, B, C) => D)
                                       (fa: F[A], fb: F[B], fc: F[C]): F[D] =
  fa flatMap { a =>
    fb flatMap { b =>
      fc map { c =>
        compute(a, b, c)
      }
    }
  }
```

2. Aside: Effects

What is *F[_]*?

F[_] is a type constructor.

It represents the **computational context** of the operation, also called the **effect** of the operation.

effect != side effect

With the context bound *F[_]: Monad* we constrain the effect to be a Monad.

This makes *F* a place holder for any Monad which provides us *map* and *flatMap*.

Some Effects

F[_]	Effect
Option	a possibly missing value
List	an arbitrary number of values of the same type
Future	an asyncronously computed value
Either	either a value of this type or a value of that type
Id	the effect of having no effect
IO	the effect of having a side effect
Tuple2	two values of possibly different type
Function1	a pure computation taking an input and returning an output

3. Aside: Curried functions

See: *examples.CurriedFunctions*

Every function is a *Function1* ...

Every function is a *Function1* ...

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
// sum3Ints: (Int, Int, Int) => Int = $$Lambda$6510/1947502277@3c418454
```

Every function is a *Function1* ...

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
// sum3Ints: (Int, Int, Int) => Int = $$Lambda$6510/1947502277@3c418454
```

... if you curry it.

Every function is a *Function1* ...

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
// sum3Ints: (Int, Int, Int) => Int = $$Lambda$6510/1947502277@3c418454
```

... if you curry it.

```
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
// sumCurried: Int => (Int => (Int => Int)) = scala.Function3$$Lambda$4346/9914305
```

Every function is a *Function1* ...

```scala
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
// sum3Ints: (Int, Int, Int) => Int = $$Lambda$6510/1947502277@3c418454
```

... if you curry it.

```scala
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
// sumCurried: Int => (Int => (Int => Int)) = scala.Function3$$Lambda$4346/9914305
```

The type arrow (=>) is right associative. Hence we can omit the parentheses.

Every function is a *Function1* ...

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
// sum3Ints: (Int, Int, Int) => Int = $$Lambda$6510/1947502277@3c418454
```

... if you curry it.

```
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
// sumCurried: Int => (Int => (Int => Int)) = scala.Function3$$Lambda$4346/9914305
```

The type arrow (=>) is right associative. Hence we can omit the parentheses.

The type arrow (=>) is syntactic sugar for Function1.
A => *B* is the same as *Function1[A, B]*.

Every function is a *Function1* ...

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
// sum3Ints: (Int, Int, Int) => Int = $$Lambda$6510/1947502277@3c418454
```

... if you curry it.

```
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
// sumCurried: Int => (Int => (Int => Int)) = scala.Function3$$Lambda$4346/9914305
```

The type arrow (=>) is right associative. Hence we can omit the parentheses.

The type arrow (=>) is syntactic sugar for Function1.
A => *B* is the same as *Function1[A, B]*.

```
val sumCurried2: Function1[Int, Function1[Int, Function1[Int, Int]]] = sumCurried
// sumCurried2: Int => (Int => (Int => Int)) = scala.Function3$$Lambda$4346/991430
```

Every function is a *Function1* ...

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
// sum3Ints: (Int, Int, Int) => Int = $$Lambda$6510/1947502277@3c418454
```

... if you curry it.

```
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
// sumCurried: Int => (Int => (Int => Int)) = scala.Function3$$Lambda$4346/9914305
```

The type arrow (=>) is right associative. Hence we can omit the parentheses.

The type arrow (=>) is syntactic sugar for Function1.
A => *B* is the same as *Function1[A, B]*.

```
val sumCurried2: Function1[Int, Function1[Int, Function1[Int, Int]]] = sumCurried
// sumCurried2: Int => (Int => (Int => Int)) = scala.Function3$$Lambda$4346/991430
```

If uncurried again, you get the original function back.

```
val sumUncurried: (Int, Int, Int) => Int = Function.uncurried(sumCurried)
// sumUncurried: (Int, Int, Int) => Int = scala.Function$$$Lambda$6605/301079867@1
```

Partial application of curried functions

Partial application of curried functions

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
```

Partial application of curried functions

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
```

```
val applied1st = sumCurried(1)
// applied1st: Int => (Int => Int) = scala.Function3$$Lambda$4348/1531035406@5a231
```

Partial application of curried functions

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
```

```
val applied1st = sumCurried(1)
// applied1st: Int => (Int => Int) = scala.Function3$$Lambda$4348/1531035406@5a231
```

```
val applied2nd = applied1st(2)
// applied2nd: Int => Int = scala.Function3$$Lambda$4349/402963549@117e96fb
```

Partial application of curried functions

```scala
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
```

```scala
val applied1st = sumCurried(1)
// applied1st: Int => (Int => Int) = scala.Function3$$Lambda$4348/1531035406@5a231
```

```scala
val applied2nd = applied1st(2)
// applied2nd: Int => Int = scala.Function3$$Lambda$4349/402963549@117e96fb
```

```scala
val applied3rd = applied2nd(3)
// applied3rd: Int = 6
```

Partial application of curried functions

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
```

```
val applied1st = sumCurried(1)
// applied1st: Int => (Int => Int) = scala.Function3$$Lambda$4348/1531035406@5a231
```

```
val applied2nd = applied1st(2)
// applied2nd: Int => Int = scala.Function3$$Lambda$4349/402963549@117e96fb
```

```
val applied3rd = applied2nd(3)
// applied3rd: Int = 6
```

```
val appliedAllAtOnce = sumCurried(1)(2)(3)
// appliedAllAtOnce: Int = 6
```

Partial application of curried functions

```scala
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _
val sumCurried: Int => Int => Int => Int = sum3Ints.curried
```

```scala
val applied1st = sumCurried(1)
// applied1st: Int => (Int => Int) = scala.Function3$$Lambda$4348/1531035406@5a231
```

```scala
val applied2nd = applied1st(2)
// applied2nd: Int => Int = scala.Function3$$Lambda$4349/402963549@117e96fb
```

```scala
val applied3rd = applied2nd(3)
// applied3rd: Int = 6
```

```scala
val appliedAllAtOnce = sumCurried(1)(2)(3)
// appliedAllAtOnce: Int = 6
```

As curried functions can be partially applied,
they are better composable than their uncurried counterparts.

Curring in Haskell?

Curring in Haskell?

Nope!

Curring in Haskell?

Nope!

Every function is curried.
It can take only one parameter …
… and returns only one value which might be another function.

Curring in Haskell?

Nope!

Every function is curried.
It can take only one parameter …
… and returns only one value which might be another function.

Invoking functions with more than one parameter is possible.
That is just syntactic sugar for curried functions.

4. Applicative Processing

See: *examples.ApplicativeProcessing*

Trait Applicative

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

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

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

  def pure[A](a: A): F[A]

  def ap[A, B](ff: F[A => B])(fa: F[A]): F[B]
}
```

Replacing monadic with applicative context

Replacing monadic with applicative context

Monad is more powerful than Applicative.
Applicative is weaker, but sufficient for our problem.

Replacing monadic with applicative context

Monad is more powerful than Applicative.
Applicative is weaker, but sufficient for our problem.

Monadic context:

```
def processABC[F[_]: Monad, A, B, C, D](compute: (A, B, C) => D)
                                       (fa: F[A], fb: F[B], fc: F[C]): F[D] =
  ??? // monadic solution
```

Replacing monadic with applicative context

Monad is more powerful than Applicative.
Applicative is weaker, but sufficient for our problem.

Monadic context:

```
def processABC[F[_]: Monad, A, B, C, D](compute: (A, B, C) => D)
                                       (fa: F[A], fb: F[B], fc: F[C]): F[D] =
  ??? // monadic solution
```

Applicative context:

```
def processABC[F[_]: Applicative, A, B, C, D](compute: (A, B, C) => D)
                                             (fa: F[A], fb: F[B], fc: F[C]): F[D] =
  ??? // applicative solution
```

Replacing monadic with applicative context

Monad is more powerful than Applicative.
Applicative is weaker, but sufficient for our problem.

Monadic context:

```
def processABC[F[_]: Monad, A, B, C, D](compute: (A, B, C) => D)
                                        (fa: F[A], fb: F[B], fc: F[C]): F[D] =
  ??? // monadic solution
```

Applicative context:

```
def processABC[F[_]: Applicative, A, B, C, D](compute: (A, B, C) => D)
                                              (fa: F[A], fb: F[B], fc: F[C]): F[D] =
  ??? // applicative solution
```

Let's start with *Option[Int]* again and then abstract the solution.

Solution with *Applicative#ap*:

Example context/effect: *Option*

```
def processInts(compute: (Int, Int, Int) => Int)
               (oi1: Option[Int], oi2: Option[Int], oi3: Option[Int])
               : Option[Int] = {
  val f3Curried: Int => Int => Int => Int = compute.curried
  val of3: Option[Int => Int => Int => Int] = Some(f3Curried)
  val of2: Option[Int => Int => Int] = of3 ap oi1
  val of1: Option[Int => Int] = of2 ap oi2
  val result: Option[Int] = of1 ap oi3
  result
}
```

Some(f3Curried) lifts the curried function into the *Option* context.

Solution with *Applicative#ap*:

Example context/effect: *Option*

```
def processInts(compute: (Int, Int, Int) => Int)
               (oi1: Option[Int], oi2: Option[Int], oi3: Option[Int])
               : Option[Int] = {
  val f3Curried: Int => Int => Int => Int = compute.curried
  val of3: Option[Int => Int => Int => Int] = Some(f3Curried)
  val of2: Option[Int => Int => Int] = of3 ap oi1
  val of1: Option[Int => Int] = of2 ap oi2
  val result: Option[Int] = of1 ap oi3
  result
}
```

Some(f3Curried) lifts the curried function into the *Option* context.

```
val sum3Ints: (Int, Int, Int) => Int = _ + _ + _

val result1 = processInts(sum3Ints, Some(1), Some(2), Some(3))
// result1: Option[Int] = Some(6)

val result2 = processInts(sum3Ints, Some(1), Some(2), None)
// result2: Option[Int] = None
```

Abstracting *Option* to *F[_]: Applicative*

```
def processInts[F[_]: Applicative](compute: (Int, Int, Int) => Int)
                                  (fi1: F[Int], fi2: F[Int], fi3: F[Int])
                                  : F[Int] = {
  val fCurried: Int => Int => Int => Int = compute.curried
  val ff: F[Int => Int => Int => Int] = Applicative[F].pure(fCurried)
  ff ap fi1 ap fi2 ap fi3
}
```

Applicative[F].pure(fCurried) lifts the curried function into the generic *F* context.

Abstracting away the *Ints*

```
def processABC[F[_]: Applicative, A, B, C, D](compute: (A, B, C) => D)
                                             (fa: F[A], fb: F[B], fc: F[C])
                                             : F[D] = {
  val fCurried: A => B => C => D = compute.curried
  val ff: F[A => B => C => D] = Applicative[F].pure(fCurried)
  ff ap fa ap fb ap fc
}
```

Using *pure* syntax and *<*>* (alias for *ap*)

```
def processABC[F[_]: Applicative, A, B, C, D](compute: (A, B, C) => D)
                                              (fa: F[A], fb: F[B], fc: F[C])
                                              : F[D] = {
  compute.curried.pure[F] <*> fa <*> fb <*> fc
}
```

Using *Applicative#ap3* saves us from currying

```
def processABC[F[_]: Applicative, A, B, C, D](compute: (A, B, C) => D)
                                              (fa: F[A], fb: F[B], fc: F[C])
                                              : F[D] = {
  Applicative[F].ap3(compute.pure[F])(fa, fb, fc)
}
```

Using *Applicative#map3* saves us from lifting the function with *pure*

```
def processABC[F[_]: Applicative, A, B, C, D](compute: (A, B, C) => D)
                                              (fa: F[A], fb: F[B], fc: F[C])
                                              : F[D] = {
  Applicative[F].map3(fa, fb, fc)(compute)
}
```

map2, *map3* .. *map22* are available for *Applicative*.

Apply is just *Applicative* without *pure*

```
def processABC[F[_]: Apply, A, B, C, D](compute: (A, B, C) => D)
                                       (fa: F[A], fb: F[B], fc: F[C])
                                       : F[D] = {
  Apply[F].map3(fa, fb, fc)(compute)
}
```

map2, *map3* .. *map22* come from *Apply*.

Using *Tuple3#mapN* for convenience

```
def processABC[F[_]: Apply, A, B, C, D](compute: (A, B, C) => D)
                                      (fa: F[A], fb: F[B], fc: F[C])
                                      : F[D] = {
  (fa, fb, fc) mapN compute
}
```

We just tuple up the three *F*'s and invoke *mapN* with a function that takes three parameters and fuses the *F*'s into an *F*-result.

mapN is provided as an enrichment for *Tuple2*, *Tuple3* .. *Tuple22*.

Comparing the solutions

Comparing the solutions

Monadic solution:

```
def processABC[F[_]: Monad, A, B, C, D](compute: (A, B, C) => D)
                                       (fa: F[A], fb: F[B], fc: F[C]): F[D] =
  for {
    a <- fa
    b <- fb
    c <- fc
  } yield compute(a, b, c)
```

Comparing the solutions

Monadic solution:

```
def processABC[F[_]: Monad, A, B, C, D](compute: (A, B, C) => D)
                                        (fa: F[A], fb: F[B], fc: F[C]): F[D] =
  for {
    a <- fa
    b <- fb
    c <- fc
  } yield compute(a, b, c)
```

Applicative solution:

```
def processABC[F[_]: Apply, A, B, C, D](compute: (A, B, C) => D)
                                        (fa: F[A], fb: F[B], fc: F[C]) : F[D] =
  (fa, fb, fc) mapN compute
```

Currying again

Currying again

We provided *processABC* with two parameter lists.

```
def processABC[F[_]: Apply, A, B, C, D](compute: (A, B, C) => D)
                                       (fa: F[A], fb: F[B], fc: F[C]) : F[D] =
  (fa, fb, fc) mapN compute
```

Currying again

We provided *processABC* with two parameter lists.

```
def processABC[F[_]: Apply, A, B, C, D](compute: (A, B, C) => D)
                                        (fa: F[A], fb: F[B], fc: F[C]) : F[D] =
  (fa, fb, fc) mapN compute
```

This improves composability and allows us provide the *compute* function and the effectful *F*'s in separate steps.

```
// providing the computation
def processEffectfulInts[F[_]: Apply](fi1: F[Int], fi2: F[Int], fi3: F[Int])
                                      : F[Int] =
  processABC(sum3Ints)(fi1, fi2, fi3)
```

```
val result1 = processEffectfulInts(Option(1), Option(2), Option(3))
// result1: Option[Int] = Some(6)
val result2 = processEffectfulInts(List(1, 2), List(10, 20), List(100, 200))
// result2: List[Int] = List(111, 211, 121, 221, 112, 212, 122, 222)
```

5. Comparing Monad with Applicative

Sequential vs. parallel

Monads enforce sequential operation!

Imagine a for-comprehension.
The generators are executed in sequence, one after the other.
The yield clause is executed after all generators have finished.

Applicatives allow for parallel operations!

The operations are by nature <u>independent</u> of each other.
(That does not mean, they are asynchronous.)

Fail-fast semantics

Monads enforce short-circuiting!

If an 'erraneous' value (e.g. *None*, *Nil* or *Left*) is found in a monadic computation, the computation stops immediately, because the result will not change if processing were continued.
E.g.: We are not able to collect errors.

Applicatives do not short-circuit!

As applicative operations are independent of each other processing does not stop if an error occurs.
It is possible to collect errors (see: *cats.data.Validated*)
cats.data.Validated has an Applicative instance, but no Monad instance.

Composition

Monads do not compose!

To kind of 'compose' Monads you need an extra construct such as a Monad Transformer which itself is a Monad. I.e.: If we want to compose two Moands we need a third Monad to stack them up.

Applicatives compose!

Composition of (Functor and) Aplicative is a breeze.
It is genuinely supported by these type classes.

See: *examples.Composition*

Composition - Code

```
val loi1 = List(Some(1), Some(2))
val loi2 = List(Some(10), Some(20))
```

Monads do not compose!

```
def processMonadic(xs: List[Option[Int]], ys: List[Option[Int]]): List[Option[Int]
  val otli: OptionT[List, Int] =
    for {
      x <- OptionT[List, Int](xs)
      y <- OptionT[List, Int](ys)
    } yield x + y
  otli.value
}
val result1 = processMonadic(loi1, loi2)
// result1: List[Option[Int]] = List(Some(11), Some(21), Some(12), Some(22))
```

Applicatives compose!

```
def processApplicative(xs: List[Option[Int]], ys: List[Option[Int]]): List[Option[
  Applicative[List].compose[Option].map2(xs, ys)((_:Int) + (_:Int))

val result2 = processApplicative(loi1, loi2)
```

When to use which

Use Applicative if all computations are independent of each other.

Howto:

- Write a monadic solution with a for-comprehension.
- If the generated values (to the left of the generator arrow <-) are only used in the *yield* clause at the end (not in an other generator) the computations are independent and allow for an applicative solution.
- Tuple up the computations and *mapN* them with a function which fuses the computation results to a final result.

(This howto is applicable only if the effect in question also has a Monad instance.)

Principle of Least Power

**Given a choice of solutions,
pick the least powerful solution
capable of solving your problem.**

-- Li Haoyi

6. The Applicative trait

Cats typeclass hierarchy (a small section of it)

Complete hierarchy here

Typeclass Functor

```
trait Functor[F[_]] { // simplified

  // ----- intrinsic abstract Functor method

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

  // ----- method implementations in terms of map

  def fmap[A, B](fa: F[A])(f: A => B): F[B] = map(fa)(f) // alias for map

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

  def as[A, B](fa: F[A], b: B): F[B] = map(fa)(_ => b)

  def void[A](fa: F[A]): F[Unit] = as(fa, ())

  def compose[G[_]: Functor]: Functor[Lambda[X => F[G[X]]]] = ???
}
```

Typeclass Applicative

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

  // ----- intrinsic abstract Applicative methods
  def pure[A](a: A): F[A]
  def ap[A, B](ff: F[A => B])(fa: F[A]): F[B]

  // ----- method implementations in terms of pure and ap
  override def map[A, B](fa: F[A])(f: A => B): F[B] = ap(pure(f))(fa)

  def <*>[A, B](ff: F[A => B])(fa: F[A]): F[B] = ap(ff)(fa) // alias for ap

  def ap2[A, B, Z](ff: F[(A, B) => Z])(fa: F[A], fb: F[B]): F[Z] = {
    val ffBZ: F[B => Z] = ap(map(ff)(f => (a:A) => (b:B) => f(a, b)))(fa)
    ap(ffBZ)(fb)
  }
  // continued with ap3, ap4 .. ap22

  def map2[A, B, Z](fa: F[A], fb: F[B])(f: (A, B) ⇒ Z): F[Z] =
    ap(map(fa)(a => f(a, _: B)))(fb)
  // continued with map3, map4 .. map22

  def product[A, B](fa: F[A], fb: F[B]): F[(A, B)] = map2(fa, fb)((_, _))

  def tuple2[A, B](fa: F[A], fb: F[B]): F[(A, B)] = product(fa, fb)
  // continued with tuple3, tuple4 .. tuple22

  def compose[G[_]: Applicative]: Applicative[Lambda[X => F[G[X]]]] = ???
}
```

7. Either vs. Validated

See: *examples.EitherVsValidated*

Monadic processing with *Either*

Monadic processing enforces fail-fast semantics.
After an erraneous operation subsequent operations are not executed.
Hence we get only the first of possibly multiple errors.

Monadic processing with *Either*

Monadic processing enforces fail-fast semantics.
After an erraneous operation subsequent operations are not executed.
Hence we get only the first of possibly multiple errors.

```scala
val result: Either[List[String], Int] =
  for {
    x <- 5.asRight[List[String]]
    y <- List("Error 1").asLeft[Int]
    z <- List("Error 2").asLeft[Int] // List("Error 2") is lost!
  } yield x + y + z

// result: Either[List[String],Int] = Left(List(Error 1))
```

Applicative processing with *Either*

Applicative processing disables fail-fast semantics.
But with *Either* we get the same result as before.
Because *Either* has a Monad instance.

Applicative processing with *Either*

Applicative processing disables fail-fast semantics.
But with *Either* we get the same result as before.
Because *Either* has a Monad instance.

```
val result: Either[List[String], Int] =
  (5.asRight[List[String]],
    List("Error 1").asLeft[Int],
    List("Error 2").asLeft[Int] // List("Error 2") is lost!
  ) mapN ((_: Int) + (_: Int) + (_: Int))

// result: Either[List[String],Int] = Left(List(Error 1))
```

Applicative processing with *Validated*

cats.data.Validated is designed in analogy to *Either*.
Instead of *Right* and *Left* it has *Valid* and *Invalid*.
Validated has an Applicative instance but no Monad instance.

Applicative processing with *Validated*

cats.data.Validated is designed in analogy to *Either*.
Instead of *Right* and *Left* it has *Valid* and *Invalid*.
Validated has an Applicative instance but no Monad instance.

```scala
val result: Validated[List[String], Int] =
  (5.valid[List[String]],
    List("Error 1").invalid[Int],
    List("Error 2").invalid[Int] // List("Error 2") is preserved!
  ) mapN ((_: Int) + (_: Int) + (_: Int))

// result: cats.data.Validated[List[String],Int] = Invalid(List(Error 1, Error 2))
```

Conversions between *Either* and *Validated*

Conversion is supported in both directions
Either#toValidated converts an *Either*. to a *Validated.*
Validated#toEither *converts a* Validated *to an* Either.

Conversions between *Either* and *Validated*

Conversion is supported in both directions
Either#toValidated converts an *Either*. to a *Validated.*
Validated#toEither *converts a* Validated *to an* Either.

```
val result: Validated[List[String], Int] =
  (5.asRight[List[String]].toValidated,
    List("Error 1").asLeft[Int].toValidated,
    List("Error 2").asLeft[Int].toValidated // List("Error 2") is preserved!
  ) mapN ((_: Int) + (_: Int) + (_: Int))

// result: cats.data.Validated[List[String],Int] = Invalid(List(Error 1, Error 2))

val resultAsEither = result.toEither
// resultAsEither: Either[List[String],Int] = Left(List(Error 1, Error 2))
```

8. Traversals

See: *examples.Traversals*

Recap: *Future.traverse* and *Future.sequence*

Most Scala devs know these methods defined on the *Future* companion object.

Recap: *Future.traverse* and *Future.sequence*

Most Scala devs know these methods defined on the *Future* companion object.

Let's start with *Future.sequence*. It is the simpler one.

Future.sequence API doc: *Simple version of Future.traverse. Asynchronously and non-blockingly transforms a TraversableOnce[Future[A]] into a Future[TraversableOnce[A]]. Useful for reducing many Futures into a single Future.*

Recap: *Future.traverse* and *Future.sequence*

Most Scala devs know these methods defined on the *Future* companion object.

Let's start with *Future.sequence*. It is the simpler one.

Future.sequence API doc: *Simple version of Future.traverse. Asynchronously and non-blockingly transforms a TraversableOnce[Future[A]] into a Future[TraversableOnce[A]]. Useful for reducing many Futures into a single Future.*

Simply spoken: *sequence* turns a *List[Future[A]]* into a *Future[List[A]]*.

```
val lfd: List[Future[Double]] = List(Future(2.0), Future(4.0), Future(6.0))
val fld: Future[List[Double]] = Future.sequence(lfd)
```

Recap: *Future.traverse* and *Future.sequence*

Most Scala devs know these methods defined on the *Future* companion object.

Let's start with *Future.sequence*. It is the simpler one.

Future.sequence API doc: *Simple version of Future.traverse. Asynchronously and non-blockingly transforms a TraversableOnce[Future[A]] into a Future[TraversableOnce[A]]. Useful for reducing many Futures into a single Future.*

Simply spoken: *sequence* turns a *List[Future[A]]* into a *Future[List[A]]*.

```
val lfd: List[Future[Double]] = List(Future(2.0), Future(4.0), Future(6.0))
val fld: Future[List[Double]] = Future.sequence(lfd)
```

Typically in an application we don't have a *List[Future[B]]*, but we produce it by processing a *List[A]* asynchronously. We can first map the *A => List[Future[B]]* and then invoke *sequence* on the resulting list.

Recap: *Future.traverse* and *Future.sequence*

Most Scala devs know these methods defined on the *Future* companion object.

Let's start with *Future.sequence*. It is the simpler one.

Future.sequence API doc: *Simple version of Future.traverse. Asynchronously and non-blockingly transforms a TraversableOnce[Future[A]] into a Future[TraversableOnce[A]]. Useful for reducing many Futures into a single Future.*

Simply spoken: *sequence* turns a *List[Future[A]]* into a *Future[List[A]]*.

```
val lfd: List[Future[Double]] = List(Future(2.0), Future(4.0), Future(6.0))
val fld: Future[List[Double]] = Future.sequence(lfd)
```

Typically in an application we don't have a *List[Future[B]]*, but we produce it by processing a *List[A]* asynchronously. We can first map the *A => List[Future[B]]* and then invoke *sequence* on the resulting list.

```
val li: List[Int] = List(1, 2, 3)
val doubleItAsync: Int => Future[Double] = x => Future { x * 2.0 }
val lfi: List[Future[Double]] = li.map(doubleItAsync)
val fld: Future[List[Double]] = Future.sequence(lfi)
```

Recap: *Future.traverse* and *Future.sequence*

Mapping and sequencing traverses the list twice.
traverse fuses mapping and sequencing into a single traversal.

Recap: *Future.traverse* and *Future.sequence*

Mapping and sequencing traverses the list twice.
traverse fuses mapping and sequencing into a single traversal.

Future.traverse takes a *List[A]* and a mapping function from *A => Future[B]* and returns a *Future[List[B]]*.

```
val li: List[Int] = List(1, 2, 3)
val doubleItAsync: Int => Future[Double] = x => Future { x * 2.0 }
val fld: Future[List[Double]] = Future.traverse(li)(doubleItAsync)
```

Recap: *Future.traverse* and *Future.sequence*

Mapping and sequencing traverses the list twice.
traverse fuses mapping and sequencing into a single traversal.

Future.traverse takes a *List[A]* and a mapping function from *A* => *Future[B]* and returns a *Future[List[B]]*.

```scala
val li: List[Int] = List(1, 2, 3)
val doubleItAsync: Int => Future[Double] = x => Future { x * 2.0 }
val fld: Future[List[Double]] = Future.traverse(li)(doubleItAsync)
```

Providing *identity* as mapping function to *traverse* has the same effect as *sequence*.

```scala
val lfd: List[Future[Double]] = List(Future(2.0), Future(4.0), Future(6.0))
val fld: Future[List[Double]] = Future.traverse(lfd)(identity)
```

Traverse

Cats generalizes this concept into the Traverse typeclass.

Traverse

Cats generalizes this concept into the Traverse typeclass.

Mentally replace *List* by *F[_]: Foldable, Functor* and *Future* by *G[_]: Applicative*.

Traverse

Cats generalizes this concept into the Traverse typeclass.

Mentally replace *List* by *F[_]: Foldable, Functor* and *Future* by *G[_]: Applicative*.

```
trait Traverse[F[_]] extends Foldable[F] with Functor[F] { // simplified

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

  def sequence[G[_], A](fga: F[G[A]])(implicit AG: Applicative[G]): G[F[A]] =
    traverse(fga)(identity)

  // map in terms of traverse using the Id context
  override def map[A, B](fa: F[A])(f: A => B): F[B] = traverse(fa)(f: A => Id[B])
}
```

Traverse

Cats generalizes this concept into the Traverse typeclass.

Mentally replace *List* by *F[_]: Foldable, Functor* and *Future* by *G[_]: Applicative*.

```
trait Traverse[F[_]] extends Foldable[F] with Functor[F] { // simplified

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

  def sequence[G[_], A](fga: F[G[A]])(implicit AG: Applicative[G]): G[F[A]] =
    traverse(fga)(identity)

  // map in terms of traverse using the Id context
  override def map[A, B](fa: F[A])(f: A => B): F[B] = traverse(fa)(f: A => Id[B])
}
```

Both methods require the *G[_]* to be an Applicative (*Future* before).

Traverse

Cats generalizes this concept into the Traverse typeclass.

Mentally replace *List* by *F[_]: Foldable, Functor* and *Future* by *G[_]: Applicative*.

```
trait Traverse[F[_]] extends Foldable[F] with Functor[F] { // simplified

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

  def sequence[G[_], A](fga: F[G[A]])(implicit AG: Applicative[G]): G[F[A]] =
    traverse(fga)(identity)

  // map in terms of traverse using the Id context
  override def map[A, B](fa: F[A])(f: A => B): F[B] = traverse(fa)(f: A => Id[B])
}
```

Both methods require the *G[_]* to be an Applicative (*Future* before).

The structure traversed over (*List* before) must be *Foldable with Functor*.
Hence *Traverse* extends these two traits.

Traverse

Cats generalizes this concept into the Traverse typeclass.

Mentally replace *List* by *F[_]: Foldable*, *Functor* and *Future* by *G[_]: Applicative*.

```
trait Traverse[F[_]] extends Foldable[F] with Functor[F] { // simplified

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

  def sequence[G[_], A](fga: F[G[A]])(implicit AG: Applicative[G]): G[F[A]] =
    traverse(fga)(identity)

  // map in terms of traverse using the Id context
  override def map[A, B](fa: F[A])(f: A => B): F[B] = traverse(fa)(f: A => Id[B])
}
```

Both methods require the *G[_]* to be an Applicative (*Future* before).

The structure traversed over (*List* before) must be *Foldable with Functor*.
Hence *Traverse* extends these two traits.

The *traverse* function can indeed be implemented with *foldLeft* or *foldRight*.
On the other hand *foldLeft*, *foldRight* and *map* can be implemented with *traverse*.

Traverse in practice

Instead of *Future.sequence* we can now use *Traverse[List].sequence*.

Traverse in practice

Instead of *Future.sequence* we can now use *Traverse[List].sequence*.

```
val lfd: List[Future[Double]] = List(Future(2.0), Future(4.0), Future(6.0))
val fld1: Future[List[Double]] = Traverse[List].sequence(lfd)
import cats.syntax.traverse._ // supports 'sequence' as an enrichment of List
val fld2: Future[List[Double]] = lfd.sequence
```

Traverse in practice

Instead of *Future.sequence* we can now use *Traverse[List].sequence*.

```scala
val lfd: List[Future[Double]] = List(Future(2.0), Future(4.0), Future(6.0))
val fld1: Future[List[Double]] = Traverse[List].sequence(lfd)
import cats.syntax.traverse._ // supports 'sequence' as an enrichment of List
val fld2: Future[List[Double]] = lfd.sequence
```

Instead of *Future.traverse* we can now use *Traverse[List].traverse*.

Traverse in practice

Instead of *Future.sequence* we can now use *Traverse[List].sequence*.

```
val lfd: List[Future[Double]] = List(Future(2.0), Future(4.0), Future(6.0))
val fld1: Future[List[Double]] = Traverse[List].sequence(lfd)
import cats.syntax.traverse._ // supports 'sequence' as an enrichment of List
val fld2: Future[List[Double]] = lfd.sequence
```

Instead of *Future.traverse* we can now use *Traverse[List].traverse*.

```
val li: List[Int] = List(1, 2, 3)
val doubleItAsync: Int => Future[Double] = x => Future { x * 2.0 }
val fld1: Future[List[Double]] = Traverse[List].traverse(li)(doubleItAsync)
import cats.syntax.traverse._ // supports 'traverse' as an enrichment of List
val fld2: Future[List[Double]] = li traverse doubleItAsync
```

Traverse in practice

Instead of *Future.sequence* we can now use *Traverse[List].sequence*.

```scala
val lfd: List[Future[Double]] = List(Future(2.0), Future(4.0), Future(6.0))
val fld1: Future[List[Double]] = Traverse[List].sequence(lfd)
import cats.syntax.traverse._ // supports 'sequence' as an enrichment of List
val fld2: Future[List[Double]] = lfd.sequence
```

Instead of *Future.traverse* we can now use *Traverse[List].traverse*.

```scala
val li: List[Int] = List(1, 2, 3)
val doubleItAsync: Int => Future[Double] = x => Future { x * 2.0 }
val fld1: Future[List[Double]] = Traverse[List].traverse(li)(doubleItAsync)
import cats.syntax.traverse._ // supports 'traverse' as an enrichment of List
val fld2: Future[List[Double]] = li traverse doubleItAsync
```

We now can traverse over any other Foldable that has a *Traverse* instance, e.g. Vector. The mapping function may produce any Applicate, e.g. Option.

Traverse in practice

Instead of *Future.sequence* we can now use *Traverse[List].sequence*.

```
val lfd: List[Future[Double]] = List(Future(2.0), Future(4.0), Future(6.0))
val fld1: Future[List[Double]] = Traverse[List].sequence(lfd)
import cats.syntax.traverse._ // supports 'sequence' as an enrichment of List
val fld2: Future[List[Double]] = lfd.sequence
```

Instead of *Future.traverse* we can now use *Traverse[List].traverse*.

```
val li: List[Int] = List(1, 2, 3)
val doubleItAsync: Int => Future[Double] = x => Future { x * 2.0 }
val fld1: Future[List[Double]] = Traverse[List].traverse(li)(doubleItAsync)
import cats.syntax.traverse._ // supports 'traverse' as an enrichment of List
val fld2: Future[List[Double]] = li traverse doubleItAsync
```

We now can traverse over any other Foldable that has a *Traverse* instance, e.g. Vector. The mapping function may produce any Applicate, e.g. Option.

```
val vi: Vector[Int] = Vector(3, 2, 1)
val divideBy: Int => Option[Double] = x => if (x == 0) None else Some { 6.0 / x }
val ovd1: Option[Vector[Double]] = Traverse[Vector].traverse(vi)(divideBy)
import cats.syntax.traverse._ // supports 'traverse' as an enrichment of Vector
val ovd2: Option[Vector[Double]] = vi traverse divideBy
```

9. Resources

Resources (1/2)

- Code and Slides of this Talk:
  https://github.com/hermannhueck/use-applicative-where-applicable
- Cats documentation:
  https://typelevel.org/cats/typeclasses/applicative.html
  https://typelevel.org/cats/typeclasses/traverse.html
  https://typelevel.org/cats/datatypes/validated.html
- Herding Cats, Day 3:
  http://eed3si9n.com/herding-cats/Semigroupal.html
  http://eed3si9n.com/herding-cats/Apply.html
  http://eed3si9n.com/herding-cats/Applicative.html
- "Scala with Cats", Chapters 6 and 7
  Book by Noel Welsh and Dave Gurnell
  https://underscore.io/books/scala-with-cats/

Resources (2/2)

- Learn You a Haskell for Great Good!, Chapter 11
  Online book by Miran Lipovaca
  http://learnyouahaskell.com/functors-applicative-functors-and-monoids

- Applicative Programming with Effects
  Conor McBride and Ross Paterson in Journal of Functional Programming 18:1 (2008), pages 1-13
  http://www.staff.city.ac.uk/~ross/papers/Applicative.pdf

- The Essence of the Iterator Pattern
  Jeremy Gibbons and Bruno C. d. S. Oliveira, Oxford University Computing Laboratory
  https://www.cs.ox.ac.uk/jeremy.gibbons/publications/iterator.pdf

Thanks for Listening

Q & A

https://github.com/hermannhueck/use-applicative-where-applicable