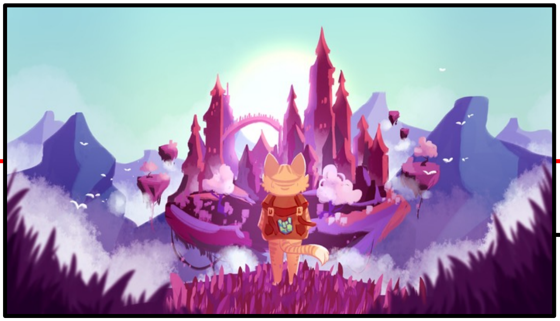
a sighting of

filterA

in

Typelevel Rite of Passage



Build a Full-Stack Application with Scala 3 and the Typelevel Stack

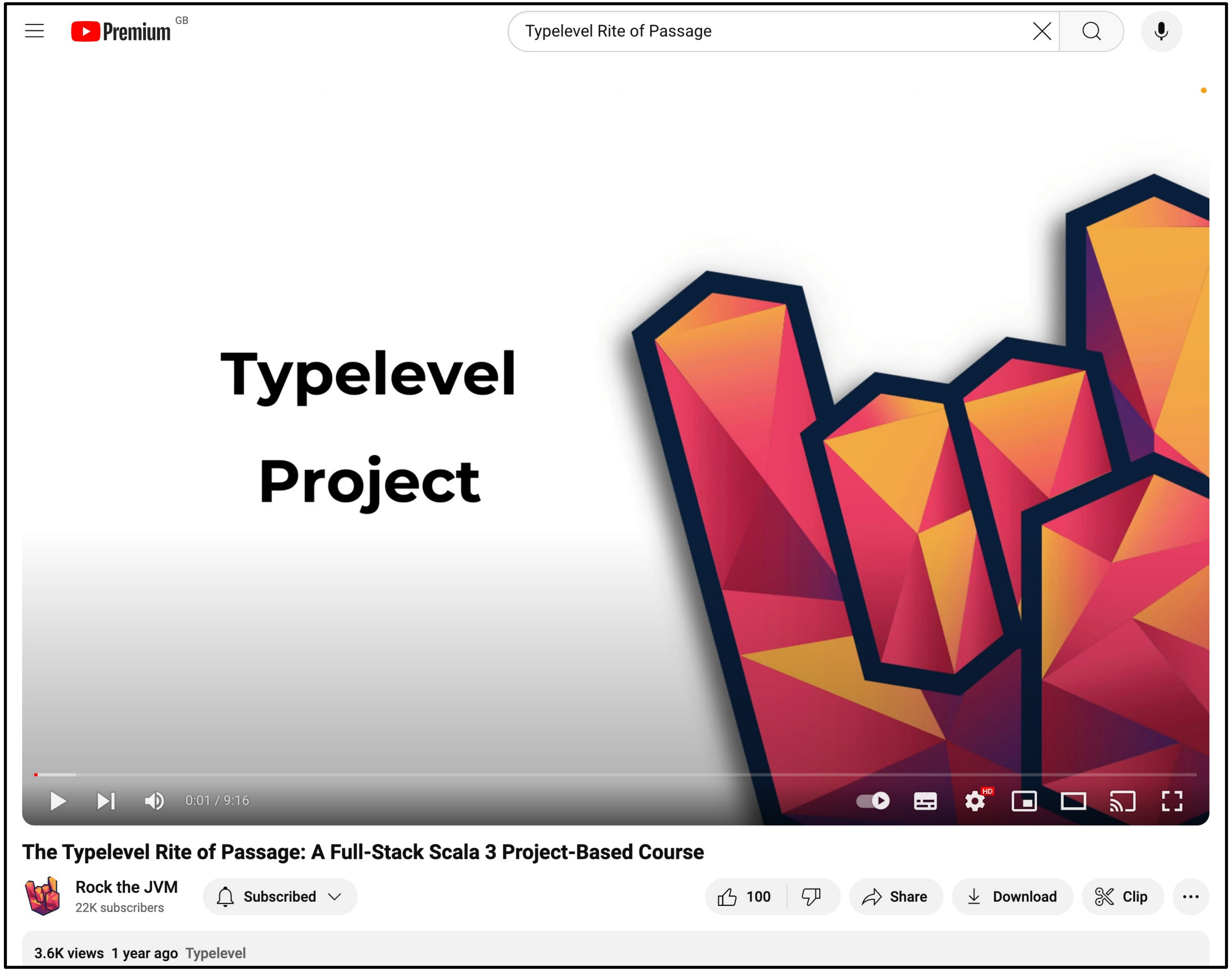
by

Daniel Ciocirlan
@rockthejvm
in/danielciocirlan/
/company/rockthejvm/

slides by  @philip_schwarz FP IλλUminated http://fpilluminated.com/

This deck is based on about 60 seconds of a 35-hour video course called **Typelevel Right of Passage**, an introduction to which is available on **YouTube**.

@philip_schwarz



Daniel Ciocirlan

@rockthejvm

The Typelevel Rite of Passage

I am going to compare this **user**'s **password** against the **hash** that is already stored in the database.

```scala
def login(email: String, password: String): F[Option[JwtToken]] =

  for
    // find the user in the DB - return None if no user
    maybeUser <- users.find(email)

    // check password
    maybeValidatedUser <- maybeUser.filter(user =>
      BCrypt.checkpwBool[F](password, PasswordHash[BCrypt](user.hashedPassword)))

    // Return a new token if password matches
    maybeJwtToken <- maybeValidatedUser.traverse(user => authenticator.create(user.email))

  yield maybeJwtToken
```

```scala
def find(email: String): F[Option[User]]
```

BCrypt

```scala
/** Check against a bcrypt hash in a pure way
  *
  * It may raise an error for a malformed hash
  */
def checkpwBool[F[_]](password: String, hash: PasswordHash[A])
                     (implicit P: PasswordHasher[F, A]): F[Boolean] = …
```

Daniel Ciocirlan

@rockthejvm

The Typelevel Rite of Passage

However, the call to checkpwBool returns an F[Boolean], so our types are a little bit screwed up here.

```scala
def login(email: String, password: String): F[Option[JwtToken]] =

  for
    // find the user in the DB - return None if no user
    maybeUser <- users.find(email)

    // check password
    maybeValidatedUser <- maybeUser.filter(user =>
      BCrypt.checkpwBool[F](password, PasswordHash[BCrypt](user.hashedPassword)))

    // Return a new token if password matches
    maybeJwtToken <- maybeValidatedUser.traverse(user => authenticator.create(user.email))

  yield maybeJwtToken
```

```scala
def find(email: String): F[Option[User]]
```

BCrypt

```scala
/** Check against a bcrypt hash in a pure way
  *
  * It may raise an error for a malformed hash
  */
def checkpwBool[F[_]](password: String, hash: PasswordHash[A])
                     (implicit P: PasswordHasher[F, A]): F[Boolean] = …
```

Daniel Ciocirlan

@rockthejvm

The Typelevel Rite of Passage

We have an Option[User], and then I am going to call filter on a function User => IO[Boolean], and I would need to return an IO[Option[User]], so that later I can use maybeValidatedUser.

```scala
def login(email: String, password: String): F[Option[JwtToken]] =

  for
    // find the user in the DB - return None if no user
    maybeUser <- users.find(email)

    // Option[User].filter(User => IO[Boolean]) => IO[Option[User]]
    maybeValidatedUser <- maybeUser.filter(user =>
      BCrypt.checkpwBool[F](password, PasswordHash[BCrypt](user.hashedPassword)))

    // Return a new token if password matches
    maybeJwtToken <- maybeValidatedUser.traverse(user => authenticator.create(user.email))

  yield maybeJwtToken
```

```scala
def find(email: String): F[Option[User]]
```

BCrypt

```scala
/** Check against a bcrypt hash in a pure way
  *
  * It may raise an error for a malformed hash
  */
def checkpwBool[F[_]](password: String, hash: PasswordHash[A])
                     (implicit P: PasswordHasher[F, A]): F[Boolean] = …
```

Daniel Ciocirlan

@rockthejvm

The Typelevel Rite of Passage

At this point we have two **improper types**. One is the IO[Boolean], due to the fact that filter does not accept a function returning an IO[Boolean], but rather a simple Boolean.

```
def login(email: String, password: String): F[Option[JwtToken]] =

  for
    // find the user in the DB - return None if no user
    maybeUser <- users.find(email)

    // Option[User].filter(User => IO[Boolean]) => IO[Option[User]]
    maybeValidatedUser <- maybeUser.filter(user =>
      BCrypt.checkpwBool[F](password, PasswordHash[BCrypt](user.hashedPassword)))

    // Return a new token if password matches
    maybeJwtToken <- maybeValidatedUser.traverse(user => authenticator.create(user.email))

  yield maybeJwtToken
```

```
def find(email: String): F[Option[User]]
```

BCrypt

```
/** Check against a bcrypt hash in a pure way
  *
  * It may raise an error for a malformed hash
  */
def checkpwBool[F[_]](password: String, hash: PasswordHash[A])
                     (implicit P: PasswordHasher[F, A]): F[Boolean] = …
```

Daniel Ciocirlan

@rockthejvm

The Typelevel Rite of Passage

And the other is the return value of **filter**, because it is an `Option[User]`, rather than an effect wrapping `Option[User]`.

```
def login(email: String, password: String): F[Option[JwtToken]] =

  for
    // find the user in the DB - return None if no user
    maybeUser <- users.find(email)

    // Option[User].filter(User => IO[Boolean]) => IO[Option[User]]
    maybeValidatedUser <- maybeUser.filter(user =>
      BCrypt.checkpwBool[F](password, PasswordHash[BCrypt](user.hashedPassword)))

    // Return a new token if password matches
    maybeJwtToken <- maybeValidatedUser.traverse(user => authenticator.create(user.email))

  yield maybeJwtToken
```

```
def find(email: String): F[Option[User]]
```

BCrypt

```
/** Check against a bcrypt hash in a pure way
  *
  * It may raise an error for a malformed hash
  */
def checkpwBool[F[_]](password: String, hash: PasswordHash[A])
                     (implicit P: PasswordHasher[F, A]): F[Boolean] = …
```

Daniel Ciocirlan

@rockthejvm

The Typelevel Rite of Passage

Which is why I am going to use a **little trick**. I am going to call **filterA**, which is an **extension method** (I think it comes from the **Traverse** typeclass).

On the **Option** of a particular type [e.g. **User**], you can call **filterA** with a function returning a different kind of effect **G** wrapping a **Boolean**, so you'll then return an effect **G** wrapping this **Option**[**User**].

```
def login(email: String, password: String): F[Option[JwtToken]] =

  for
    // find the user in the DB - return None if no user
    maybeUser <- users.find(email)

    // Option[User].filter(User => G[Boolean]) => G[Option[User]]
    maybeValidatedUser <- maybeUser.filterA(user =>
      BCrypt.checkpwBool[F](password, PasswordHash[BCrypt](user.hashedPassword)))

    // Return a new token if password matches
    maybeJwtToken <- maybeValidatedUser.traverse(user => authenticator.create(user.email))

  yield maybeJwtToken
```

```
def find(email: String): F[Option[User]]
```

BCrypt

```
/** Check against a bcrypt hash in a pure way
  *
  * It may raise an error for a malformed hash
  */
def checkpwBool[F[_]](password: String, hash: PasswordHash[A])
                     (implicit P: PasswordHasher[F, A]): F[Boolean] = …
```

filterA is to **traverse** what **filter** is to **map**

Function	From	Given	To	Type Class
`map`	`F[A]`	`A => B`	`F[B]`	`Functor[F]`
`filter`	`F[A]`	`A => Boolean`	`F[A]`	`FunctorFilter[F]`
`traverse`	`F[A]`	`A => G[B]`	`G[F[B]]`	`Traverse[F]`
`filterA`	`F[A]`	`A => G[Boolean]`	`G[F[A]]`	`TraverseFilter[F]`

`G` is an `Applicative` (every `Monad` is an `Applicative`)


```scala
assert(List(1,2,3,4).map(_.toString) == List("1","2","3","4"))
```

map	F[A]	A => B	F[B]	Functor[F]

```scala
def isEven(n: Int): Boolean = n % 2 == 0

assert(List(1,2,3,4).filter(isEven) == List(2,4))
```

filter	F[A]	A => Boolean	F[A]	FunctorFilter[F]

```scala
def maybeDigit(c: Char): Option[Int] =
  Option.when(c.isDigit)(c.asDigit)

assert(List('1','2','3','4').traverse(maybeDigit) == Some(List(1,2,3,4)))
assert(List('1','2','x','4').traverse(maybeDigit) == None)
```

traverse	F[A]	A => G[B]	G[F[B]]	Traverse[F]

```scala
def maybeEvenDigit(c: Char): Option[Boolean] =
  maybeDigit(c).map(isEven)

assert(List('1','2','3','4').filterA(maybeEvenDigit) == Some(List('2','4')))
assert(List('1','2','x','4').filterA(maybeEvenDigit) == None)
```

filterA	F[A]	A => G[Boolean]	G[F[A]]	TraverseFilter[F]

A = Int
B = String
F = List

A = Char
B = Int
F = List
G = Option

Function	From	Given	To
filterA	F[A]	A => G[Boolean]	G[F[A]]

```scala
def isEven(n: Int): Boolean = n % 2 == 0

def maybeDigit(c: Char): Option[Int] = Option.when(c.isDigit)(c.asDigit)

def maybeIsEvenDigit(c: Char): Option[Boolean] = maybeDigit(c).map(isEven)

assert(List('1','2','3','4').filterA(maybeIsEvenDigit) == Some(List('2','4')))
assert(List('1','2','x','4').filterA(maybeIsEvenDigit) == None)
```

filterA	List[Char]	Char => Option[Boolean]	Option[List[Char]]

Here we compare our example of using **filterA**, with the usage of **filterA** seen in the course.

```scala
def checkpwBool[F[_]](p: String, hash: PasswordHash[A])(implicit P: PasswordHasher[F, A]): F[Boolean] = …

for
  // find the user in the DB - return None if no user
  maybeUser <- users.findEmail(email)

  // check password - Option[User].filter(User => IO[Boolean]) => IO[Option[User]]
  maybeValidatedUser <- maybeUser.filterA(user =>
    BCrypt.checkpwBool[F](p, PasswordHash[BCrypt](user.hashedPassword)))
```

filterA	Option[User]	User => IO[Boolean]	IO[Option[User]]

Obviously the behaviour of **filterA**, which is reflected in its result, depends on the behaviour of a particular **Applicative G**.

So far we have seen examples with **G** = **Option** and **G** = **IO**.

Just as an example of the type of behaviour that we can achieve when **G** = **List**, here is a function that uses **filterA** to compute the **powerset** of a set (the list of sublists of a list).

```
import cats.implicits.*

def powerset[A](as: List[A]): List[List[A]] = as.filterA(_ => List(true, false))

assert(powerset(List(1, 2, 3))
       ==
       List(List(1, 2, 3),
            List(1, 2),
            List(1, 3),
            List(1),
            List(2, 3),
            List(2),
            List(3),
            List()
       )
      )
```

```
A = Int
F = List
G = List
```

Function	From	Given	To
filterA	F[A]	A => G[Boolean]	G[F[A]]
filterA	List[Int]	Int => List[Boolean]	List[List[Int]]

I just realised the using **filterA** to compute a **powerset** is actually one of the examples in the documentation of **filterA**!

cats.TraverseFilter

```
def filterA[G[_], A](fa: F[A])(f: A => G[Boolean])(implicit G: Applicative[G]): G[F[A]]
```

Filter values inside a G context.

This is a generalized version of Haskell's filterM ↗ . This StackOverflow question ↗ about filterM may be helpful in understanding how it behaves.

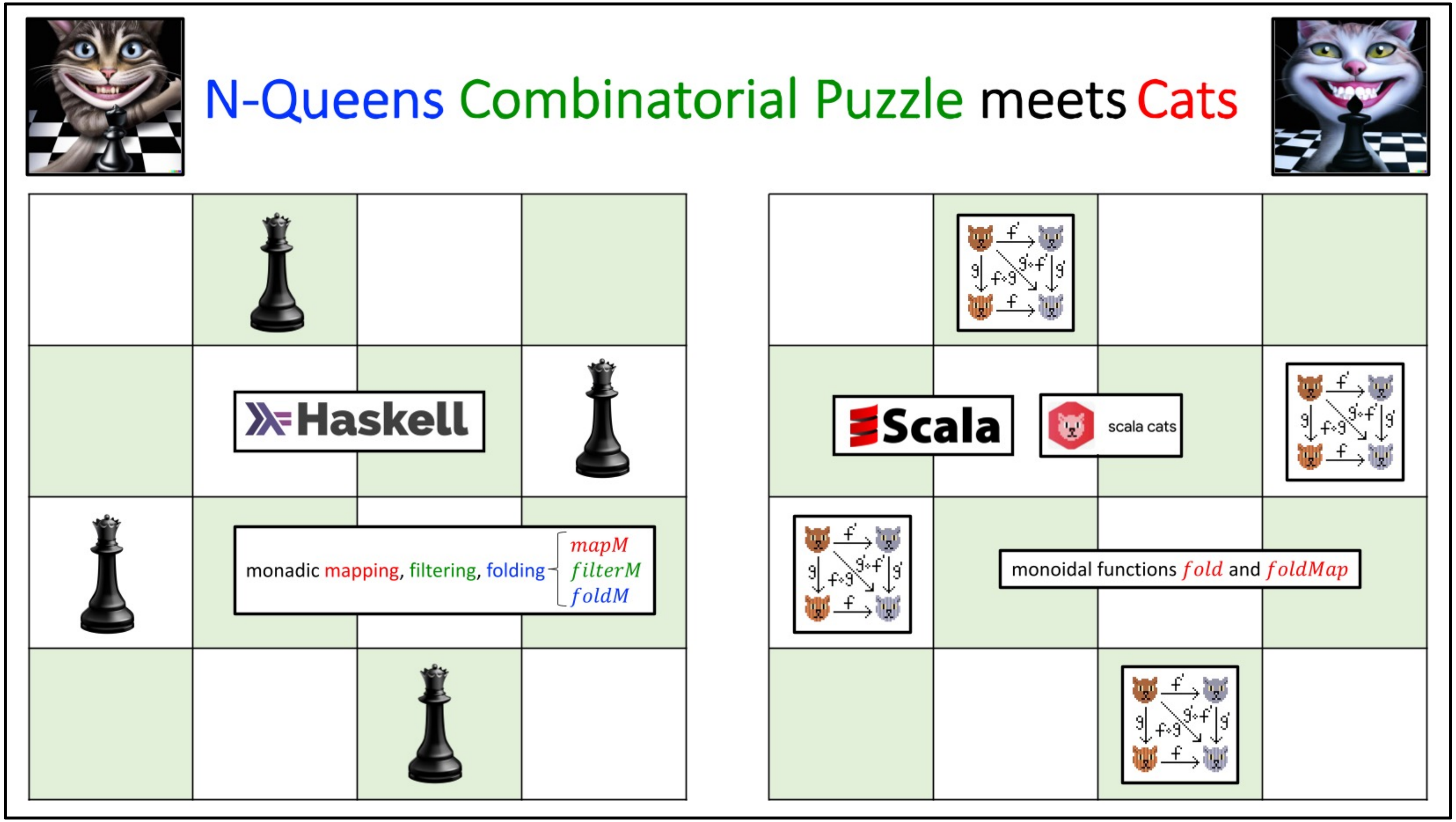
Example:

```
scala> import cats.implicits._
scala> val l: List[Int] = List(1, 2, 3, 4)
scala> def odd(i: Int): Eval[Boolean] = Now(i % 2 == 1)
scala> val res: Eval[List[Int]] = l.filterA(odd)
scala> res.value
res0: List[Int] = List(1, 3)

scala> List(1, 2, 3).filterA(_ => List(true, false))
res1: List[List[Int]] = List(List(1, 2, 3), List(1, 2), List(1, 3), List(1), List(2, 3),
List(2), List(3), List())
```

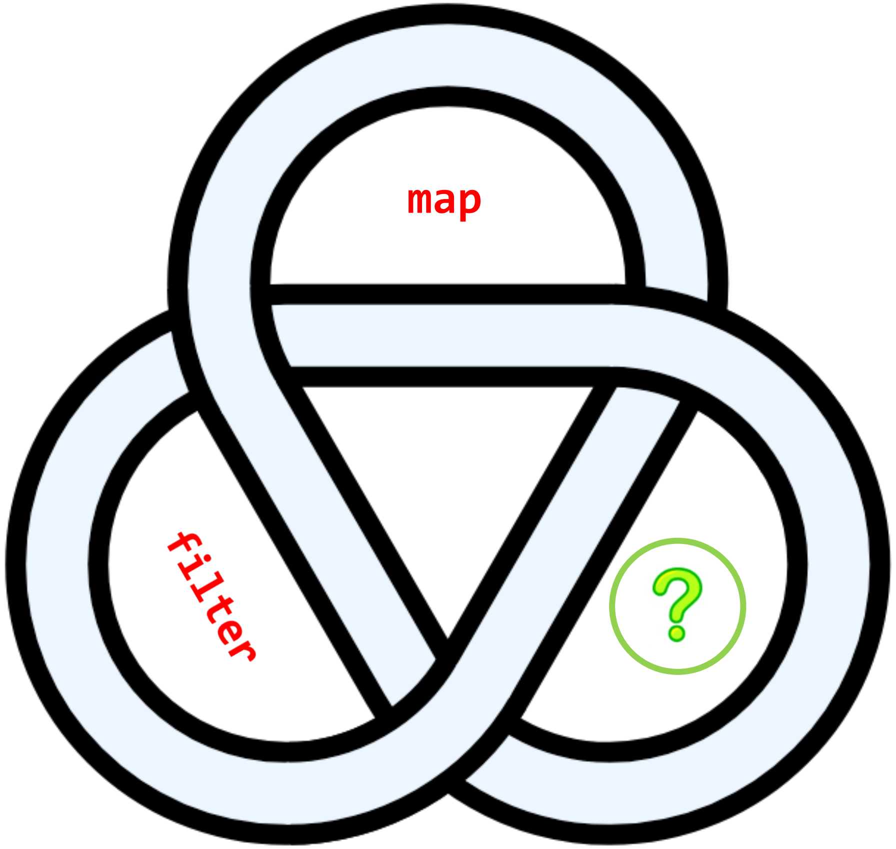
Cats

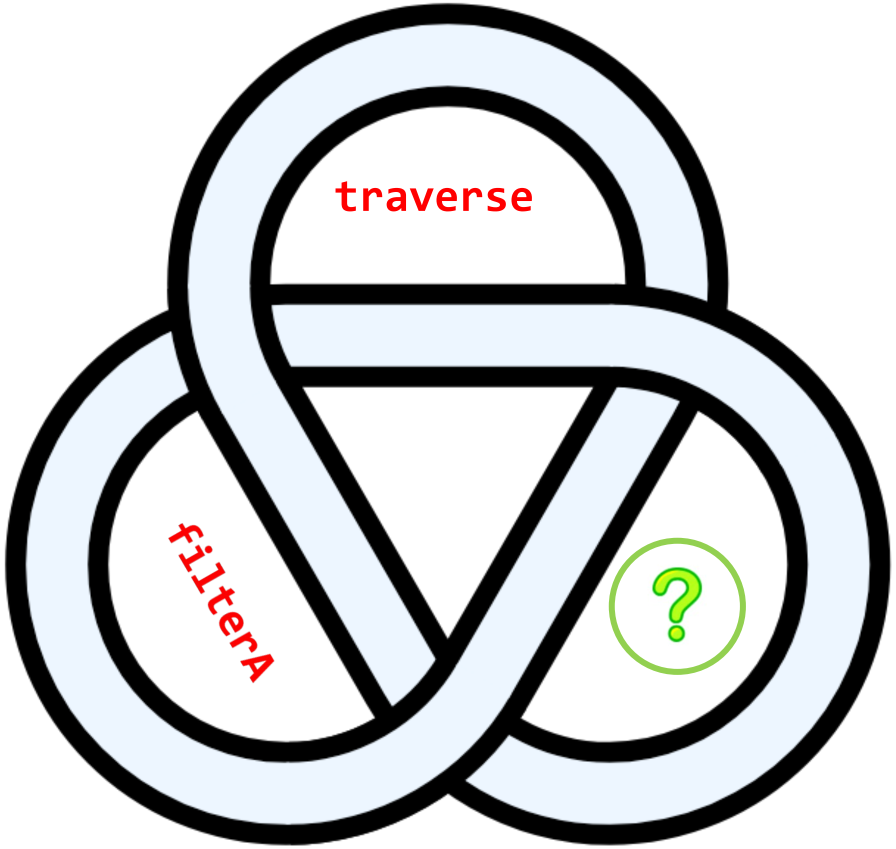
If you want to know more about how **filterA** works when computing the **`powerset`** function, see slides 256 – 276 of the following slide deck.



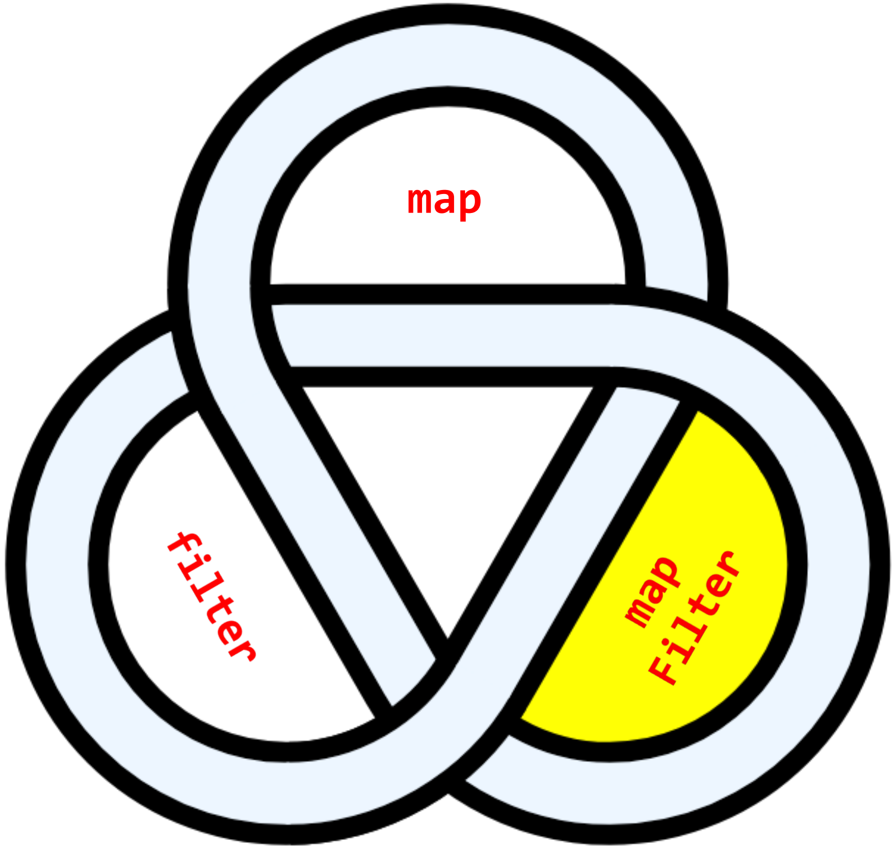
By the way, did you know that there is a further function that is a **combination** of **map** and **filter**, and another one that is a **combination** of **traverse** and **filterA**?

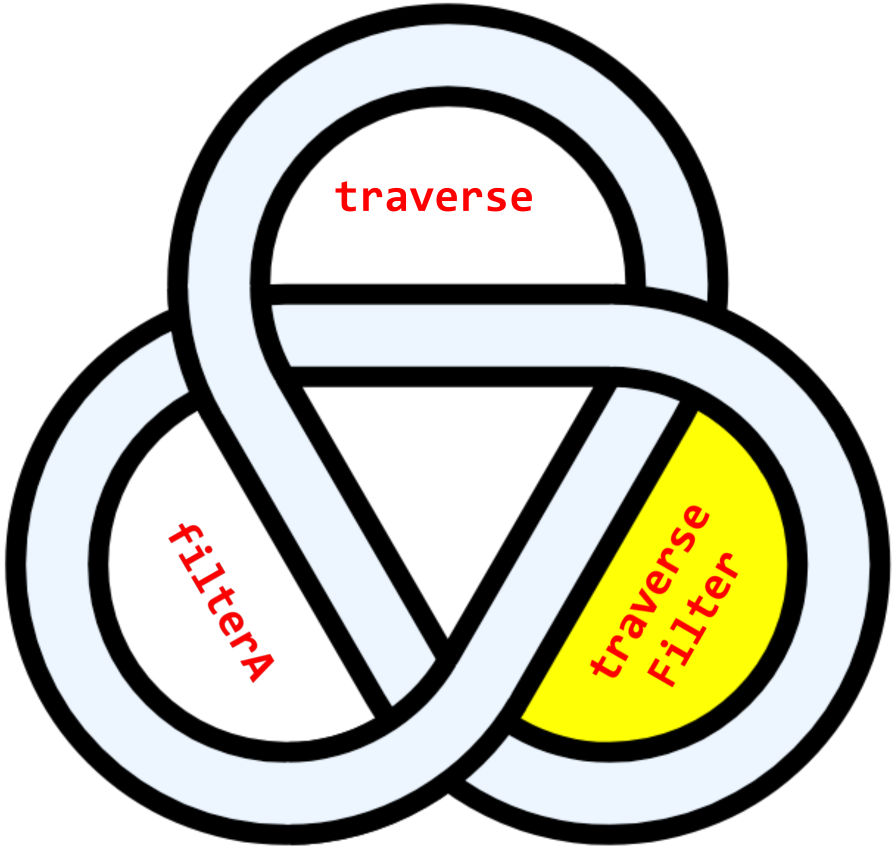
@philip_schwarz





Not surprisingly, the functions are called **mapFilter** and **traverseFilter**.





cats.FunctorFilter

```scala
def mapFilter[A, B](fa: F[A])(f: A => Option[B]): F[B]
```

A combined map and filter. Filtering is handled via Option instead of Boolean such that the output type B can be different than the input type A.

Example:

```scala
scala> import cats.implicits._
scala> val m: Map[Int, String] = Map(1 -> "one", 3 -> "three")
scala> val l: List[Int] = List(1, 2, 3, 4)
scala> def asString(i: Int): Option[String] = m.get(i)
scala> l.mapFilter(asString)
res0: List[String] = List(one, three)
```

Cats

cats.TraverseFilter

```scala
def traverseFilter[G[_], A, B](fa: F[A])(f: A => G[Option[B]])(implicit G: Applicative[G]): G[F[B]]
```

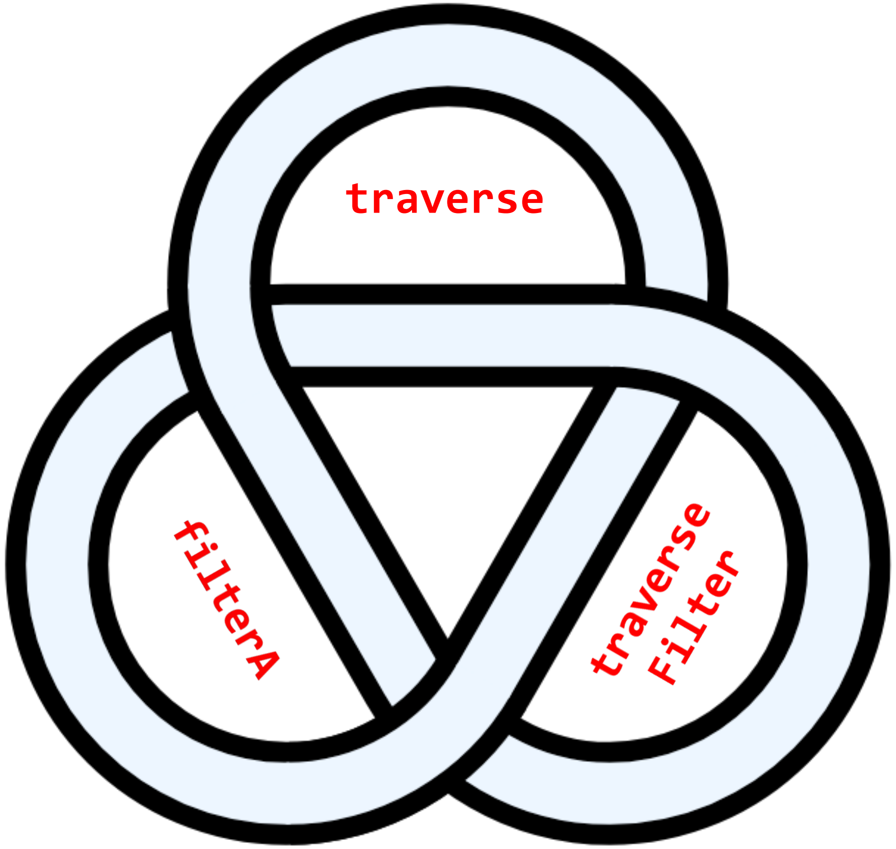
A combined `traverse` and `filter`. Filtering is handled via Option instead of Boolean such that the output type B can be different than the input type A.

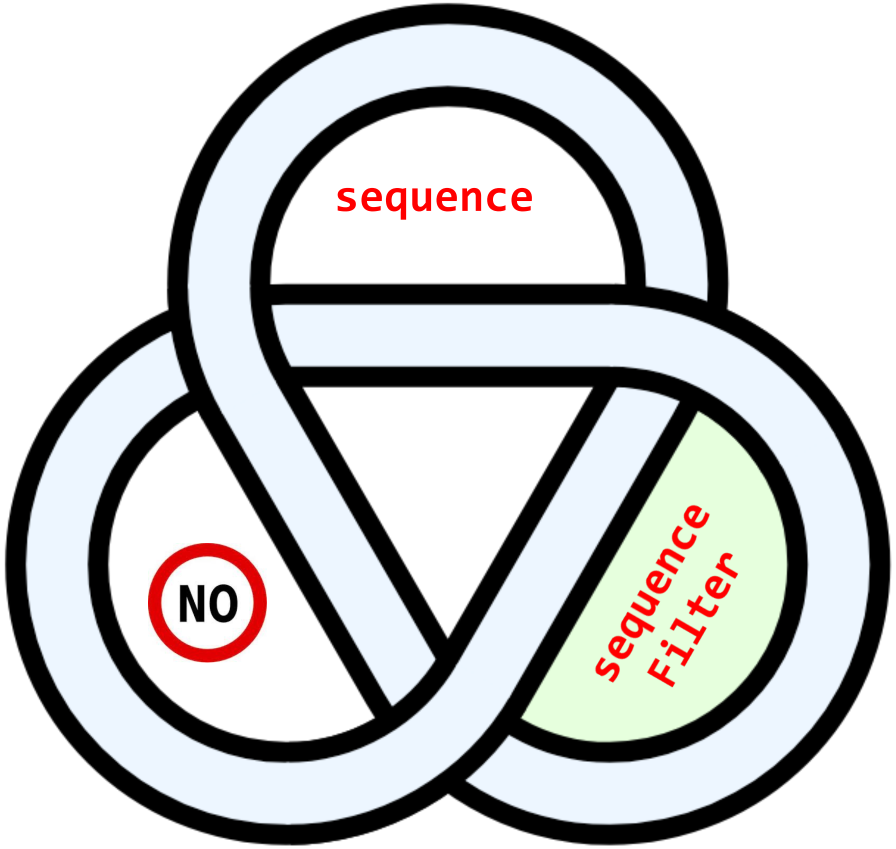
Example:

```scala
scala> import cats.implicits._
scala> val m: Map[Int, String] = Map(1 -> "one", 3 -> "three")
scala> val l: List[Int] = List(1, 2, 3, 4)
scala> def asString(i: Int): Eval[Option[String]] = Now(m.get(i))
scala> val result: Eval[List[String]] = l.traverseFilter(asString)
scala> result.value
res0: List[String] = List(one, three)
```

What about **sequence**, which is closely related to **traverse**?

Because **sequence** is just **traverse(x => x)**, it doesn't make sense for the **sequence** equivalent of **filterA** to exist, but **sequenceFilter** does exist (see next slide).

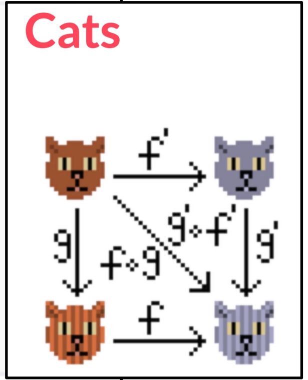




cats.TraverseFilter

```
def sequenceFilter[G[_], A](fgoa: F[G[Option[A]]])(implicit G: Applicative[G]): G[F[A]]
```

```
scala> import cats.implicits._
scala> val a: List[Either[String, Option[Int]]] = List(Right(Some(1)), Right(Some(5)),
Right(Some(3)))
scala> val b: Either[String, List[Int]] = TraverseFilter[List].sequenceFilter(a)
b: Either[String, List[Int]] = Right(List(1, 5, 3))
```



See next slide for a few more variations of the example given above.

Scastie — Feedback — philipschwarz

Editor | Run | New | Format | Clear Messages | Worksheet | Download | Embed | Metals Status

Build Settings

```scala
import cats.implicits.*
import cats.TraverseFilter

val a: List[Either[String, Option[Int]]] = List(Right(Some(1)), Right(Some(5)), Right(Some(3)))
a.sequenceFilter Right(List(1, 5, 3)): scala.util.Either[scala.Predef.String, scala.collection.immutable.List[scala.Int]]

val b: List[Either[String, Option[Int]]] = List(Right(Some(1)), Left("boom"), Right(Some(3)))
b.sequenceFilter Left(boom): scala.util.Either[scala.Predef.String, scala.collection.immutable.List[scala.Int]]

val c: List[Either[String, Option[Int]]] = List(Right(Some(1)), Right(None), Right(Some(3)))
c.sequenceFilter Right(List(1, 3)): scala.util.Either[scala.Predef.String, scala.collection.immutable.List[scala.Int]]

val d: List[Either[String, Option[Int]]] = List(Right(Some(1)), Left("boom"), Left("bang"))
d.sequenceFilter Left(boom): scala.util.Either[scala.Predef.String, scala.collection.immutable.List[scala.Int]]
```

In conclusion, the next slide recaps the signatures of all the functions that we have mentioned.

@philip_schwarz

Function	From	Given	To	Type Class	
map	F[A]	A => B	F[B]	Functor[F]	
filter	F[A]	A => Boolean	F[A]	FunctorFilter[F]	Apply a filter to a structure such that the output structure contains all A elements in the input structure that satisfy the predicate f but none that don't.
mapFilter	F[A]	A => Option[B]	F[B]	FunctorFilter[F]	A combined map and filter. Filtering is handled via Option instead of Boolean such that the output type B can be different than the input type A.
traverse	F[A]	A => G[B]	G[F[B]]	Traverse[F]	Given a function which returns a G effect, thread this effect through the running of this function on all the values in F, returning an F[B] in a G context.
filterA	F[A]	A => G[Boolean]	G[F[A]]	TraverseFilter[F]	Filter values inside a G context. This is a generalized version of Haskell's filterM . This StackOverflow question about filterM may be helpful in understanding how it behaves.
traverseFilter	F[A]	A => G[Option[B]]	G[F[B]]	TraverseFilter[F]	A combined traverse and filter. Filtering is handled via Option instead of Boolean such that the output type B can be different than the input type A.
sequence	F[G[A]]		G[F[A]]	Traverse[F]	Thread all the G effects through the F structure to invert the structure from F[G[A]] to G[F[A]].
sequenceFilter	F[G[Option[A]]]		G[F[A]]	TraverseFilter[F]	traverseFilter with identity