10 — Traversable Functors

Einführung in die Funktionale Programmierung

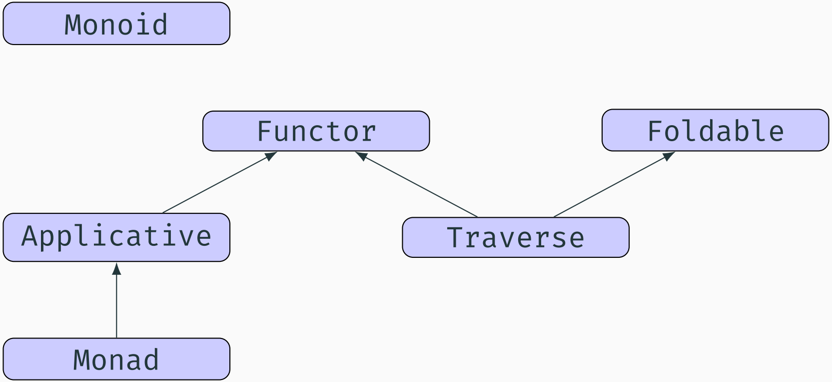
Prof. Frank Puppe, Felix Herrmann, Alexander Gehrke

Sommersemester 2023

Lehrstuhl für Informatik VI, Uni Würzburg

Traversable Functors

Typeclass hierarchy



Recap: Applicatives

We introduced the abstraction of *applicative functors*:

- defined by `pure` and either `ap` or `map2`
- less powerful than Monads: cannot remove a layer of their type constructor
- All monads are applicative functors

Reasons for this abstraction:

- Some applicatives cannot be monads (we saw `Validated`)
- Applicatives can be composed, monads (in general) can't.

Sequence and traverse without List?

We found the *applicative functor* abstraction by noticing that `sequence` (and `traverse`) did not directly depend on `flatMap`.

Let's look at their signatures in `Applicative[F[_]]` again:

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

Can these only work with `List`?

Exercise: Implement sequence for maps

Implement `sequence` for maps inside the `Applicative` trait:

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

Hints:

- The template contains the sequence implementation for lists.
- `foldRight` on Map gives `(K,V)` tuples to its function.
- To add an entry to a map, use `map + ((key, value))`.

Exercise: Implement sequence for maps — Solution

```scala
def sequenceMap[K,V](m: Map[K, F[V]]): F[Map[K,V]] =
  m.foldRight(pure(Map.empty[K,V])){
    //function is given a (K,F[V]) tuple and the F[Map[K,V]] accumulator
    case ((k, fv), acc) => acc.map2(fv)((map, v) => map +((k, v)))
  }
```

Traversable Functors

We call types that can be traversed *traversable functors*. As there are lots of them, we define a new type class:

```
trait Traverse[F[_]]:
  def traverse[G[_]: Applicative, A, B](fa: F[A])(f: A => G[B]): G[F[B]]

  def sequence[G[_]: Applicative, A](fga: F[G[A]]): G[F[A]]
```

Compare to previous signatures:

- `List` replaced by variable `F`.
- Other type `G` must still be applicative (defined before on `Applicative` trait).

Meaning of `sequence`

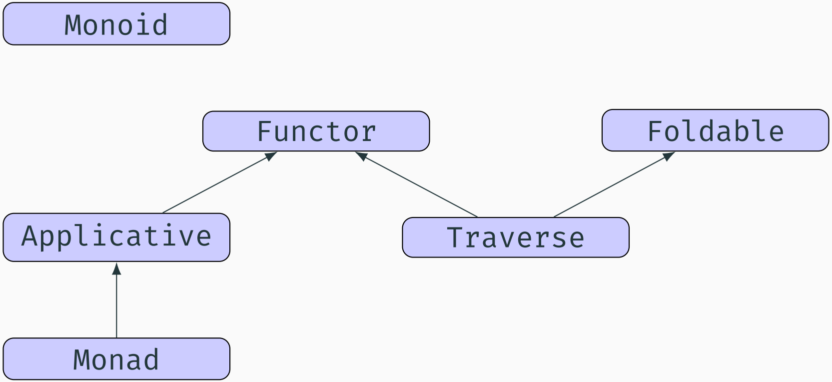
```scala
def sequence[G[_]: Applicative, A](fga: F[G[A]]): G[F[A]]
```

`Sequence` swaps nested `F` and `G`. What does that mean for different types?

- `List[Option[A]] => Option[List[A]]`: returns `None` if any element of list is `None`, else list wrapped in `Some` (see monad lecture)
- `Map[K, Option[V]] => Option[Map[K, V]]`: returns `None` if any value in the map is `None`, else map wrapped in `Some`
- `Option` is traversable too:
  `Option[List[A]] => List[Option[A]]`: returns a list with a single `None` if the original `Option` is `None`, else List with all elements wrapped in `Some`

Traversable Functors in the hierarchy

Typeclass hierarchy



How powerful is `traverse`?

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

`traverse` is equivalent to a `map` operation, followed by `sequence`, and we can provide a default implementation with them.

But it is also possible to implement `map` via `traverse`, which makes `Traverse` a functor!

Traverse as Functor

```scala
trait Traverse[F[_]] extends Functor[F]:
  def traverse[G[_]: Applicative, A, B](fa: F[A])(f: A => G[B]): G[F[B]] =
    sequence(map(fa)(f))

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

  /** can be implemented using traverse */
  extension [A](fa: F[A]) def map[B](f: A => B): F[B] = ??? // => Exercise sheet
```

This means, an instance of `Traverse` only needs to implement either `traverse` or `sequence` and `map`

Traverse can provide more

Let's define a type, that throws away its parameter:

```
type ConstInt[A] = Int
```

We can use this, when a type constructor is expected, e.g. for `G` in `traverse`

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

this results in

```
def traverse[A, B](fa: F[A])(f: A => Int): Int
```

Does this signature remind you of any function or structure we have seen before?

`Traverse` can provide folds

This is similar to `foldMap`:

```
def traverse[A, B](fa: F[A])(f: A => Int): Int
```

```
def  foldMap[A, M](as: F[A])(f: A => M)(M: Monoid[M]): M
```

And we can implement `foldMap` via `traverse`! But we need an applicative for `Const` first.

An applicative for monoids

```
type Const[M, A] = M // like ConstInt, but with parameter instead of fixed int
```

```
def monoidApplicative[M](M: Monoid[M]) =
  new Applicative[Const[M, _]]:
    def pure[A](a: A): M = M.zero
    extension [A](m1: M)
      override def map2[B,C](m2: M)(f: (A, B) =>  C): M = M.combine(m1,m2)
```

As our `Const` throws away it's second parameter, our applicative also never uses the type parameters of its functions: the value passed to `pure` is ignored, so is the function passed to `map2`.

Traverse with Foldable

This lets us implement `foldMap` via `traverse`, and extend `Foldable`:

```
trait Traverse[F[_]] extends Functor[F] with Foldable[F]:
  def traverse[G[_]: Applicative, A, B](fa: F[A])(f: A => G[B]): G[F[B]] = ???
  // ...
  extension [A](as: F[A])
    override def foldMap[B](f: A => B)(using mb: Monoid[B]): B =
      traverse[Const[B,_], A, Nothing](as)(f)(monoidApplicative(mb))
```

Type parameters for `traverse`:

- `G[_] = Const[M,_]`: Evaluates to `M` for any passed parameter
- `A`: same meaning for both, type of elements in our traversed object.
- `B = Nothing`: `Const` throws away its parameter. Therefore all uses of `B` are meaningless.

`Traverse` is functor and foldable

We've seen that the `traverse` function is powerful enough to implement `Functor` and `Foldable` with it.

Notably, `Foldable` cannot extend `Functor`, as you can not write a map in terms of a fold *in general*, although it is possible for specific foldable data structures.

So what kind of *generalized* functions does `Traverse` allow us to write?

Traversals with State

A useful method also found in the standard library for several types is `zipWithIndex`. This takes a traversable data structure and adds an index to each element by tupling. Example:

```
List(
  "Here",
  "are",
  "some",
  "elements"
): List[String]
```

→ `zipWithIndex` →

```
List(
  ("Here", 0),
  ("are", 1),
  ("some", 2),
  ("elements", 3)
): List[(String, Int)]
```

By using the `State` applicative functor, we can keep some iteration state, like the current index, during a traversal.

Implementing `zipWithIndex`

Idea:

1. Define a function: For an element of type `A`, we create a State monad, that
   - takes an Int as state
   - returns a tuple of the element and that int as its result (type `(A, Int)`)
   - returns the int + 1 as the new state

   This has type `State[Int, (A, Int)]`
2. we call `traverse`, passing an `F[A]` and our function from step 1.
   This results in a `State[Int, F[(A, Int)]]`
3. we run this state with 0 as start value (first index)
4. The result is a tuple `(lastIndex + 1 ,fWithIndices)`, because `State.run` also always returns the next state.

Implementing `zipWithIndex` — Step 1

1. Define a function: For an element of type `A`, we create a State monad, that
   - takes an Int as state
   - returns a tuple of the element and that int as its result (type `(A, Int)`)
   - returns the int + 1 as the new state

```
(a: A) => for
  i <- State.get[Int]   // get the passed in state, save it in i
  _ <- State.set(i + 1) // set the new state to i + 1
yield (a, i)            // return the A together with the passed in index
```

Another way of writing the same function:

```
(a:A) => State((i: Int) => (i+1, (a,i)))
```

Implementing `zipWithIndex` — Step 2

2. we call **`traverse`**, passing an `F[A]` and our function from step 1.
   This results in a `State[Int, F[(A, Int)]]`

```
traverse(fa)(          // fa: F[A]
  (a: A) => for        // A => State[Int, (A, Int)]
    i <- State.get[Int]
    _ <- State.set(i + 1)
  yield (a, i)
)
```

Implementing `zipWithIndex` — Step 3

3. we run this state with 0 as start value (first index)

```
traverse(fa)(
  (a: A) => for
    i <- State.get[Int]
    _ <- State.set(i + 1)
  yield (a, i)
).run(0) // start with index 0
```

Implementing `zipWithIndex` — Step 4

4. The result is a tuple `(lastIndex + 1 ,fWithIndices)`, because `State.run` also always returns the next state.
   We only care about our indexed `F`.

```
def zipWithIndex[A](fa:F[A]): F[(A,Int)] =
  traverse(fa)(
    (a: A) => for
      i <- State.get[Int]
      _ <- State.set(i + 1)
    yield (a, i)
  ).run(0)._2
```

If we're using the `cats` library, we could also use `runA` instead of `run` for the same result.

Implementing `toList`

With a similar approach, we can write a generic `toList`: We keep a list as state (starting with an empty list) and append each element as a state change. Our State monads can be of type `State[List[A],Unit]`, because we don't need intermediate results.

```
def toList[A](fa:F[A]): List[A] =
  traverse(fa)(
    (a: A) => for
      as <- State.get[List[A]] // keep list as state
      _  <- State.set(a :: as) // prepend each element
    yield ()
  ).run(Nil) // start with empty list
    ._1      // state at the end is list of elements
    .reverse // but in reverse order
```

Factoring out

Compare our two methods:

```
def zipWithIndex[A](fa:F[A]): F[(A,Int)] =
  traverse(fa)(
    (a: A) => for
      i <- State.get[Int]
      _ <- State.set(i + 1)
    yield (a, i)
  ).run(0)._2
```

```
def toList[A](fa:F[A]): List[A] =
  traverse(fa)(
    (a: A) => for
      as <- State.get[List[A]]
      _  <- State.set(a :: as)
    yield ()
  ).run(Nil)._1.reverse
```

These look pretty similar. And so do lots of traversals with state. Let's factor out the common part.

Mapping with Accumulator

What we are doing is like mapping, while passing an accumulator to our function. Therefore we define a function `mapAccum`, which looks similar to `map`, but with an accumulator of type `S` added to any other value:

```
//  map       [A,B](fa: F[A])      (f:   (A) =>   (B)):     F[B]
def mapAccum[S,A,B](fa: F[A], s: S)(f: (S,A) => (S,B)): (S, F[B]) =
  traverse(fa)((a: A) => for
      s1 <- State.get[S]  // get the state
      (s2, b) = f(s1, a)  // get map result and new state
      _  <- State.set(s2) // store new state
    yield b               // yield map result
  ).run(s)                // run with start state
```

Using mapAccum

```scala
def zipWithIndex[A](fa: F[A]): F[(A,Int)] =
  mapAccum(fa, 0)((s, a) => (s + 1, (a, s)))._2

def toList[A](fa :F[A]): List[A] =
  mapAccum(fa, List.empty[A])((s, a) => (a::s, ()))._1.reverse
```

This has also similarities with folding, additionaly saving a value for every recursion step. In fact, `toList` could also be written with a fold. On the exercise sheet, you'll implement `foldLeft` with `mapAccum`.

Fusing traversals

We've already seen ways to combine several traversals of a data structure into one, e.g. LazyList combines several operations in one pass by using lazy evaluation

On a former task sheet we saw how we can combine `Applicative`s with `product`. We can use this to fuse two traversals into one pass.

Exercise: Implement `fuse`

Implement `fuse` using applicative functor products. It should traverse `fa` a single time and collect the results of both given functions at once.

```
def fuse[G[_],H[_],A,B](fa: F[A])(f: A => G[B], g: A => H[B])
  (using G: Applicative[G], H: Applicative[H])
  : (G[F[B]], H[F[B]]) =
```

Hint: you may have to specify the type parameters in your call: `traverse[[b] =>> (G[b],H[b]), A, B](...)`
Also, you'll have to specify the applicative explicitly.

Exercise: Implement **fuse** — Solution

```scala
def fuse[G[_],H[_],A,B](fa: F[A])(f: A => G[B], g: A => H[B])
  (using G: Applicative[G], H: Applicative[H])
  : (G[F[B]], H[F[B]]) =
    traverse[[b] =>> (G[b],H[b]), A, B](fa)(
      a => (f(a), g(a)))(Applicative.product(using G, H)
    )
```

- traversal function: apply both given functions, pack in tuple
- Applicative: product of `G` and `H` (tupled combination)