Chapter 12: Traversing the Stone

So far, in our cirque du conteneur, you've seen us tame the ferocious functor, bending it to our will to perform any operation that strikes our fancy. You've been dazzled by the juggling of many dangerous effects at once using function application to collect the results. Sat there in amazement as containers vanished in thin air by joining them together. At the side effect sideshow, we've seen them composed into one. And most recently, we've ventured beyond what's natural and transformed one type into another before your very eyes.

And now for our next trick, we'll look at traversals. We'll watch types soar over one another as if they were trapeze artists holding our value intact. We'll reorder effects like the trolleys in a tilt-a-whirl. When our containers get intertwined like the limbs of a contortionist, we can use this interface to straighten things out. We'll witness different effects with different orderings. Fetch me my pantaloons and slide whistle, let's get started.

Types n' Types

Let's get weird:

```
// readFile :: FileName -> Task Error String

// firstWords :: String -> String
const firstWords = compose(join(' '), take(3), split(' '));

// tldr :: FileName -> Task Error String
const tldr = compose(map(firstWords), readFile);

map(tldr, ['file1', 'file2']);
// [Task('hail the monarchy'), Task('smash the patriarchy')]
```

Here we read a bunch of files and end up with a useless array of tasks. How might we fork each one of these? It would be most agreeable if we could switch the types around to have `Task Error [String]` instead of `[Task Error String]`. That way, we'd have one future value holding all the results, which is much more amenable to our async needs than several future values arriving at their leisure.

Here's one last example of a sticky situation:

```
// getAttribute :: String -> Node -> Maybe String
// $ :: Selector -> IO Node
```

```
// getControlNode :: IO (Maybe (IO Node))
const getControlNode = compose(map(map($)), map(getAttribute('aria-controls')), $);
```

Look at those `IO` s longing to be together. It'd be just lovely to `join` them, let them dance cheek to cheek, but alas a `Maybe` stands between them like a chaperone at prom. Our best move here would be to shift their positions next to one another, that way each type can be together at last and our signature can be simplified to `IO (Maybe Node)` .

Type Feng Shui

The *Traversable* interface consists of two glorious functions: `sequence` and `traverse` .

Let's rearrange our types using `sequence` :

```
sequence(List.of, Maybe.of(['the facts'])); // [Just('the facts')]
sequence(Task.of, new Map({ a: Task.of(1), b: Task.of(2) })); // Task(Map({ a: 1, b
sequence(IO.of, Either.of(IO.of('buckle my shoe'))); // IO(Right('buckle my shoe'))
sequence(Either.of, [Either.of('wing')]); // Right(['wing'])
sequence(Task.of, left('wing')); // Task(Left('wing'))
```

See what has happened here? Our nested type gets turned inside out like a pair of leather trousers on a humid summer night. The inner functor is shifted to the outside and vice versa. It should be known that `sequence` is bit particular about its arguments. It looks like this:

```
// sequence :: (Traversable t, Applicative f) => (a -> f a) -> t (f a) -> f (t a)
const sequence = curry((of, x) => x.sequence(of));
```

Let's start with the second argument. It must be a *Traversable* holding an *Applicative*, which sounds quite restrictive, but just so happens to be the case more often than not. It is the `t (f a)` which gets turned into a `f (t a)` . Isn't that expressive? It's clear as day the two types do-si-do around each other. That first argument there is merely a crutch and only necessary in an untyped language. It is a type constructor (our *of*) provided so that we can invert map-reluctant types like `Left` - more on that in a minute.

Using `sequence` , we can shift types around with the precision of a sidewalk thimblerigger. But how does it work? Let's look at how a type, say `Either` , would implement it:

```js
class Right extends Either {
  // ...
  sequence(of) {
    return this.$value.map(Either.of);
  }
}
```

Ah yes, if our `$value` is a functor (it must be an applicative, in fact), we can simply `map` our constructor to leap frog the type.

You may have noticed that we've ignored the `of` entirely. It is passed in for the occasion where mapping is futile, as is the case with `Left`:

```js
class Left extends Either {
  // ...
  sequence(of) {
    return of(this);
  }
}
```

We'd like the types to always end up in the same arrangement, therefore it is necessary for types like `Left` who don't actually hold our inner applicative to get a little help in doing so. The *Applicative* interface requires that we first have a *Pointed Functor* so we'll always have a `of` to pass in. In a language with a type system, the outer type can be inferred from the signature and does not need to be explicitly given.

Effect Assortment

Different orders have different outcomes where our containers are concerned. If I have `[Maybe a]`, that's a collection of possible values whereas if I have a `Maybe [a]`, that's a possible collection of values. The former indicates we'll be forgiving and keep "the good ones", while the latter means it's an "all or nothing" type of situation. Likewise, `Either Error (Task Error a)` could represent a client side validation and `Task Error (Either Error a)` could be a server side one. Types can be swapped to give us different effects.

```js
// fromPredicate :: (a -> Bool) -> a -> Either e a

// partition :: (a -> Bool) -> [a] -> [Either e a]
const partition = f => map(fromPredicate(f));

// validate :: (a -> Bool) -> [a] -> Either e [a]
```

```
const validate = f => traverse(Either.of, fromPredicate(f));
```

Here we have two different functions based on if we `map` or `traverse`. The first, `partition` will give us an array of `Left`s and `Right`s according to the predicate function. This is useful to keep precious data around for future use rather than filtering it out with the bathwater. `validate` instead will give us the first item that fails the predicate in `Left`, or all the items in `Right` if everything is hunky dory. By choosing a different type order, we get different behavior.

Let's look at the `traverse` function of `List`, to see how the `validate` method is made.

```
traverse(of, fn) {
    return this.$value.reduce(
      (f, a) => fn(a).map(b => bs => bs.concat(b)).ap(f),
      of(new List([])),
    );
  }
```

This just runs a `reduce` on the list. The reduce function is `(f, a) => fn(a).map(b => bs => bs.concat(b)).ap(f)`, which looks a bit scary, so let's step through it.

1. `reduce(..., ...)`

   Remember the signature of `reduce :: [a] -> (f -> a -> f) -> f -> f`. The first argument is actually provided by the dot-notation on `$value`, so it's a list of things. Then we need a function from a `f` (the accumulator) and a `a` (the iteree) to return us a new accumulator.

2. `of(new List([]))`

   The seed value is `of(new List([]))`, which in our case is `Right([]) :: Either e [a]`. Notice that `Either e [a]` will also be our final resulting type!

3. `fn :: Applicative f => a -> f a`

   If we apply it to our example above, `fn` is actually `fromPredicate(f) :: a -> Either e a`.

> fn(a) :: Either e a

4. `.map(b => bs => bs.concat(b))`

When `Right`, `Either.map` passes the right value to the function and returns a new `Right` with the result. In this case the function has one parameter (`b`), and returns another function (`bs => bs.concat(b)`, where `b` is in scope due to the closure). When `Left`, the left value is returned.

> fn(a).map(b => bs => bs.concat(b)) :: Either e ([a] -> [a])

5. `.ap(f)`

   Remember that `f` is an Applicative here, so we can apply the function `bs => bs.concat(b)` to whatever value `bs :: [a]` is in `f`. Fortunately for us, `f` comes from our initial seed and has the following type: `f :: Either e [a]` which is by the way, preserved when we apply `bs => bs.concat(b)`.
   When `f` is `Right`, this calls `bs => bs.concat(b)`, which returns a `Right` with the item added to the list. When `Left`, the left value (from the previous step or previous iteration respectively) is returned.

> fn(a).map(b => bs => bs.concat(b)).ap(f) :: Either e [a]

This apparently miraculous transformation is achieved with just 6 measly lines of code in `List.travserse`, and is accomplished with `of`, `map` and `ap`, so will work for any Applicative Functor. This is a great example
of how those abstraction can help to write highly generic code with only a few assumptions (that can, incidentally, be declared and checked at the type level!).

Waltz of the Types

Time to revisit and clean our initial examples.

```
// readFile :: FileName -> Task Error String

// firstWords :: String -> String
const firstWords = compose(join(' '), take(3), split(' '));

// tldr :: FileName -> Task Error String
const tldr = compose(map(firstWords), readFile);

traverse(Task.of, tldr, ['file1', 'file2']);
// Task(['hail the monarchy', 'smash the patriarchy']);
```

Using `traverse` instead of `map`, we've successfully herded those unruly `Task`s into a nice coordinated array of results. This is like `Promise.all()`, if you're familiar, except it isn't just a one-off, custom function, no, this works for any *traversable* type. These mathematical apis tend to capture most things we'd like to do in an interoperable, reusable way, rather than each library reinventing these functions for a single type.

Let's clean up the last example for closure (no, not that kind):

```
// getAttribute :: String -> Node -> Maybe String
// $ :: Selector -> IO Node

// getControlNode :: IO (Maybe Node)
const getControlNode = compose(chain(traverse(IO.of, $)), map(getAttribute('aria-co
```

Instead of `map(map($))` we have `chain(traverse(IO.of, $))` which inverts our types as it maps then flattens the two `IO` s via `chain`.

No Law and Order

Well now, before you get all judgemental and bang the backspace button like a gavel to retreat from the chapter, take a moment to recognize that these laws are useful code guarantees. 'Tis my conjecture that the goal of most program architecture is an attempt to place useful restrictions on our code to narrow the possibilities, to guide us into the answers as designers and readers.

An interface without laws is merely indirection. Like any other mathematical structure, we must expose properties for our own sanity. This has a similar effect as encapsulation since it protects the data, enabling us to swap out the interface with another law abiding citizen.

Come along now, we've got some laws to suss out.

Identity

```
const identity1 = compose(sequence(Identity.of), map(Identity.of));
const identity2 = Identity.of;

// test it out with Right
identity1(Either.of('stuff'));
// Identity(Right('stuff'))

identity2(Either.of('stuff'));
// Identity(Right('stuff'))
```

This should be straightforward. If we place an `Identity` in our functor, then turn it inside out with `sequence` that's the same as just placing it on the outside to begin with. We chose `Right` as our guinea pig as it is easy to try the law and inspect. An arbitrary functor there is normal, however, the use of a concrete functor here, namely `Identity` in the law itself might raise some eyebrows. Remember a category is defined by morphisms between its objects that have associative composition and identity. When dealing with the category of functors, natural transformations are the morphisms and `Identity` is, well identity. The `Identity` functor is as fundamental in demonstrating laws as our `compose` function. In fact, we should give up the ghost and follow suit with our Compose type:

Composition

```
const comp1 = compose(sequence(Compose.of), map(Compose.of));
const comp2 = (Fof, Gof) => compose(Compose.of, map(sequence(Gof)), sequence(Fof));


// Test it out with some types we have lying around
comp1(Identity(Right([true])));
// Compose(Right([Identity(true)]))

comp2(Either.of, Array)(Identity(Right([true])));
// Compose(Right([Identity(true)]))
```

This law preserves composition as one would expect: if we swap compositions of functors, we shouldn't see any surprises since the composition is a functor itself. We arbitrarily chose `true`, `Right`, `Identity`, and `Array` to test it out. Libraries like quickcheck or jsverify can help us test the law by fuzz testing the inputs.

As a natural consequence of the above law, we get the ability to fuse traversals, which is nice from a performance standpoint.

Naturality

```
const natLaw1 = (of, nt) => compose(nt, sequence(of));
const natLaw2 = (of, nt) => compose(sequence(of), map(nt));

// test with a random natural transformation and our friendly Identity/Right functo

// maybeToEither :: Maybe a -> Either () a
const maybeToEither = x => (x.$value ? new Right(x.$value) : new Left());

natLaw1(Maybe.of, maybeToEither)(Identity.of(Maybe.of('barlow one')));
// Right(Identity('barlow one'))
```

```js
natLaw2(Either.of, maybeToEither)(Identity.of(Maybe.of('barlow one')));
// Right(Identity('barlow one'))
```

This is similar to our identity law. If we first swing the types around then run a natural transformation on the outside, that should equal mapping a natural transformation, then flipping the types.

A natural consequence of this law is:

```js
traverse(A.of, A.of) === A.of;
```

Which, again, is nice from a performance standpoint.

In Summary

Traversable is a powerful interface that gives us the ability to rearrange our types with the ease of a telekinetic interior decorator. We can achieve different effects with different orders as well as iron out those nasty type wrinkles that keep us from `join`ing them down. Next, we'll take a bit of a detour to see one of the most powerful interfaces of functional programming and perhaps even algebra itself: Monoids bring it all together

Exercises

Considering the following elements:

```js
// httpGet :: Route -> Task Error JSON

// routes :: Map Route Route
const routes = new Map({ '/': '/', '/about': '/about' });
```

{% exercise %}
Use the traversable interface to change the type signature of `getJsons` to
Map Route Route → Task Error (Map Route JSON)

{% initial src="./exercises/ch12/exercise_a.js#L11;" %}
`js // getRoutes :: Map Route Route -> Map Route (Task Error JSON) const getJsons =`

`map(httpGet);`

{% solution src="./exercises/ch12/solution_a.js" %}
{% validation src="./exercises/ch12/validation_a.js" %}
{% context src="./exercises/support.js" %}
{% endexercise %}

We now define the following validation function:

```
// validate :: Player -> Either String Player
const validate = player => (player.name ? Either.of(player) : left('must have name'
```

{% exercise %}
Using traversable, and the `validate` function, update `startGame` (and its signature) to only start the game if all players are valid

{% initial src="./exercises/ch12/exercise_b.js#L7;" %}
`js // startGame :: [Player] -> [Either Error String] const startGame = compose(map(always('game started!')), map(validate));`

{% solution src="./exercises/ch12/solution_b.js" %}
{% validation src="./exercises/ch12/validation_b.js" %}
{% context src="./exercises/support.js" %}
{% endexercise %}

Finally, we consider some file-system helpers:

```
// readfile :: String -> Task Error String
// readdir :: String -> Task Error [String]
```

{% exercise %}
Use traversable to rearrange and flatten the nested Tasks & Maybe

{% initial src="./exercises/ch12/exercise_c.js#L8;" %}
`js // readFirst :: String -> Task Error (Task Error (Maybe String)) const readFirst = compose(map(map(readfile('utf-8'))), map(safeHead), readdir);`

{% solution src="./exercises/ch12/solution_c.js" %}
{% validation src="./exercises/ch12/validation_c.js" %}
{% context src="./exercises/support.js" %}
{% endexercise %}