Chapter 10: Applicative Functors

Applying Applicatives

The name **applicative functor** is pleasantly descriptive given its functional origins. Functional programmers are notorious for coming up with names like `mappend` or `liftA4`, which seem perfectly natural when viewed in the math lab, but hold the clarity of an indecisive Darth Vader at the drive thru in any other context.

Anyhow, the name should spill the beans on what this interface gives us: *the ability to apply functors to each other*.

Now, why would a normal, rational person such as yourself want such a thing? What does it even *mean* to apply one functor to another?

To answer these questions, we'll start with a situation you may have already encountered in your functional travels. Let's say, hypothetically, that we have two functors (of the same type) and we'd like to call a function with both of their values as arguments. Something simple like adding the values of two `Container`s.

```
// We can't do this because the numbers are bottled up.
add(Container.of(2), Container.of(3));
// NaN

// Let's use our trusty map
const containerOfAdd2 = map(add, Container.of(2));
// Container(add(2))
```

We have ourselves a `Container` with a partially applied function inside. More specifically, we have a `Container(add(2))` and we'd like to apply its `add(2)` to the `3` in `Container(3)` to complete the call. In other words, we'd like to apply one functor to another.

Now, it just so happens that we already have the tools to accomplish this task. We can `chain` and then `map` the partially applied `add(2)` like so:

```
Container.of(2).chain(two => Container.of(3).map(add(two)));
```

The issue here is that we are stuck in the sequential world of monads wherein nothing may be

evaluated until the previous monad has finished its business. We have ourselves two strong, independent values and I should think it unnecessary to delay the creation of `Container(3)` merely to satisfy the monad's sequential demands.

In fact, it would be lovely if we could succinctly apply one functor's contents to another's value without these needless functions and variables should we find ourselves in this pickle jar.

Ships in Bottles

`ap` is a function that can apply the function contents of one functor to the value contents of another. Say that five times fast.

```
Container.of(add(2)).ap(Container.of(3));
// Container(5)

// all together now

Container.of(2).map(add).ap(Container.of(3));
// Container(5)
```

There we are, nice and neat. Good news for `Container(3)` as it's been set free from the jail of

the nested monadic function. It's worth mentioning again that `add`, in this case, gets partially applied during the first `map` so this only works when `add` is curried.

We can define `ap` like so:

```
Container.prototype.ap = function (otherContainer) {
  return otherContainer.map(this.$value);
};
```

Remember, `this.$value` will be a function and we'll be accepting another functor so we need only `map` it. And with that we have our interface definition:

> An *applicative functor* is a pointed functor with an `ap` method

Note the dependence on **pointed**. The pointed interface is crucial here as we'll see throughout the following examples.

Now, I sense your skepticism (or perhaps confusion and horror), but keep an open mind; this `ap` character will prove useful. Before we get into it, let's explore a nice property.

```
F.of(x).map(f) === F.of(f).ap(F.of(x));
```

In proper English, mapping `f` is equivalent to `ap`ing a functor of `f`. Or in properer English, we can place `x` into our container and `map(f)` OR we can lift both `f` and `x` into our container and `ap` them. This allows us to write in a left-to-right fashion:

```
Maybe.of(add).ap(Maybe.of(2)).ap(Maybe.of(3));
// Maybe(5)

Task.of(add).ap(Task.of(2)).ap(Task.of(3));
// Task(5)
```

One might even recognise the vague shape of a normal function call if viewed mid squint. We'll look at the pointfree version later in the chapter, but for now, this is the preferred way to write such code. Using `of`, each value gets transported to the magical land of containers, this parallel universe where each application can be async or null or what have you and `ap` will apply functions within this fantastical place. It's like building a ship in a bottle.

Did you see there? We used `Task` in our example. This is a prime situation where applicative

functors pull their weight. Let's look at a more in-depth example.

Coordination Motivation

Say we're building a travel site and we'd like to retrieve both a list of tourist destinations and local events. Each of these are separate, stand-alone api calls.

```
// Http.get :: String -> Task Error HTML

const renderPage = curry((destinations, events) => { /* render page */ });

Task.of(renderPage).ap(Http.get('/destinations')).ap(Http.get('/events'));
// Task("<div>some page with dest and events</div>")
```

Both `Http` calls will happen instantly and `renderPage` will be called when both are resolved. Contrast this with the monadic version where one `Task` must finish before the next fires off. Since we don't need the destinations to retrieve events, we are free from sequential evaluation.

Again, because we're using partial application to achieve this result, we must ensure `renderPage` is curried or it will not wait for both `Tasks` to finish. Incidentally, if you've ever had to do such a thing manually, you'll appreciate the astonishing simplicity of this interface. This is the kind of beautiful code that takes us one step closer to the singularity.

Let's look at another example.

```
// $ :: String -> IO DOM
const $ = selector => new IO(() => document.querySelector(selector));

// getVal :: String -> IO String
const getVal = compose(map(prop('value')), $);

// signIn :: String -> String -> Bool -> User
const signIn = curry((username, password, rememberMe) => { /* signing in */ });

IO.of(signIn).ap(getVal('#email')).ap(getVal('#password')).ap(IO.of(false));
// IO({ id: 3, email: 'gg@allin.com' })
```

`signIn` is a curried function of 3 arguments so we have to `ap` accordingly. With each `ap`, `signIn` receives one more argument until it is complete and runs. We can continue this pattern with as many arguments as necessary. Another thing to note is that two arguments end up naturally in `IO` whereas the last one needs a little help from `of` to lift it into `IO` since

`ap` expects the function and all its arguments to be in the same type.

Bro, Do You Even Lift?

Let's examine a pointfree way to write these applicative calls. Since we know `map` is equal to `of/ap`, we can write generic functions that will `ap` as many times as we specify:

```
const liftA2 = curry((g, f1, f2) => f1.map(g).ap(f2));

const liftA3 = curry((g, f1, f2, f3) => f1.map(g).ap(f2).ap(f3));

// liftA4, etc
```

`liftA2` is a strange name. It sounds like one of the finicky freight elevators in a rundown factory or a vanity plate for a cheap limo company. Once enlightened, however, it's self explanatory: lift these pieces into the applicative functor world.

When I first saw this 2-3-4 nonsense it struck me as ugly and unnecessary. After all, we can check the arity of functions in JavaScript and build this up dynamically. However, it is often useful to partially apply `liftA(N)` itself, so it cannot vary in argument length.

Let's see this in use:

```
// checkEmail :: User -> Either String Email
// checkName :: User -> Either String String

const user = {
  name: 'John Doe',
  email: 'blurp_blurp',
};

//  createUser :: Email -> String -> IO User
const createUser = curry((email, name) => { /* creating... */ });

Either.of(createUser).ap(checkEmail(user)).ap(checkName(user));
// Left('invalid email')

liftA2(createUser, checkEmail(user), checkName(user));
// Left('invalid email')
```

Since `createUser` takes two arguments, we use the corresponding `liftA2`. The two statements are equivalent, but the `liftA2` version has no mention of `Either`. This makes it more generic and flexible since we are no longer married to a specific type.

Let's see the previous examples written this way:

```
liftA2(add, Maybe.of(2), Maybe.of(3));
// Maybe(5)

liftA2(renderPage, Http.get('/destinations'), Http.get('/events'));
// Task('<div>some page with dest and events</div>')

liftA3(signIn, getVal('#email'), getVal('#password'), IO.of(false));
// IO({ id: 3, email: 'gg@allin.com' })
```

Operators

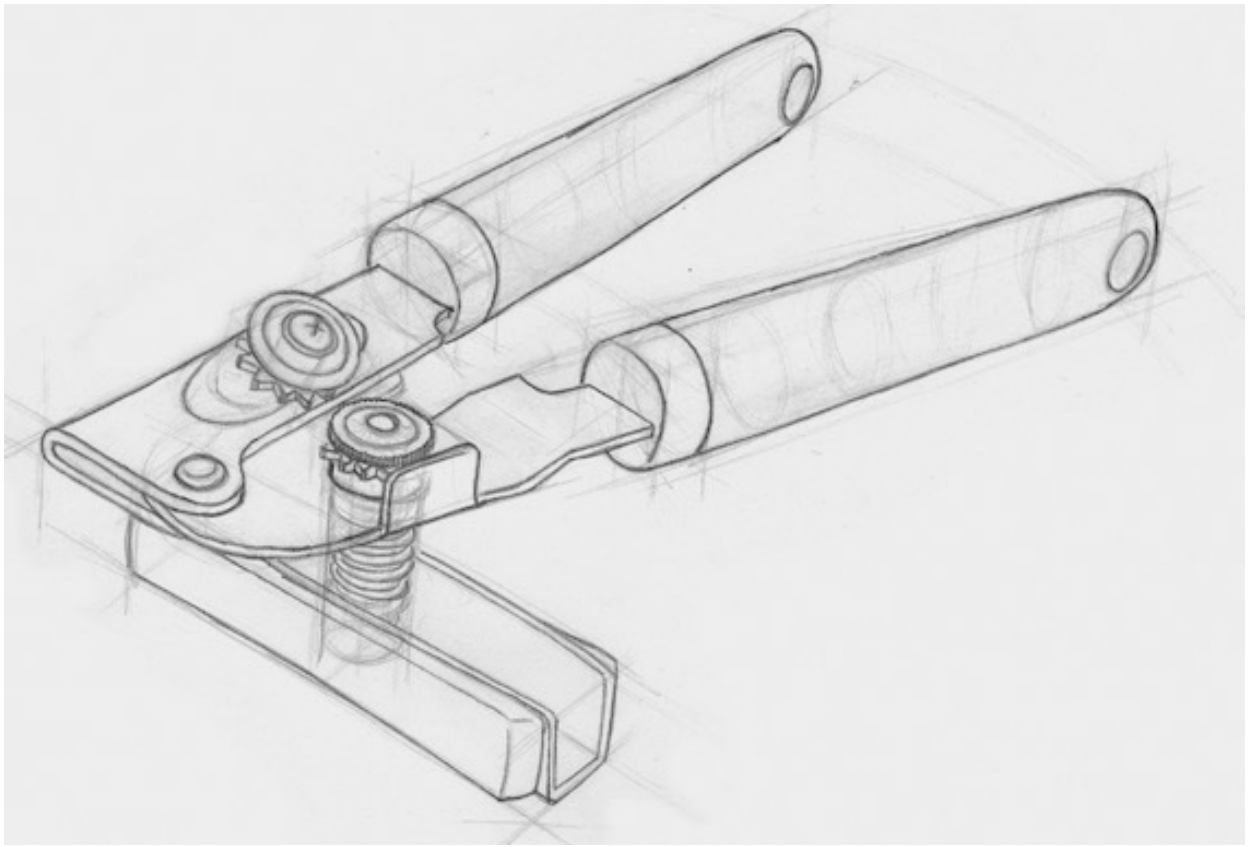
In languages like Haskell, Scala, PureScript, and Swift, where it is possible to create your own infix operators you may see syntax like this:

```
-- Haskell / PureScript
add <$> Right 2 <*> Right 3
```

```
// JavaScript
map(add, Right(2)).ap(Right(3));
```

It's helpful to know that `<$>` is `map` (aka `fmap`) and `<*>` is just `ap`. This allows for a more natural function application style and can help remove some parenthesis.

Free Can Openers



We haven't spoken much about derived functions. Seeing as all of these interfaces are built off of each other and obey a set of laws, we can define some weaker interfaces in terms of the stronger ones.

For instance, we know that an applicative is first a functor, so if we have an applicative instance, surely we can define a functor for our type.

This kind of perfect computational harmony is possible because we're working within a mathematical framework. Mozart couldn't have done better even if he had torrented Ableton as a child.

I mentioned earlier that `of/ap` is equivalent to `map`. We can use this knowledge to define `map` for free:

```
// map derived from of/ap
X.prototype.map = function map(f) {
  return this.constructor.of(f).ap(this);
};
```

Monads are at the top of the food chain, so to speak, so if we have `chain`, we get functor and applicative for free:

```
// map derived from chain
X.prototype.map = function map(f) {
  return this.chain(a => this.constructor.of(f(a)));
};

// ap derived from chain/map
X.prototype.ap = function ap(other) {
  return this.chain(f => other.map(f));
};
```

If we can define a monad, we can define both the applicative and functor interfaces. This is quite remarkable as we get all of these can openers for free. We can even examine a type and automate this process.

It should be pointed out that part of `ap`'s appeal is the ability to run things concurrently so defining it via `chain` is missing out on that optimization. Despite that, it's good to have an immediate working interface while one works out the best possible implementation.

Why not just use monads and be done with it, you ask? It's good practice to work with the level of power you need, no more, no less. This keeps cognitive load to a minimum by ruling out possible functionality. For this reason, it's good to favor applicatives over monads.

Monads have the unique ability to sequence computation, assign variables, and halt further execution all thanks to the downward nesting structure. When one sees applicatives in use, they needn't concern themselves with any of that business.

Now, on to the legalities …

Laws

Like the other mathematical constructs we've explored, applicative functors hold some useful properties for us to rely on in our daily code. First off, you should know that applicatives are "closed under composition", meaning `ap` will never change container types on us (yet another reason to favor over monads). That's not to say we cannot have multiple different effects - we can stack our types knowing that they will remain the same during the entirety of our application.

To demonstrate:

```
const tOfM = compose(Task.of, Maybe.of);

liftA2(liftA2(concat), tOfM('Rainy Days and Mondays'), tOfM(' always get me down'))
// Task(Maybe(Rainy Days and Mondays always get me down))
```

See, no need to worry about different types getting in the mix.

Time to look at our favorite categorical law: *identity*:

Identity

```
// identity
A.of(id).ap(v) === v;
```

Right, so applying `id` all from within a functor shouldn't alter the value in `v`. For example:

```
const v = Identity.of('Pillow Pets');
Identity.of(id).ap(v) === v;
```

`Identity.of(id)` makes me chuckle at its futility. Anyway, what's interesting is that, as we've already established, `of/ap` is the same as `map` so this law follows directly from functor identity: `map(id) == id`.

The beauty in using these laws is that, like a militant kindergarten gym coach, they force all of our interfaces to play well together.

Homomorphism

```
// homomorphism
A.of(f).ap(A.of(x)) === A.of(f(x));
```

A *homomorphism* is just a structure preserving map. In fact, a functor is just a *homomorphism* between categories as it preserves the original category's structure under the mapping.

We're really just stuffing our normal functions and values into a container and running the computation in there so it should come as no surprise that we will end up with the same result if we apply the whole thing inside the container (left side of the equation) or apply it outside, then place it in there (right side).

A quick example:

```
Either.of(toUpperCase).ap(Either.of('oreos')) === Either.of(toUpperCase('oreos'));
```

Interchange

The *interchange* law states that it doesn't matter if we choose to lift our function into the left or right side of `ap`.

```
// interchange
v.ap(A.of(x)) === A.of(f => f(x)).ap(v);
```

Here is an example:

```
const v = Task.of(reverse);
const x = 'Sparklehorse';

v.ap(Task.of(x)) === Task.of(f => f(x)).ap(v);
```

Composition

And finally composition which is just a way to check that our standard function composition holds when applying inside of containers.

```
// composition
A.of(compose).ap(u).ap(v).ap(w) === u.ap(v.ap(w));
```

```
const u = IO.of(toUpperCase);
const v = IO.of(concat('& beyond'));
const w = IO.of('blood bath ');

IO.of(compose).ap(u).ap(v).ap(w) === u.ap(v.ap(w));
```

In Summary

A good use case for applicatives is when one has multiple functor arguments. They give us the

ability to apply functions to arguments all within the functor world. Though we could already do so with monads, we should prefer applicative functors when we aren't in need of monadic specific functionality.

We're almost finished with container apis. We've learned how to `map`, `chain`, and now `ap` functions. In the next chapter, we'll learn how to work better with multiple functors and disassemble them in a principled way.

Chapter 11: Transformation Again, Naturally

Exercises

{% exercise %}
Write a function that adds two possibly null numbers together using `Maybe` and `ap`.

{% initial src="./exercises/ch10/exercise_a.js#L3;" %}
`js // safeAdd :: Maybe Number -> Maybe Number -> Maybe Number const safeAdd = undefined;`

{% solution src="./exercises/ch10/solution_a.js" %}
{% validation src="./exercises/ch10/validation_a.js" %}
{% context src="./exercises/support.js" %}
{% endexercise %}

{% exercise %}
Rewrite `safeAdd` from exercise_b to use `liftA2` instead of `ap`.

{% initial src="./exercises/ch10/exercise_b.js#L3;" %}
`js // safeAdd :: Maybe Number -> Maybe Number -> Maybe Number const safeAdd = undefined;`

{% solution src="./exercises/ch10/solution_b.js" %}
{% validation src="./exercises/ch10/validation_b.js" %}
{% context src="./exercises/support.js" %}
{% endexercise %}

For the next exercise, we consider the following helpers:

```js
const localStorage = {
```

player1: { id:1, name: 'Albert' },
player2: { id:2, name: 'Theresa' },
};

// getFromCache :: String -> IO User
const getFromCache = x => new IO(() => localStorage[x]);

// game :: User -> User -> String
const game = curry((p1, p2) => `${p1.name} vs ${p2.name}`);
```

{% exercise %}
Write an IO that gets both player1 and player2 from the cache and starts the game.

{% initial src="./exercises/ch10/exercise_c.js#L16;" %}
```
js // startGame :: IO String const startGame = undefined;
```

{% solution src="./exercises/ch10/solution_c.js" %}
{% validation src="./exercises/ch10/validation_c.js" %}
{% context src="./exercises/support.js" %}
{% endexercise %}
```