Chapter 05: Coding by Composing

Functional Husbandry

Here's `compose`:

```
const compose = (...fns) => (...args) => fns.reduceRight((res, fn) => [fn.call(null
```

… Don't be scared! This is the level-9000-super-Saiyan-form of *compose*. For the sake of reasoning, let's drop the variadic implementation and consider a simpler form that can compose two functions together. Once you get your head around that, you can push the abstraction further and consider it simply works for any number of functions (we could even prove that)!
Here's a more friendly *compose* for you my dear readers:

```
const compose2 = (f, g) => x => f(g(x));
```

`f` and `g` are functions and `x` is the value being "piped" through them.

Composition feels like function husbandry. You, breeder of functions, select two with traits you'd like to combine and mash them together to spawn a brand new one. Usage is as follows:

```
const toUpperCase = x => x.toUpperCase();
const exclaim = x => `${x}!`;
const shout = compose(exclaim, toUpperCase);

shout('send in the clowns'); // "SEND IN THE CLOWNS!"
```

The composition of two functions returns a new function. This makes perfect sense: composing two units of some type (in this case function) should yield a new unit of that very type. You don't plug two legos together and get a lincoln log. There is a theory here, some underlying law that we will discover in due time.

In our definition of `compose`, the `g` will run before the `f`, creating a right to left flow of data.

This is much more readable than nesting a bunch of function calls. Without compose, the above would read:

```
const shout = x => exclaim(toUpperCase(x));
```

Instead of inside to outside, we run right to left, which I suppose is a step in the left direction (boo!). Let's look at an example where sequence matters:

```
const head = x => x[0];
const reverse = reduce((acc, x) => [x].concat(acc), []);
const last = compose(head, reverse);

last(['jumpkick', 'roundhouse', 'uppercut']); // 'uppercut'
```

`reverse` will turn the list around while `head` grabs the initial item. This results in an effective, albeit inefficient, `last` function. The sequence of functions in the composition should be apparent here. We could define a left to right version, however, we mirror the mathematical version much more closely as it stands. That's right, composition is straight from the math books. In fact, perhaps it's time to look at a property that holds for any composition.

```
// associativity
compose(f, compose(g, h)) === compose(compose(f, g), h);
```

Composition is associative, meaning it doesn't matter how you group two of them. So, should we choose to uppercase the string, we can write:

```
compose(toUpperCase, compose(head, reverse));
// or
compose(compose(toUpperCase, head), reverse);
```

Since it doesn't matter how we group our calls to `compose`, the result will be the same. That allows us to write a variadic compose and use it as follows:

```
// previously we'd have to write two composes, but since it's associative,
// we can give compose as many fn's as we like and let it decide how to group them.
const arg = ['jumpkick', 'roundhouse', 'uppercut'];
const lastUpper = compose(toUpperCase, head, reverse);
const loudLastUpper = compose(exclaim, toUpperCase, head, reverse);
```

```
lastUpper(arg); // 'UPPERCUT'
loudLastUpper(arg); // 'UPPERCUT!'
```

Applying the associative property gives us this flexibility and peace of mind that the result will be equivalent. The slightly more complicated variadic definition is included with the support libraries for this book and is the normal definition you'll find in libraries like lodash, underscore, and ramda.

One pleasant benefit of associativity is that any group of functions can be extracted and bundled together in their very own composition. Let's play with refactoring our previous example:

```
const loudLastUpper = compose(exclaim, toUpperCase, head, reverse);

// -- or ----------------------------------------------------------

const last = compose(head, reverse);
const loudLastUpper = compose(exclaim, toUpperCase, last);

// -- or ----------------------------------------------------------

const last = compose(head, reverse);
const angry = compose(exclaim, toUpperCase);
const loudLastUpper = compose(angry, last);

// more variations...
```

There's no right or wrong answers - we're just plugging our legos together in whatever way we please. Usually it's best to group things in a reusable way like `last` and `angry`. If familiar with Fowler's "Refactoring", one might recognize this process as "extract method"…except without all the object state to worry about.

Pointfree

Pointfree style means never having to say your data. Excuse me. It means functions that never mention the data upon which they operate. First class functions, currying, and composition all play well together to create this style.

> Hint: Pointfree versions of `replace` & `toLowerCase` are defined in the Appendix C - Pointfree Utilities. Do not hesitate to have a peek!

```
// not pointfree because we mention the data: word
const snakeCase = word => word.toLowerCase().replace(/\s+/ig, '_');

// pointfree
const snakeCase = compose(replace(/\s+/ig, '_'), toLowerCase);
```

See how we partially applied `replace`? What we're doing is piping our data through each function of 1 argument. Currying allows us to prepare each function to just take its data, operate on it, and pass it along. Something else to notice is how we don't need the data to construct our function in the pointfree version, whereas in the pointful one, we must have our `word` available before anything else.

Let's look at another example.

```
// not pointfree because we mention the data: name
const initials = name => name.split(' ').map(compose(toUpperCase, head)).join('. ')

// pointfree
const initials = compose(join('. '), map(compose(toUpperCase, head)), split(' '));

initials('hunter stockton thompson'); // 'H. S. T'
```

Pointfree code can again, help us remove needless names and keep us concise and generic. Pointfree is a good litmus test for functional code as it lets us know we've got small functions that take input to output. One can't compose a while loop, for instance. Be warned, however, pointfree is a double-edged sword and can sometimes obfuscate intention. Not all functional code is pointfree and that is O.K. We'll shoot for it where we can and stick with normal functions otherwise.

Debugging

A common mistake is to compose something like `map`, a function of two arguments, without first partially applying it.

```
// wrong - we end up giving angry an array and we partially applied map with who kn
const latin = compose(map, angry, reverse);

latin(['frog', 'eyes']); // error

// right - each function expects 1 argument.
const latin = compose(map(angry), reverse);
```

```
latin(['frog', 'eyes']); // ['EYES!', 'FROG!'])
```

If you are having trouble debugging a composition, we can use this helpful, but impure trace function to see what's going on.

```
const trace = curry((tag, x) => {
  console.log(tag, x);
  return x;
});

const dasherize = compose(
  join('-'),
  toLower,
  split(' '),
  replace(/\s{2,}/ig, ' '),
);

dasherize('The world is a vampire');
// TypeError: Cannot read property 'apply' of undefined
```

Something is wrong here, let's `trace`

```
const dasherize = compose(
  join('-'),
  toLower,
  trace('after split'),
  split(' '),
  replace(/\s{2,}/ig, ' '),
);

dasherize('The world is a vampire');
// after split [ 'The', 'world', 'is', 'a', 'vampire' ]
```

Ah! We need to `map` this `toLower` since it's working on an array.

```
const dasherize = compose(
  join('-'),
  map(toLower),
  split(' '),
  replace(/\s{2,}/ig, ' '),
);

dasherize('The world is a vampire'); // 'the-world-is-a-vampire'
```

The `trace` function allows us to view the data at a certain point for debugging purposes. Languages like Haskell and PureScript have similar functions for ease of development.

Composition will be our tool for constructing programs and, as luck would have it, is backed by a powerful theory that ensures things will work out for us. Let's examine this theory.

Category Theory

Category theory is an abstract branch of mathematics that can formalize concepts from several different branches such as set theory, type theory, group theory, logic, and more. It primarily deals with objects, morphisms, and transformations, which mirrors programming quite closely. Here is a chart of the same concepts as viewed from each separate theory.

Types	Logic	Sets	Homotopy
A	proposition	set	space
$a : A$	proof	element	point
$B(x)$	predicate	family of sets	fibration
$b(x) : B(x)$	conditional proof	family of elements	section
$\mathbf{0}, \mathbf{1}$	$\bot, \top$	$\emptyset, \{\emptyset\}$	$\emptyset, *$
$A + B$	$A \vee B$	disjoint union	coproduct
$A \times B$	$A \wedge B$	set of pairs	product space
$A \to B$	$A \Rightarrow B$	set of functions	function space
$\sum_{(x:A)} B(x)$	$\exists_{x:A} B(x)$	disjoint sum	total space
$\prod_{(x:A)} B(x)$	$\forall_{x:A} B(x)$	product	space of sections
Id_A	equality $=$	$\{ (x, x) \mid x \in A \}$	path space A^I

Sorry, I didn't mean to frighten you. I don't expect you to be intimately familiar with all these concepts. My point is to show you how much duplication we have so you can see why category theory aims to unify these things.

In category theory, we have something called… a category. It is defined as a collection with the following components:

- A collection of objects
- A collection of morphisms
- A notion of composition on the morphisms

- A distinguished morphism called identity

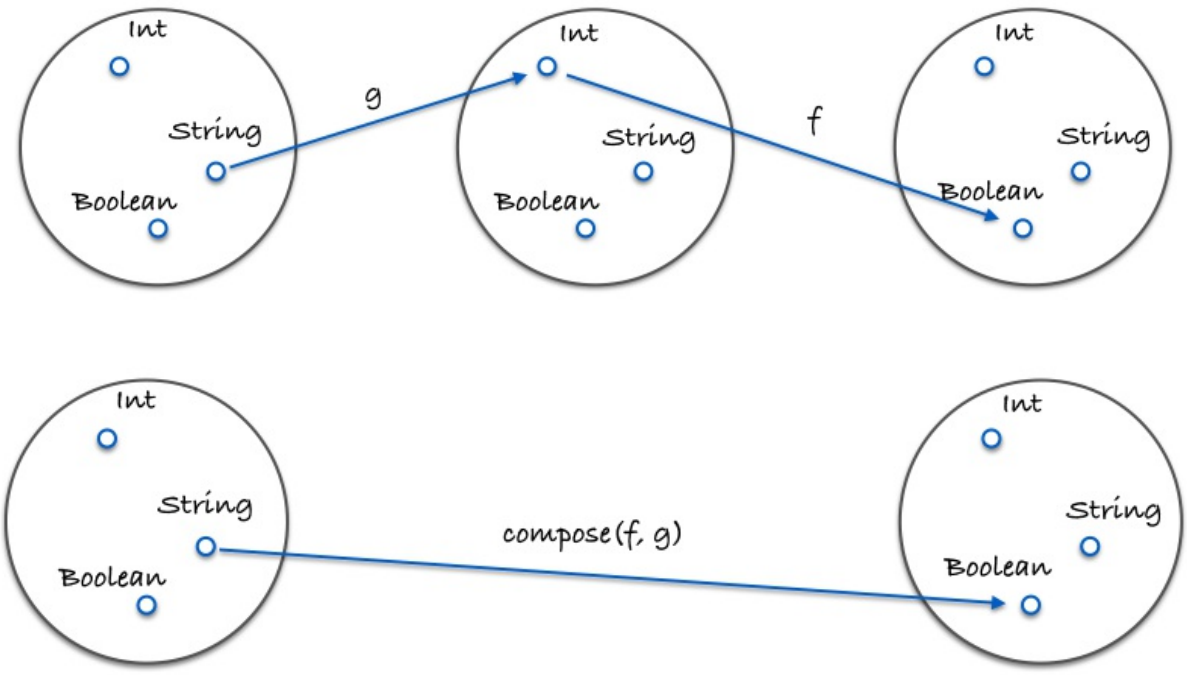
Category theory is abstract enough to model many things, but let's apply this to types and functions, which is what we care about at the moment.

A collection of objects
The objects will be data types. For instance, `String`, `Boolean`, `Number`, `Object`, etc. We often view data types as sets of all the possible values. One could look at `Boolean` as the set of `[true, false]` and `Number` as the set of all possible numeric values. Treating types as sets is useful because we can use set theory to work with them.

A collection of morphisms
The morphisms will be our standard every day pure functions.

A notion of composition on the morphisms
This, as you may have guessed, is our brand new toy - `compose`. We've discussed that our `compose` function is associative which is no coincidence as it is a property that must hold for any composition in category theory.

Here is an image demonstrating composition:



Here is a concrete example in code:

```
const g = x => x.length;
const f = x => x === 4;
const isFourLetterWord = compose(f, g);
```

A distinguished morphism called identity
Let's introduce a useful function called `id`. This function simply takes some input and spits it back at you. Take a look:

```
const id = x => x;
```

You might ask yourself "What in the bloody hell is that useful for?". We'll make extensive use of this function in the following chapters, but for now think of it as a function that can stand in for our value - a function masquerading as every day data.

`id` must play nicely with compose. Here is a property that always holds for every unary (unary: a one-argument function) function f:

```
// identity
compose(id, f) === compose(f, id) === f;
// true
```

Hey, it's just like the identity property on numbers! If that's not immediately clear, take some time with it. Understand the futility. We'll be seeing `id` used all over the place soon, but for now we see it's a function that acts as a stand in for a given value. This is quite useful when writing pointfree code.

So there you have it, a category of types and functions. If this is your first introduction, I imagine you're still a little fuzzy on what a category is and why it's useful. We will build upon this knowledge throughout the book. As of right now, in this chapter, on this line, you can at least see it as providing us with some wisdom regarding composition - namely, the associativity and identity properties.

What are some other categories, you ask? Well, we can define one for directed graphs with nodes being objects, edges being morphisms, and composition just being path concatenation. We can define with Numbers as objects and `>=` as morphisms (actually any partial or total order can be a category). There are heaps of categories, but for the purposes of this book, we'll only concern ourselves with the one defined above. We have sufficiently skimmed the surface and must move on.

In Summary

Composition connects our functions together like a series of pipes. Data will flow through our application as it must - pure functions are input to output after all, so breaking this chain would disregard output, rendering our software useless.

We hold composition as a design principle above all others. This is because it keeps our app simple and reasonable. Category theory will play a big part in app architecture, modelling side effects, and ensuring correctness.

We are now at a point where it would serve us well to see some of this in practice. Let's make an example application.

Chapter 06: Example Application

Exercises

In each following exercise, we'll consider Car objects with the following shape:

```
{
  name: 'Aston Martin One-77',
  horsepower: 750,
  dollar_value: 1850000,
  in_stock: true,
}
```

{% exercise %}
Use `compose()` to rewrite the function below.

{% initial src="./exercises/ch05/exercise_a.js#L12;" %}
`js const isLastInStock = (cars) => { const lastCar = last(cars); return prop('in_stock', lastCar); };`

{% solution src="./exercises/ch05/solution_a.js" %}
{% validation src="./exercises/ch05/validation_a.js" %}
{% context src="./exercises/support.js" %}
{% endexercise %}

Considering the following function:

```
const average = xs => reduce(add, 0, xs) / xs.length;
```

{% exercise %}
Use the helper function `average` to refactor `averageDollarValue` as a composition.

{% initial src="./exercises/ch05/exercise_b.js#L7;" %}
```
js const averageDollarValue = (cars) => { const dollarValues = map(c => c.dollar_value, cars); return average(dollarValues); };
```

{% solution src="./exercises/ch05/solution_b.js" %}
{% validation src="./exercises/ch05/validation_b.js" %}
{% context src="./exercises/support.js" %}
{% endexercise %}

{% exercise %}
Refactor `fastestCar` using `compose()` and other functions in pointfree-style. Hint, the `flip` function may come in handy.

{% initial src="./exercises/ch05/exercise_c.js#L4;" %}
```
js const fastestCar = (cars) => { const sorted = sortBy(car => car.horsepower); const fastest = last(sorted); return concat(fastest.name, ' is the fastest'); };
```

{% solution src="./exercises/ch05/solution_c.js" %}
{% validation src="./exercises/ch05/validation_c.js" %}
{% context src="./exercises/support.js" %}
{% endexercise %}