Appendix B: Algebraic Structures Support

In this appendix, you'll find some basic JavaScript implementations of various algebraic structures described in the book. Keep in mind that these implementations may not be the fastest or the
most efficient implementation out there; they *solely serve an educational purpose*.

In order to find structures that are more production-ready, have a peek at folktale or fantasy-land.

Note that some methods also refer to functions defined in the Appendix A

Compose

```
const createCompose = curry((F, G) => class Compose {
  constructor(x) {
    this.$value = x;
  }

  inspect() {
    return `Compose(${inspect(this.$value)})`;
  }

  // ----- Pointed (Compose F G)
  static of(x) {
    return new Compose(F(G(x)));
  }

  // ----- Functor (Compose F G)
  map(fn) {
    return new Compose(this.$value.map(x => x.map(fn)));
  }

  // ----- Applicative (Compose F G)
  ap(f) {
    return f.map(this.$value);
  }
});
```

Either

```
class Either {
  constructor(x) {
    this.$value = x;
  }

  // ----- Pointed (Either a)
  static of(x) {
    return new Right(x);
  }
}

class Left extends Either {
  get isLeft() {
    return true;
  }

  get isRight() {
    return false;
  }

  static of(x) {
    throw new Error(``of` called on class Left (value) instead of Either (type)');
  }

  inspect() {
    return `Left(${inspect(this.$value)})`;
  }

  // ----- Functor (Either a)
  map() {
    return this;
  }

  // ----- Applicative (Either a)
  ap() {
    return this;
  }

  // ----- Monad (Either a)
  chain() {
    return this;
  }

  join() {
    return this;
  }

  // ----- Traversable (Either a)
  sequence(of) {
    return of(this);
  }

  traverse(of, fn) {
    return of(this);
  }
```

```
}

class Right extends Either {
  get isLeft() {
    return false;
  }

  get isRight() {
    return true;
  }

  static of(x) {
    throw new Error('`of` called on class Right (value) instead of Either (type)');
  }

  inspect() {
    return `Right(${inspect(this.$value)})`;
  }

  // ----- Functor (Either a)
  map(fn) {
    return Either.of(fn(this.$value));
  }

  // ----- Applicative (Either a)
  ap(f) {
    return f.map(this.$value);
  }

  // ----- Monad (Either a)
  chain(fn) {
    return fn(this.$value);
  }

  join() {
    return this.$value;
  }

  // ----- Traversable (Either a)
  sequence(of) {
    return this.traverse(of, identity);
  }

  traverse(of, fn) {
    fn(this.$value).map(Either.of);
  }
}
```

Identity

```
class Identity {
  constructor(x) {
    this.$value = x;
  }

  inspect() {
    return `Identity(${inspect(this.$value)})`;
  }

  // ----- Pointed Identity
  static of(x) {
    return new Identity(x);
  }

  // ----- Functor Identity
  map(fn) {
    return Identity.of(fn(this.$value));
  }

  // ----- Applicative Identity
  ap(f) {
    return f.map(this.$value);
  }

  // ----- Monad Identity
  chain(fn) {
    return this.map(fn).join();
  }

  join() {
    return this.$value;
  }

  // ----- Traversable Identity
  sequence(of) {
    return this.traverse(of, identity);
  }

  traverse(of, fn) {
    return fn(this.$value).map(Identity.of);
  }
}
```

IO

```
class IO {
  constructor(fn) {
    this.unsafePerformIO = fn;
  }
```

```
  inspect() {
    return `IO(?)`;
  }

  // ----- Pointed IO
  static of(x) {
    return new IO(() => x);
  }

  // ----- Functor IO
  map(fn) {
    return new IO(compose(fn, this.unsafePerformIO));
  }

  // ----- Applicative IO
  ap(f) {
    return this.chain(fn => f.map(fn));
  }

  // ----- Monad IO
  chain(fn) {
    return this.map(fn).join();
  }

  join() {
    return this.unsafePerformIO();
  }
}
```

List

```
class List {
  constructor(xs) {
    this.$value = xs;
  }

  inspect() {
    return `List(${inspect(this.$value)})`;
  }

  concat(x) {
    return new List(this.$value.concat(x));
  }

  // ----- Pointed List
  static of(x) {
    return new List([x]);
  }

  // ----- Functor List
  map(fn) {
```

```
    return new List(this.$value.map(fn));
  }

  // ----- Traversable List
  sequence(of) {
    return this.traverse(of, identity);
  }

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

Map

```
class Map {
  constructor(x) {
    this.$value = x;
  }

  inspect() {
    return `Map(${inspect(this.$value)})`;
  }

  insert(k, v) {
    const singleton = {};
    singleton[k] = v;
    return Map.of(Object.assign({}, this.$value, singleton));
  }

  reduceWithKeys(fn, zero) {
    return Object.keys(this.$value)
      .reduce((acc, k) => fn(acc, this.$value[k], k), zero);
  }

  // ----- Functor (Map a)
  map(fn) {
    return this.reduceWithKeys(
      (m, v, k) => m.insert(k, fn(v)),
      new Map({}),
    );
  }

  // ----- Traversable (Map a)
  sequence(of) {
    return this.traverse(of, identity);
  }
```

```
  traverse(of, fn) {
    return this.reduceWithKeys(
      (f, a, k) => fn(a).map(b => m => m.insert(k, b)).ap(f),
      of(new Map({})),
    );
  }
}
```

Maybe

> Note that `Maybe` could also be defined in a similar fashion as we did for `Either` with two
> child classes `Just` and `Nothing`. This is simply a different flavor.

```
class Maybe {
  get isNothing() {
    return this.$value === null || this.$value === undefined;
  }

  get isJust() {
    return !this.isNothing;
  }

  constructor(x) {
    this.$value = x;
  }

  inspect() {
    return `Maybe(${inspect(this.$value)})`;
  }

  // ----- Pointed Maybe
  static of(x) {
    return new Maybe(x);
  }

  // ----- Functor Maybe
  map(fn) {
    return this.isNothing ? this : Maybe.of(fn(this.$value));
  }

  // ----- Applicative Maybe
  ap(f) {
    return this.isNothing ? this : f.map(this.$value);
  }

  // ----- Monad Maybe
  chain(fn) {
    return this.map(fn).join();
  }
```

```
  join() {
    return this.isNothing ? this : this.$value;
  }

  // ----- Traversable Maybe
  sequence(of) {
    this.traverse(of, identity);
  }

  traverse(of, fn) {
    return this.isNothing ? of(this) : fn(this.$value).map(Maybe.of);
  }
}
```

Task

```
class Task {
  constructor(fork) {
    this.fork = fork;
  }

  inspect() {
    return 'Task(?)';
  }

  static rejected(x) {
    return new Task((reject, _) => reject(x));
  }

  // ----- Pointed (Task a)
  static of(x) {
    return new Task((_, resolve) => resolve(x));
  }

  // ----- Functor (Task a)
  map(fn) {
    return new Task((reject, resolve) => this.fork(reject, compose(resolve, fn)));
  }

  // ----- Applicative (Task a)
  ap(f) {
    return this.chain(fn => f.map(fn));
  }

  // ----- Monad (Task a)
  chain(fn) {
    return new Task((reject, resolve) => this.fork(reject, x => fn(x).fork(reject,
  }

  join() {
```

```
      return this.chain(identity);
    }
  }
```