Appendix C: Pointfree Utilities

In this appendix, you'll find pointfree versions of rather classic JavaScript functions described in the book. All of the following functions are seemingly available in exercises, as part of the global context. 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 functions that are more production-ready, have a peek at ramda, lodash, or folktale.

Note that functions refer to the `curry` & `compose` functions defined in Appendix A

add

```
// add :: Number -> Number -> Number
const add = curry((a, b) => a + b);
```

chain

```
// chain :: Monad m => (a -> m b) -> m a -> m b
const chain = curry((fn, m) => m.chain(fn));
```

concat

```
// concat :: String -> String -> String
const concat = curry((a, b) => a.concat(b));
```

eq

```
// eq :: Eq a => a -> a -> Boolean
const eq = curry((a, b) => a === b);
```

filter

```
// filter :: (a -> Boolean) -> [a] -> [a]
const filter = curry((fn, xs) => xs.filter(fn));
```

flip

```
// flip :: (a -> b) -> (b -> a)
const flip = curry((fn, a, b) => fn(b, a));
```

forEach

```
// forEach :: (a -> ()) -> [a] -> ()
const forEach = curry((fn, xs) => xs.forEach(fn));
```

head

```
// head :: [a] -> a
const head = xs => xs[0];
```

intercalate

```
// intercalate :: String -> [String] -> String
const intercalate = curry((str, xs) => xs.join(str));
```

join

```
// join :: Monad m => m (m a) -> m a
const join = m => m.join();
```

last

```
// last :: [a] -> a
const last = xs => xs[xs.length - 1];
```

map

```
// map :: Functor f => (a -> b) -> f a -> f b
const map = curry((fn, f) => f.map(fn));
```

match

```
// match :: RegExp -> String -> Boolean
const match = curry((re, str) => re.test(str));
```

prop

```
// prop :: String -> Object -> a
const prop = curry((p, obj) => obj[p]);
```

reduce

```
// reduce :: (b -> a -> b) -> b -> [a] -> b
const reduce = curry((fn, zero, xs) => xs.reduce(fn, zero));
```

replace

```
// replace :: RegExp -> String -> String -> String
const replace = curry((re, rpl, str) => str.replace(re, rpl));
```

reverse

```
// reverse :: [a] -> [a]
const reverse = x => Array.isArray(x) ? x.reverse() : x.split('').reverse().join(''
```

safeHead

```
// safeHead :: [a] -> Maybe a
const safeHead = compose(Maybe.of, head);
```

safeLast

```
// safeLast :: [a] -> Maybe a
const safeLast = compose(Maybe.of, last);
```

safeProp

```
// safeProp :: String -> Object -> Maybe a
const safeProp = curry((p, obj) => compose(Maybe.of, prop(p))(obj));
```

sequence

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

sortBy

```
// sortBy :: Ord b => (a -> b) -> [a] -> [a]
```

```
const sortBy = curry((fn, xs) => {
  return xs.sort((a, b) => {
    if (fn(a) === fn(b)) {
      return 0;
    }

    return fn(a) > fn(b) ? 1 : -1;
  });
});
```

split

```
// split :: String -> String -> [String]
const split = curry((sep, str) => str.split(sep));
```

take

```
// take :: Number -> [a] -> [a]
const take = curry((n, xs) => xs.slice(0, n));
```

toLowerCase

```
// toLowerCase :: String -> String
const toLowerCase = s => s.toLowerCase();
```

toString

```
// toString :: a -> String
const toString = String;
```

toUpperCase

```
// toUpperCase :: String -> String
```

```
const toUpperCase = s => s.toUpperCase();
```

traverse

```
// traverse :: (Applicative f, Traversable t) => (a -> f a) -> (a -> f b) -> t a ->
const traverse = curry((of, fn, f) => f.traverse(of, fn));
```

unsafePerformIO

```
// unsafePerformIO :: IO a -> a
const unsafePerformIO = io => io.unsafePerformIO();
```