JavaScript Programmers Should Learn Algebraic Data Types

LambdaConf 2018

by Seth House

@whiteinge
seth@eseth.com

http://talks.eseth.com/#js-adts

JavaScript!

Why ADTs?

Common advice for functional-style coding.

- Write small, single-purpose functions.

Common advice for functional-style coding.

- Write small, single-purpose functions.
- Make them pure.

Common advice for functional-style coding.

- Write small, single-purpose functions.
- Make them pure.
- Compose them together into a larger whole.

```
const tableContents = _.chain(allRecords)
      .filter(_.overEvery(_.values(predicateFns)))
      .slice(curPageStart, curPageOffset)
      .orderBy(..._.unzip(orderedColumns))
      .map(x => _.pick(x, visibleColumns))
      .groupBy(currentSelection)
      .value()
```

```
const valOrDefault = ifThenElse(
    mq => mq.selected == null && 'defaultall' in $attrs,
    () => true,
    get('selected'));

const valShouldBeSelected = ifThenElse(
    () => selectionOverrides != null,
    mq => selectionOverrides.includes(mq.name),
    valOrDefault);

const markAsSelected = ifThenElse(valShouldBeSelected,
    mq => Object.assign({}, mq, {selected: true}),
    mq => Object.assign({}, mq, {selected: false}));

const getOptions = pipe([
    sortBy('name'),
    map(markAsSelected),
    map(createOption),
]);
```

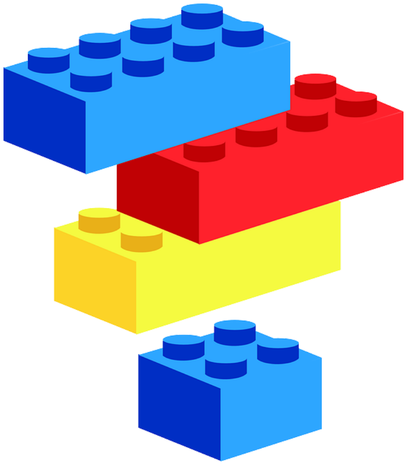
...now what?

Composition.

Favor composition over inheritance

– Gang of Four

> Designing is fundamentally about taking things apart in such a way that they can be put back together. Separating things into things that can be composed.
>
> – Rich Hickey



[Programming is] all about decomposing the problem and then recomposing solutions.

– Bartoz Milewski

[Programming is] all about decomposing the problem and then recomposing solutions.

There are so many ways of composing things and each of them is different.

– Bartoz Milewski

[Programming is] all about decomposing the problem and then recomposing solutions.

There are so many ways of composing things and each of them is different.

Category theory describes all these various ways of composing things.

– Bartoz Milewski

“The perfect API”.

“The perfect API”.

(Brought to you by math.)

Too hard?

If you write anything in italics so it looks like math developers go, "Ooh, I can't do that." And I'm really surprised that they have that reaction because they can do JavaScript. And JavaScript sometimes involves a lot of little picky tiny rules that are quite illogical.

– Philip Wadler

It really does constrain your ability to think when you're thinking in terms of a programming language. Code makes you miss the forest for the trees: It draws your attention to the working of individual pieces, rather than to the bigger picture of how your program fits together, or what it's supposed to do—and whether it actually does what you think.

– Leslie Lamport

Prerequisites.

- Desire to learn more about functional-style JavaScript.

Prerequisites.

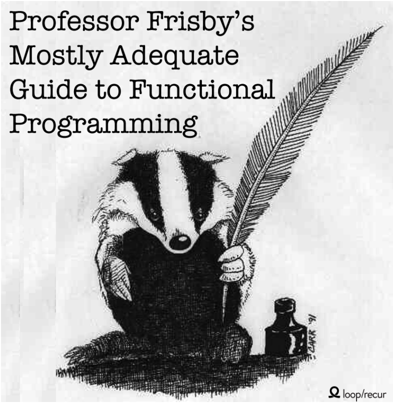
- Desire to learn more about functional-style JavaScript.
- ES5 Array extras: `map`, `filter`, `reduce`.

Prerequisites.

- Desire to learn more about functional-style JavaScript.
- ES5 Array extras: `map`, `filter`, `reduce`.
- FP baseline: compose, closures, decorators, partial application, currying.

Prerequisites.

- Desire to learn more about functional-style JavaScript.
- ES5 Array extras: `map`, `filter`, `reduce`.
- FP baseline: compose, closures, decorators, partial application, currying.



Goal.

- Practical use-cases.

Goal.

- Practical use-cases.
- Ship code.

Goal.

- Practical use-cases.
- Ship code.
- Adopt these ideas incrementally with your team.

Goal.

- Practical use-cases.
- Ship code.
- Adopt these ideas incrementally with your team.

Tom Harding

Fantas, Eel, and Specification

Why JavaScript?

- Ubiquitous. (For now.)

Why JavaScript?

- Ubiquitous. (For now.)
- We need to up our game.

Why JavaScript?

- Ubiquitous. (For now.)
- We need to up our game.
- Our async primitives are not good enough.

Data types not type checking.

What are ADTs?

User-defined types.

Examples.

Example: Box

Problem: Build a pipeline of computations.

```
const capitalize = ([h, ...t]) => `${h.toUpperCase()}${t.join('')}`
const exclaim = x => `${x}!`
```

Problem: Build a pipeline of computations.

```
const capitalize = ([h, ...t]) => `${h.toUpperCase()}${t.join('')}`
const exclaim = x => `${x}!`
```

```
Box.of('foo')
    .map(capitalize)
    .map(exclaim)
// => Box('Foo!')
```

Problem: Build a pipeline of computations.

```
const capitalize = ([h, ...t]) => `${h.toUpperCase()}${t.join('')}`
const exclaim = x => `${x}!`
```

```
Box.of('foo')
    .map(capitalize)
    .map(exclaim)
// => Box('Foo!')
```

```
const step1 = Box.of('foo')
    .map(capitalize)

const val1 = step1.map(exclaim)
// => Box('Foo!')

const val2 = step1.map(question)
// => Box('Foo?')
```

Summary: Build a pipeline of computations.

```
class Box {
    constructor(x) { this._x = x }
    map(f) { return new Box(f(this._x)) }
    chain(f) { return f(this._x) }

    static of(x) { return new Box(x) }
}
```

(Evaluated eagerly but there is a lazy variant.)

Problem: Build a pipeline of computations.

> map is not about iterating. It's about going inside of an object or a data structure and running a function from within that data structure on its properties or behavior.
>
> – Brian Lonsdorf

Problem: Build a pipeline of computations.

```
Box.of('foo')
    .map(capitalize)
    .map(exclaim)
```

Equivalent to:

```
const compose = (f, g) => (...args) => f(g(...args));

Box.of('foo')
    .map(compose(exclaim, capitalize))
```

Problem: Build a pipeline of computations.

```
Box.of('foo')
    .map(capitalize)
    .map(exclaim)
```

Equivalent to:

```
const compose = (f, g) => (...args) => f(g(...args));

Box.of('foo')
    .map(compose(exclaim, capitalize))
```

`map` is `compose`!

Problem: Build a pipeline of computations.

Combine values from two `Box` containers:

```
const getBoxOne = () => Box.of('one')
const combined = Box.of('two')
    .chain(two =>
        getBoxOne().map(one => `${one} and ${two}`))
```

Summary: Build a pipeline of computations.

- `map` is `compose`.
- Term: functor has the `map` interface.
- Term: pointed functor has the `of` interface.
- Term: monad has the `chain` interface.

Summary: Build a pipeline of computations.

You've already used these (similar anyway).

Summary: Build a pipeline of computations.

You've already used these (similar anyway).

- Promises? Almost, minus fundamental design flaws:

Summary: Build a pipeline of computations.

You've already used these (similar anyway).

- Promises? Almost, minus fundamental design flaws:
    - `then` is both `map` and `chain` (limits flexibility).

Summary: Build a pipeline of computations.

You've already used these (similar anyway).

- Promises? Almost, minus fundamental design flaws:
  - `then` is both `map` and `chain` (limits flexibility).
  - Eager, not lazy (drastically limits composability).

Summary: Build a pipeline of computations.

You've already used these (similar anyway).

- Promises? Almost, minus fundamental design flaws:
  - `then` is both `map` and `chain` (limits flexibility).
  - Eager, not lazy (drastically limits composability).
- Lodash?
  - `flatMap` nested arrays.

Summary: Build a pipeline of computations.

You've already used these (similar anyway).

- Promises? Almost, minus fundamental design flaws:
  - `then` is both `map` and `chain` (limits flexibility).
  - Eager, not lazy (drastically limits composability).
- Lodash?
  - `flatMap` nested arrays.
- Ramda?
  - `chain`.

Summary: Build a pipeline of computations.

You've already used these (similar anyway).

- Promises? Almost, minus fundamental design flaws:
  - `then` is both `map` and `chain` (limits flexibility).
  - Eager, not lazy (drastically limits composability).
- Lodash?
  - `flatMap` nested arrays.
- Ramda?
  - `chain`.
- RxJS?
  - `flatMap` & `concatMap`.

Example: Maybe

Problem: A value might be `null`.

AKA, any JavaScript function ever.

Problem: A value might be `null`.

AKA, any JavaScript function ever.

```
const match = 'foo bar baz qux'.match(/grault/) // null
const word = match[0] // TypeError or ""

const wordAsCap = capitalize(word)
const wordAsExclaim = exclaim(wordAsCap)
```

Problem: A value might be `null`.

```
class Maybe {
    constructor(val, type) { this.val = val; this.type = type }
    map(f) { return this.type === 'Ok'
        ? Maybe.Ok(f(this.val)) : this }
    chain(f) { return this.type === 'Ok' ? f(this.val) : this }
    getOrElse(def) { return this.type === 'Ok' ? this.val : def }

    static Nothing(x) { return new Maybe(undefined, 'Nothing') }
    static Ok(x) { return new Maybe(x, 'Ok') }

    static fromNull(x) { return x == null ?
        Maybe.Nothing() : Maybe.Ok(x) }
    static tryCatch(f, ...args) {
        try { return Maybe.Ok(f(...args)) }
        catch(e) { return Maybe.Nothing() }
    }
}
```

Problem: A value might be `null`.

```
Maybe.fromNull('foo bar baz qux'.match(/grault/))
    .chain(x => Maybe.tryCatch(() => x[0]))
    .chain(x => x == "" ? Maybe.Nothing() : Maybe.Ok(x))
    .map(capitalize)
    .map(exclaim)
    .getOrElse('Not found!')
```

Common example: Either

Problem: Conditional code branches.

- Success or Error.
- Thing or Other-Thing.
- If / else.
- Same benefits of `Maybe` plus you can retain some data from the 'else' path.

Solution: Conditional code branches.

```
class Either {
    constructor(val, type) { this.val = val; this.type = type }

    map(f) { return this.type === 'Right'
        ? Either.Right(f(this.val)) : this }
    chain(f) { return this.type === 'Right' ? f(this.val) : this }
    fold(f, g) {
        switch(this.type) {
            case 'Left': return f(this.val);
            case 'Right': return g(this.val);
        }
    }

    static Left(x) { return new Either(x, 'Left') }
    static Right(x) { return new Either(x, 'Right') }

    static of(x) { return Either.Right(x) }
    static fromNullable(x) { return x != null
        ? Either.Right(x) : Either.Left(x) }
}
```

Common example: Task

Problem: Compose lazy, async pipelines.

- Like a promise but lazy and doesn't auto-flatten.

Problem: Compose lazy, async pipelines.

- Like a promise but lazy and doesn't auto-flatten.
- Enables broad composition.

Problem: Compose lazy, async pipelines.

- Like a promise but lazy and doesn't auto-flatten.
- Enables broad composition.
- Sequential execution with `chain`.

Problem: Compose lazy, async pipelines.

- Like a promise but lazy and doesn't auto-flatten.
- Enables broad composition.
- Sequential execution with `chain`.
- Parallel execution with `ap` (and currying).

Problem: Compose lazy, async pipelines.

- Like a promise but lazy and doesn't auto-flatten.
- Enables broad composition.
- Sequential execution with `chain`.
- Parallel execution with `ap` (and currying).
- Race multiple executions with `alt`.

Solution: Compose lazy, async pipelines.

```
const compose = (f, g) => (...args) => f(g(...args));

class Task {
    constructor(fork) { this.fork = fork }
    static of(fork) { return new Task(fork) }

    map(f) { return Task.of((rej, res) =>
        this.fork(rej, compose(res, f))) }
    chain(f) { return Task.of((rej,res) =>
        this.fork(rej, x => f(x).fork(rej, res))) }
    ap(task) { return Task.of((rej, res) =>
        this.fork(rej, f => task.fork(rej, compose(res, f))) ) }

    static fromPromise(f, ...args) {
        return Task.of((reject, resolve) =>
            f(...args).then(resolve, reject)) }
}
```

Solution: Compose lazy, async pipelines.

```
const getTimer = wait => Task.of((rej, res) =>
    setTimeout(res, wait, `Waited for ${wait}`));
```

Solution: Compose lazy, async pipelines.

```
const getTimer = wait => Task.of((rej, res) =>
    setTimeout(res, wait, `Waited for ${wait}`));
```

```
Task.of((rej, res) => res(x => y => ({x, y})))
    .ap(getTimer(500).map(x => `First ${x}`))
    .ap(getTimer(500).map(x => `Second ${x}`))
    .fork(console.error, console.log)
```

Summary: Compose lazy, async pipelines.

- Term: applicative has the `ap` interface.

Common example: Combine Values

Problem: Combine the same kind of things.

```
'foo'.concat('bar').concat('baz');
```

Problem: Combine the same kind of things.

```
'foo'.concat('bar').concat('baz');
```

```
['foo'].concat(['bar', 'baz'])
```

Problem: Combine the same kind of things.

```
'foo'.concat('bar').concat('baz');
```

```
['foo'].concat(['bar', 'baz'])
```

```
Sum(3).concat(Sum(2))
// ⇒ Sum(5)
```

Solution: Combine the same kind of things.

`concat` combines things in a predicable order.

Problem: Reduce needs an external seed value.

```
['foo', 'bar'].reduce(
    (acc, cur) => { acc[cur] = cur; return acc },
    {})
```

Solution: Reduce needs an external seed value.

Generalize:

```
String.empty = () ⇒ '';
Array.empty = () ⇒ [];
```

Solution: Reduce needs an external seed value.

Generalize:

```
String.empty = () => '';
Array.empty = () => [];
```

```
const fold = M => xs => xs.reduce(
    (acc, x) => acc.concat(x),
    M.empty());
```

Solution: Reduce needs an external seed value.

Generalize:

```
String.empty = () => '';
Array.empty = () => [];
```

```
const fold = M => xs => xs.reduce(
    (acc, x) => acc.concat(x),
    M.empty());
```

```
fold(String)(['foo', 'bar'])
fold(Array)(['foo', 'bar'])
```

Solution: Reduce needs an external seed value.

Generalize:

```
String.empty = () => '';
Array.empty = () => [];
```

```
const fold = M => xs => xs.reduce(
    (acc, x) => acc.concat(x),
    M.empty());
```

```
fold(String)(['foo', 'bar'])
fold(Array)(['foo', 'bar'])
```

What about objects?

Solution: Reduce needs an external seed value.

```
class Collection {
    static of(x) { return Collection.Add(x) }
    static empty() { return Collection.Add({}) }
    static Add(x) { return new Collection(x, 'Add') }
    static Del(x) { return new Collection(x, 'Del') }
    constructor(val, type) { this.val = val; this.type = type }
    concat(x) {
        if (x.type === 'Add') {
            Object.assign(this.val, x.val); return this }
        if (x.type === 'Del') {
            Object.keys(x.val).forEach(key => delete this.val[key]);
            return this;
        }
    }
}
```

Solution: Reduce needs an external seed value.

Manage an object over time:

```
fold(Collection)([
    Collection.Add({foo: 'Foo!'}),
    Collection.Add({bar: 'Bar!'}),
    Collection.Add({baz: 'Baz!'}),
    Collection.Del({bar: 'Bar!'}),
]).val
```

Solution: Reduce needs an external seed value.

```
// Flux! (Seriously.)
const Dispatcher = new Rx.Subject()
const send = (tag, arg) => ev => Dispatcher.onNext({
    tag,
    data: typeof arg === 'function' ? arg(ev) : arg,
})
```

```
const ThingStore = Dispatcher
    .filter(({tag}) => tag === 'THING_ADDED'
        || tag === 'THING_REMOVED')
    .scan(function(acc, {tag, data}) {
        switch (tag) {
            case 'THING_ADDED':
                acc[data] = data;
                return acc;
            case 'THING_REMOVED':
                delete acc[data];
                return acc;
            default:
                return acc;
        }
    }, {})
    .startWith({})
    .map(things => `
        <p><input> <button onclick="send('THING_ADDED', ev =>
            previousElementSibling.value)(event)">Add</button></p>
        <ul>${Object.entries(things).map(([key, val]) =>
            `<li>${val}
                <button onclick="send('THING_REMOVED',
                    '${val}')()">X</button>
            </li>`).join('')}</ul>
    `)
```

```
const foldp = (M, seed = M.empty()) => o => o
    .scan((acc, x) => acc.concat(x), seed)
    .startWith(seed)
    .pluck('val');

const ThingStore = Dispatcher
    .filter(({tag}) => tag === 'THING')
    .pluck('data')
    .let(foldp(Collection))
    .map(things => `
        <p><input> <button onclick="send('THING',
                ({target: {previousElementSibling: {value}}}) =>
            Collection.Add({
                [value]: value,
            }))(event)">Add</button></p>
        <ul>
            ${Object.entries(things).map(([key, val]) =>
                `<li>${val}
                    <button onclick="send('THING', Collection.Del({
                        ['${val}']: '${val}',
                    }))()">X</button>
                </li>`).join('')}
        </ul>
    `)
```

Summary: Combine and reduce values.

- Term: semigroup has the `concat` interface.
- Term: monoid has the `empty` interface.

Summary: Combine and reduce values.

- Term: semigroup has the `concat` interface.
- Term: monoid has the `empty` interface.
  - Sum.empty = () => Sum(0);
  - Product.empty = () => Product(1);
  - Max.empty = () => Max(-Infinity);
  - Min.empty = () => Min(Infinity);
  - All.empty = () => All(true);
  - Any.empty = () => Any(false);

Make your own types.

Example: XhrResult.

Problem: An XHR request/response has four states.

- Initial -> Loading -> Success/Error.

Problem: An XHR request/response has four states.

- Initial -> Loading -> Success/Error.
- Often spread across different levels of the app:
  - Angular: `$scope.spinner = true`
  - Redux: `{...state, spinner: true}`

Problem: An XHR request/response has four states.

- Initial -> Loading -> Success/Error.
- Often spread across different levels of the app:
  - Angular: `$scope.spinner = true`
  - Redux: `{...state, spinner: true}`
- Must check current state at every level before accessing data.

Problem: An XHR request/response has four states.

- Initial -> Loading -> Success/Error.
- Often spread across different levels of the app:
  - Angular: `$scope.spinner = true`
  - Redux: `{...state, spinner: true}`
- Must check current state at every level before accessing data.
- Often leads to business logic in the view.

Solution: An XHR request/response has four states.

```
class XhrResult {
    constructor(val, type) { this.val = val; this.type = type }

    inspect() { return `${this.type}: ${JSON.stringify(this.val)}` }
    map(f) { return this.type === 'Right'
        ? XhrResult.Right(f(this.val)) : this }
    chain(f) { return this.type === 'Right' ? f(this.val) : this }
    fold(f, g, h, i) {
        switch(this.type) {
            case 'Left': return f(this.val);
            case 'Right': return g(this.val);
            case 'Loading': return h(this.val);
            case 'Initial': return i(this.val);
        }
    }

    static Left(x) { return new XhrResult(x, 'Left') }
    static Right(x) { return new XhrResult(x, 'Right') }
    static Loading(x) { return new XhrResult(x, 'Loading') }
    static Initial(x) { return new XhrResult(x, 'Initial') }
    static of(x) { return XhrResult.Right(x) }
}
```

Solution: An XHR request/response has four states.

```
function wrapResponse(resp) {
    return !resp.errors || (resp.status >= 200 && resp.status < 300)
        ? XhrResult.Right(resp)
        : XhrResult.Left(resp);
}

function wrapXhr(ox) {
    const cacheLookupTimeout = 10;
    return ox.publish(oy => oy
        .map(resp => Rx.Observable.just(resp).map(wrapResponse))
        .takeUntilWithTime(cacheLookupTimeout)
        .defaultIfEmpty(Rx.Observable.just(XhrResult.Loading())
            .concat(oy.map(wrapResponse)))
        .mergeAll());
}
```

Solution: An XHR request/response has four states.

```
// Fake a list of user names.
// A map of all possible users to use for the initial render.
const userList = _.range(30).map(x => `user-${x}`);
const defaultUserMap = userList.reduce(function(acc, user) {
    acc[user] = XhrResult.Initial();
    return acc;
}, {});

// Make an xhr observable for each user; limit the number of calls.
const maxParallelCalls = 5;
const allTheXhrs = Rx.Observable.from(userList)
    .flatMapWithMaxConcurrent(maxParallelCalls, user =>
        makeXhr(user)
            .let(wrapXhr)
            .map(ret => ({user, ret})));

// Emit the initials all at once then accumulate updates.
const allTheUsers = allTheXhrs
    .scan(function(acc, {user, ret}) {
        acc[user] = ret;
        return acc;
    }, defaultUserMap)
    .startWith(defaultUserMap);
```

Solution: An XHR request/response has four states.

```
// Demo!
const content = document.querySelector('#content');
const sub = allTheUsers.subscribe(x =>
    content.innerHTML = _.chain(x)
        .map((ret, user) => `<li>${user}: ${ret.inspect()}</li>`)
        .value()
        .join('\n'));

function makeXhr(user) {
    // return xhr.xhrNext(PUT, `/some/url/${user}`);
    return Rx.Observable.just(user)
        .delay(_.random(100, 3000))
        .map(body => ({status: 200, body}));
}
```

Conclusion.

Summary of types.

- Function composition – Functor (`map`)
- Sequential execution – Monad (`map`, `chain`)
- Parallel, recursive execution – Applicative (`ap`, `map`)
- Combination – Semigroup (`concat`)
- Reduction – Monoid (`concat`, `empty`)

Closing thoughts.

- Use the simplest abstraction needed to solve the problem.

Closing thoughts.

- Use the simplest abstraction needed to solve the problem.
- RxJS Observables may be a good first step:

Closing thoughts.

- Use the simplest abstraction needed to solve the problem.
- RxJS Observables may be a good first step:
  - Popular, pragmatic, useful for other things.

Closing thoughts.

- Use the simplest abstraction needed to solve the problem.
- RxJS Observables may be a good first step:
    - Popular, pragmatic, useful for other things.
    - Highly composable.

Closing thoughts.

- Use the simplest abstraction needed to solve the problem.
- RxJS Observables may be a good first step:
  - Popular, pragmatic, useful for other things.
  - Highly composable.
  - Functor, pointed functor, monad, semigroup, monoid, and more...

Closing thoughts.

- Use the simplest abstraction needed to solve the problem.
- RxJS Observables may be a good first step:
  - Popular, pragmatic, useful for other things.
  - Highly composable.
  - Functor, pointed functor, monad, semigroup, monoid, and more...
  - Subscription combinations allow central management of effects (SerialDisposable, CompositeDisposable, RefCountDisposable).