Algebraic Data Types Composition

and

Software Complexity

Utah Clojure 2019-05-22

by Seth House

@whiteinge
seth@eseth.com

Why JavaScript Programmers Should Learn Algebraic Data Types

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

Why JavaScript?

Why JavaScript?

- "JavaScript fatigue".

Why JavaScript?

- "JavaScript fatigue".
- Inadequate primitives.

Why JavaScript?

- "JavaScript fatigue".
- Inadequate primitives.
- Frameworks over fundamentals.

Two equivalent approaches?

Two equivalent approaches?

- Dynamic vs. static.
- Type-checking vs. (more) tests.
- Compile-time checks vs. run-time checks.

Discussion!

Discussion!

- What are we trying to protect against, and how?
- Refactoring. (From both you and your teammates.)
- Renaming and code churn. (Is it worth it ever?)
- Feature additions and changing business requirements.
- Safety from bad external data.
- Safety from security breaches.
- Optimistic vs. defensive programming. (Boundaries?)
- Can you carefully prevent failures? Or do you plan for failures and have robust recovery (e.g. Erlang's OTP)?
- The goal is to have confidence that the thing you wrote is going to do the thing you say it's going to do, and that it will be resilient in the face of unknown or surprising usage.
- Composition for the UI like composition at the CLI?

Software Complexity

Programmers: Stop Calling Yourselves Engineers

It undermines a long tradition of designing and building infrastructure in the public interest.

IAN BOGOST NOV 5, 2015

Lynn Scurfield

The Coming Software Apocalypse

A small group of programmers wants to change how we code—before catastrophe strikes.

JAMES SOMERS

SEP 26, 2017 | TECHNOLOGY

The stakes keep rising, but programmers aren’t stepping up—they haven’t developed the chops required to handle increasingly complex problems. “In the 15th century,” he said, “people used to build cathedrals without knowing calculus, and nowadays I don’t think you’d allow anyone to build a cathedral without knowing calculus.

I look at the past century and I can't find one industry that improved safety or security without being forced. Cars, planes, pharmaceuticals, medical devices, food, restaurants, workplace, consumer goods, most recently financial products.

In every case the economics rewards skimping on security and safety. Taking the chance, hoping you do ok, rolling the dice in the courts if you don't, and stalling regulation as far as possible.

– Bruce Schneier

Abstraction

Reduce boilerplate
vs.
Think on another level

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

Composition.

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?

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

Algebraic Data Types

Make everything a value.
Make your data a value.
Make your functions a value.
Make your effects a value.
Make your errors a value.

– Paul Snively #lambdaconf

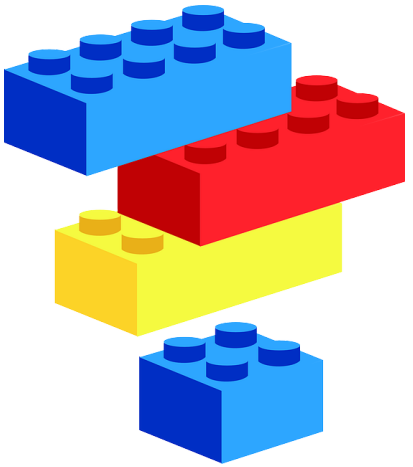
> Make everything a value.
> Make your data a value.
> Make your functions a value.
> Make your effects a value.
> Make your errors a value.
>
> – Paul Snively #lambdaconf

This allows you to retain control of the execution of your program.

“The perfect API”.

“The perfect API”.

(Brought to you by math.)



Group-like structures

	Totality[α]	Associativity	Identity	Invertibility	Commutativity
Semigroupoid	Unneeded	Required	Unneeded	Unneeded	Unneeded
Small Category	Unneeded	Required	Required	Unneeded	Unneeded
Groupoid	Unneeded	Required	Required	Required	Unneeded
Magma	Required	Unneeded	Unneeded	Unneeded	Unneeded
Quasigroup	Required	Unneeded	Unneeded	Required	Unneeded
Loop	Required	Unneeded	Required	Required	Unneeded
Semigroup	Required	Required	Unneeded	Unneeded	Unneeded
Inverse Semigroup	Required	Required	Unneeded	Required	Unneeded
Monoid	Required	Required	Required	Unneeded	Unneeded
Group	Required	Required	Required	Required	Unneeded
Abelian group	Required	Required	Required	Required	Required

^α Closure, which is used in many sources, is an equivalent axiom to totality, though defined differently.

Too hard?

Too hard?

> People say why do you need this thing? I can't prove that you need it. I would say try it without it and see how much work you have to do. See how much discipline you have to have. See how much structure you have to invent. And then maybe you'll come to appreciate what philosophies these systems embody.
>
> – David Nolen (on Om.Next / Falcor / et al)

Too hard?

> I find that when someone's taking time to do something right in the present, they're a perfectionist with no ability to prioritize, whereas when someone took time to do something right in the past, they're a master artisan of great foresight.
>
> – https://xkcd.com/974/

Abstraction

Abstraction

> I want to forget about the details of implementation for this particular thing so I can use it later with a hundred other things. [...] I don't care how this thing is implemented. I just care how this thing relates to other things.
>
> – Bartoz Milewski

Simple vs. Easy

Simple vs. Easy

```
// Simple
const numberList = [1, 5, 9]
let sum1 = 0
for (let i = 0; i < numberList.length; i += 1) {
  sum1 += numberList[i]
}
```

Simple vs. Easy

```
// Simple
const numberList = [1, 5, 9]
let sum1 = 0
for (let i = 0; i < numberList.length; i += 1) {
  sum1 += numberList[i]
}
```

```
// Somewhat simple; somewhat easy
const sum2 = [1, 5, 9].reduce((acc, cur) ⇒ acc + cur, 0)
```

Simple vs. Easy

```
// Simple
const numberList = [1, 5, 9]
let sum1 = 0
for (let i = 0; i < numberList.length; i += 1) {
  sum1 += numberList[i]
}
```

```
// Somewhat simple; somewhat easy
const sum2 = [1, 5, 9].reduce((acc, cur) ⇒ acc + cur, 0)
```

```
// Easy
const fold = M ⇒ xs ⇒
    xs.reduce((acc, x) ⇒ acc.concat(x), M.empty())
const Sum = val ⇒ ({ val, concat: x ⇒ Sum(val + x.val) })
Sum.empty = () ⇒ Sum(0)

const sum3 = fold(Sum)([Sum(1), Sum(5), Sum(9)])
```

Simple vs. Easy

```
Sum.empty = () => Sum(0);
Product.empty = () => Product(1);
Max.empty = () => Max(-Infinity);
Min.empty = () => Min(Infinity);
All.empty = () => All(true);
Any.empty = () => Any(false);
```

Richer Primitives

- Folding.
- State management.
- State transitions.
- Async.
- Concurrency.
- Tree traversal.
- Map over a list. Map over a rose tree.
- Computing deltas.
- Distributed computation & collecting the results.
- Reactivity.
- Having a value or not (null or future).

Example: Cartesian Product

- Lists are applicatives (in Ramda). `lift` takes a vanilla function into an applicative context so it can then combine things as you wish.

Example: Cartesian Product

- Lists are applicatives (in Ramda). `lift` takes a vanilla function into an applicative context so it can then combine things as you wish.

```
const combineFn = (a, b) ⇒ [a, b]
const result = R.lift(combineFn)(
    ['Women', 'Men', 'Kids'],
    ['Red', 'Green', 'Blue']
)
```

Example: Remote Data

Problem: An XHR request/response has four states.

- Initial -> Loading -> Success/Error.

Example: Remote 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}`

Example: Remote 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.

Example: Remote 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.

Example: Remote 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.
- https://gist.github.com/whiteinge/7721a637afd4c001313514062bd1bdbb

Abstraction

What are your favorite abstractions?