framp.me

Algebraic Data Types in JS

slides on framp.me/fp-workshop/2

disclaimer

we're going to refer to Fantasy Land Specification

https://github.com/fantasyland/fantasy-land

Algebra

it's a container

it has methods

it obeys to laws

an Algebra is a container

```
// Container :: a -> Container a
const Container = (a) => ({
  __value: a
})

assert(Container(5).__value, 5) // Noice!
```

Setoid

a Setoid is a container with a `.equals()`

```
// NaturalList :: a -> NaturalList a
const List = (a) => ({
  __value: a,
  // equals :: NaturalList a -> NaturalList b -> Boolean
  equals: (b) => a.every((v, i) =>
    Math.abs(v) === Math.abs(b.__value[i]))
})

assert(List([1,2,3]).equals(List([1,2,3])) === true)
assert(List([1,2,3]).equals(List([-1,-2,-3])) === true)
assert(List([1,2,3]).equals(List([1,2,42])) === false)
```

Setoid laws

a.equals(a) === true (reflexivity)

a.equals(b) === b.equals(a) (symmetry)

If a.equals(b) and b.equals(c), then a.equals(c) (transitivity)

```
const a = [1,2,-3,4,5]
const b = [1,2,3,4,-5]
const c = [-1,-2,-3,-4,-5]
assert(List(a).equals(List(a)) === true)
assert(List(a).equals(List(b)) === true)
assert(List(b).equals(List(c)) === true)
assert(List(a).equals(List(c)) === true)
```

Exercise time

Define a Setoid which works for colours

Colours can be one of

```
["red", "blue", "yellow"]
```

or a variation like:

```
["golden red", "apple red",
 "blue cyan", "magic yellow sunpower"]
```

```
Colour("red").equals(Colour("super red")) // true
Colour("super red").equals(Colour("red")) // true
Colour("red").equals(Colour("super blue")) // false
```

`"green"` is not a creative colour

```
const validColours = ["red", "blue", "yellow"]
const matchColour = (name) =>
  validColours.find((colour) =>
    name.indexOf(colour) !== -1)
// Colour :: a -> Colour a
const Colour = (a) => matchColour(a)
? ({
    __value: a,
    // equals :: Colour a -> Colour b -> Boolean
    equals: (b) =>
      matchColour(a) === matchColour(b.__value)
  })
: null

assert(Colour("red")
  .equals(Colour("pink red")) === true)
assert(Colour("red")
  .equals(Colour("creative blue")) === false)
```

Semigroup

a Semigroup is a container with a `.concat()`

```
// FSet :: a -> FSet a
const FSet = (a) => ({
  __value: a,
  // concat :: FSet -> FSet -> FSet
  concat: (b) => FSet(a.concat(b.__value))
})

assert.deepEqual(
  FSet([1]).concat(FSet([42])).__value,
  FSet([1,42]).__value)
```

Semigroup laws

a.concat(b).concat(c) === a.concat(b.concat(c)) (associativity)

```
assert.deepEqual(
  FSet([1,2]).concat(FSet([3])).concat(FSet([4])).__value
  FSet([1,2]).concat(FSet([3]).concat(FSet([4]))).__value
```

Monoid

a Semigroup is a container with a `.concat()`

a Monoid is a Semigroup with a `.empty()`

```
// FSet :: a -> FSet a
const FSet = (a) => ({
  __value: a,
  // concat :: FSet -> FSet -> FSet
  concat: (b) => FSet(a.concat(b.__value))
})
// empty :: () -> FSet
FSet.empty = () => FSet([])

assert.deepEqual(
  FSet([1]).concat(FSet([42])).__value,
  FSet([1,42]).__value)
assert.deepEqual(
  FSet([1]).concat(FSet.empty()).__value,
  FSet([1]).__value)
```

Monoid laws

m.concat(M.empty()) === m (right identity)

M.empty().concat(m) === m (left identity)

```
assert.deepEqual(
  FSet([1,2]).concat(FSet.empty()).__value,
  FSet.empty().concat(FSet([1,2])).__value)
```

Exercise time

Define a DotString Monoid which always add a dot between 2 strings every time we're concatenating them

```
DotString("lol").concat(DotString("asd")) // lol.asd
DotString("lol").concat(DotString.empty()) // lol
DotString.empty().concat(DotString("asd")) // asd
```

```
// DotString :: a -> DotString a
const DotString = (a) => ({
  __value: a,
  // concat :: DotString -> DotString -> DotString
  concat: (b) =>
    DotString(a + (a && b.__value ? '.' : '') + b.__value
})
// empty :: () -> DotString
DotString.empty = () => DotString('')

assert.equal(
  DotString("line").concat(DotString("line")).__value,
  "line.line")
assert.equal(
  DotString("line").concat(DotString.empty()).__value,
  "line")
```

Functor

a Functor is a container with a `.map()`

```
// List :: a -> List a
const List = (a) => ({
  __value: a,
  // map :: List a -> (a -> b) -> List b
  map: (fn) => List(a.map(fn))
})

assert.deepEqual(
  List([1,2]).map(a => a*2).__value,
  List([2,4]).__value)
```

a Functor is a container with a `.map()`

```
// Container :: a -> Container a
const Container = (a) => ({
  __value: a,
  // map :: Container a -> (a -> b) -> Container b
  map: (fn) => Container(fn(a))
})

assert(
  Container(777).map(a => a*2).__value,
  Container(1554).__value)
```

Functor laws

u.map(a => a) === u (identity)

u.map(x => f(g(x))) === u.map(g).map(f) (composition)

```
assert(
  Container(777).map(a => a).__value,
  Container(777).__value)
assert(
  Container(777).map(a => a*2*5).__value,
  Container(777).map(a => a*2).map(a => a*5).__value)
```

Exercise time

Can you use functors to abstract null checks?

```
const find = (predicate) => (list) =>
  list.find(predicate)

const accounts = [
  { owner: "pam", credit: 50 },
  { owner: "sam", debt: 10 },
]
const selector = (name) => ({ owner }) => owner === name
const pam = find(selector('pam'))(accounts)
if (pam && pam.credit) {
  console.log(100/pam.credit)
}
```

Maybe Functor

a Functor is a container with a `.map()`

```
// Maybe :: a -> Maybe a
const Maybe = (a) => ({
  __value: a,
  // map :: Maybe a -> (a -> b) -> Maybe b
  map: (fn) => a ? Maybe(fn(a)) : Maybe(null),
})

assert(Maybe(42).map(a => a*3).__value === 126)
assert(Maybe(42).map(a => a*3)
                .map(a => a-126)
                .map(a => a+10).__value === null)
assert(Maybe(0).map(a => a*3).__value === null)
```

```javascript
// find :: f -> [a] -> Maybe a
const find = (predicate) => (list) =>
  Maybe(list.find(predicate))

const accounts = [
  { owner: "pam", credit: 50 },
  { owner: "sam", debt: 10 },
]
const selector = (name) => ({ owner }) => owner === name
const pam = find(selector('pam'))(accounts)
  .map(({ credit }) => credit)
  .map(credit => 100/credit)
```

Applicative

a Functor is a container with a `.map()`

an Applicative is a Functor with `.ap()` and `.of()`

```
// Container :: a -> Container a
const Container = (a) => ({
  __value: a,
  // map :: Container a -> (a -> b) -> Container b
  map: (fn) => Container(fn(a)),
  // ap :: Container a -> Container (a->b) -> Container b
  ap: (fnAp) => Container(fnAp.__value(a))
})
Container.of = (a) => Container(a)

assert(
  Container("WAT").ap(Container(a => a+a)).__value,
  Container("WATWAT").__value)

assert(Container.of("WAT").__value,
       Container("WAT").__value)
```

Applicative laws

v.ap(u.ap(a.map(f=>g=>x=>f(g(x))))) == v.ap(u).ap(a) (composition)

v.ap(A.of(x => x)) === v (identity)

A.of(x).ap(A.of(f)) === A.of(f(x)) (homomorphism)

A.of(y).ap(u) === u.ap(A.of(f => f(y))) (interchange)

```
const v = Container('A')
const u = Container(a => a+a)
const a = Container(a => "A" + a)
assert.equal(
  v.ap(u.ap(a.map(f => g => x => f(g(x))))).__value,
  v.ap(u).ap(a).__value)
assert.equal(
  v.ap(Container.of(x => x)).__value,
  v.__value)
const x = 'A'
const f = a => a+a
assert.equal(
  Container.of(x).ap(Container.of(f).__value,
  Container.of(f(x))).__value)
assert.equal(
  Container.of(x).ap(a).__value,
  a.ap(Container.of(f => f(x))).__value)
```

what's the point of Applicatives?

```
// get :: a -> b (a -> c) -> Maybe c
const get = (property, object) =>
  Maybe(object[property])

const data = { cat: 'starlord' }
const transformers = { cat: (a) => a.toUpperCase() }

get('cat', data).map(get('cat', transformers)) // nup
get('cat', data).ap(get('cat', transformers)) // 'STARLORD
```

what's the point of Applicatives again?

a Functor is a container with a `.map()`

an Applicative is a Functor with `.ap()` and `.of()`

```
// List :: a -> List a
const List = (a) => ({
  __value: a,
  // map :: List a -> (a -> b) -> List b
  map: (fn) => List(a.map(fn)),
  // ap :: List a -> List (a -> b) -> List b
  ap: (fnAp) => List(a.reduce((acc, item) =>
    acc.concat(fnAp.__value.map(fn => fn(item))), []))
})
List.of = (a) => List(a)
```

Exercise time

Use the List applicative to generate a deck of cards.

A deck of cards has 52 cards and 4 seeds: ♠ ❤ ♦ ♣

There are 13 cards for each seed

```
// Generate cards
const concat = a => b => "" + b + a
const seedFns = List.of("♠❤♦♣".split('').map(concat))
const numbers = [1,2,3,4,5,6,7,8,9,10,11,12,13]
const cards = List.of(numbers).ap(seedFns)

//Haskell equivalent
(map (,) "♠❤♦♣") <*> [1..13]
```

more fun with Maybe

```
// find :: f -> [a] -> Maybe a
const find = (predicate) => (list) =>
  Maybe(list.find(predicate))
// get :: a -> b -> Maybe c
const get = (property) => (object) =>
  Maybe(object[property])

const accounts = [
  { owner: "pam", credit: 5000 },
  { owner: "sam", debt: 1000 },
]
const selector = (name) => ({ owner }) => owner === name
const pam = find(selector('pam'))(accounts)
        .map(get('credit'))
//{ __value: { __value: 5000 } } :(
const sam = find(selector('sam'))(accounts)
        .map(get('credit'))
//{ __value: { __value: undefined } } :(
```

Monad

a Functor is a container with a `.map()`

an Applicative is a Functor with `.of()` and `.ap()`

a Monad is an Applicative with `.chain()`

```
// Container :: a -> Container a
const Container = (a) => ({
  __value: a,
  // map :: Container a -> (a -> b) -> Container b
  map: (fn) => Container(fn(a)),
  // ap :: Container a -> Container (a -> b) -> Container
  ap: (fnAp) => Container(fnAp.__value(a)),
  // chain :: Container a -> (a -> Container b) -> Contai
  chain: (fnCh) => fnCh(a)  //equivalent to map(fnCh).__v
})
Container.of = (a) => Container(a)
```

Monad laws

m.chain(f).chain(g) === m.chain(x => f(x).chain(g)) (associativity)

M.of(a).chain(f) === f(a) (left identity)

m.chain(M.of) === m (right identity)

```
const a = 10
const m = Container(a)
const f = a => Container(a*2)
const g = a => Container(a*4)
assert.equal(
  m.chain(f).chain(g).__value,
  m.chain(x => f(x).chain(g)).__value)
assert.equal(
  Container.of(a).chain(f).__value,
  f(a).__value)
assert.equal(m.chain(Container.of).__value, m.__value)
```

MORE fun with Maybe

```
// find :: f -> [a] -> Maybe a
const find = (predicate) => (list) =>
  Maybe(list.find(predicate))
// get :: a -> b -> Maybe c
const get = (property) => (object) =>
  Maybe(object[property])

const accounts = [
  { owner: "pam", credit: 5000 },
  { owner: "sam", debt: 1000 },
]
const selector = (name) => ({ owner }) => owner === name
const pam = find(selector('pam'))(accounts)
        .chain(get('credit'))
//{ __value: 5000 } :)
const sam = find(selector('sam'))(accounts)
        .chain(get('credit'))
//{ __value: undefined } :)
```

Dad's message reminder

```
const phoneBook = [
  { name: "Pai Mei", number: '06-14'},
  { name: "Rai Mei", number: '09-11'},
  { name: "Lei Mei", number: '09-11'}
]
const familyBook = [{ name: 'Pai Mei', sons: ['Rai Mei',
```

Input: `name`

Output: if `name` has a phone number and if they have at least a son, send a message to `name` with their first son's number

```
// findPhone :: Name -> PhoneBook -> PhoneBookEntry
const findPhone = (target, source) =>
  source.find(({ name }) => name === target)

// findFirstSon :: Name -> FamilyBook -> FamilyEntry
const findFirstSon = (target, source) => {
  const r = source.find(({ name }) => name === target)
  return r ? r.sons[0] : null
}
const send = (son, dad) => true

const dadPhoneEntry = findPhone('Pai Mei', phoneBook)
if (dadPhoneEntry) {
  const son = findFirstSon(dadPhoneEntry.name, familyBook
  if (son) {
    const sonPhoneEntry = findPhone(son, phoneBook)
    if (sonPhoneEntry) {
      send(sonPhoneEntry.number, dadPhoneEntry.number)
    }
  }
}
```

Exercise time

Can you use monads to improve the code in the previous example?

Maybe Monad

`Applicative .of(), Functor .map(),`

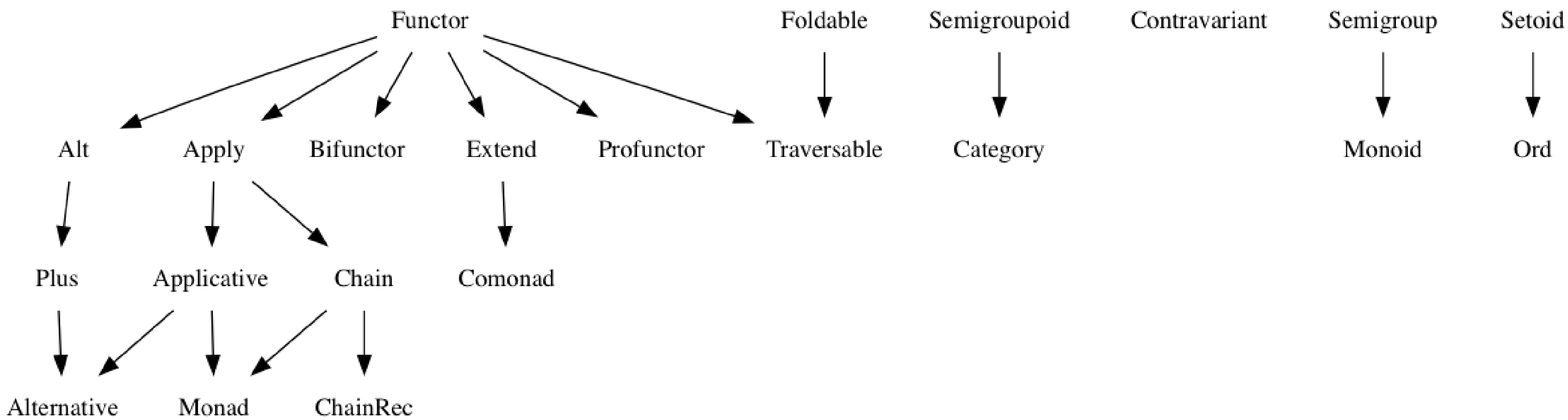
`Apply .ap(), Chain .chain()`

```
// Maybe :: a -> Maybe a
const Maybe = (a) => ({
  __value: a,
  // map :: Maybe a -> (a -> b) -> Maybe b
  map: (fn) => a ? Maybe(fn(a)) : Maybe(null),
  // ap :: Maybe a -> Maybe (a -> b) -> Maybe b
  ap: (fnAp) => Maybe(fnAp.__value(a)),
  // chain :: Maybe a -> (a -> Maybe b) -> Maybe b
  chain: (fnCh) => a ? fnCh(a) : Maybe(null)
})
Maybe.of = (a) => Maybe(a)
```

```
// findByName :: Name -> b c -> Maybe c
const findName = (target, source) =>
  Maybe(source.find(({ name }) => name === target))
// send :: PhoneNumber -> PhoneNumber -> Boolean
const send = dad => son => true

findByName('Pai Mei', phoneBook)
  .chain(dadPhoneEntry =>
    findByName(dadPhoneEntry.name, familyBook)
      .map(({ sons }) => sons[0])
      .chain(son => findByName(son, phoneBook))
      .map(sonPhoneEntry => sonPhoneEntry.number)
      .map(send(dadPhoneEntry.number)))
```

Fantasy Land Algebras Dependencies



Monadic Promises

are Promises Monads?

```
Promise.resolve(1).then(a => a+1) // 2
Promise.resolve(1).then(a => Promise.resolve(a+1)) // 2
```

They're behave differently, `then` behave like `chain` and `map` are Promises Monads?

```
new Promise((res, rej) => console.log('YOLO') || res())
//YOLO
```

They execute (with possible side effects) as soon as you define them

They're not pure

Folktale to the rescue

http://folktalegithubio.readthedocs.io/en/latest/index.html

data.task

```
const tasks = [
  Task.of(1).chain(a => Task.of(a+1)),
  Task.of(1).map(a => a+1),
  new Task((res, rej) => setTimeout(_ => res(42), 1000))
]
const log = (label) => console.log.bind(console, label)
tasks.map(task =>
  task.fork(log("ERR"), log("YUP"))
```

Task represents the intention of doing something

If you return a Task your function is still pure

You get side effects calling `.fork`

Final Exercise

Using GitHub's REST API (here and here, GraphQL is cheating), retrieve a list of repos of all the members of the organisation

Input: Github organisation, eg: 'tes'

Output: List of repos of all the members in the format

```
Name: [${name}](${link})
Author: ${author}
Description: ${description} (if not available drop the li
Stars: ${stars}
```

sorted by `stars` and `name`

Notes:

Try to use Task, other Folktale structures or your own structures to abstract tedious tasks and isolate side effects.

Fin.

slides on framp.me/fp-workshop/2