framp.me

understanding Monads

disclaimer

we're going to:

1. gloss on some implementation details
2. keep it in layman's terms
3. refer to Fantasy Land Specification (kinda)

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()`

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

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

Semigroup

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

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

assert.deepEqual(
  List([1]).concat(List([42])).value,
  List([1,42]).value))
```

Monoid

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

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

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

assert.deepEqual(
  List([1]).concat(List.empty()).value,
  List([1]).value)
```

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()`

```javascript
// 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)
```

Maybe

Maybe Functor

`Functor .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(43).map(a => a*3)
                .map(a => a-126)
                .map(a => a+10).value === 13)
assert(Maybe(null).map(a => a*3).value === null)
```

Apply / Applicative

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

an Apply is a Functor with `.ap()`

```
// 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))
})

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

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

an Apply is a Functor with `.ap()`

an Applicative is an Apply with `.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
  ap: (fnAp) => Container(fnAp.value(a))
})
Container.of = (a) => Container(a)

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

what's the point of Applicatives?

```
// get :: a -> b -> 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)) // 'STARLORI
```

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 } } :(
```

Chain

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

an Apply is a Functor with `.ap()`

a Chain is an Apply with `.chain()`

```javascript
// 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 -> Chain b) -> Chain b
  chain: (fnCh) => fnCh(a)  // better: map(a).value
})

assert(Container("WAT")
  .chain(a => Container("B" + a))
  .chain(a => Container(a + "C")).value,
  "BWATC")
```

Monad

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

an Apply is a Functor with `.ap()`

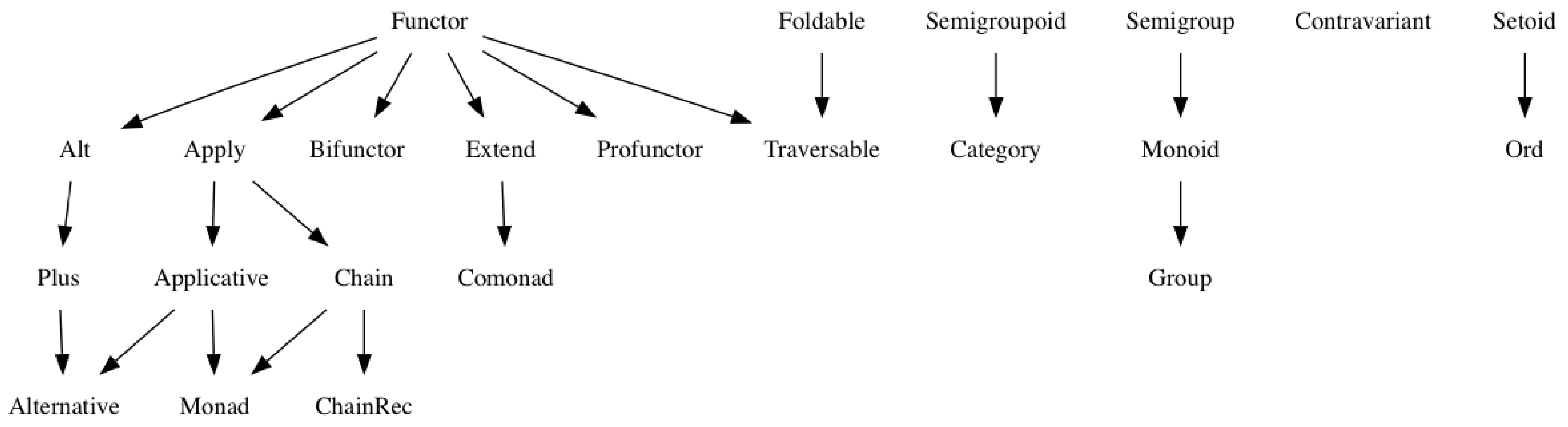
an Applicative is an Apply with `.of()`

a Chain is an Apply with `.chain()`

a Monad is a Chain and an Applicative

```
// 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 -> Chain b) -> Chain b
  chain: (fnCh) => fnCh(a)  // better: map(a).value
})
Container.of = (a) => Container(a)
//Definitely a monad
```

Fantasy Land Algebras Dependencies



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) => fnCh(a)  // or map(a).value
})
Maybe.of = (a) => Maybe(a)
```

more 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 } :)
```

Monadic Promises

are Promises Monads?

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

are Promises Monads?

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

kind of, but not really

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"))
```

http://folktalegithubio.readthedocs.io/en/latest/api/data/task/Task.html#data.task.Task

Fin.

slides on framp.me/monads