ALGEBRAIC DATA TYPES

by Giorgi Bagdavadze / @notgiorgi

PRODUCT TYPES

```
class Pair<U, V> {
    constructor(
        x: U,
        y: V,
    ) { /* ... */ }
}
```

```
data Pair u v = Pair u v
```

```
class Person {
  constructor(
    name: String,
    age: Number,
    address: String,
  ) { /* ... */ }
}

class Box<T> {
  constructor(private value: T) { }
  /* ... */
}
```

```
data Person = Person
  { name    :: String
  , age     :: Int
  , address :: String
  }

data Box a = Box a
```

INTIUTION OF HASKELL

```
class Pair<U, V> {
    constructor(x: U, y: V) { }
}

let pair = new Pair<boolean, string>(false, "Foo")
```

```
data Pair u v = Pair u v
--   ^ Type      ^ Constructor

-- Type
pair :: Pair Bool String
-- Value
pair = Pair False "Foo"
```

WHY ALGEBRAIC? WHY PRODUCT?

SYMMETRICAL

a * b == b * a

```
Pair<U,V> ~~ Pair<V, U>
Pair<boolean, string> ~~ Pair<string, boolean>
```

```
Pair u v ~~ Pair v u
Pair Bool String ~~ Pair String Bool
```

ISOMORPHISM

```
to   :: A -> B
from :: B -> A
```

```
to(from(x)) == x == from(to(x))
```

```
function swap<U, V>(pair: Pair<U, V>): Pair<V, U> {
  return new Pair<V, U>(pair.y, pair.x)
}
```

ASSOCIATIVE

`a * (b * c) == (a * b) * c == a * b * c`

```
Pair<Pair<U, V>, W>
~~
Pair<U, Pair<V, W>>
~~
Triplet<U, V, W>
```

```
Pair u (Pair v w) ~~ Pair (u Bool v) w
```

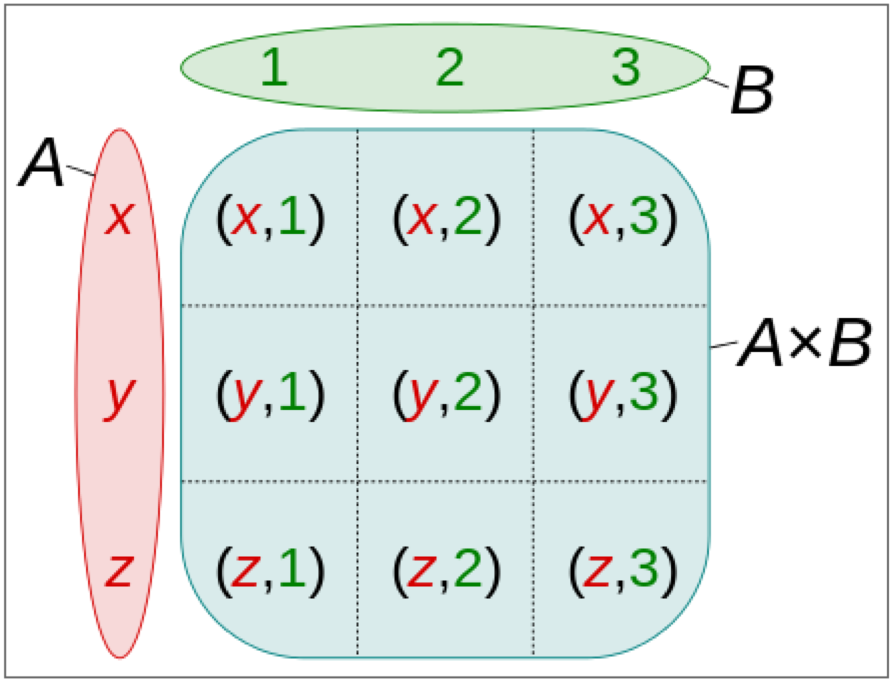
IDENTITY

```
a * 1 == a == 1 * a
```

```
Pair<T, Unit> ~~ T
```

```
Pair T Unit ~~ Bool
```

CARTESIAN PRODUCT



(x,1)	(x,2)	(x,3)
(y,1)	(y,2)	(y,3)
(z,1)	(z,2)	(z,3)

SUM (UNION) TYPES

```
enum PromiseState = {
  Pending,
  Fulfilled,
  Rejected,
}
```

```
data PromiseState = Pending | Fulfilled | Rejected
```

MORE THAN JUST ENUMS

```
interface Optional<T> {/* ... */}
class Some<T> implements Optional<T> {/* .. */}
class None<T> implements Optional<T> {/* ... */}

interface List<T> {/* ... */}
class Nil<T> implements List<T> {/* ... */}
class Cons<T> implements List<T> {/* ... */}
```

```
data Optional a = Some a | None

data List a = Nil | Cons a (List a)

data Promise u v
  = Pending (Array Listener)
  | Rejected (Array Listener) u
  | Fulfilled (Array Listener) v
```

SYMMETRICAL

```
a + b == b + a
```

```
Result<Response, string> ~~ Result<string, Response>
```

```
Result Response String ~~ Result String Response
```

ASSOCIATIVE

```
(a + b) + c == a + (b + c)
```

```
Result<Result<DBError, Response>, number>
~~
Result<DBError, Result<Response, number>>
```

```
Result (Result<DBError Response) Int
~~
Result<DBError (Result Response Int)
```

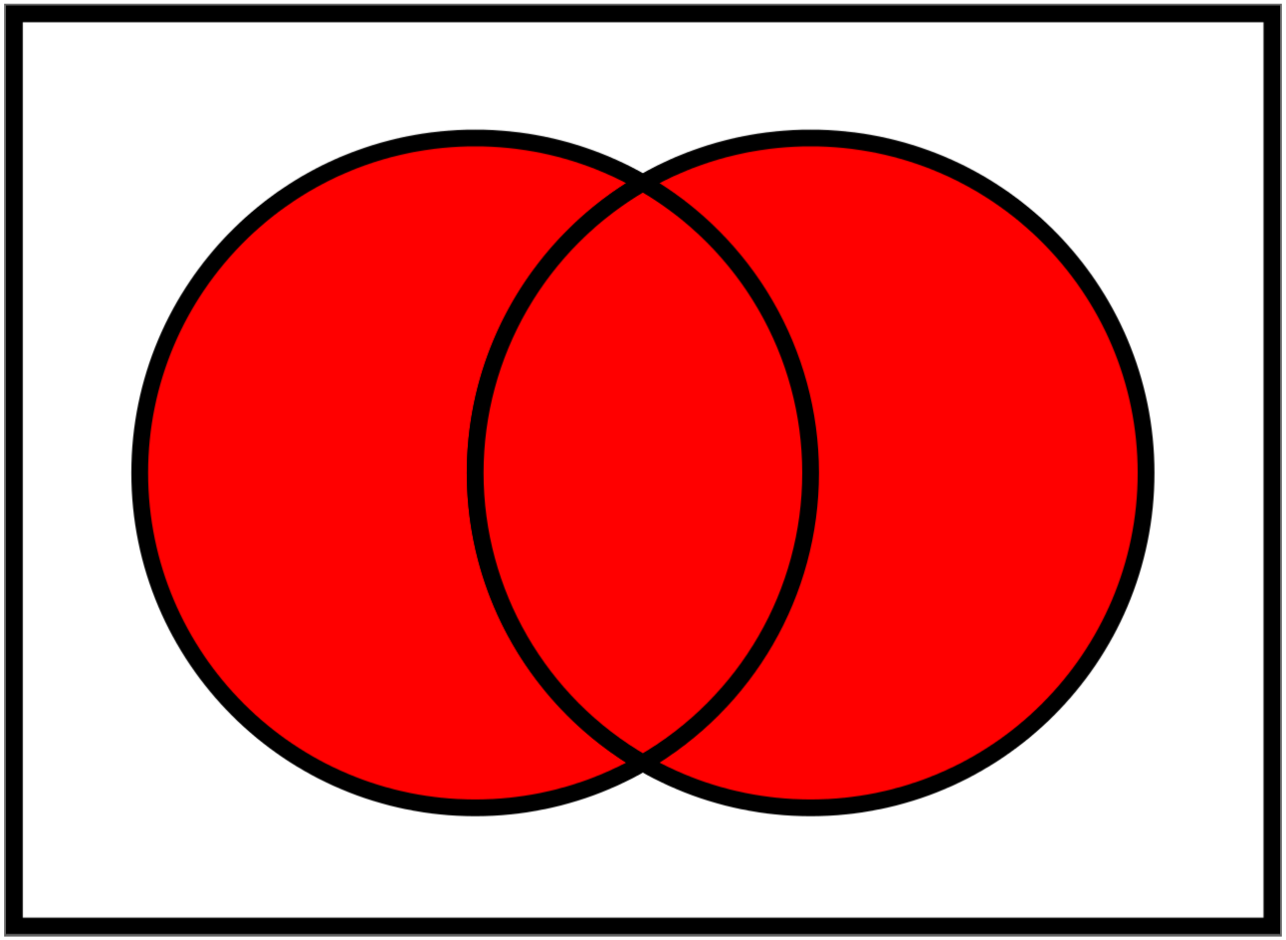
IDENTITY

$a + 0 = a = 0 + a$

```
class Foo { /* ... */ }

class Void {
  private Void() {/* ... */}
}

Result<Foo, Void> ~~ Foo
```

```
Foo a | Void ~~ Foo a
```

UNION



```
class Pending<U,V> implements Promise<U, V> {
  constructor(
    private listeners: Array<Listener>
  ) { }
}

class Rejected<U, V> implements Promise<U, V> {
  constructor(
    private listeners: Array<Listener>,
    err: U
  ) { }
}

class Fulfilled<U, V> implements Promise<U, V> {
  constructor(
    private listeners: Array<Listener>,
    value: V
  ) { }
}
```

```
data Promise u v
  = Pending (Array Listener)
  | Rejected (Array Listener) u
  | Fulfilled (Array Listener) v
```

DISTRIBUTIVE PROPERTY

```
a * x + c * x == x * (a + b)
```

```
type PromiseState<U, V> = Pending | Rejected<U> | Fulfilled<V>

class Promise<U, V> {
  constructor(
    private listeners: Array<Listener>,
    private state: PromiseState<U, V>
  )
}
```

```
data PromiseState a b
    = Pending
    | Rejected a
    | Fulfilled b

data Promise a b = Promise (Array Listener) (PromiseState a b)
```

EXPOTENTIAL (FUNCTION) TYPE

$a \rightarrow b \sim\sim b^a$

```
function f(p: PromiseState): Boolean {/* ... */}
```

```
f :: PromiseState → Bool
```

$a^b * a^c == a^{b + c}$

```
interface PromiseStates<U, V> {
    /* ... */
    then<W>(
      onFulfill: (value: U) => W,
      onReject: (error: V) => W
    ): W

    // ~~

    then<W>(
      handler: (valueOrError: Result<U, V>) => W
    ): W
}
```

```
Pair (b -> a) (c -> a) ~~ Result b c -> a
```

$$c^{(a * b)} == (c^{a})^{b}$$

```
function f(a: U, b: V): W
~~
function f(a: U): ((b: V) ⇒ W)
```

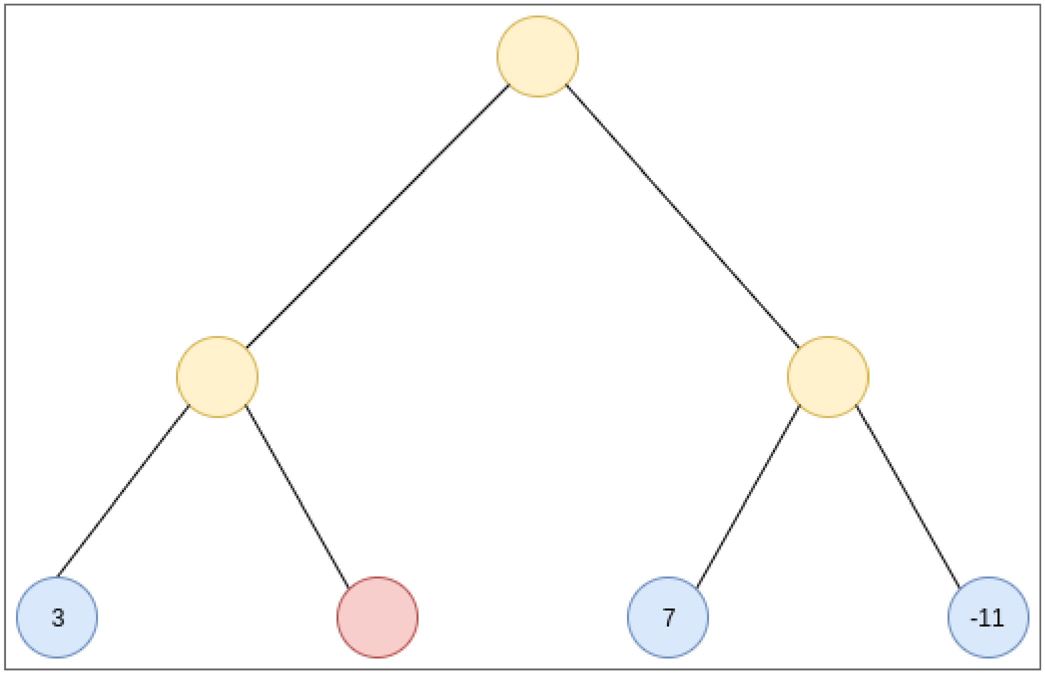
```
f :: Pair u v → w
~~
f :: u → v → w
```

FLAGS

```
type YesNo<T> = Yes<T> | No<T>

Pair<string, boolean> ~~ YesNo<string>

new Pair("barrab", true) ~~ new Yes("barrab")
new Pair("foo", false) ~~ new No("foo")
```

```
data YesNo a = Yes a | No a

Pair String Bool ~~ YesNo String
```

DECLARE WITH INVARIANTS!



NAIVE APPROACH

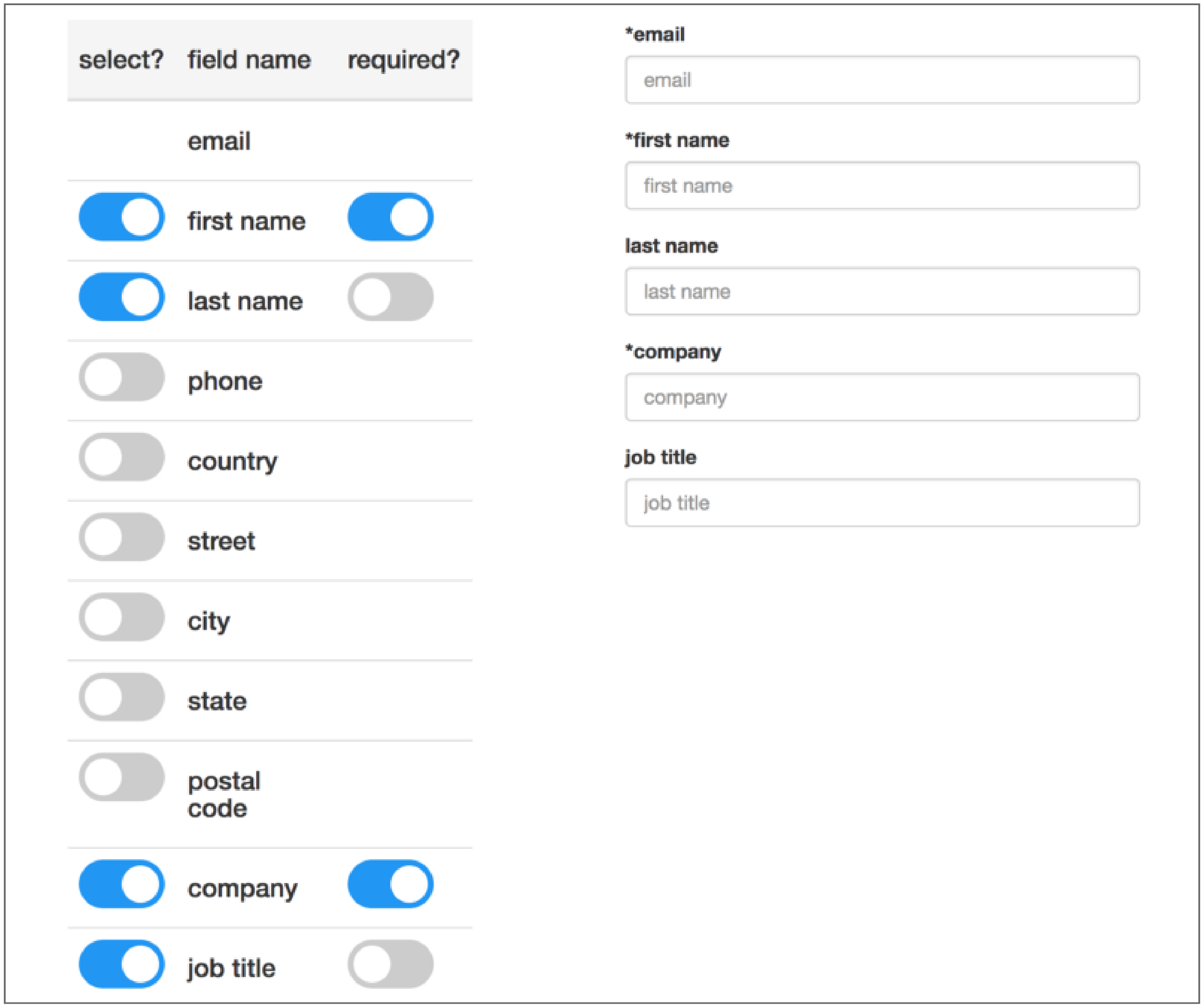
```
class Node<T> {
  constructor(
    left: Node,
    value: T,
    right: Node,
  ) { }
}
```

```
data Node a = Node (Optional (Node a)) (Optional a) (Optional (Node a))

depth :: Node a -> Int
depth (Node left value right) =
  case left of
    None   -> ...
    Some a ->
      case right of
        None -> ...
        ...
```

DECLARE WITH INVARIANTS!

```
type Tree<T> = Empty<T> | Leaf<T> | Node<T>
```

```
data Tree a
  = Empty
  | Leaf a
  | Node Tree Tree

depth :: Tree -> Int
depth Empty = 0
depth (Leaf n) = 1
depth (Node l r) = 1 + max (depth l) (depth r)
```



select?	field name	required?
	email	
●	first name	●
●	last name	○
○	phone	
○	country	
○	street	
○	city	
○	state	
○	postal code	
●	company	●
●	job title	○

***email**

email

***first name**

first name

last name

last name

***company**

company

job title

job title

```haskell
data Field = Field String SelectionStatus

data SelectionStatus
    = Sealed
    | Unselected
    | Selected FieldRequirement

data FieldRequirement
    = Required
    | Optional
```

- No invalid states
- You won't forget edge cases
- Compiler is your friend

RECURSIVE TYPES

```
L(a) = 1 + (a * L(a))
     = 1 + a * (1 + a * L(a))
     = 1 + a + (a * a * L(a))
     = ...
```

```
interface List<T> {/* ... */}
class Nil<T> implements List<T> {/*  ...  */}
class Cons<T> implements List<T> {/*  ...  */}

let l: List<Number> =
  new Cons(3,
    new Cons(4,
      new Nil()
    )
  )
// 3 -> 4 -> Nil
```

```
--            1   +      1 * L(x)
data List a = Nil | Cons a (List a)
```

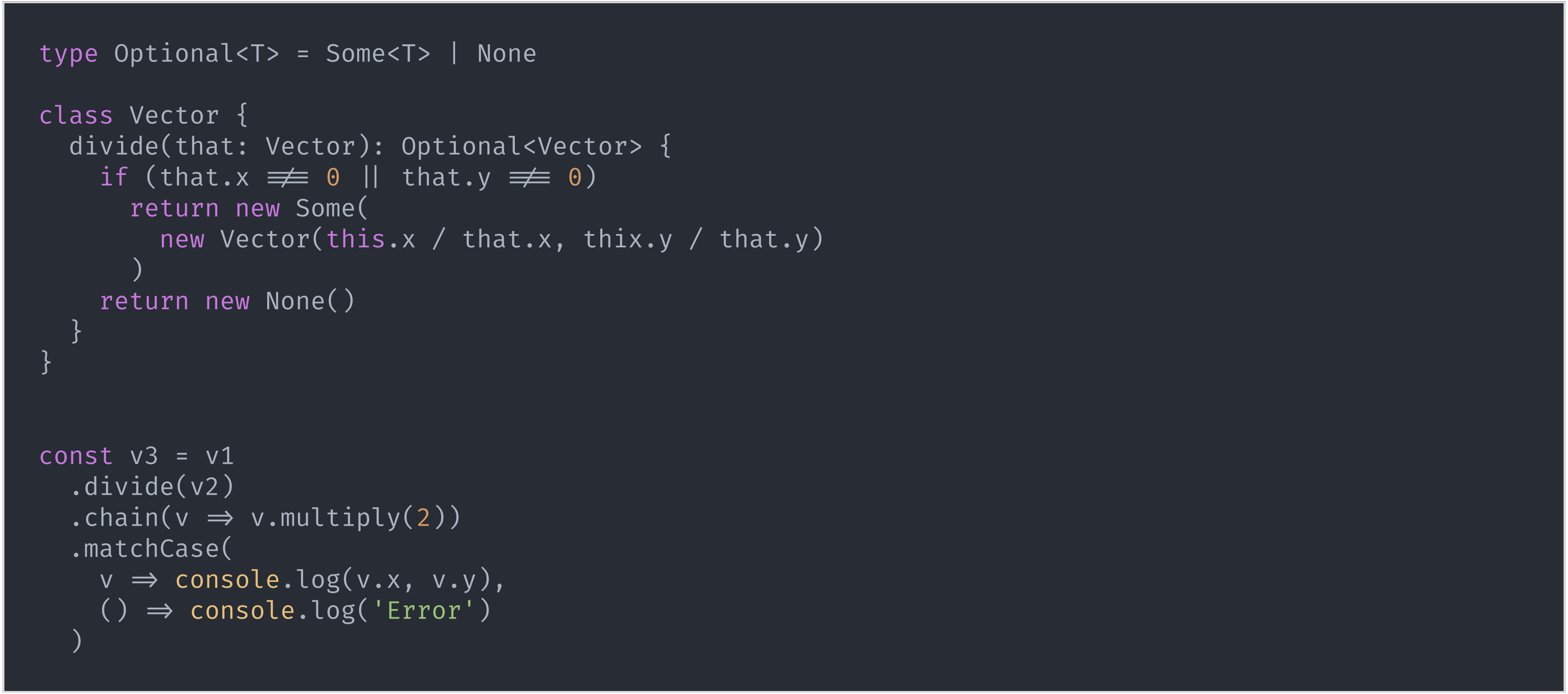
IF (X != NULL) - BAD

```
class Vector {
  /* ... */
  divide(that: Vector): Vector {
    if (that.x !== 0 || that.y !== 0)
      return new Vector(this.x / that.x, thix.y / that.y)
    return null
  }
}

/* ... */
const v3 = v1.divide(v2)
if (v3 !== null) {
  v3.add(/* ... */)
}
```

BETTER

```
interface Vector {/* ... */}

class FullVector implements Vector{
  divide(that: Vector): Vector {
    if (that.x !== 0 || that.y !== 0)
      return new FullVector(this.x / that.x, thix.y / that.y)
    return new NullVector()
  }
  render() {/* ... */}
}

class NullVector implements Vector {
  divide(that: Vector): Vector { return this }
  render() {/* ... */}
}

const v3 = v1
  .divide(v2)
  .add(v5)
  .render()
```

THE BEST



```
type Optional<T> = Some<T> | None

class Vector {
  divide(that: Vector): Optional<Vector> {
    if (that.x !== 0 || that.y !== 0)
      return new Some(
        new Vector(this.x / that.x, thix.y / that.y)
      )
    return new None()
  }
}


const v3 = v1
  .divide(v2)
  .chain(v => v.multiply(2))
  .matchCase(
    v => console.log(v.x, v.y),
    () => console.log('Error')
  )
```

THANKS!

¿Questions?