Introducción a la programación funcional en JavaScript

Hola!

Soy Ricardo Pallás

https://github.com/RPallas92/congreso-web-2016

Índice

1. Introducción (primeros ejercicios y conceptos)
2. Trabajando con arrays sin bucles
3. Teoría de Categorías
4. Ejercicio final (App web vídeos: React + FP)
5. Conclusiones

1.

Introducción

Primeros ejercicios
y conceptos:
Funciones puras

¿Qué es la programación funcional?

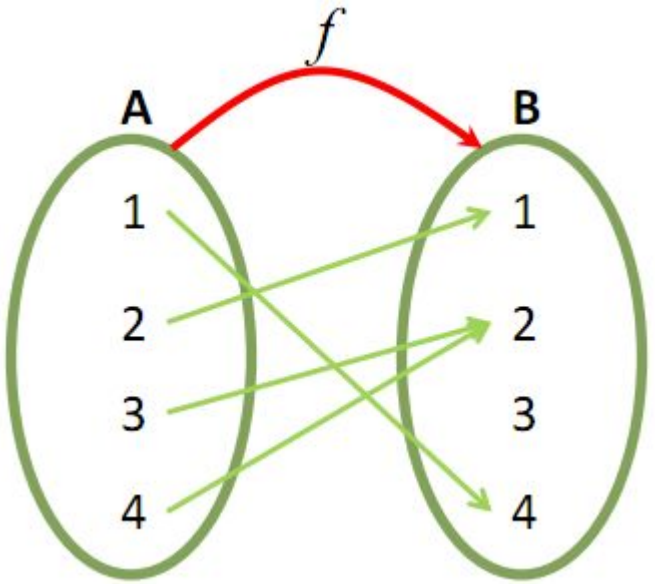
La programación funcional es un paradigma de programación que se basa en el uso de funciones modeladas como funciones matemáticas.

La esencia es que los programas son la combinación de **expresiones**. Una expresión puede ser un valor concreto, variables y también **funciones.**

¿Qué es una función?

Una función es una relación entre un conjunto de posibles entradas y un conjunto de posibles salidas.

La función en sí misma, representa la relación.



Gaviotas

Unir: unir dos bandadas de gaviotas en una sola

Engendrar: la bandada se multiplica por el número de gaviotas que engendra cada una

```
var Bandada = function(n) { this.gaviotas = n }

Bandada.prototype.unir = function(otra) {
  this.gaviotas += otra.gaviotas;
  return this;
}

Bandada.prototype.engendrar = function(otra) {
  this.gaviotas = this.gaviotas * otra.gaviotas;
  return this;
}

var bandada1 = new Bandada(4);
var bandada2 = new Bandada(2);
var bandada3 = new Bandada(0);

var resultado = bandada1.unir(bandada3)
      .engendrar(bandada2).unir(bandada1.engendrar(bandada2)).gaviotas;
//=> 32
```

```
var unir = function(bandada_x, bandada_y) { return bandada_x + bandada_y; };
var engendrar = function(bandada_x, bandada_y) { return bandada_x * bandada_y; };

var bandada_a = 4;
var bandada_b = 2;
var bandada_c = 0;

var result = unir(
  engendrar(bandada_b, unir(bandada_a, bandada_c)), engendrar(bandada_a, bandada_b)
);
//=>16
```

```
var sumar = function(x, y) { return x + y; };
var multiplicar = function(x, y) { return x * y; };

var bandada_a = 4;
var bandada_b = 2;
var bandada_c = 0;

var result = sumar(
  multiplicar(bandada_b, sumar(bandada_a, bandada_c)), multiplicar(bandada_a, bandada_b)
);
//=>16
```

```
// associative
sumar(sumar(x, y), z) === sumar(x, sumar(y, z));

// commutative
sumar(x, y) === sumar(y, x);

// identity
sumar(x, 0) === x;

// distributive
multiplicar(x, sumar(y,z)) === sumar(multiplicar(x, y), multiplicar(x, z));
```

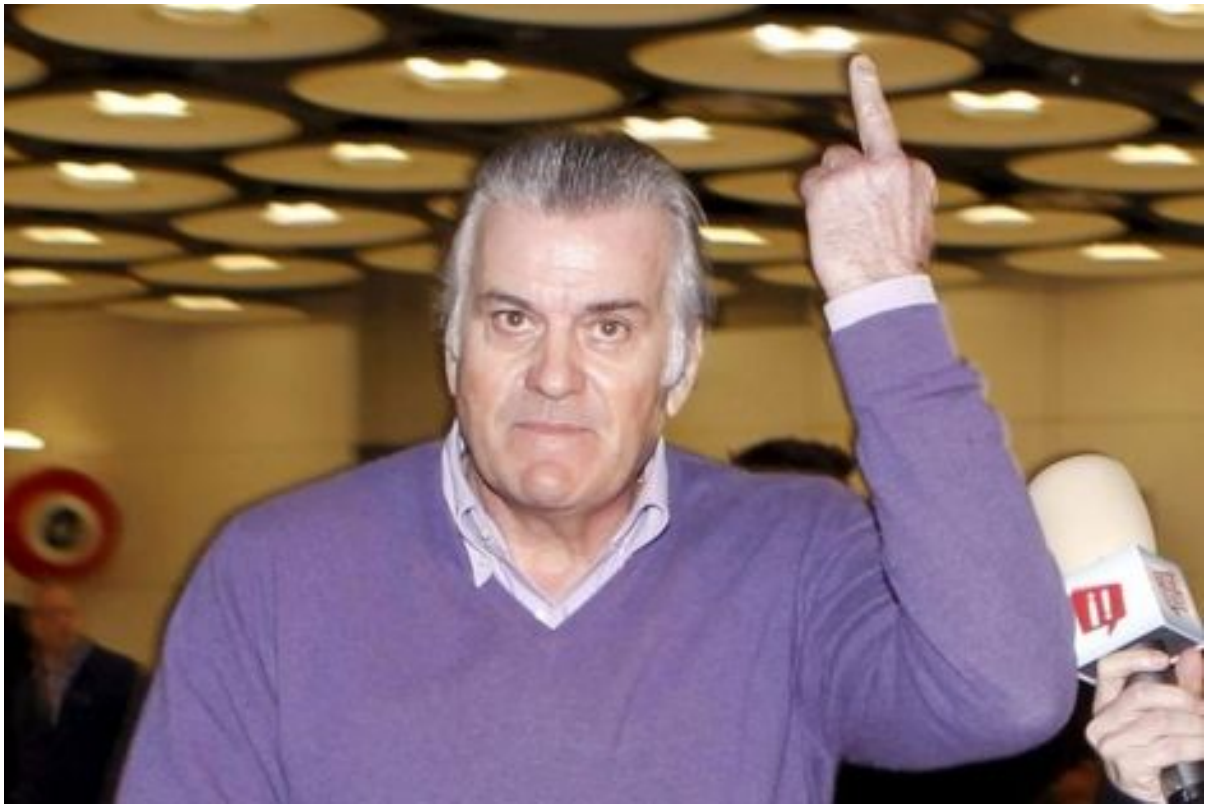
Funciones puras

Una función pura es una función que, dada la misma entrada, siempre devuelve la misma salida y no tiene ningún efecto colateral.

¿Qué es la programación funcional, otra vez?

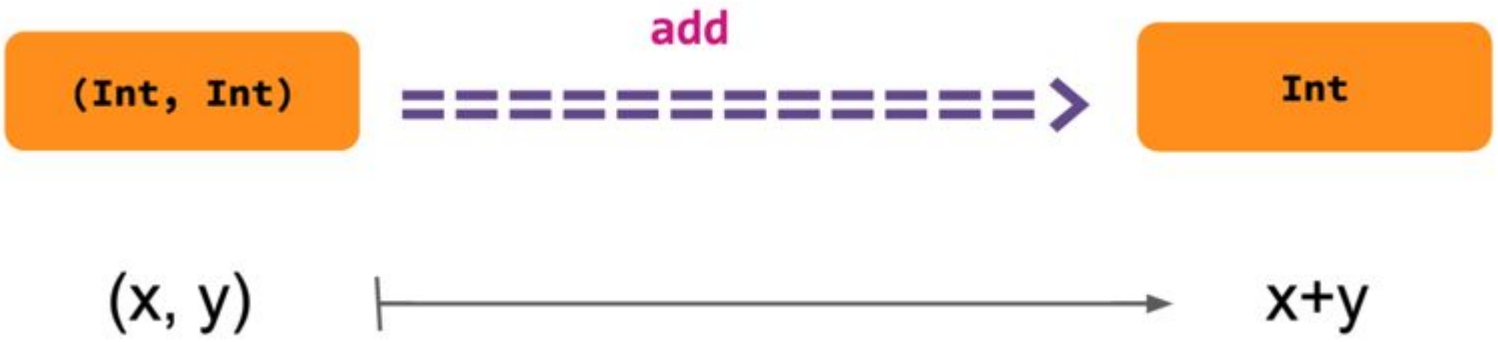
En gran medida, programas con **funciones puras**: funciones que reciben *valores* de entrada, devuelve valores de salida, y **no hacen de más.**

El problema: los efectos colaterales



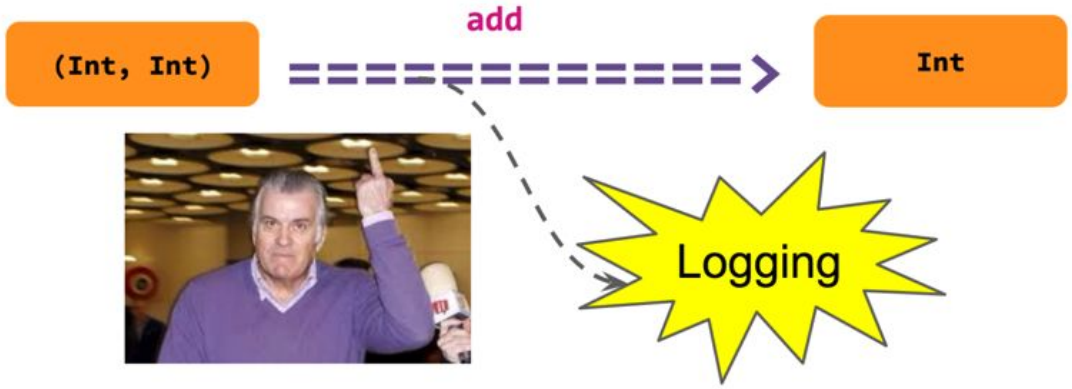
Una función pura: libre de efectos colaterales

```
function add(x,y) {
  return x + y;
}
```



Una función impura: corrupta (no declara todo lo que hace realmente)

```
function add(x,y) {
  var z = x + y;
  console.log('result of the sum: ', z);
  return z;
}
```



```
var xs = [1,2,3,4,5]
```

```
// impure
xs.splice(0,3)
//=> [1,2,3]

xs.splice(0,3)
//=> [4,5]

xs.splice(0,3)
//=> []
```

```
// pure
xs.slice(0,3)
//=> [1,2,3]

xs.slice(0,3)
//=> [1,2,3]

xs.slice(0,3)
//=> [1,2,3]
```

```
// impure
var minimum = 21;

var checkAge = function(age) {
  return age >= minimum;
}
```

```
// pure
var checkAge = function(age) {
  var minimum = 21;
  return age >= minimum;
}
```

```
// impure
var greeting = function(name) {
  console.log("hi, " + name + "!")
}
```

```
// pure
var greeting = function(name) {
  return "hi, " + name + "!";
}

console.log(greeting("Jonas"))
```

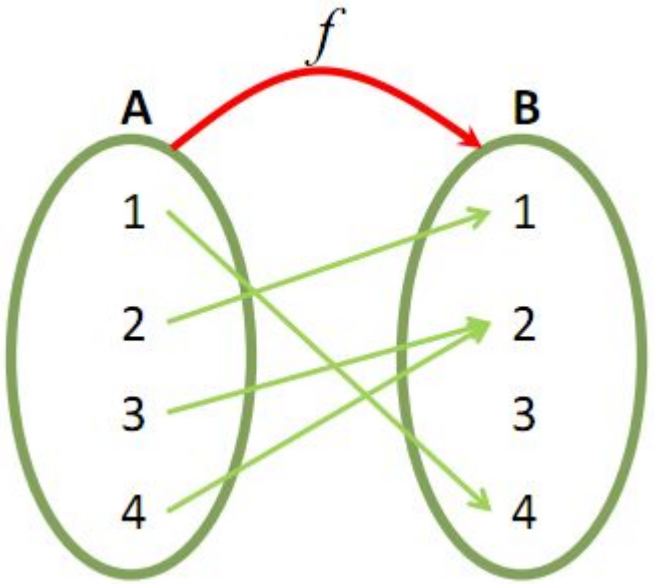
```
//impure
var signUp = function(attrs) {
  var user = saveUser(attrs)
  welcomeUser(user)
}
```

```
//pure
var signUp = function(Db, Email, attrs) {
  return saveUser(Db, attrs).chain(function(user){
    return welcomeUser(Email, user)
  })
}
```

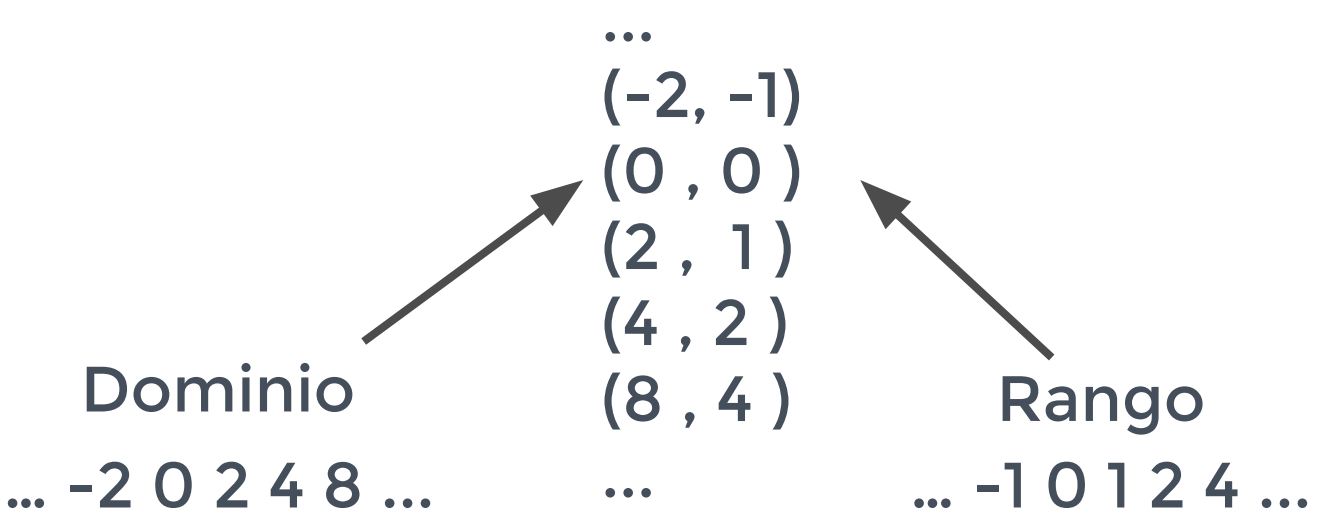
Recordatorio

Una función es una relación entre un conjunto de posibles entradas y un conjunto de posibles salidas.

La función en sí misma, representa la relación.



Una entrada, una salida



Una entrada, una salida

```
var toLowerCase = {"A":"a", "B": "b", "C": "c", "D": "d", "E": "e", "D": "d"}

toLowerCase["C"]
//=> "c"
```

```
var isPrime = {1: false, 2: true, 3: true, 4: false, 5: true, 6: false}

isPrime[3]
//=> true
```

¿Y si hay múltiples argumentos?

```
//+ add :: [Number, Number] -> Number
var add = function(numbers) {
  return numbers[0] + numbers[1]
}
```

¿Y si hay múltiples argumentos?

```
//+ add :: ? -> Number
var add = function() {
  return arguments[0] + arguments[1]
}
```

Currying

Función *"curried":*

"Una función que devuelve una nueva función hasta que recibe todos sus argumentos"

Currying

```
//+ add :: Number -> Number -> Number
var add = curry(function(x, y) {
  return x + y
})
```

Currying

```
add
//=> function(x,y) { return x + y }

add(2,3)
//=> 5

add(2)
//=> function(y) { return 2 + y }
```

Currying

```
//+ get :: String -> {String: a} -> a
var get = curry(function(prop, obj) {
  return obj[prop];
})

var user = {id: 15, name: "Jorge Aznar", email: "jaznar@email.com"}

get('email', user)
//=> jaznar@email.com

//+ email :: {String: a} -> a
var email = get('email')
//=> function(obj){ return obj[‘email’] }

email(user)
//=> jaznar@email.com
```

Currying

```
//+ filter :: (a -> Bool) -> [a] -> [a]
var filter = curry(function(f, xs) {
  return xs.filter(f);
})

//+ odds :: [a] -> [a]
var odds = filter(isOdd)


odds([1,2,3])
//=> [1, 3]
```

Currying

```
var emails = map(email)
var users = [jesus, jorge, ja]

emails(users)
//=> ["jesus@lopez.org", "jorge@aznar.net", "ja@samitier.com"]
```

Currying

```
//+ goodArticles :: [Article] -> [Article]
var goodArticles = function(articles) {
  return _.filter(articles, function(article){
    return _.isDefined(article)
  })
}


//+ goodArticles :: [Article] -> [Article]
var goodArticles = filter(isDefined)
```

Currying

```
//+ getChildren :: DOM -> [DOM]
var getChildren = function(el) {
  return el.childNodes
}
//+ getALLChildren :: [DOM] -> [[DOM]]
var getAllChildren = function(els) {
  return _.map(els, function(el) {
    return getChildren(el)
  })
}

var getChildren = get('childNodes')
var getAllChildren = map(getChildren)
```

Composición

La composición de funciones es aplicar una función a los resultados de otra

Composición

```
//+ compose :: (b -> c) -> (a -> b) -> a -> c
var compose = curry(function(f, g, x) {
  return f(g(x))
})
```

Composición

```
//+ head :: [a] -> a
var head = function(x) { return x[0] }

//+ last :: [a] -> a
var last = compose(head, reverse)

last(['Java', 'JavaScript', 'PHP'])
//=> PHP
```

Composición

```
//+ wordCount :: String -> Number
var wordCount = function(sentence) {
  var count = split(' ', sentence)
  return length(count)
}


//+ wordCount :: String -> Number
var wordCount = compose(length, split(' '))


wordCount("Soy una frase de 6 palabras")
//=> 6
```

Composición

```
'B' <- 'b' <- 'Congreso Web'

compose(toUpperCase, last)('Congreso Web')
```

Composición

```
// Ley asociativa
compose(f, compose(g, h)) == compose(compose(f, g), h)
```

Composición

```
compose(toUpperCase, compose(head, reverse))

// ó

compose(compose(toUpperCase, head), reverse)
```

Composición

```
//+ lastUpper :: [String] -> String
var lastUpper = compose(toUpperCase, head, reverse)

lastUpper(['Java', 'PHP', 'JavaScript'])
//=> 'JAVASCRIPT'

//+ lastUpper :: [String] -> String
var loudLastUpper = compose(exclaim, toUpperCase, head,
reverse)

loudLastUpper(['Java', 'PHP', 'JAVASCRIPT'])
//=> 'JAVASCRIPT!'
```

Composición

```
? <- map(['JavaScript', 'Rust']) <- ['JavaScript', 'Rust'] <- ['Rust', 'JavaScript']
compose(toUpperCase, map, reverse)(['Rust', 'JavaScript'])
```

Composición

```
? <- map(['JavaScript', 'Rust']) <- ['JavaScript', 'Rust'] <- ['Rust', 'JavaScript']
compose(toUpperCase, map, reverse)(['Rust', 'JavaScript'])
```

Composición

```
? <- map(['JavaScript', 'Rust']) <- ['JavaScript', 'Rust'] <- ['Rust', 'JavaScript']
compose(toUpperCase, map, reverse)(['Rust', 'JavaScript'])
```

Composición

```
['JAVASCRIPT', 'RUST'] <- ['JavaScript', 'Rust'] <- ['Rust', 'JavaScript']
compose(map(toUpperCase), reverse)(['Rust', 'JavaScript'])
```

2.

Trabajando con arrays

Sin bucles, con funciones puras

Inciso: Trabajar con colecciones (sin bucles)

La mayoría de las operaciones que hacemos se pueden conseguir con 5 funciones:

- map
- filter
- reduce
- concatAll
- zip

Prácticamente, todos los bucles se pueden sustituir por estas funciones.

Ejercicios

Vamos a practicar el uso de las 5 funciones

https://github.com/RPallas92/congreso-web-2016

Directorio parte2

3.

Teoría de Categorías

Primeros ejercicios y conceptos: Funciones puras

Teoría de Categorías

Es una rama abstracta de las matemáticas que formaliza conceptos de diferentes ramas como teoría de conjuntos, teoría de tipos, teoría de grupos, lógica y más.

Principalmente, se trabaja con objetos, morfismos y transformaciones. (Similitud con la programación)

Categoría

Es una colección con los siguientes componentes:

- Una colección de objetos
- Una colección de morfismos
- Una noción de composición de los morfismos
- Un morfismo en concreto, llamado identidad

Categoría

- La Teoría de Categorías es suficientemente abstracta para modelar muchas cosas, pero vamos a centrarnos en **tipos** y **funciones**

Categoría

Una **colección de objetos:**

- Los objetos serán tipos de datos. String, Boolean, Number, Object.
- **Boolean** como el conjunto [true, false]
- **Number** como el conjunto de todos los posibles valores numéricos

Categoría

Una **colección de morfismos:**

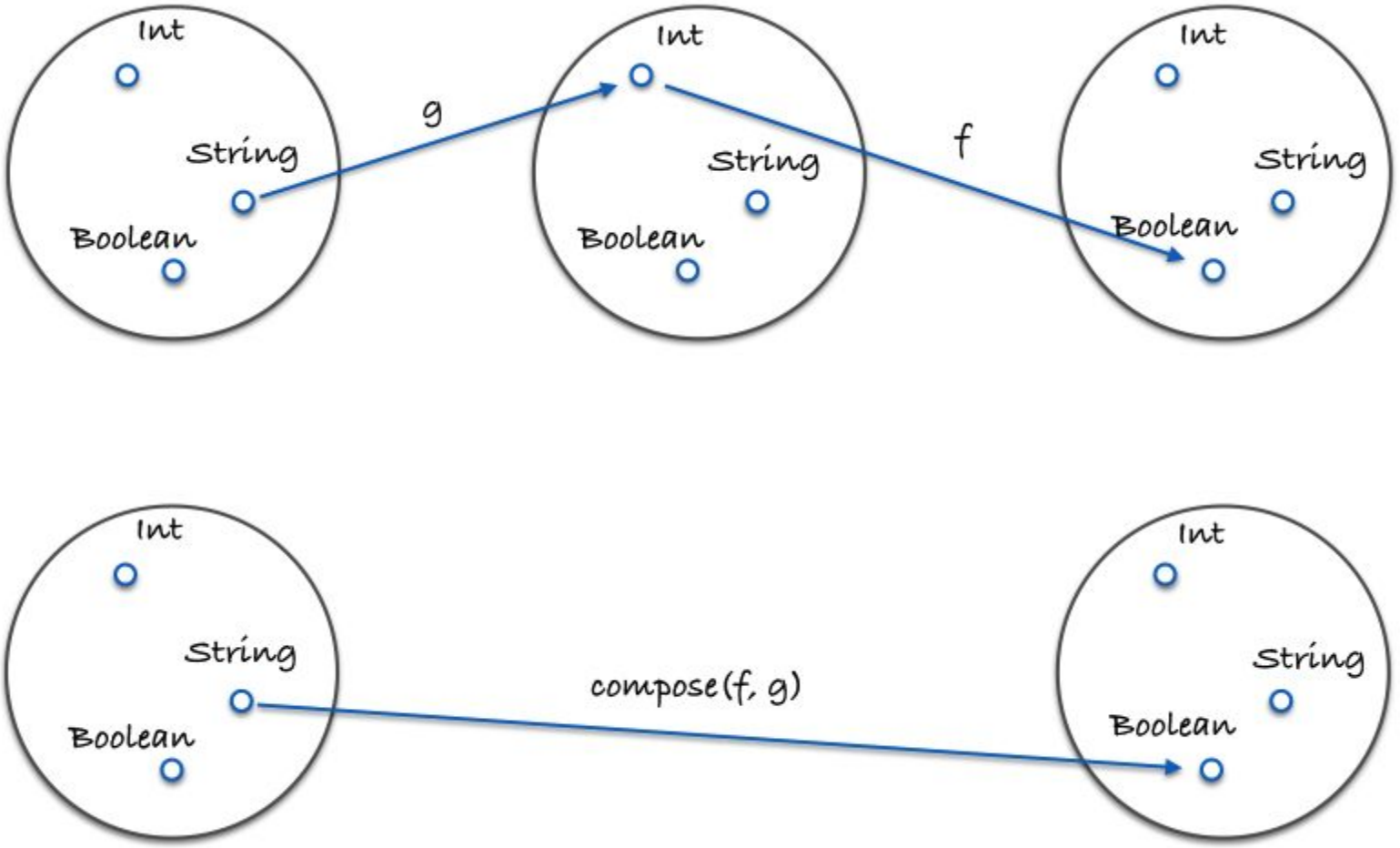
- Los morfismos serán las funciones normales que escribimos diariamente.

Categoría

Una **noción de composición de los morfismos:**

- Nuestra función **compose**, presentada anteriormente
- Que **compose** cumpla la ley asociativa no es casualidad.
- Cualquier composición en la Teoría de Categorías debe cumplirla

Categoría: Compsición



Categoría

Un **morfismo concreto llamado identidad:**

- Nueva función **id**
- Recibe una entrada, y la devuelve, nada más

```
var id = function(x) {
  return x;
};
```

Almacena un valor, será utilizada más adelante

Identidad

```
// identity
compose(id, f) == compose(f, id) == f;
// true
```

- Esta propiedad se cumple para **todas** las funciones **unarias**.
- Es como la identidad en los **números**

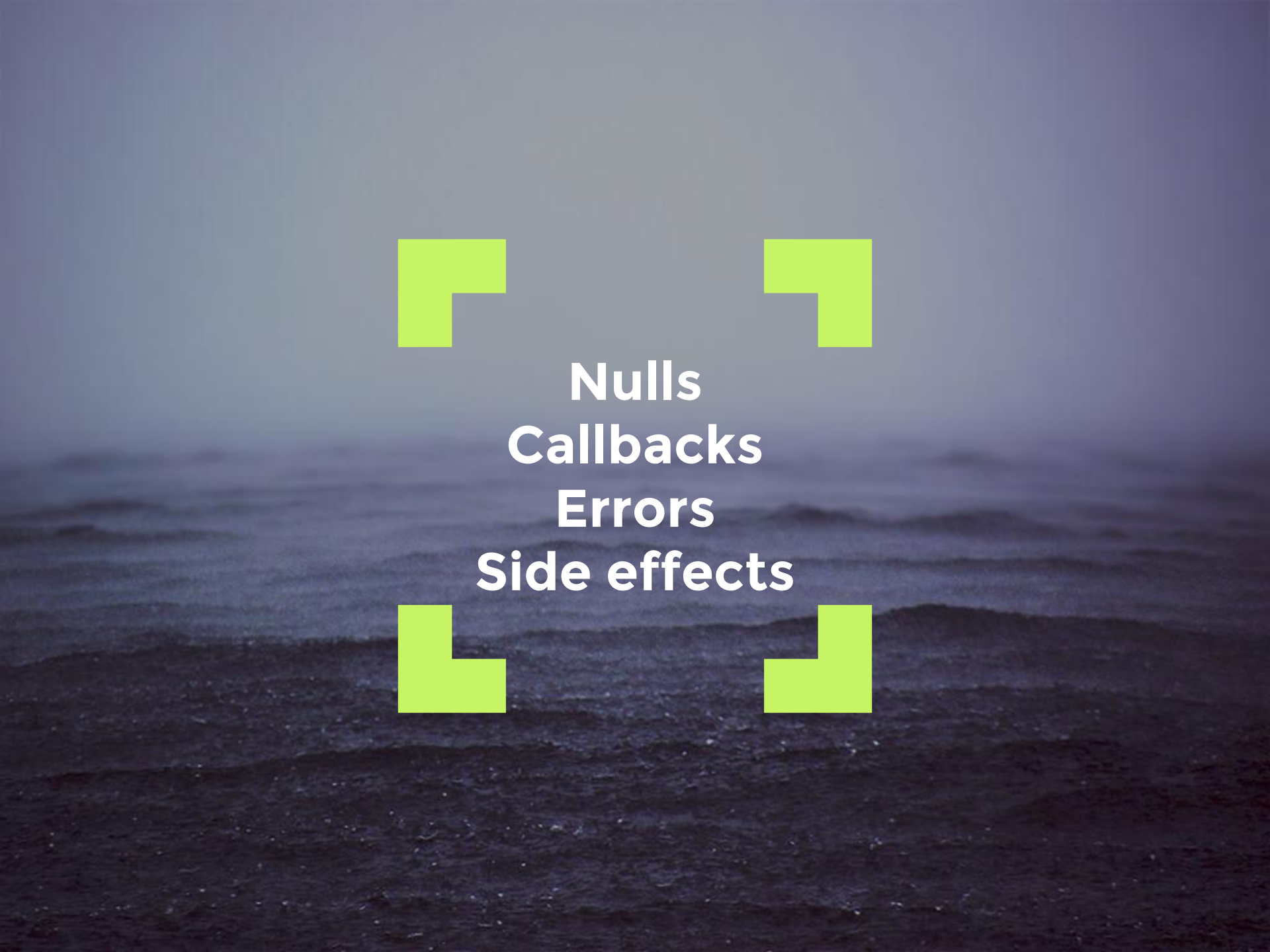
Identidad

```
// identity
compose(id, f) == compose(f, id) == f;
// true
```

- Esta propiedad se cumple para **todas** las funciones **unarias**.
- Es como la identidad en los **números**

Ejercicio

Application de ejemplo
https://github.com/RPallas92/congreso-web-2016
Directorio parte3


Nulls
Callbacks
Errors
Side effects

Objetos

- Contenedores/Wrappers para valores
- No tienen métodos
- No tienen propiedades
- Seguramente, no crees los tuyos propios

Objetos

```
var Container = function(val) {
  this.val = val;
}

//+ Container :: a -> Container(a)
Container.of = function(x) {
   return new Container(x);
};

Container.of(3)
//=> Container {val: 3}
```

Objetos

```
capitalize("flamethrower")
//=> "Flamethrower"


capitalize(Container.of("flamethrower"))
//=> [object Object]
```

Objetos

```
//+ map :: (a ->  b) -> Container(a) -> Container(b)
Container.prototype.map = function(f) {
  return Container.of(f(this.val));
}

Container.of("flamethrower").map(function(s){
    return capitalize(s)
})

//=> Container(“Flamethrower”)
```

Objetos

```
//+ map :: (a ->  b) -> Container(a) -> Container(b)
Container.prototype.map = function(f) {
  return Container.of(f(this.val));
}

Container.of("flamethrower").map(capitalize)

//=> Container(“Flamethrower”)
```

Objetos

```
Container.of(3).map(add(1))
//=> Container(4)


[3].map(add(1))
//=> [4]
```

Objetos

```
Container.of([1,2,3]).map(reverse).map(first)
//=> Container(3)


Container.of("flamethrower").map(length).map(add(1))
//=> Container(13)
```

Objetos

```
//+ map :: Functor F => (a ->  b) -> F a -> F b
var map = _.curry(function(f, obj) {
  return obj.map(f)
})

Container.of(3).map(add(1)) // Container(4)

map(add(1), Container.of(3)) // Container(4)
```

Objetos

```
map(match(/cat/g), Container.of("catsup"))
//=> Container([“cat”])

map(compose(first, reverse),Container.of("dog"))
//=> Container(“g”)
```

FUNCTOR

"Un objeto o estructura de datos que puede ser mapeada"

- Funciones: **map**

Aquellos problemáticos null

```
//+ getElement :: String -> DOM
var getElement = document.querySelector
//+ getNameParts :: String -> [String]
var getNameParts = compose(split(' '), getText, getElement)

getNameParts('#full_name')
//=> ['José', 'Luis', 'Bárcenas', 'Gutiérrez']
```

Aquellos problemáticos null

```
//+ getElement :: String -> DOM
var getElement = document.querySelector
//+ getNameParts :: String -> [String]
var getNameParts = compose(split(' '), getText, getElement)

getNameParts('#fullname')
//=> BOOOOOOOOMM!!!!!!
```

Maybe

- Captura un **null check**
- Es posible que el valor no esté
- A veces tienen dos subclases: **Just/Nothing**
- En otros lenguajes se llama **Option**

Maybe

```
//+ map :: (a -> b) -> Maybe(a) -> Maybe(b)
var _Maybe.prototype.map = function(f) {
  return this.val ? Maybe.of(f(this.val)) : Maybe.of(null);
}

map(capitalize, Maybe.of("flamethrower"))
//=> Maybe(“Flamethrower”)
```

Maybe

```
//+ map :: (a -> b) -> Maybe(a) -> Maybe(b)
var _Maybe.prototype.map = function(f) {
  return this.val ? Maybe.of(f(this.val)) : Maybe.of(null);
}

map(capitalize, Maybe.of(null))
//=> Maybe(null)
```

Maybe

```
//+ firstMatch :: String -> String
var firstMatch = compose(first, match(/java/g))



firstMatch("python")
//=> BOOOM!!!
```

Maybe

```
//+ firstMatch :: String -> Maybe(String)
var firstMatch = compose(map(first), Maybe.of, match(/java/g))


firstMatch("python")
//=> Maybe(null)
```

Maybe

```
//+ firstMatch :: String -> Maybe(String)
var firstMatch = compose(map(first), Maybe.of, match(/java/g))


firstMatch("javascript")
//=> Maybe(“java”)
```

Either

- Utilizado para manejo de excepciones **pura**
- Es como Maybe, pero con un mensaje de error embebido
- Tienen dos subclases: **Left/Right**
- Mapea la función sobre el **Right**, ignora el **Left**

Either

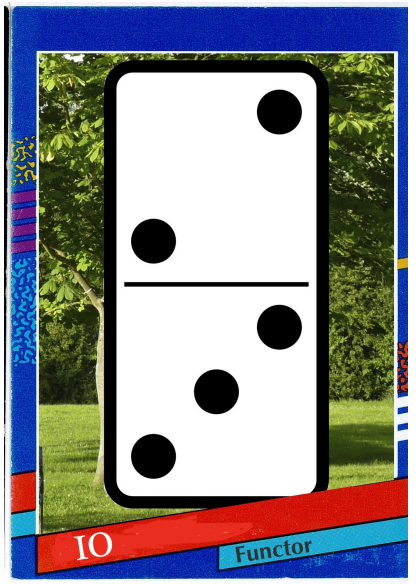
```
map(function(x) { return x + 1; }, Right.of(2))
//=> Right(3)

map(function(x) { return x + 1; }, Left.of('some message'))
//=> Left('some message')
```

Either

```
//+ determineAge :: User -> Either(String, Number)
var determineAge = function(user){
  return user.age ? Right.of(user.age) : Left.of("couldn't get
age");
}
//+ yearOlder :: User -> Either(String, Number)
var yearOlder = compose(map(add(1)), determineAge)

yearOlder({age: 22})
//=> Right(23)



var jordiHurtado = {age: null}

yearOlder(jordiHurtado)
//=> Left("couldn't get age")
```

IO

- Utilizado para ***lazy computation***
- Utilizado para manejar los efectos colaterales
- Para ejectuar la operación, se debe llamar a ***runIO***
- **map** añade la función a una lista de operaciones a ejecutar



IO

```
//+ email_io :: IO(String)
var email_io = new IO(function(){ return $("#email").val() })
//+ msg_io :: IO(String)
var msg_io = map(concat("Bienvenido "), email_io)

msg_io.unsafePerformIO()
//=> ”Bienvenido jbalbas@email.net”
```

unsafePerfomIO()

IO

```
//+ getBgColor :: JSON -> Color
var getBgColor = compose(get("background-color"), JSON.parse)
//+ bgPref :: _ -> IO(Color)
var bgPref = compose(map(getBgColor), Store.get("preferences"))

//+ app :: _ -> IO(Color)
var app = bgPref()
//=> IO()

app.unsafePerformIO()
//=> #efefef
```

IO

```
//+ email_io :: _ -> IO(String)
var email_io = new IO(function(){ return $("#email").val() }

//+ getValue :: String -> IO(String)
var getValue = function(sel){
    return new IO(function(){ return $(sel).val() });
}
```

Task / Future

- Se utiliza para operaciones **asíncronas**
- Representa un valor en el **tiempo**
- Para ejecutar la operación, se debe llamar a ***fork***
- Fork recibe una función como valor
- Se llama a la función con el resultado del Task una vez que lo tiene

Task / Future

```
//+ makeHtml :: Post -> HTML
var makeHtml = function(post){ return "<div>"+post.title+"</div>"};
//+ page_f :: Future(Html)
var page_f = map(makeHtml, http.get('/posts/2'))

page_f.fork(
    function(err) { throw(err) },
    function(page){ $('#container').html(page) }
)
```

Pointed Functors

- Es un Functor con un método **of**
- Of :: a -> F a
- Permite poner valores en un contexto mínimo que retiene el valor (o contexto puro)
- También llamado **pure**

Pointed Functors

- Es un Functor con un método **of**
- Of :: a -> F a
- Permite poner valores en un contexto mínimo que retiene el valor (o contexto puro)
- También llamado **pure**

Pointed Functors

```
Container.of(2)
// Container(2)
```

```
Future.of(match(/dubstep/))
// Future(match(/dubstep/))
```

```
Maybe.of(reverse)
// Maybe(reverse)
```

```
IO.of('hack')
// IO(function(){ return 'hack'})
```

Task / Future

```
//+ lineCount :: String -> Number
var lineCount = compose(length, split(/\n/))

//+ fileLineCount :: String -> Future(Number)
var fileLineCount = compose(map(lineCount), readFile)

fileLineCount("mydoc.txt").fork(log, log)
//=> 34
```

Leyes de los functores

```
// identity
map(id) == id

// composition
compose(map(f), map(g)) == map(compose(f, g))
```

Juego al vuelo

Haz una llamada a una api con una id, y obtén un post (puede existir o no)

Juego al vuelo

Future(Maybe(Post))

Mónadas

- Patrón de diseño utilizado para **describir computaciones** como series de pasos (o computaciones anidadas)
- Utilizadas para gestionar los efectos colaterales
- O para controlar la complejidad

Funciones:

- join
- chain

Mónadas

- join :: M M a -> M a
- chain :: (a -> M b) -> M a -> M b

Una mónada es un Pointed Functor añadiendo los métodos join y chain

Mónadas

```
join(Container.of(Container.of(2)))

//=> Container(2)
```

Mónadas

```
//+ getTrackingId :: Order -> Maybe(TrackingId)
var getTrackingId = compose(Maybe.of, get("tracking_id"))

//+ findOrder :: Number -> Maybe(Order)
var findOrder = compose(Maybe.of, Api.findOrder)

//+ getOrderTracking :: Number -> Maybe(Maybe(TrackingId))
var getOrderTracking = compose(map(getTrackingId),  findOrder)

//+ renderPage :: Number -> Maybe(Maybe(Html))
var renderPage = compose(map(map(renderTemplate)), getOrderTracking)
```

Mónadas

```
//+ getTrackingId :: Order -> Maybe(TrackingId)
var getTrackingId = compose(Maybe.of, get("tracking_id"))

//+ findOrder :: Number -> Maybe(Order)
var findOrder = compose(Maybe.of, Api.findOrder)

//+ getOrderTracking :: Number -> Maybe(TrackingId)
var getOrderTracking = compose(join, map(getTrackingId),  findOrder)


//+ renderPage :: Number -> Maybe(Html)
var renderPage = compose(map(renderTemplate), getOrderTracking)
```

Mónadas

```
//+ :: setSearchInput :: String -> IO(_)
var setSearchInput = function(x) { return new IO(function() { ("#input").val(x); }) }

//+ :: searchTerm :: IO(String)
var searchTerm = new IO(function() {  getParam("term", location.search) })

//+ :: initSearchForm :: IO(IO(_))
var initSearchForm = map(setSearchInput, searchTerm)


initSearchForm.unsafePerformIO().unsafePerformIO();
```

Mónadas

```
//+ :: setSearchInput :: String -> IO(_)
var setSearchInput = function(x) { return new IO(function() { ("#input").val(x); }) }

//+ :: searchTerm :: IO(String)
var searchTerm = new IO(function() {  getParam("term", location.search) })

//+ :: initSearchForm :: IO(IO(_))
var initSearchForm = map(setSearchInput, searchTerm)


join(initSearchForm).unsafePerformIO();
```

Mónadas

```
//+ sendToServer :: Params -> Future(Response)
var sendToServer = httpGet('/upload')

//+ uploadFromFile :: String -> Future(Response)
var uploadFromFile = compose(join, map(sendToServer),  readFile)

uploadFromFile("/tmp/my_file.txt").fork(logErr, alertSuccess)
```

Mónadas

```
//+ sendToServer :: Params -> Future(Response)
var sendToServer = httpGet('/upload')

//+ uploadFromFile :: String -> Future(Response)
var uploadFromFile = compose(join, map(sendToServer), join, map(readFile), askUser)

uploadFromFile('what file?').fork(logErr, alertSuccess)
```

Mónadas

```
//+ chain :: (a -> M b) -> M a -> M b
var chain = function(f) {
  return compose(join, map(f))
}

// también llamado flatMap
```

Mónadas

```
//+ sendToServer :: Params -> Future(Response)
var sendToServer = httpGet('/upload')

//+ uploadFromFile :: String -> Future(Response)
var uploadFromFile = compose(chain(sendToServer), chain(readFile), askUser)

uploadFromFile('what file?').fork(logErr, alertSuccess)
```

Functores aplicativos

- Ejecutar computaciones completas en un contexto
- Permite aplicar un functor a otro

Funciones:

- ap
- liftA2
- liftA3
- lifA..N

Functores aplicativos

```
add(Container.of(2), Container.of(3));

//NaN
```

```
var container_of_add_2 = map(add, Container.of(2));

// Container(add(2))
```

Functores aplicativos

```
map(add(1), Container(2))
//=> Container(3)


add(Container.of(2), Container.of(3));
//NaN


map(add, Container(2))
//=> Container(add(2))
```

Functores aplicativos

- **ap :: A (a -> b) -> A a -> A b**

- Un functor aplicativo es un Pointed Functor con un método **ap**

Functores aplicativos

```
Container.of(f).ap(Container(x))

//=> Container(f(x))
```

```
Container.of(f).ap(Container(x)).ap(Container(y))

//=> Container(f(x, y))
```

Functores aplicativos

```
Container.of(add).ap(Container(1)).ap(Container(3))

//=> Container(4)
```

Functores aplicativos

```
Maybe.of(add).ap(Maybe(1)).ap(Maybe(3))
//=> Maybe(4)

Maybe.of(add).ap(Maybe(1)).ap(Maybe(null))
//=> Maybe(null)
```

Functores aplicativos

```
var loadPage = _.curry(function(products, reviews){
    render(products.zip(reviews)
})

Future.of(loadPage)
   .ap(Api.get('/products'))
   .ap(Api.get('/reviews'))
```

Functores aplicativos

```
var getVal = compose(Maybe, get('value'), document.querySelector)
var save = _.curry(function(email, pass){ return User(email, pass) })

Maybe.of(save).ap(getVal('#email')).ap(getVal('#password'))
//=> Maybe(user)
```

Functores aplicativos

```
var getVal = compose(Maybe, pluck('value'), document.querySelector)
var save = _.curry(function(email, pass){ return User(email, pass) })

liftA2(save, getVal('#email'), getVal('#password'))
//=> Maybe(user)
```

Fantasy Land

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

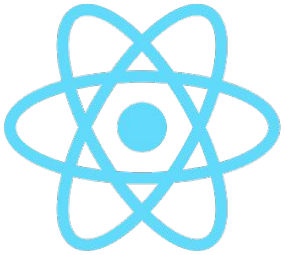
4.

Ejercicio final

Aplicación de ejemplo

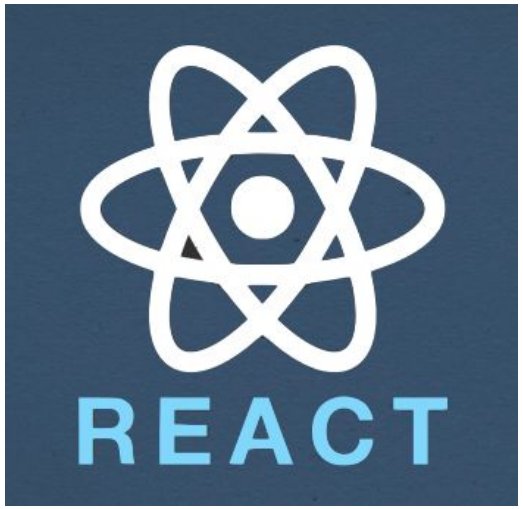
Aplicación buscador de vídeos Youtube

- Vista: **React**
- Manipulación de datos: **Ramda**
- Maybe: **folktale/data.maybe**
- Either: **folktale/data.either**
- Task: **folktale/data.task**



Folktale
JAVASCRIPT MEETS DRY

React



- Biblioteca para crear interfaces de usuario en JavaScript
- **Simple**
- **Declarativa**
- **Modular**: componentes reusables
- Utiliza la **composición** de los componentes para crear nuevos

https://facebook.github.io/react

Ramda

- Biblioteca para programación funcional *práctica* en JavaScript
- Similar a lodash o undescore
- Todas las funciones están **curried**
- Parámetros de las funciones ordenados para favorecer el curry
- Trabaja bien con implementaciones Mónadas, functores, etc.

http://ramdajs.com

Folktale

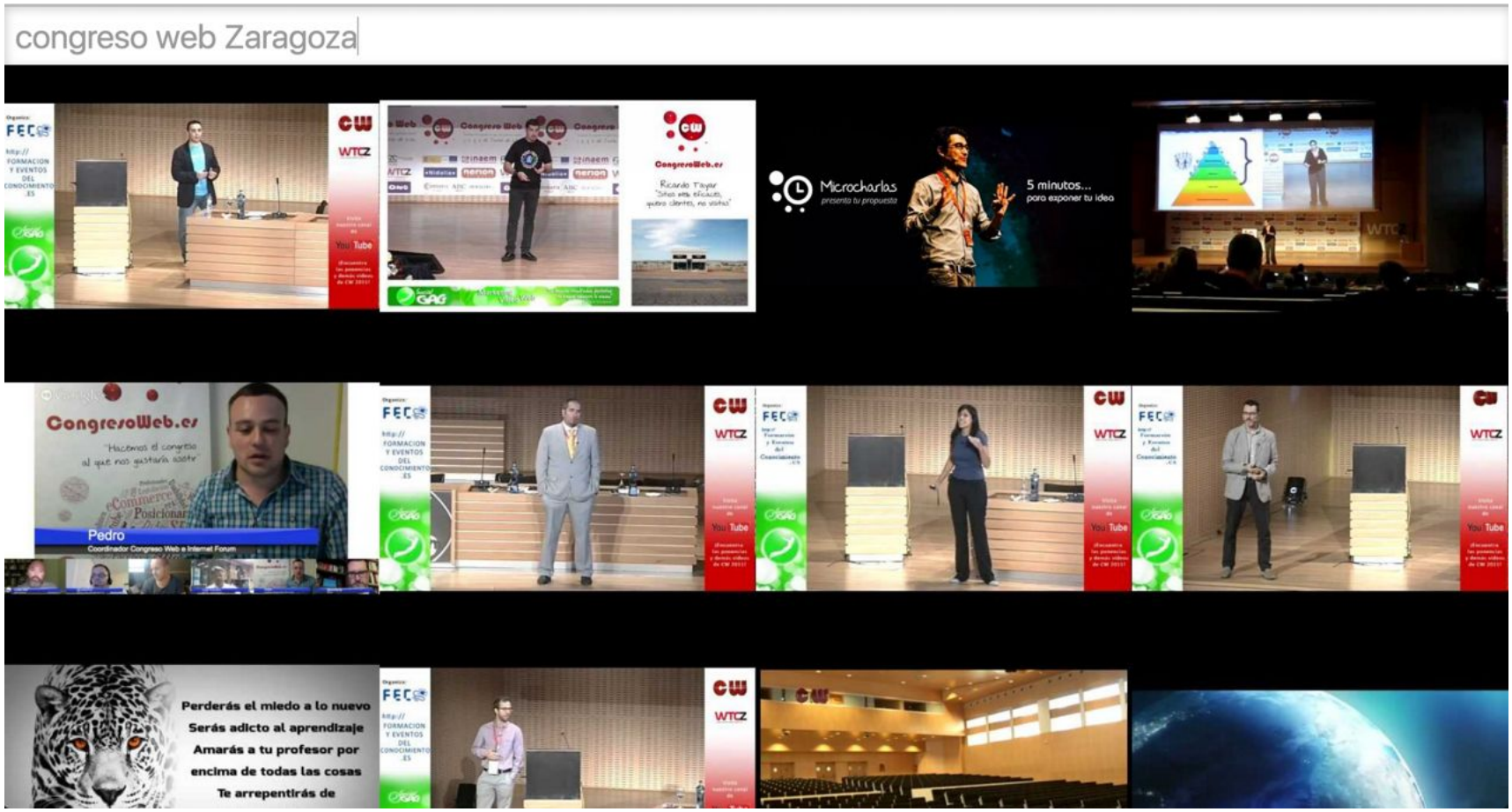
- Suite de bibliotecas para programación funcional genérica en JavaScript
- Utilizaremos su implementación de
  - **Maybe**
  - **Either**
  - **Task (Future)**

http://folktalejs.org

App vídeos YouTube

- Permite buscar vídeos “al vuelo”
- Muestra los vídeos como un grid
- Al hacer clic sobre uno, se abre su página de YouTube

App vídeos YouTube



App vídeos YouTube

- Permite buscar vídeos “al vuelo”
- Muestra los vídeos como un grid
- Al hacer clic sobre uno, se abre su página de YouTube

5.

Conclusiones

Conclusiones

- Crear funciones pequeñas (Divide and conquer (and reuse))
- **Composición** de esas funciones
- Utilizar funciones puras
- Tener aislada la parte impura del programa
- Estilo declarativo vs imperativo

¡Gracias!

¿Preguntas?

@pallasr

https://github.com/MostlyAdequate/mostly-adequate-guide