

Functional Programming

In Javascript

{ with: categories }

Gain advanced understanding of the mathematics
behind modern functional programming



Dimitris Papadimitriou

FUNCTIONAL PROGRAMMING IN JS WITH CATEGORIES

Gain advanced understanding of the mathematics behind modern functional programming

Dimitris Papadimitriou

This book is for sale at <https://leanpub.com/functional-programming-in-js-with-categories>

This version was published on 2019-06-08

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Feel free to contact me at:

<https://www.linkedin.com/in/dimitrispapadimitriou/>

<https://leanpub.com/u/dimitrispapadim>

<https://medium.com/@dimpapadim3>

<https://github.com/dimitris-papadimitriou-chr>

dimitrispapadim@live.com

Resources

Functional Resources: <https://github.com/dimitris-papadimitriou-chr/functional-resources>

Source Code: <https://github.com/dimitris-papadimitriou-chr/FunctionalJsWithCategories>

Video Course : Also, there is a **video course** on its way on Leanpub .

Acknowledgments

<https://pixabay.com/illustrations/parrot-bird-macaw-beak-tropical-2979285/>

© 2019 Dimitris Papadimitriou

Contents

About this book coding conventions	7
Type Safety	7
Fantasy land.....	9
1.1 Categories	12
1.2 Essential Lambda calculus with Js.....	14
1.3 SKI combinators.....	15
2 The Pillars.....	18
2.1 Monoids	18
2.1.1 Function composition as a monoid.....	21
2.1.2 Folding monoids	24
2.2 folding functions under composition	25
2.2.1 Composing monoids	26
2.2.2 Monoid homomorphisms and Parallelism.....	28
2.3 Higher-order functions	31
2.3.1 The Strategy Design pattern	31
2.4 Currying and partial application.....	33
3 Algebraic Data Types.....	38
3.1 The product structure:.....	39
3.1.1 Introduction / Elimination.....	41
3.2 The co-product (aka Union) structure:.....	42
3.2.1 Introduction / Elimination.....	45
3.3 Extending Union Types.....	45
3.3.1 Adding Pattern Matching extensions to Union Types	46
3.3.2 Rewriting Union Type methods with matchWith	47
3.3.3 Pattern matching in functional libraries.....	48
3.4 One	49
3.5 Recursive Algebraic Types	50
3.5.1 Rewriting map with matchWith.....	51
3.5.2 On the value of the symbolic representation	52
3.5.3 Adding Pattern Matching extension to native array	52
4 Functors.....	57
4.1 The Identity Functor	57
4.2 Commutative Diagrams	59
4.3 The Functor Laws	60
4.4 A brief mentioning of <i>Catamorphisms</i>	61

4.5	Id Functor on the Fly	63
4.6	Extending Promise to Functor	63
4.6.1	The Promise - functor laws	65
4.7	IO Functor, a Lazy Id Functor	66
4.7.1	Lazy as Functor	66
4.7.2	IO Functor	68
4.8	Reader Functor	69
4.9	Maybe Functor	71
	Dealing with null - Null object Design pattern	71
4.9.1	The Null Object Design pattern	71
4.9.2	The Functional equivalent - Maybe as Functor	72
4.10	Either Functor	74
4.10.1	Using Either for exception handling	76
4.10.2	Either and Promises	78
4.11	Functors from Algebraic Data Types	81
4.12	Functor Composition	83
4.13	Applicative Functor	85
4.14	Reader Applicative Functor	86
4.15	Composing Applicatives	88
4.16	Decorator design pattern functional idiom with Functor	88
5	Catamorphisms Again	90
5.1	A brief mentioning of F-algebras	90
5.2	Catamorphisms	91
5.3	Initial algebra	94
5.3.1	F-Algebras Homomorphisms	94
5.4	Catamorphisms for Trees	96
5.5	Reversing a Tree	97
5.6	Catamorphisms with the Visitor Design pattern	98
5.6.1	F-Coalgebra	99
5.6.2	A brief mentioning of Anamorphisms	99
5.6.3	Corecursion	100
5.6.4	A brief mentioning of Hylomorphisms	102
5.6.5	Hylomorphism example: Mergesort	104
5.7	Fold relation with Cata	105
6	Traversable	106
6.1	Traversable Array with Either applicative for validation	108
6.2	Traversable Algebraic data structures	109

6.3	Identity Functor Traversable	110
6.4	Traversing with The Promise - Parallel.....	111
6.5	Applicative Reader Isomorphism with the Interpreter Design pattern	113
6.5.1	A Catamorphism implementation	114
6.5.2	Object Oriented Interpreter Pattern implementation	114
6.6	Composing Traversables	115
6.7	Foldable.....	116
6.7.1	FoldMap.....	118
6.7.2	Filterable structures	119
6.8	Fold and FoldMap Derivation from Catamorphism	120
6.9	Traverse Derivation from Catamorphism.....	121
6.9.1	Mixins	122
6.9.2	Composing Foldables	122
6.9.3	Iterators	123
7	The Yoneda Lemma	125
7.1	Co-Yoneda	125
8	Natural Transformations.....	128
8.1.1	Natural Transformation Between Maybe and Either	130
9	Monads.....	132
9.1	The List Monad	134
9.2	The Identity Monad	134
9.2.1	Monad laws for Identity Monad	135
9.3	Monads -Kleisli Composition in Javascript	136
9.4	Monad laws following Kleisli formulation	137
9.5	Maybe Monad	138
9.6	Either Monad	140
9.6.1	Using Either Monad exception handling	141
9.7	State Monad	143
9.7.1	Traversing with State	146
9.8	Reader Monad	148
9.9	IO monad.....	150
9.10	Writer Monad	151
9.10.1	Traversing with Writer	152
9.11	Deriving the Continuation Monad.....	153
1)	Use callback and remove return type	153
2)	Curry the callback	153
3)	Generalize the returned continuation	154

4) Extract the continuation structure.....	155
9.12 The Continuation monad.....	156
9.13 Extending Promises to Monads.....	157
9.14 Async Error handling with either.....	158
9.14.1 Implementing Bind	159
10 Comonads.....	161
10.1 The Identity Comonad.....	161
10.2 Co-Monad laws for Identity Co-Monad	162
10.3 CoKleisli Composition	163
10.3.1 Co-Monad laws following Co-Kleisli formulation	164
10.4 Store Comonad.....	164
10.5 Lazy.....	165
10.6 Pair Comonad.....	166
10.7 Spatial Comonad and the <i>Game of Life</i>	169
10.8 Stream Comonad	172
10.9 Tree Comonad.....	173
10.9.1 Tree annotation example	174
11 F-Algebras.....	176
11.1 F-Algebras Homomorphisms	180
11.2 Initial Algebras	182
12 F-Coalgebra	182
12.1 Catmorphisms.....	183

Purpose

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

— Edsger Dijkstra

OO makes code understandable by encapsulating moving parts.
FP makes code understandable by minimizing moving parts.

—Michael Feathers (Twitter)

One of the main reasons for this book is to transfer in the community of object-oriented developers some of the ideas and advancements happening to the functional side. This book wants to be pragmatic. Unfortunately, some great ideas are coming from academia that cannot be used by the average developer in his day to day coding.

This is not an Introductory book to functional programming, even if I restate some of the most basic concepts in the light of category theory and the object-oriented paradigm. This book covers the middle ground. Nonetheless, it is written in a way that if someone pays attention and goes through the examples, he or she could understand the concepts. An introductory book will follow in [Leanpub](#), covering more basic ideas, along [with the extended version](#) of this book covering more advanced topics. If you think the content of this book is more difficult than what you expected, please contact me to give you a free copy of the book “The Simplified Functional Programming in JavaScript, with categories” when it is finished in [Leanpub](#).

In this book, I will try to simplify the mathematical concepts in a way that can be displayed with code. If something cannot be easily displayed with code probably will not be something that can be readily available to a developer’s arsenal of techniques and patterns.

If you think a section is boring, then skip it and maybe finish it later.

One thing I admire in Software Engineers is their ability to infer things. The ability to connect the dots is what separates the exceptional developer from the good one. In some parts of the book, I might indeed have gaps or even assume things that you are not familiar with. Nonetheless, I am sure that even an inexperienced developer will connect the dots and assign meaning, as I usually do when left with unfinished

About this book coding conventions

A quick stop here. Most of the code in this book use vanilla JavaScript. Instead of the standard way to create a class:

```
var Rectangle = function (height, width) {
  this.height = height;
  this.width = width;
  this.area = function () {
    return this.height * this.width;
  }
}
```

Most of the times I will use this brief notation:

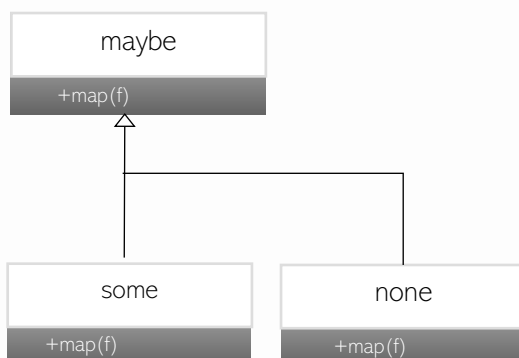
```
const Rectangle = (height, width) => ({
  area: () => height * width,
  scale: (s) => Rectangle (s * height, s * width),
});
```

Object literal notation forces immutability. In this way of writing, you must always return a new instance. In a larger Domain Model, this way of writing might not be viable. Here we do not need to use `new` to create an object. Also, we cannot access the `height` and `width` variables. If you want to mutate the state, you either must question your design of the object that forces you to think in a defensive programming style. Let's be pragmatic here, this style of coding that promotes purity should **not be an end in itself**, and we should always be able to convince ourselves to go back into an object-oriented style of coding even if it is not that pure. The important thing is to write **functional code that provides business value** to the organization and to the end-user that will eventually use it. We must be pragmatic and utilitarian when we code and abstractionists when we think about code.

Type Safety

Well. In this book, we will use minimal dependencies; this means that there is an orientation to vanilla JavaScript. And **let me tell you, there is no type safety in vanilla JavaScript**. Type safety became so prevalent because the benefits are exciding all the drawbacks and overheads. The current popularity of Typescript is not random. In Some difficult sections there would be some brief **TypeScript** paragraphs to allow a quick pick at the underlying Types. There is also a [translation of this book in TypeScript on Leanpub](#). **The importance of paying attention to types in a dynamic type system becomes even more important for the developer**. That's why type theory is an essential part of this book. Duck typing is used throughout the book. What is duck typing? We use the duck test—"If it walks like a

duck and it quacks like a duck, then it must be a duck"—to determine if an object can be used for a particular purpose. When we have hierarchies like the one below, for example, we will be using duck typing to implement it



```

var some = (v) => ({
  map: (f) => Some(f(v))
});

var none = () => ({
  map: (f) => none()
});
  
```

Run This: [Js Fiddle](#)

This way of writing is for explanatory purposes. In production code or code that belongs to libraries and application framework etc. proper inheritance might be considered, depending on the scope of the model

```

class Maybe {
  map(f) {
    throw new Error('You have to implement the method map!');
  }
}

class Some extends Maybe {
  constructor(v) {
    super();
    this.v = v;
  }
  map(f) {
    return new Some(f(this.v));
  }
}

class None extends Maybe {
  map(f) {
    return new None();
  }
}
  
```

Run This: [Js Fiddle](#)

We will also be using Javascript mixins, to add functionality without involving inheritance (This became very famous with scala traits).

Fantasy land

Fantasy land is a set of specifications for javascript functional structures. Some functional libraries out there are following the naming conventions and structure of the fantasy land specifications. In this book, we will adopt fantasy-land conventions loosely.

Overview

I want to give a very general idea about the structures that we are going to meet in this book in the most simplified way I could come up with. This is the 10.000 feet view of the first half of the book.

1. Two ways to look an evaluation

Let's start with this simple expression.

```
var s = a => b => a(b)
```

(that's a simple application by the way) we could write the following

```
s(console.log)(5) // prints 5
```

We have an expression that waits for a value, and when we finally give the value, it evaluates the expression. Also, we can separate the two parts

```
var t = s(console.log);

setTimeout(() => { t(5) }, 3000);
```

Run This: [Js Fiddle](#)

We can save the execution in a variable, and at some later time when we have a value, inject it to the expression and finish our execution.

We can write another expression-based on our s

```
s(x => x(5))(console.log)
```

Here we reverse the order and first give the value that we want to use and then at the end the action that we want to perform. The reason behind `x => x(5)` is that because in our `s` we apply the `a` on `b` like this `a(b)` we have to use a callback to reverse the order so it's cancel out [try to create this by yourself]. Now I will rename this `x` variable to `resolve`:

```
s(resolve => resolve(5))(console.log)
```

Hopefully, a promise comes to mind. The above expression is equivalent to this promise

```
new Promise (resolve=>resolve(5)).then(console.log)
```

as you would expect this should work in a delayed fashion as well. Or it would not be a true promise

```
s(resolve => setTimeout(() => { resolve(5) }, 3000) )(console.log)
```

Run This: [Js Fiddle](#)

2. Adding a mapping function

We will extend our `s`.

```
var s = a => f => b => a(f(b))
```

here I added another variable `f` in the middle of all this. now we can write this

```
s(console.log)(x => x + 2)(5)
```

luckily for us because of the signature we could probably add many `f`'s

```
s(console.log) (x => x + 2) (x => x + 3) (5)
```

so nice... but it does not work. This does though:

```
s(console.log) (s (x => x + 2) (x => x + 2)) (5)
```

Run This: [Js Fiddle](#)

I got carried away for a moment (pun intended). So, let us go back to the other way of looking at things

```
s(resolve => resolve(5))(x => x + 2)(console.log)
```

this also does not work because everything is reverse, but this could

```
s(resolve => resolve(5))(f => x => f(x + 2))(console.log)
```

now I can define this map operator and use this

```
var map = g => f => x => f(g(x))
```

```
s(next => next(5))(map(x => x + 2))(console.log)
```

[Run This: Js Fiddle](#)

does it look like this [map from Rx.js](#) ?

We saw two ways of making a computation consisting of two parts a value and an action on the value.

Having that in mind will help you recognize in the first form `s(console.log)(5)` the Reader, the IO and the State functors, and in the second form `s(resolve => resolve(5))(console.log)` the continuation monad, promises, and the observables.

A quick glance at Lambda

And basic category theory

1.1 Categories

Category Theory is a mathematical discipline with a wide range of applications in theoretical computer science. Concepts like *Category*, *Functor*, *Monad*, and others, which were originally defined in Category Theory, have become pivotal for the understanding of modern Functional Programming (FP) languages and paradigms.

A category C consists of the following three mathematical entities:

1. A class $\text{ob}(C)$, whose elements are called **objects**. Any object-oriented programmer would find this as a great way to start a definition.



Our favourite category in programming is the **category of types** `int`, `bool`, `char`, etc. There are many interesting categories in programming and in this book, we will explore some of them.

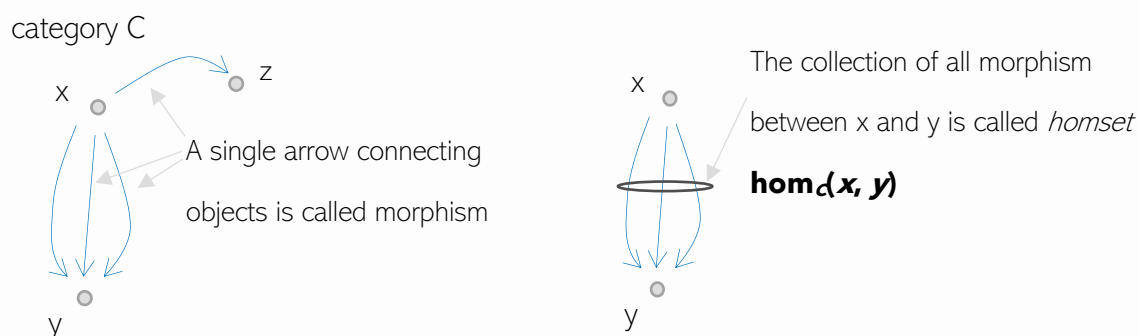
2. A class $\text{hom}(C)$, whose elements are called **morphisms** or *arrows*. Each morphism f has a *source object* a and *target object* b . For our type category, any arrow from `int` \rightarrow `bool` is a function. For example, this one:

```
var isEven = a => a % 2 === 0;
```

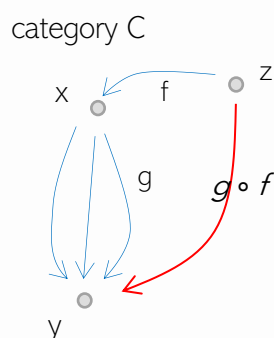
Or this one:

```
var isLessThan10 = a => a < 10;
```

the set of all those morphisms is called **HomSet**, and we write **$\text{hom}(\text{int}, \text{bool})$** . In general, in any category C the expression **$\text{hom}(a, b)$** means all morphisms from object a to object b . Let me note that there are some categories where the arrows could “be represented as objects” inside the category like in our Type category, where the functions are themselves types.



3. A binary operation • called *composition of morphisms*. The composition of $f: z \rightarrow x$ and $g: x \rightarrow y$ is written as $g \circ f$ and is read g **after** f since the f comes before g.



Two laws govern the composition:

- Associativity: If $f: a \rightarrow b$, $g: b \rightarrow c$ and $h: c \rightarrow d$ then $h \circ (g \circ f) = (h \circ g) \circ f$, and
- Identity: For every object x , there exists a morphism $1_x: x \rightarrow x$ called the identity morphism for x , such that for every morphism $f: a \rightarrow b$, we have $1_b \circ f = f = f \circ 1_a$.

In our **type category**, the composition law is luckily true, and both the axioms are satisfied. In any other case, we would have to abandon our idea of a category of types from the very beginning.

The basic focus of category theory **is the relations between objects** and not the objects per se, in contrast with the Set theory that primarily focuses on sets of objects. Functional programmers quickly endorsed this unique perspective of category theory.

1.2 Essential Lambda calculus with Js

Lambda calculus is a formal system in mathematical logic, that was used to express computations. It was created by the great Alonzo Church in the 1930s. The formulation of the lambda expressions is constructed bottom up- in an inductive manner using as it base the term **Term**. A term can be one of three things:

1. A variable: x
2. An abstraction: $(\lambda x. M)$ where M is a **Term**
3. An Application: $(M N)$ where M and N are **Terms**

[As you can see that the first rule is something that is terminating, something that cannot be decomposed or expand, while the other two are recursive because in their definition they use the **Term**, *which is the same thing we want to define*]

Now with those 3 rules, we can create infinite expressions in this simple language bottom-up that are well-formed. For example:

$$(M N) \xrightarrow{\text{replace } M} ((\lambda x. M) N) \xrightarrow{\text{replace } M \text{ again}} ((\lambda x. (\lambda y. M)) N) \rightarrow \dots$$

After we have built a term, we can use some rules to evaluate them.

The important rule here is called **β -reduction**.

1. $((\lambda x. M) E) \rightarrow_{\beta} (M [x:=E])$ this is a simple substitution of E inside M wherever there is the variable x

This simple language might not have all the comforts of a language like JavaScript, but it has the exact same expressive power with JavaScript. Lambda calculus and JavaScript are both Turing complete.

In λ -calculus, functions are defined using λ (lambda) and are not named. The lambda term $(\lambda x. x)$ in JavaScript terms would be an anonymous function having as an argument a free variable named x but since the term `function(x){ }` is not syntactically correct for Js we have to rewrite it like this :

```
(function(x){  })();
```

This is valid expression called **IIFE** and is pronounced as *"iffy"* in JavaScript slung. The name comes from the term ***Immediately-invoked Function Expression*** which is one of the often-used coding patterns with functions (the module and revealing module patterns in javascript are some of the most famous examples of "iffies").

If we use the lambda arrow notation, we can now write this, in a minimalistic way that we are going to be using heavily in this book:

```
(x=>x)();
```

Let's say we have this expression in js

```
var discountedPrice = 100 - 0.1 * 100;
```

If we want to abstract the discount 0.1 to make it more generic, we can use a common refactoring move that **is called extract method**.

```
var discountedPriceCalculation = function (discount) {return 100 - discount * 100}
var discountedPrice = discountedPriceCalculation(0.1);
```

in lambda we could write loosely (since there are no integers or -, * defined):

```
M = (λ discount. (100-discount*100))
```

and the whole computation to get the discounted price would be this evaluation

```
(λ discount. (100-discount*100))(0.1)
```

This uses the rule 3 (M N) where we apply 0.1 on our abstraction of (λ discount. (100-discount*100))

Which is the same as writing in javascript

```
var discountedPrice = (discount=>100 - discount * 100 )(0.1);
```

one could further abstract the value 100, which is the price in a new abstraction move

```
var discountedPrice = (price=>(discount=>price - discount * price)(0.1))(100);
```

we are going to do that a lot in this book. (In the above example one can see that the last expression is curried as we will discuss in a later section) in lambda terms, this would be

```
M=(λ price. (λ discount. (price - discount* price)))
```

Now we could evaluate the expression for a given discount and price using the β -reduction rule of substitution.

```
(λ price. (λ discount. (price - discount* price)) (100) →β (λ discount. (100- discount* 100))
```

That is all the lambda calculus for now.

1.3 Optional SKI combinators

Combinatory logic is a notation to eliminate the need for quantified variables in mathematical logic. A **combinator** is a λ -term with no free variables. One can intuitively understand combinators as 'completely specified' operations since they have no free variables.

One book named "To Mock a Mockingbird" by the mathematician and logician Raymond Smullyan, that contains puzzles in Combinatory Logic, singlehandedly spawn a subculture

of lambda combinators inside functional communities. You can find combinators libraries in javascript, like this one called [combinators-js](#) that has a wide collection of combinators, aka "[birds](#)." [Combinators](#) are not commonly used in real-life development scenarios, but they offer an exercise in our lambda understanding. Nonetheless, some of the most popular js functional libraries, like ramda and [sanctuary](#) support the most important of the combinators.

Here I want to mention some of the classic combinators that have a profound meaning in programming :

1. **I** combinator:

```
const I = x => x;
```

What can someone say for the identity? This is the alpha and the omega. Well maybe not the omega which is this similar-looking combinator $\omega = (\lambda x.xx)$ that when we apply to **I** we get **I** since: $\omega I \rightarrow (\lambda x.xx) I \rightarrow II \rightarrow I$ but when we apply it to itself we get an infinite loop: $\omega\omega \rightarrow (\lambda x.xx) \omega \rightarrow \omega\omega \dots$ if we slightly modify the ω following this idea of $\omega\omega$ copying itself we will get the great Y combinator [unfortunately I won't get to Y combinator in this book, but you can check [this article for insight](#)].

2. K combinator aka constant. $Kab = b$

```
const K = a => b => a
```

K, when applied to any argument *a*, yields a one-argument constant function **Ka**, which, when applied to any argument, returns *a*. K simply discards b.

3. S combinator aka substitution $S f g x = f(x)(g(x))$

```
const S = f => g => x => f(x)(g(x));
```

S, takes x and applies it to both f and g and then applies the g(x) onto f(x). We will find this in the [applicative reader functor section](#). The x should be understood like a common environment of variables (or a configuration) that is provided to both f and g, and then we apply the result of g(x) to the f(x), which is a function.

```
const S = f => g => x => f(x)(g(x));
```

```
var pricingCalculation = (config => price => price - config.discount * price);
var getPrice = (config => config.productPrice);
var config = ({ discount: 0.1, productPrice: 100 });
```

```
var discountedPrice = S(pricingCalculation)(getPrice)(config);
```

```
console.log(discountedPrice)//90
```

Run This: [Js Fiddle](#)

4. B combinator Aka Compose - $B x y z = x (y z)$

```
const B = f => g => x => f(g(x));
```

B is just the classic function composition

5. Y combinator Aka fixed point combinator

This is the famous Y combinator that was introduced by Haskell Curry to represent recursion in his lambda calculus. Unfortunately, in spite of its beauty Y combinator is not used often. [If you are interested you can find a bit more in this article]

Amazingly S and **K** can be composed to produce combinators that are extensionally equal to *any* lambda term, and therefore, by Church's thesis, to any computable function whatsoever

The Pillars

2 The Pillars

2.1 Monoids

"Alternatively, the fundamental notion of category theory is that of a Monoid"

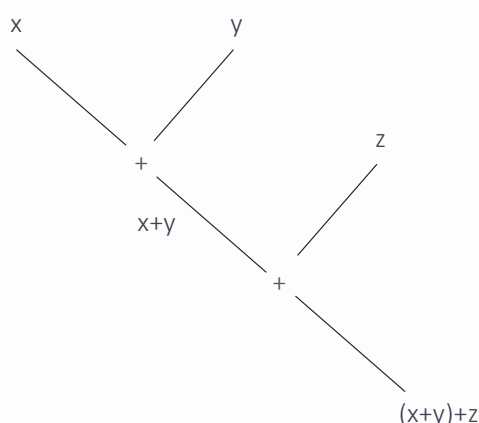
– Categories for the Working Mathematician

Monoids are one of those profound ideas that are easy to understand and are everywhere. Monoids belong to the category of Algebras. There are a couple of very important ways to organize different structures that appear in functional programming. Algebras is one of them. Fantasy land has some of the most important algebras in its specification.

Wikipedia says a **monoid** is an algebraic structure with a single associative binary operation and an identity element. Any structure that has those two elements is a monoid.

Suppose that S is a structure and \otimes is some binary operation $S \otimes S \rightarrow S$, then S with \otimes is a **monoid** if it satisfies the following two axioms:

1. **Associativity:** For all a, b and c in S , the equation $(a \otimes b) \otimes c = a \otimes (b \otimes c)$ holds.
2. **Identity element:** There exists an element e in S such that for every element a in S , the equations $e \otimes a = a \otimes e = a$ hold.



Let us take the integers we can take addition as a binary operation between two integers. The addition is associative. This means there is no importance on the order of the addition.

$$(x+(y+z)) = ((x+y) +z)$$

And 0 is the identity element for addition since :

$$x + 0 = x$$

For those reasons, the pair **(+,0)** forms a monoid over the integers. However, we can form different other monoids over the integers. For example, multiplication ***** also can be viewed

as a monoid with its respective identity element the **1**. In this way the pair **(*,1)** is another monoid over the integers. Strings in javascript have a default monoid, which is formed by the concatenation and the empty string (**' '**, **concat**). String type in javascript has natively a **concat** method that refers to this specific monoid we all know. However, for integers, for example, it would not make any sense to pick a specific monoid as default.

We will use monoids in our code by defining simple object literals that have two functions, one that returns the identity element called **empty** and one that represents the binary operation called **concat**.

[In this book, we will use the fantasy land specification of the monoid regarding naming conventions, especially since javascript already uses concat for strings and arrays.]

```
var Sum = { empty: () => 0, concat: (x, y) => x + y }
var Product = { empty: () => 1, concat: (x, y) => x * y }
```

the types of those operations are

which we will use it like this:

```
Product.concat(Product.empty(), 5);
```

This definition would allow us to use it with the **reduce** function of a list in a direct manner

```
var Sum = { empty: 0, concat: (x, y) => x + y }
var reduce = [1, 2, 3, 4, 5, 6].reduce(Sum.concat, Sum.empty);
```

Run This: [Js Fiddle](#)

Alternatively, if we want to have closure [meaning that the return type of the **concat** and **empty** is of the same type with the initial object], we could define one new class for each monoid.

```
Class Sum {
  constructor(v) { this.v = v; }
  static empty() { return new Sum(0); }
  concat(b) { return new Sum(this.v + b.v); }
};
```

The return type is **Sum** allowing for fluent chaining

```
var result = Sum.empty() .concat(new Sum(10)) .concat(new Sum(20)) // Sum(30)
```

Run This: [Js Fiddle](#)

This formulation is nice in the sense that allows for fluent chaining `var result = Sum.empty() .concat(new Sum(10)) .concat(new Sum(20))` because of the fact that empty and concat both return a result of the same type. This also is more congruent with the mathematical definition that says that if S is a structure (in a programming language, this translates to a Type), then $S \times S \rightarrow S$, means that the operation returns a structure of the same kind. The use of reduce is a bit more tedious thought

```
var reduce = [1, 2, 3, 4, 5, 6]
  .map(x=>new Sum(x))
  .reduce((a, b) => a .concat(b), Sum.empty());
```

In practice we do not care about strict definitions as long as we are aware that we are dealing with a monoid.

We already said that the javascript **Array** is a monoid having **concat** as the binary operation

```
[x].concat([y].concat([z])) == ([x].concat([y])).concat([z])
```

Furthermore, `[]` is an identity element for concat:

```
[x].concat([])=[x]
```

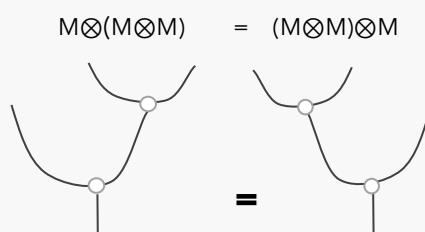
Category Theory

Monoids

In Category theory we would say that a monoid in a **monoidal** category **C** is an object **M** equipped with a multiplication $\mu: M \otimes M \rightarrow M$ and a unit $\eta: I \rightarrow M$



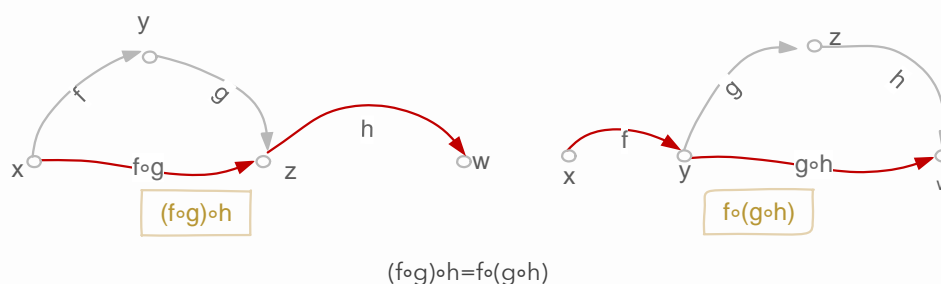
The requirements of associativity have the structure of the following graphical rules in string diagrams (don't worry about the rigorous definition) :



2.1.1 Function composition as a monoid

Let us look at a more interesting example that will help us understand monoid in a more abstract way. Let's say that we have a category where the objects are the functions, and the binary operation is the **composition** of two functions $_ \circ _$, then associativity still works since:

$$(f \circ g) \circ h = f \circ (g \circ h)$$



Furthermore, $id = x \Rightarrow x$ is an identity element for composition \circ that is:

$$(f \circ id) = f$$

In JavaScript the function composition for two functions is

```
var compose = f => g => x => f(g(x))
```

associativity means that

```
compose(h)(compose(g)(f)) === compose(compose(h)(g))(f)
```

Run This: [Js Fiddle](#)

for any h, g, f .

because of monoid nature of composition we can extend the definition that takes only two functions on any amount of functions by just keep applying the composition. For example, for four functions would be `compose(h)(compose(g)(compose(g)(k)))`. We will see how to use the `Array.reduce` to give a generic definition of an arbitrary number of functions in the next section.

In this way, the composition is a monoid $(\circ, x \Rightarrow x)$ in the category of functions. Functional composition is probably the most important tool of functional programming, and we are going to use it heavily throughout the book, so we are not going to spend much time here.

2.1.1.1 Functional idiomatic Decorator Design pattern with composition

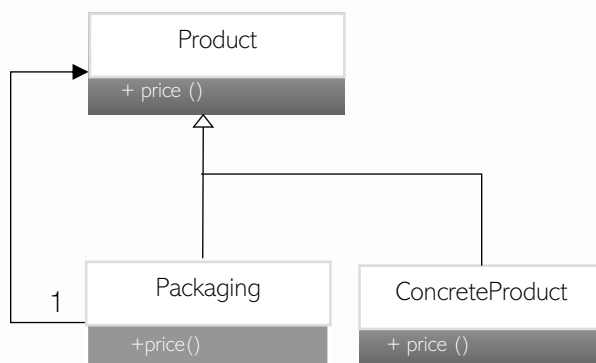
Functional composition is in the heart of many object-oriented best practices. Here we will see an idiomatic expression of the Decorator design pattern, which is one of the most used object-oriented design patterns from Gang of Four.

Let start with simple example where we have a product, and we can decorate it with packaging. Adding a packaging would affect the price, but we want to treat a product and a packaged product interchangeably in our code. The decorator pattern allows us to do just that.

```
var ConcreteProduct = function (price) {
  this.price = function () {
    return price;
  }
}

var Packaging = function (product) {
  this.price = function () {
    return product.price() + 0.5;
  }
}
```

Run This: [Js Fiddle](#)



The decorator pattern is based on the idea/theme that the decorator exposes the same interface with the decorated object. In this example, the `Packaging` exposes the same `price` method with the decorated `Product`. This allows the rest of the application to handle a simple product or a decorated product in the same way without needing to know more details in order to get the price [this is a nice display of **information hiding/encapsulation** in object-oriented programming, since the decorated object hides the details of implementation from the outside world and only reveals the price].

In the functional realm we can achieve this by functional composition.

```
var createProduct = (value) => ({ price: () => value, })
var addPackaging = (product) => ({ price: () => product.price() + 0.5, })
var addRibbons = (product) => ({ price: () => product.price() + 0.3, })
```

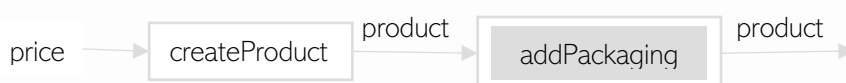
```
var finalProduct = addRibbons (addPackaging (createProduct (20)))
```

Run This: [Js Fiddle](#)

Or by using our defined function for function composition, we can write

```
var finalProduct = compose(addRibbons, addPackaging, createProduct)(20);
```

The idea here is for the decorating composing functions that come after the decorated function that has a return type Product.



Here the type of the function `addPackaging` is `product → product`. Which means that it does not affect the signature.

```
createProduct ◦ addPackaging: int → product
```

```
createProduct: int → product
```

after pre-composing `int → product` with `product → product` the composition remains of type `int → product`

This is a very common functional design pattern, that is used in practice. We will see some different variations as we go along.

Libraries

Composition in Js functional libraries

There is no surprise that functional composition almost in every functional library available

```
lodash    : _.compose
Ramda     : R.compose
Sanctuary : S.compose
```


2.1.2 Folding monoids

Monoids have this very desirable property that if we keep applying this binary operation, we can always reduce the computation to a single element of the same type:

$$(M \otimes \dots (M (\otimes (M \otimes M))) \rightarrow M$$

this is called folding. We already know folding as **reduce** from the js Array type. However, we are going to generalize fold in a latter chapter for data structures that we will call traversables. Coming back to fold, let us try to derive reduce.

Lets say we have some sequence of integers 2,4,5,6 and we want to add them this a beginner level problem:

```
var array = [2,4,5,6 ]

var accumulation = 0;
for (let i = 0; i < array.length; i++) {
  const element = array[i];
  accumulation = accumulation + element
}
```

I highlighted there the elements of a monoid. So let's abstract away. If we replace the (0,+) with any other monoid this should work, so why not abstract and reuse.

```
var fold = function (array, monoid) {
  var accumulation = monoid.empty(); // abstracted 0
  for (let i = 0; i < array.length; i++) {
    const element = array[i];
    accumulation = monoid.concat(accumulation, element) // abstracted +
  }
  return accumulation;
}
```

Now we can use this function with all kind of monoids

```
var sum = fold([2, 4, 5, 6], { empty: () => 0, concat: (a, b) => a + b })
var product = fold([2, 4, 5, 6], { empty: () => 1, concat: (a, b) => a * b })
var max = fold([2, 4, 5, 6], { empty: () => -Infinity, concat: (a, b) => Math.max(a, b) })
```

this is ok, but we might want to be more flexible instead of just providing the monoid

```
var fold = function (array, accumulation, f) {
  for (let i = 0; i < array.length; i++) {
    const element = array[i];
    accumulation = f(accumulation, element)
  }
  return accumulation;
}
```

```
var sum = fold([2, 4, 5, 6], 0, (accumulation, element) => accumulation + element)
```

this is more flexible because we can also manipulate the elements before doing the addition `accumulation + element`, for example, we could add the square of the elements

```
var sum = fold([2, 4, 5, 6], 0, (accumulation, element) => accumulation + element*element)
```

which is the same as first mapping a square function and then sum up

```
var sum = fold([2, 4, 5, 6].map(x => x * x), 0, (accumulation, element) => accumulation + element)
```

this is the exact signature of `array.reduce` (we will take a deeper look at fold in chapter 6) in this example we assumed that `2,4,5,6` where stored in an array, but we could have had a structure like this one `({v:2,r:{v:4,r:{v:6,r:{ v:6,r: null }}}})`, this is isomorphic with the array, but we cannot use the previous fold function to sum the numbers. But how about this fold ?

```
var listF = ({ v: 2, r: { v: 4, r: { v: 6, r: { v: 6, r: null } } } })
```

```
var fold = function (array, monoid) {
  var accumulation = array.v;
  var next = array.r;

  while (next) {
    accumulation= monoid.concat(accumulation,next.v);
    next = next.r
  }

  return accumulation;
}
```

```
var t = fold(listF, { empty: () => 0, concat: (a, b) => a + b })
```

2.2 folding functions under composition

What if we pass an array of functions into the fold function, and since functions have a monoid consisting of the id and composition (`id`, `o`) use this to fold the array?

```
var functions = [x=>x+1, x=>x*x, x=>`thats the result: ${x}`];
```

```
var id = x=>x;
```

```
var composeM = { empty: () => id, concat: (f, g) => x =>g(f(x)) }
var compositon = fold(functions, composeM)
var compositionResult =compositon(2);//`thats the result: 9`
```

Run This: [Js Fiddle](#)

We can write a general compose function in javascript using iterator and reduce

```
const compose = (...fns) => fns.reduce((f, g) => (...args) => f(g(...args)))
```

this is a practical definition in order to compose an arbitrary number of functions and we will use it throughout the rest of the book

2.2.1 Composing monoids

Monoids can also be composed in various ways and give more monoids. In this section, we will see three such examples of composition.

Our first monoidal composition is the one that is formed by functions that have the same type signature:

$$(A \rightarrow B) \otimes (A \rightarrow B) \rightarrow (A \rightarrow B)$$

[this is different from functional composition where we have the composition operation

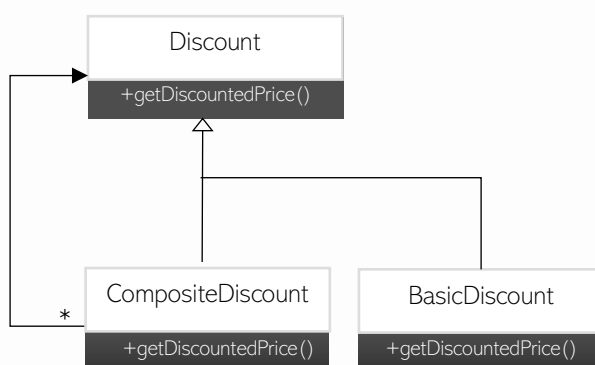
$$\circ : (A \rightarrow B) \otimes (B \rightarrow C) \rightarrow (A \rightarrow C)]$$

Let me illustrate how this particular monoid arises in a practical way. Let's say for example we have this discount function `var repeatCustomerDiscount = (price) => price * 0.2` that type of this function is `double → double` we can define a higher order function `ComposeDiscounts` that composes two functions of this type:

```
var repeatCustomerDiscount = (price) => price * 0.2;
var newProductDiscount = (price) => price * 0.1;
var ComposeDiscounts = (f1, f2) => (price) => f1(price) + f2(price);
var totalDiscount = ComposeDiscounts(repeatCustomerDiscount, newProductDiscount);
```

Run This: [Js Fiddle](#)

One can prove that we can create a monoid from a function $f : (A \rightarrow B)$ when B is a monoid. This idea is in the core of the famous composite design pattern from the [Gang of Four design patterns](#).



The following is an implementation of a Composite design pattern for a cart that contains bundled products. The reason that this pattern works is because we apply a binary operation over a list of `getDiscountedPrice():double` and reduce them to an integer by summation. The resulting composition has the same signature `getDiscountedPrice():double`.

```
class Discount {
  getDiscountedPrice() { }
}
class BasicDiscount extends Discount {
  constructor(discount) {
    super();
    this.discount = discount;
  }
  getDiscountedPrice(price) {
    return this.discount * price;
  }
}
class CompositeDiscount extends Discount {
  constructor() {
    super();
    this.discounts = [];
  }
  addDiscount(discount) {
    this.discounts.push(discount);
    return this;
  }
  getDiscountedPrice(price) {
    return price - this.discounts.reduce((sum, discount) =>
      sum + discount.getDiscountedPrice(price), 0)
  }
}
var discountComposite = new CompositeDiscount()
  .addDiscount(new BasicDiscount(0.1))
  .addDiscount(new BasicDiscount(0.2));

console.log(discountComposite.getDiscountedPrice(100))
```

The composite discount uses the composition of monoids in `getDiscountedPrice`

Run This: [Js Fiddle](#)

Another easy monoidal composition is the one that stems from combining two monoids in a tuple (aka product). One can verify that if `m1` and `m2` are monoids then this is also a monoid :

```
var Pair = (m1, m2) => ({
  empty: ({ first: m1.empty(), second: m2.empty() }),
  concat: (x, y) => ({
    first: m1.concat(x.first, y.first),
    second: m2.concat(x.second, y.second)
  })
});
```

Run This: [Js Fiddle](#)

Using a product of monoids, one could perform two different computations on a list at the same time.

```
var pair = Pair(Sum, Sum);
var fold = [1, 2, 3, 4]
  .map(x => ({ first: x, second: 1 })))
  .reduce(pair.concat, pair.empty); //{first:10, second:4}
var average = fold.first / fold.second;
```

Run This: [Js Fiddle](#)

2.2.2 Monoid homomorphisms and Parallelism

The length of a concatenation of two arrays, equals the sum of the length of the arrays

```
([1, 2, 3, 4].concat([7, 8])).length === ([1, 2, 3, 4]).length + ([7, 8]).length
```

The length property can distribute over the array concatenation if the (concat) is replaced with (+). *This means that if we break an array in small chunks and compute a property like length and then add the results this should be the same as we computed the result to our initial array.* This fact comes from the following idea of Monoid Homomorphism.

Monoid Homomorphism

Monoid Homomorphism is a thing of beauty and a profound concept, that can be found everywhere in mathematics, logic and programming. It captures the idea of **Preserving the Structure** when we map something from one structure to another. The word *homomorphism* comes from the ancient Greek language: ὁμός (*homos*) meaning "same" and μορφή (*morphe*) meaning "form" or "shape".

A monoid homomorphism between two monoids $(*, M)$ and (\bullet, N) , is a function:

$$\phi: M \rightarrow N$$

Such that

$$\phi(a_1 * a_2) = \phi(a_1) \bullet \phi(a_2)$$

$$\phi(1_M) = 1_N$$

where 1_M And 1_N are the identities of N and M aka our monoid empties.

Let's see another example:

```
"Monoid".concat("Homomorphism").length == "Monoid".length+ "Homomorphism".length
```

Or this one

```
var totalSum = array => array.reduce((a, b) => a + b, 0)

totalSum([1, 2, 3, 4].concat([7, 8])) === totalSum([1, 2, 3, 4]) + totalSum([7, 8])
```

where the `totalSum` function is a Homomorphism over the Sum `{empty:0,concat:(x,y)=>x+y}` and Array Monoids. The point of all this is that we can parallelize over the list by splitting it to smaller chunks and process them independently. This is the idea behind the famous pattern called mapReduce

if we have a big list, we can slice it and apply a function to each chunk, then fold each slice and then fold the results of the chunks.

```
var bigList = [...Array(10e6)].map((_, i) => i);

var slice = array => size => {
  var sliced = [];
  for (var i = 0; i < array.length; i += size)
    sliced.push(array.slice(i, i + size))
  return sliced;
}

var mapReduce = array => f => m =>
  array.map(chunk => chunk.map(f).reduce(m.concat, m.empty)) //chunk map/reduce can
                                                                // be done in parallel
    .reduce(m.concat, m.empty)                                //overall reduce

var total = mapReduce(slice(bigList)(100))(x => x + 1)(Sum) ;
```

Run This: [Js Fiddle](#)

in javascript there are no threads, the nature of the language (event loop) does not allow any parallelism easily. At this point there is the new web worker API as an attempt to enable parallelism. You have to remember that the concepts discussed here go beyond javascript and we use them with any other language.

The theoretical support of the map reduce pattern came from the work of Richard Bird and Lambert Meertens that come up with the Bird-Meertens formalism and the Homomorphism lemma specifically for lists that say that a function h on lists is a list homomorphism :

$$h([]) = e$$

$$h(a.concat(b)) = h(a) \bullet h(b)$$

if only if there exist a function f and an operation \bullet so the following hold

$$h = \text{reduce}(\bullet) \circ \text{map}(f)$$

$\text{reduce}(\bullet)$ here means the operation $(e1 \bullet \dots (e2(\bullet (e3 \bullet \dots en)))$ folding the elements on a list. If there is a Homomorphism it should **be able to be decomposed** in a **map** followed by a **reduce**.

A Homomorphism in a list means Parallelism. Let's see another example.

Counting words in a large document is also something that can be done in parallel.

1. We can split the document in chunks and
2. then count the words in each chunk and add them in a dictionary.
3. Then we can merge the dictionaries because they form a monoid

Here the monoid is the dictionary as a hashMap. *We can prove that if we have a $\text{hasMap}[K, V]$ with keys of type K and values of type V and **V is a monoid** then we can create a **hasMap monoid**.*

```
var dictionary1 = {};
dictionary1["and"] = 5;
dictionary1["the"] = 3;

var dictionary2 = {};
dictionary2["and"] = 7;
dictionary2["or"] = 4;

var Sum = { empty: 0, concat: (x, y) => x + y }

var mergeHashMaps = valueMonoid => ({
  empty: {},
  concat: (x, y) => {
    var merge = Object.assign({}, y);
    for (var word in x) {
      if (merge[word]) {
        merge[word] = valueMonoid.concat(merge[word], (x[word]));
      }
      else {
        merge[word] = merge[word] = x[word];
      }
    }
    return merge;
  }
})

var mergedDictionary = mergeHashMaps(Sum).concat(dict1, dict2);
```

Run This: [Js Fiddle](#)

this idea means that we can parallelize the word counting problem, just because we were able to recognize a monoid.

Monoidal category C intuitively is a category that defines a product $\otimes: C \times C \rightarrow C$ that combine objects from the category C and the resulting combinations also belong in the same category. And there is also the notion of the *identity object or unit*, to the multiplication \otimes . The monoidal category is so profound and abstract.

Let's say for the Type category that the composite types as records e.g. If we "multiply" \otimes an integer $a=1$ and a Boolean $b=true$ we could get a new object

$\{a:1\} \otimes \{b:true\} \rightarrow \{a:1, b:true\}$

$\{a:1, b:true\}$ is itself an object in the **category of types**. In this way we can create much more new types composing the basic types like int, bool, char etc.

2.3 Higher-order functions

A **Higher-Order function** is a function that receives a function as an *argument* or *returns a function as output*. But I assume that you are already familiar with this definition. What I want to do in this section is to point some connections with the Object-oriented world and discuss why Higher-Order functions are so important in functional programming.

2.3.1 The Strategy Design pattern

Object oriented programmers used Higher order functions in indirect ways. The strategy design pattern is the closest thing to Higher order function from a purely OOP perspective since it allows to pass around functions as Objects. With the strategy pattern, we can modify the behaviour of an object based on the different strategy object that we have passed to it.

If for example, we have two different discounting scenarios for our product, we can abstract the discounting calculation from the product object and place it in a strategy object.

```
var repeatedCustomerStrategy = () => ({ getPrice: (price) => price * 0.9, })
var couponStrategy = couponDiscount => ({ getPrice: (price) => price * couponDiscount, })

var Product = (price) => ({
  getFinalPrice(pricingStrategy) {
    return pricingStrategy.getPrice(price)
  }
})
```



```
    }
  })
```

```
var finalPriceRepeatedCustomer = Product(100).getFinalPrice(repeatedCustomerStrategy());
var finalPriceWithCoupon = Product(100).getFinalPrice(couponStrategy(0.5));
```

Run This: [Js Fiddle](#)

in a functional manner we could refactor this to directly pass the discount calculation function as an argument. Thus the `getFinalPrice` method of the product now becomes :

```
var Product = (price) => ({
  getFinalPrice(pricingStrategy) {
    return pricingStrategy(price)
  }
})
```

```
var finalPriceRepeatedCustomer = Product(20).getFinalPrice(price => price * 0.9);
```

here the `getFinalPrice()` is a **Higher-order Function** since it takes another function as an argument. I call this refactoring move “**replace strategy with Higher-order Function.**”. For very simple scenarios, *we do not have to create a full class in order to use the strategy pattern*. In functional programming Higher-order functions are everywhere. This is the core idea of **functional programming; to treat functions as data**.

Or we could calculate the discount by using the S combinator (we will see later this is the Reader Functor)

```
const S = f => g => x => f(x)(g(x));

var config = ({ discount: 0.1, productPrice: 100 });

var discountedPrice = S
  (config => product => product.getFinalPrice(price => price - config.discount * price))
  (config => Product(config.productPrice))
  (config);

console.log(discountedPrice)//90
```

Run This: [Js Fiddle](#)

we are not going to use S combinator in a production code, but it illustrates the fact that in functional programming by using lambda, we pass functions around as inputs or get functions as outputs all the time.

Category Theory

Internal Hom

Internal Hom : For C a category and $X, Y \in C$ two objects, the internal hom $[X \Rightarrow Y] \in C$ is, another object of C which behaves like the “object of morphisms” from X to Y.

This captures the basis of functional programming: **That the functions can be themselves treated as data.**

The internal hom relates to the product \otimes of a category with this relation

For all X, Y, Z **if $(X \otimes Y) \rightarrow Z$ is the same as $X \rightarrow [Y \Rightarrow Z]$**

this **is the definition of currying** as we will see in the next section. What this equation says is that if we have a function $f = (x, y) \Rightarrow \{...\}$ we can replace it with $f = x \Rightarrow \{ \text{return } y \Rightarrow \{...\} \}$ in essence we can exchange a composition of objects with a function. The f now returns a function as a result. **The functions can be themselves treated as data.**

In strategy pattern, the Strategy represents a reification of a method from X to Y .

2.4 Currying and partial application

Currying is a very basic and profound concept in functional programming. That is why it carries (pun intended) the name of the great Haskell Curry. There is a growing trend in functional libraries like ramda, sanctuary.js and languages like Scala to base their syntax on currying.

Currying means that a function that takes *multiple* arguments can be translated into a series of function calls that each take a *single* argument. An expression like this:

```
var f = function (x, y, z) {}
```

can be transformed to the equivalent

```
var f = function (x) {
  return function (y) {
    return function (z) {
      }
    }
  }
}
```

Or in arrow notation that will be used heavily throughout this book:

```
var f = (x, y, z) => {...}
```

```
var f = x=>y=>z=> {...}
```

In a more specific example, a function that has a discount and a price argument

```
var discountedPrice = function (discount, price) {
  return price - discount * price;
} // a regular function
```

the equivalent curried function would look like this:

```
var discountedPrice = function (discount) { // a curry function
  return function (price) {
    return price - discount * price;
  }
}

var discountedPrice = (discount) => (price) => price - discount * price;
//arrow notation
```

Run This: [Js Fiddle](#)

If we wanted to use the uncurried function with an array of prices, the code would look like this :

```
var discountedPrices = [10, 20, 30].map(x => discountedPrice(0.1, x));
```

here is the curried version:

```
var discountedPrices = [10, 20, 30].map(curry(discountedPrice)(0.1));
```

The difference here is that in the second case, we do not have the `.map(x =>` because there is no need to refer to the array object explicitly. For this reason, the first style of writing is called pointed and the second one one-point free.

After we fixed an argument to a specific discount 0.1 we got a specialized function:

```
var discountedPricePartial = x => discountedPrice(0.1, x);
```

this process of reducing the arguments of a function is **called partial application**, and is usually mentioned alongside with currying because after we have curried a function we can get a partial application on the fly like this :

```
curry(discountedPrice)(0.1)
```

this is especially nice when we use functions like `map` that expects lambdas `x=>_`. Another similar situation where currying simplifies the syntax is function composition.

For example, let us assume that we want to compose a `toDiv` function after the discount calculation:

```
var toDiv = x=><div>price: ${x}</div>;

var discountedPricesHtml=
[10, 20, 30].map(compose(curry(discountedPrice)(0.1), toDiv)) ;

console.log(discountedPricesHtml)
//[<div>price: 9</div>", "<div>price: 18</div>", "<div>price: 27</div>"]
```

Run This: [Js Fiddle](#)

[As a side note notice that those two expressions give the same result:

```
[10, 20, 30].map(compose(curry(discountedPrice)(0.1), toDiv)) ;
```

```
[10, 20, 30].map(curry(discountedPrice)(0.1)).map(toDiv);
```

Run This: [Js Fiddle](#)

Because it doesn't matter if you map prices with the composition of discount and toDiv or first discount them and then convert them to divs.]

currying always leaves the function at hand in the same syntactical state of $x \Rightarrow y \Rightarrow z \Rightarrow \dots$ every time we use **compose** or **partially apply** a value, we consume an arrow (\Rightarrow), but the result maintained the same syntactical form; $y \Rightarrow z \Rightarrow \dots$ thus promoting uniformity. This is idempotent design in practice. The downside is that the order of arguments becomes now important, and simple reorder refactoring of arguments might have side effects.

Algebra

Idempotency

An element x of an algebraic structure that has a set of elements M and a binary operation \bullet between them [that's called magma (M, \bullet)] is said to be **idempotent** if: **$x \bullet x = x$. this means that $x \bullet x \bullet x \bullet \dots = x$** no matter how many times we apply the operation \bullet it does not affect the outcome.

In Functional programming Idempotency has a special name Referential transparency.

Idempotency is a favorable property in many systems (not only programming) since it is a mechanism that **promotes fool proofing**. Take for example the elevator button, after the initial push any other push of the button does not affect the behavior of the system

Ramda.js: In ramda's landing page, someone can immediately see the following statement:

"The primary distinguishing features of Ramda are:

- 1. Ramda functions are automatically curried.*
- 2. The parameters to Ramda functions are arranged to make it convenient for currying. "*

Sanctuary.js : Most of Sanctuary.js functions are curried. For example to apply `Array.reduce` on an array `xs` the Sanctuary syntax is completely curried
`S.reduce (S.add) (0) (xs)`

Loadash: Curry

Algebraic Data Types

«Lists come up often when discussing monoids,
and this is no accident: lists are the “most fundamental” Monoid instance. »

Monoids: Theme and Variations (Functional Pearl)

Many of the most popular data structures like a list have definitions like this:

“A list is *either* an *empty list* or a *concatenation* of an *element* and a *list*.”

In the above sentence if we replace the word element with a, the either with +, and the concatenation with * we get the algebraic definition of a list:

List (a) = [] + a * List(a)

This definition is recursive because the term List appears on both sides of the definition. In javascript, this translates to something like the following:

```
var Cons = (value, rest) => ({ value: value, rest: rest })
var Empty = () => ({} )
var list = Cons(2, Cons(2, Empty ()))
```

and the equivalent in ES6 would be

```
class List { }
class Cons extends List {
  constructor(value, rest) {
    super()
    this.value = value;
    this.rest = rest;
  }
}
class Empty extends List { }

var list = new Cons(2, new Cons(2, new Empty()));
```

remember in the folding monoids section I briefly mentioned that we could store a bunch of integers [2, 4, 5, 6] in a structure like this as well

```
{ v: 2, r: { v: 4, r: { v: 6, r: { v: 6, r: null } } } }
```

This definition is the base definition of a List. Furthermore, because of its significance, we will often use this simple algebraic form of the List to display many functional ideas.

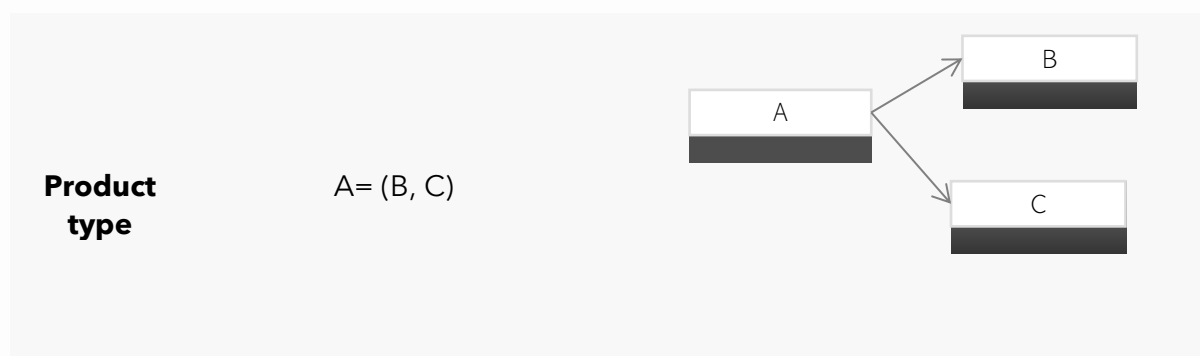
3 Algebraic Data Types

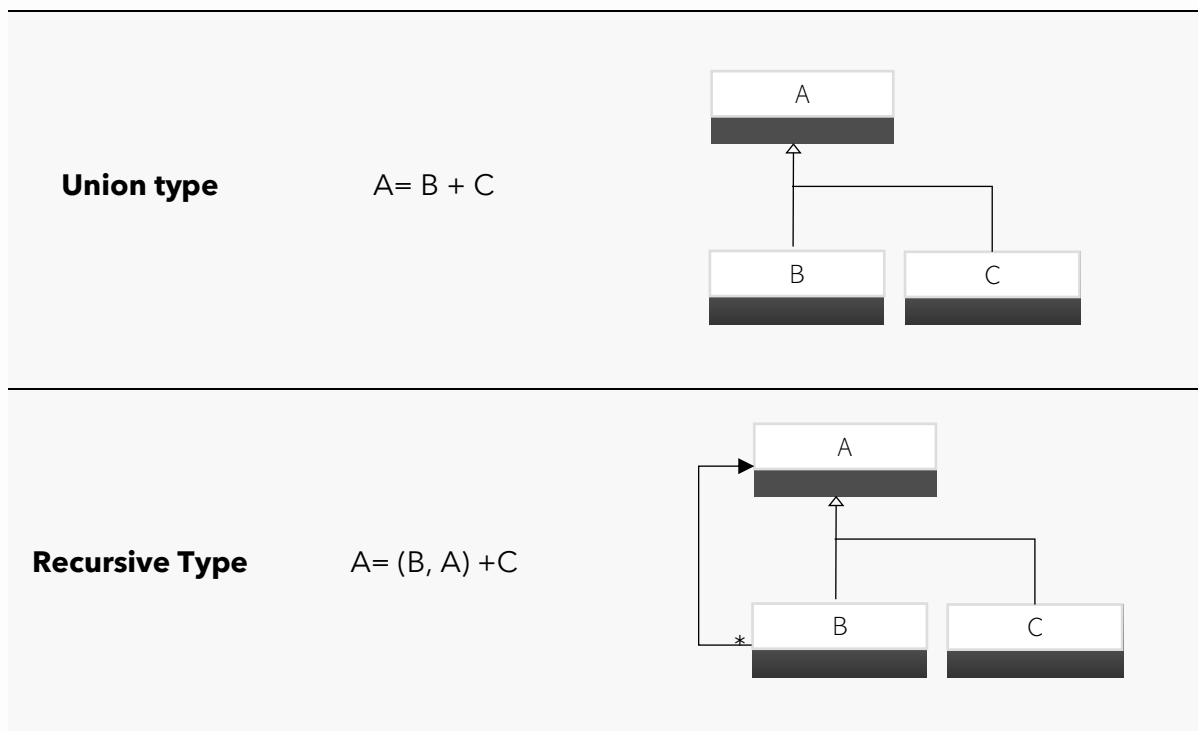
As we have said in the lambda calculus introduction, "It is common in math to give recursive definitions for things." In practical terms giving recursive definitions means that we can progressively compose more complex entities out of simpler objects by applying some operations between them. The following three ways to combine structures are the most pervasive in programming.

1. **Product** [object-oriented equivalent: **composition**]: In our definition of the list the object literal that has two properties is a product (`{ value: value, rest: rest }`)
2. **Coproduct** [object-oriented equivalent: **inheritance**]: In our definition of the list, `Cons extends List`, and `Empty extends List` form a co-product, commonly called a union type. In object-oriented terms, simple inheritance is a union type.
3. **Recursive Type** [object-oriented equivalent: **a type that contains objects of the same Type or inheriting type**]: The fact that in `Cons = (value, rest) => ({ value: value, rest: rest })` the `rest` is a `List` Type means that we have a recursive definition

The fact that a data type can be formed by applying some operations between them gives them the name **algebraic**. An **algebraic data type** is a kind of composite type.

Now, if we define those three operations in any language or programming paradigm, we can define a list by following the algebraic definition "A list is **either** an **empty list** or a **concatenation** of an **element** and a **list**." . If someone tells us how to form a product a coproduct and a recursive type in Java or python or Elm we can instantly write a definition of a list without the need to be experts on the language.





3.1 The product structure:

The product of a family of objects is the "most general" object which admits a morphism to each of the given objects. The product must be both A and B and must be the least thing containing both those elements.

We can form products using the available javascript syntax in many ways, the following are products or arity-2 formed by two objects (arity is the number of items) but there can be products of any size

```
var product = (x,y)=> [x,y]
var product = (x, y) => ({ first: x, second: y });
var product = (x, y) => ({ x, y });
```

because $(x, y) \Rightarrow \{ \{ x, y \} \}$; we can treat the tuple (x,y) as a product (they are isomorphic /contain the exact same information). If a function has a tuple as input, we can rewrite it like this (using the JavaScript destructuring assignment feature)

```
var f = (x, y) => { console.log(x) };
var f = ({ x, y }) => { console.log(x) };
function f({ x, y }) { console.log(x) }

f({ x: 4, y: 5 }) // print 4
```


This transformation commonly called **Introduce Parameter Object** refactoring move. And it is based on the fact that both $(,,,,)$ and $(\{,,, \})$ are products. The most important law in order to consider something as a product is that we must be able to write two functions that will take a product and gives us the component parts. Those functions are called **projections**.

$$\begin{array}{ccccc} A & \xleftarrow{\quad\quad\quad} & A \otimes B & \xrightarrow{\quad\quad\quad} & B \\ & \text{first} & & \text{second} & \end{array}$$

For example, for an array definition of a product $[,]$ the projections would be

```
var product = (x, y) => [x, y]

var first = product => product[0];
var second = product => product[1];
```

for $(\{first: x, second: y\})$ definition of a product the projections would be

```
var first = product => product.first;
var second = product => product.second;
```

I hope you got the idea. Someone could think that a legal definition of a product would also be:

```
var _product = (x, y, z) => [x, y, z]
```

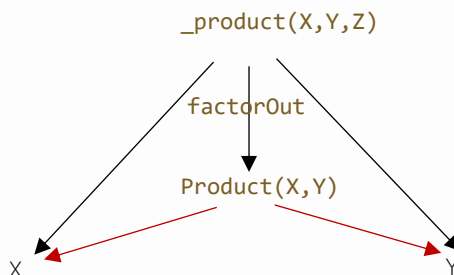
Since `first`, `second` work perfectly fine for this definition as well. The thing is that this definition contains more redundant information, which is not necessary. In essence, this is not minimal.

In order to include this idea of minimality, a more formal definition in category theory says that if there is any other definition of a product `_product`, then there must be a morphism from `_product => product` that factors out the redundant information.

```
var product = (x, y) => [x, y];
var _product = (x, y, z) => [x, y, z];

var factorOut = _product => [_product[0], _product[1]]
```

we cannot go the other way since there is information regarding the z that we cannot infer.



This way to define a property is called universal construction. The universal construction is a way of the mathematicians to say that something is minimal in some sense.

3.1.1 Optional Introduction / Elimination

The Product is the equivalent to a logical AND or a conjunction $\mathbf{A \wedge B}$ of two propositions A and, B. In order to conclude that the proposition $\mathbf{A \wedge B}$ holds we must know that both A and B are true. Or equivalently if we can prove A and prove B then we can Prove $\mathbf{A \wedge B}$. This rule is commonly named **introduction** in mathematical logic because it introduces a conjunction $\mathbf{\wedge}$ out of the previous propositions

$$\frac{A \quad B}{A \wedge B}$$

In the same spirit, in order to create a product of a type A and a type B then we must have **an instance of both types** in order to call the constructor of product `var product = (x, y) => _`. This equivalence originates from the Curry-Howard isomorphism, which is probably the most profound connection of computer programs and mathematical proofs. Going back to the logical conjunction $A \wedge B$ if we have an $A \wedge B$ then we can deduce any of the A or B. This is called conjunction elimination. From $\mathbf{A \wedge B}$ the following hold $\frac{A \wedge B}{A}$ and $\frac{A \wedge B}{B}$ or the propositions $\mathbf{A \wedge B \rightarrow A}$, $\mathbf{A \wedge B \rightarrow B}$ are tautologies (meaning are always true). In terms of programming, this translates to the two projections `first`, `second` meaning we can deduce any of the components of a product if we have an instance of a product.

```

var first = product => product.first;
var second = product => product.second;
  
```

A product is "costly to construct" since we need all components but easy to use since by taking any of the components through a projection (`first`, `second`...) would be valid.

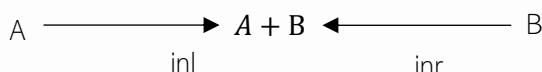
Sanctuary.js: `pair` type `S.Pair` ('js') (6)

Ramda : `R.pair`, `R.pair('js', 6); //=> ['js', 6]`

3.2 The co-product (aka Union) structure:

The coproduct (or either-or sum type or $+$) is the dual structure of the product. The coproduct of a family of objects is essentially the "least specific" object [that is in linguistic terms an attempt to describe something similar to all of the objects it sums] to which each object in the family admits a morphism. It is the category-theoretic dual notion to the categorical product, which means the definition is the same as the product but with all arrows reversed.

if a type S can be any of a set of types $\{A, B\}$ then we can say that S is a sum type of A, B or **$S=A+B$** unfortunately (or fortunately) in javascript we can assign anything to anything since javascript is not strongly typed. In a strong typed language though in order to assign any of **A, B to a variable S the A and B should be a Subclass of S** . This definition of a coproduct can be derived from the definition of the product if we reverse the arrows in the diagram.



Instead of projection the coproduct is defined by its **injections inl (for inject left) and inr (for inject right)**. There must be two functions that provide inclusion to the coproduct. In order to create a coproduct we can provide **any** of the Types it sums in contrast to the product where we need **all** of them to call the constructor.

```
class Coproduct { } // A+B
```

```
class A extends Coproduct {
  constructor( ) {
    super();
  }
}
```

This is the one Injection. That allows us to create something of type `Coproduct`

```
var AorB: Coproduct = new A();
```

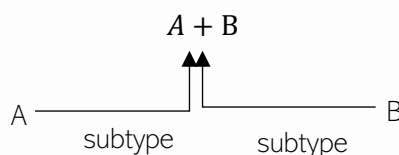
```
class B extends Coproduct {
  constructor() {
    super();
  }
}
```

This is another Injection. That allows us to write

```
var AorB: Coproduct = new B();
```

In order to create a coproduct, we can provide **any** of the Types it sums in contrast to the product where we need **all** of them to call the constructor.

If we bend the arrows a little and name injections to subtype this looks like a valid diagram resembling the definition of coproduct



If we define as $<$: the notion of subtype, that is if $A < B$, then A can be used in place of B instead of B, then $<$: this is a valid coproduct. In Javascript terms, a valid subtype scheme is $B < A$ if $A = \{x_1, x_2, x_3, \dots, x_n\}$ and $B = \{x_1, x_2, x_3, \dots, x_m\}$ where the top class has less properties $n < m$ from the subtype object.

The way to remember that is that is **always good to pass something that has more properties in a function because there is no possibility to try to access properties that do not exist.**

That's way the Rectangle should be a Subtype of the Square and not the opposite

For example lets say we have two objects where they have both x,y,z properties but B also has w,u

```
var A = ({ x: true, y: 1, z: "s" })
```

```
var B = ({ x: false, y: 1, z: "s", w: 3, u: "u" })
```

By adding properties (where same name properties are of the same type), we can treat the object with more properties as a subclass. By our definition, we must be able to replace a B with an A in a function like below without having a thrown exception.

```
var doSomethingWithA = function (type) {
  return type.x ?
    type.y.toString() :
    type.z
}

var r = doSomethingWithA(A); //valid
var r = doSomethingWithA(B); //valid also since B<A

var doSomethingWithB = function (type) {
  return type.x ?
    type.w.toString() :
    type.u
}

var r = doSomethingWithB(A); //invalid try to access w,u
```

```
//throws :Cannot read property 'toString' of undefined
```

Run This: [Js Fiddle](#)

This definition of subtyping, often seen as the Liskov substitution principle, became famous as the L letter of the SOLID principles.

So if we have a set of types { A,B, C...Z} we can take as a Supertype S a type that has as properties the union of all the properties of all types { A,B, C...Z} thus $A <: S$, $B <: S$... $Z <: S$.

Another way to discriminate would be to add a tag to each type

```
var Cons = (value, rest) => ({tag:"Cons", value: value, rest: rest})
var Empty = () => ({tag:"Empty"})

var multiply = list => list.tag== "Empty"?1 :list.value*multiply(list.rest);
var list = Cons(2, Cons(2, Empty ()));
var product = multiply(list);
```

Run This: [Js Fiddle](#)

Or by using instanceof if you are using proper classes definitions.

```
class List { }
class Cons extends List {...}
class Empty extends List { }

var list = new Cons(2, new Cons(2, new Empty()));
multiply = list => (list instanceof Empty) ? 1 : list.value * multiply(list.rest);
```

Run This: [Js Fiddle](#)

Type Theory

Liskov Substitution Principle

Barbara Liskov described the principle in a 1994 paper as follows:

«*Subtype Requirement*. Let $\phi(x)$ be a property provable about objects x of type T . Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T .»

This requirement defines a subtype relation as we saw previously $S <: T$ that can be called Behavioral Subtyping.

If $\phi(x)$ where $x: T$ then if $S <: T \Rightarrow \phi(y)$ should also hold for all $y: S$

Behavioural subtyping is a stronger notion than typical subtyping of functions defined in type theory, which relies only on the contravariance of argument types and covariance of the return type

3.2.1 Optional Introduction / Elimination

Co-Product is the equivalent to a logical OR or disjunction $A \vee B$ of two propositions A, B. In order to conclude that the proposition $A \vee B$ holds we must know that Either A or B is true. Or equivalently if we can prove A **or** prove B then we can Prove $A \vee B$. This rule is commonly named disjunction **introduction** in mathematical logic because it introduces a disjunction \vee out of the previous propositions $\frac{A}{A \vee B}$ or $\frac{B}{A \vee B}$

in the same spirit in order to create a coproduct of a type A and a type B then we must have **an instance of Any** in order to make an assignment `var unionType = subtypeA`. Going back to the logical OR - $A \vee B$ if we have an $A \vee B$ then we cannot deduce any of the A or B because we cannot know which one was initially injected to the Union type. But we can eliminate disjunction in this way $\frac{A \vee B \quad A \rightarrow C \quad B \rightarrow C}{C}$ Which mean if we have a rule that gives us a C from an A ($A \rightarrow C$) **and** a rule that gives us a C from a B ($B \rightarrow C$), **and** we have either of A or B ($A \vee B$) **then** we can deduce C. In programming terms this means that in order to use a function on an object must define the function in all Types of the union.

```
var A = (x, y, z) => ({ x: x, y: y, z: z, display: () => x + y + z })

var B = ({ x: false, y: 1, z: "s", w: 3, u: "u", display: () => x + y + z + w + u })

var EitherAorBMap = A(true, 1, "s");
console.log(EitherAorBMap.display());
```

This means that a co-product is "easy to construct" since we need any component, but difficult to use since we must discriminate between all of the types contributing to the Union.

Libraries

Union Types

union-type : union - A small JavaScript library for union types.
folktale : union
daggy: A Library for creating tagged constructors, a.k.a. "disjoint union types" or "sum types."

3.3 Extending Union Types

Here I will go one step back at the union types. We will see in this book some ways to add behaviour on Union types as we go along but for now let's see the default way using polymorphism.

As I have mentioned previously, If we want to add a new property/method to a Union of types, we must add the variations of the property/method in all the sub-types that can be in the Union. This is the use of default polymorphism in order to discriminate between the parts of the union:

```
class List {
  mult() { throw new Error('You have to implement the method mul!'); }
}

class Cons extends List {
  constructor(value, rest) {
    super()
    this.value = value;
    this.rest = rest;
  }
  multiply() { return this.value * this.rest.multiply(); }
}
class Empty extends List { multiply() { return 1; } }

var r = new Cons(2, new Cons(2, new Empty ())).multiply ();
```

Run This: [Js Fiddle](#)

Here I have added `multiply` in both parts of the union. The same structural recursive reasoning of section [3.4](#) applies here while for the definition of the two `multiply()` methods.

3.3.1 Adding Pattern Matching extensions to Union Types

Now let us add a new method called `matchWith`, this also sometimes can be found as `Case` or `Cata` or even `Fold`. We will see the distinctions latter but

```
class List {
  //consider this as an abstract method
  matchWith(pattern){
    throw new Error('You have to implement the method matchWith!');
  }
}

class Cons extends List {
  constructor(value, rest) {
    super()
    this.value = value;
    this.rest = rest;
  }

  matchWith(pattern) { return pattern.cons(this.value, this.rest); }
}
```

```
class Empty extends List {
  matchWith(pattern) { return pattern.empty( ); }
}
```

Now we can use this method in order to distinguish between the members of the union like this

```
var list = new Cons(2, new Cons(3, new Empty()));

var result = list.matchWith({
  cons: (v, r) => {
    },
  empty: () => {
    }
})
```

3.3.2 Rewriting Union Type methods with matchWith

`matchWith` is something like a universal definition that allows us to write Polymorphic method that must be added in all the branches of a union type **in a single place**.

We can now write any method definition in our `List` Union Type using just `matchWith`. Here I rewrite the `multiply` using an extension method:

```
List.prototype.multiply = function () {
  this.matchWith({
    empty: () => 1,
    cons: (v, r) => v * r.multiply()
  })
}
```

If you don't want to use prototype extension methods, you can add the body at the Base class like this:

```
class List {

  multiply() {
    this.matchWith({
      empty: () => 1,
      cons: (v, r) => v * r.multiply()
    })
  }
}
```


As another example let's generalize/abstract the multiply to a `fold` method by passing a monoid as a parameter:

```
List.prototype.fold = function (monoid) {
  this.matchWith({
    empty: () => monoid.empty(),
    cons: (v, r) => monoid.concat(v, r.fold(monoid))
  })
}
```

Here you can see that the `matchWith` pattern `{empty: ()=>{}, cons:(v,r) => {}}` looks very similar with the interface of the monoid `{empty:()=>{}, concat:(a,b) => {}}` we will see this inception again when we talk about catamorphisms extensively, non the less list has a special role because as stated in the quote at the beginning of this section from the functional pearl "Monoids: Theme and Variations":

«Lists come up often when discussing monoids, and this is no accident: lists are the "most fundamental" Monoid instance. »

3.3.3 Pattern matching in functional libraries

For example, using the union type of the folktale library we can extend the algebraic definition of a List like this :

```
var List = union('List', {
  Empty : () => {},
  Cons: (value, rest) => ({ value, rest })
});

var { Empty, Cons } = List;

List.mult = function() {
  return this.matchWith({
    Nil: () => 1,
    Cons: ({ value, rest }) => value * rest.mult()
  });
};

Cons (1, Cons (2, Empty ())).mult();
```

Run This: sourcecode

if we use the union-type library, we can use the support for recursive union types, and write the same thing as follows :

```

var List = Type({ Empty : [], Cons: [R.T, List] });

var mult = List.case({
  Cons: (value, rest) => value + ' : ' + mult(rest),
  Empty : () => Empty ,
});

var list = List.Cons(1, List.Cons(2, List.Cons(3, List.Empty )));
console.log(mult(list));

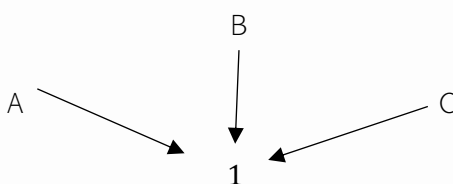
```

Run This: Js Fiddle

3.4 Optional One

If we defined $+$ and \bullet maybe, we could define a 1 item that will be consistent with our usual algebra rules of $a \bullet 1 = a$ for javascript Arrays if we assume \bullet is the concat operation then 1 should be $[]$ in order to hold $a.concat([]) = a$. The term **1** is called the terminal object of a category. $[]$ is the terminal object of the Array category.

Terminal object: The terminal object is the object with one and only one morphism coming to it from any object in the category.



In the category of Lists the terminal object can be the empty list $[]$ because we can get a function to the empty list from any other list if we drop all the elements. However, we cannot have a single function to any other list without any ad hoc specific instructions. In the category of types closed under composition [e.g. If we get the product of an integer $a=1$ and a Boolean $b=true$ we could get a new object $(\{a:1\}) \otimes (\{b:true\}) \rightarrow (\{a:1, b:true\})$] then the terminal object must be $\{\}$. It is not a very interesting idea, but its a display of the consistency of algebra across different domains. For example for the algebraic definition of a list, we have that

$$x = 1 + a \bullet x$$

we can replace the x at the left part with the definition and get:

$$x = 1 + a \bullet (1 + a \bullet x) = 1 + a + a \bullet a \bullet x$$

and if we keep doing this, we get

$$x = 1 + a + a \bullet a + a \bullet a \bullet a \bullet x \dots$$

Which says that a list can be either empty $1 = []$ or just an element $[a]$ or two elements $[a, a]$ or three elements $[a, a, a]$, and so on, which is very consistent with the definition of the list. Also, we could try to solve it like this:

$$x \cdot (1 - a) = 1 \Rightarrow x = 1 / (1 - a)$$

However, since we do not have a definition of a division, we can use Taylor expansion on $1/(1-x)$, which is expanding to the geometric series $1 + a + a^2 + a^3 + a^4 + \dots$

The consistency of the algebra is remarkable.

3.5 Recursive Algebraic Types

The fact that our list is a recursive type means that when we try to construct a new method for our type, we must give an inductive definition (or recursive definition). So, let me create a map function that will apply a function f to all the values of the list:

```
var Cons = (value, rest) => ({
  value: value,
  rest: rest,
  map: f => { ... }
})
```

```
var Empty = () => ({
  map: f => {...}
})
```

1. Firstly, because it is a coproduct, we must define the **map** in **both subtypes of the coproduct**. We start with the **Empty**, which is the easiest. **Empty** is the base case of the structural induction. We know that the map must return something of the same type, so we will return a new **Empty** since there is nothing else that we could do with f .

```
var Empty = () => ({ map: f => Empty() })
```

we can represent that symbolically with $\text{map } f([]) = []$

2. Now, let us move to **Cons**. **Cons** is a product (**value**, **rest**) the **value** is something concrete so we can apply the f directly and get the lifted value $f(\text{value})$. The second part **rest** is a list, and the only thing we know is that it must have a **map** function working as we expect. This claim constitutes the inductive hypothesis, so we assume that **rest.map(f)** returns a lifted list for the **rest**. We must finally return something of the same type, so we return a new **Cons**

```
map: f => Cons(f(value), rest.map(f))
```

we can represent that symbolically:

```
map f ([value, rest]) = [ f(v), map f (rest)]
```

Furthermore, that is the correct implementation-defined recursively.

```
var Cons = (value, rest) => ({
  value: value,
  rest: rest,
  map: f => Cons(f(value), rest.map(f))
})
```

```
var Empty = () => ({
  map: f => Empty()
})
```

```
var list = Cons(2, Cons(2, Empty()));
var liftedList = list.map(x=>x+2)
```

Run This: [Js Fiddle](#)

3.5.1 Rewriting map with matchWith

Let us again rewrite this using the matchWith method instead of polymorphism

```
List.prototype.map = function (f) {
  return this.matchWith({
    empty: () => new Empty(),
    cons: (v, r) => new Cons(f(v), r.map(f))
  })
}
var list = new Cons(2, new Cons(3, new Empty()));
var listLifted = list.map(x=>x+1)
```

3.5.2 On the value of the symbolic representation

In its most general form, algebra is the study of mathematical symbols

and the rules for manipulating these symbols

-Wikipedia:[Algebra](#)

You might want to ignore the symbolic representation, and focus on the code, or you can try to write your own **map** equation by induction. Let me write some more map definitions that i just came up with right now:

1. here I just added a \rightarrow sign to point the term mapped, and I replaced concat with * the multiplication

```
map(f)  $\rightarrow$  ([]) = []
```

```
map(f)  $\rightarrow$  ([value * rest]) = [ f(v) * map(f) $\rightarrow$  (rest)]
```

2. or look at this one I replaced fmap f with an underlining $\underline{\quad}_f$ because why not:

```
([])f = []
```

```
([value, rest])f = [ f(v), ([rest])f]
```

We can come up with a hundred different ways to represent the same thing. Thinking in graphical-symbolic terms while trying to figure out the mechanics behind some of the functional concepts, has been very valuable for me and I think it can help you too. Please take some minutes to write down your own map(f) equations for the list, on a piece of paper. The ability to abstract an idea into a symbolic representation allows us to move between paradigms and get a deeper understanding of the mechanics.

3.5.3 Adding Pattern Matching extension to native array

We will now add a `matchWith` extension on the `List<T>` type **in order to be able to use recursive definitions:**

```
Array.prototype.matchWith = function (pattern) {
  if (this.length == 0) {
    return pattern.empty();
  }
  else {
    return pattern.cons(this[0], this.slice(1));
  }
}
```

This definition will allow us to use pattern matching like syntax for `Array`

```

Array.prototype.fold = function (monoid) {
  return this.matchWith({
    empty: () => monoid.empty(),
    cons: (v, r) => monoid.concat(v, r.fold(monoid))
  })
}

var sum = [1,4,5,6,7].fold({ empty:()=> 0, concat: (x, y) => x + y })

```

In functional languages is very common to write definitions by simply pattern matching. Look at the Haskell definition of list-map:

```

map _ [] = []
map f (x:xs) = f x : map f xs

```

this is the way we defined it in section [3.5 of recursive algebraic types](#)

```

([],).map(f) = []
([value , rest]).map(f) = [ f(v), ([rest]).map(f) ]

```

As we saw this allow us to rewrite map using pattern matching with `matchWith`

```

Array.prototype.map2 = function (f) {
  return this.matchWith({
    empty: () => [],
    cons: (v, r) => [f(v),...r.map2(f)]
  })
}

var squared = [1,4,5,6,7].map2(x=>x*x);

```

When you start rewriting code using pattern-matching with `matchWith` with lists gives an insight on the functional mindset. Let's see how to zip two Lists. In an imperative style we would have a for loop that

```

for (let index = 0; index < a1.Length; index++) {
  const element1 = a1[index];
  const element2 = a2[index];
  zipped.Add(element1);
  zipped.Add(element2)
}

```

In this situation I assume that the arrays are of the same length. If this is not true, we must use more `if (...)` statements. Now if we see our arrays in the form `[x:xs] [y:ys]` this reveals a very different perspective. The general case where `zip([x:xs],[y:ys]) = [x,y:zip(xs,ys)]`

```

Array.prototype.zip = function (a2) {
  return this.matchWith({
    empty: () => a2,
    cons: (x, xs) => a2.matchWith({
      empty: () => this,
      cons: (y, ys) => [x, y].concat(xs.zip(ys))
    })
  });
}
console.log([1, 2, 3].zip([6, 7, 8, 7, 9, 10, 11])); // [1, 6, 2, 7, 3, 8, 7, 9, ...]

```

Run This: [js Fiddle](#) Run This: [TS Fiddle](#)

Also, we could generalize this instead of just having the elements that form a pair glued into a new type $T1$ that can be anything product like as long we provide a custom function f that does the gluing

```

Array.prototype.zipMap = function (a2,f) {
  return this.matchWith({
    empty: () => a2,
    cons: (x, xs) => a2.matchWith({
      empty: () => a1.map(x => f(x, null)),
      cons: (y, ys) => [f(x, y)].concat(xs.zipMap(ys, f))
    })
  });
}

console.log([1, 2, 3].zipMap([6, 7, 8, 7, 9, 10, 11],(a,b)=>[a,b])); // [[1,6], [2,7], [3,8],
, 7, 9, 10, 11]

```

Run This: [TS Fiddle](#)

For example `[1, 2, 3].zipMap([6, 7, 8, 7, 9, 10, 11],(a,b)=>[a,b])` here I used tuples to combine items so the result would be `//[[1,6],[2,7],[3,8],[_,9],[_,10],[_,11]]`

This is a perfectly nice fold. You can use some other stuff, any monoid for example would do.

In the following table, there is a summary of a progression of algebraic data types from simplest to more complex

Zero => numerals=>Lists=>Trees=>..

Algebraic data Types

Name

Signature

Natural numbers

[Peano numerals]

```
var Succ = (rest) => ({ rest: rest })
var Zero = () => ({})

var four = Succ(Succ(Succ(Succ(Zero()))));
```

List

```
var Cons = (value, rest) => ({ value: value, rest: rest, })
var Empty = () => ({})

var list = Cons(2, Cons(2, Empty()));
```

Tree

```
var Node = (left, right) => ({ left: left, right: right, })
var Leaf = (v) => ({ v: v })

var list = Node(Leaf(3), Node(Leaf(5), Leaf(7)));
```

Sidenote:

Those algebraic data structures can become more and more complex but always have a terminating condition represented by the Zero, Empty, Leaf etc (which are the initial items on the respective algebras). On the opposite side, there are the so-called co-inductive or co-recursive data structures, which are the dual constructions and usually are infinite. Unfortunately, those data structures will not be discussed in this book extensively in order to keep it simple. You can find more on the extended version of this book at Leanpub.

lambda

Church encoding

In mathematics the **Church numerals** are a representation of the natural numbers using lambda notation. The method is named for Alonzo Church, who first encoded data in the lambda calculus this way.

$0 \cong \lambda f. \lambda x. x$ $=> x$ $=> \text{Zero}()$

$$1 \cong \lambda f. \lambda x. f(x) \quad \Rightarrow f(x) \quad \Rightarrow \text{Succ}(\text{Zero}())$$
$$2 \cong \lambda f. \lambda x. f(f(x)) \quad \Rightarrow f(f(x)) \quad \Rightarrow \text{Succ}(\text{Succ}(\text{Zero}()))$$

This notation of numerals is one of the simplest recursive algebraic structure that terminates besides just `Zero()`.

Peano numerals: Church encoding is an implementation on Lambda of the Peano numerals idea of representing the Natural numbers with the use of a zero and a successor function.

Functors

«It should be observed first that the whole concept of a category is essentially an auxiliary one; Our basic concepts are essentially those of a functor and of a natural transformation.»

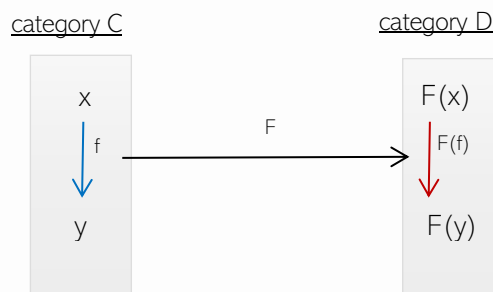
S. Eilenberg and S. MacLane »

The Idea:

In Javascript the most famous functional programming idea is to use `array.map` to replace iterations instead of *for loops* in order to transform the values of the array. That is because an array is a Functor, which is a more abstract idea that we will explore in this section. Functors can be considered the core concept of category theory.

4 Functors

In mathematics, a **functor** is a map between categories.



This Functor F must map two requirements

1. map each object x in C with an object $F(x)$ in D ,
2. map each morphism f in C with a morphism $F(f)$ in D

For object-oriented programming, the best metaphor for functors is a **container**, together with a **mapping** function. The Array as a **data structure** is a Functor, **together** with the **map** method. The **map** is the array method that transforms the items of the array by applying a function f .

4.1 The Identity Functor

We will start by looking at the minimum structure that qualifies as a functor in javascript:

```
const Id = (v) => ({
  map: (f) => Id (f(v))
});
```

that's requirement 1

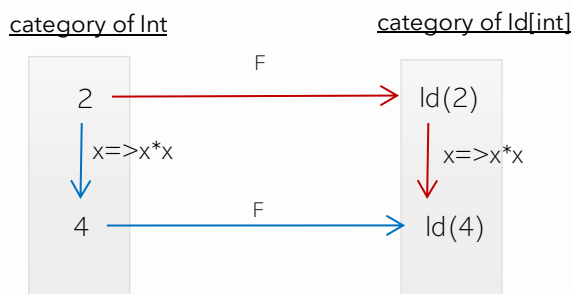
that's requirement 2

Run This: [Js Fiddle](#)

This is the minimum construction that we could call a functor because it has exactly two things

1. A “**constructor**” that maps a value v to an object literal `Id = (v) => {value: v}`
2. and it has a **mapping** method `map:(f) => { }` that lifts functions f

Because it's the minimal functor structure it goes by the name **Identity functor**. Let us see a simple example where we have two integers 2 and 4 (here we take for simplicity the category of integers as our initial category C) also in this category there is the function `square = x=>x*x` that maps 2 to 4.



If we apply the `Id(_)` constructor we can map each integers to the `Id[int]` category. For example 2 will be mapped to **Id(2)** and 4 maps to **Id(4)**, the only part missing is the correct lifting of the function f **Id[f]** to this new category. It's easy to see that the correct mapping is:

```
this.map = (f) => Id(f(value));
```

Because:

```
var square = x=>x*x;
```

```
Id(2).map(square) = Id(square(2))
```

The type of the functor map method is

$\text{map}: (a \rightarrow b) \rightarrow f(a) \rightarrow f(b)$

this means that if you give me a function from a to b ($a \rightarrow b$) and I have an $f(a)$, I can get an $f(b)$

In order to cover the first requirement that a functor should map each object x in C with an object $F(x)$ in D we have used a “**constructor**” that maps a value v to an object literal `Id = (v) => { }`. In many functional libraries online you may find the explicit definition of an `.of()` that does the same thing

```
Id.of = v => ({ v: v, map: f => Id.of(f(v)) })
```

Run This: [Js Fiddle](#)

4.2 Commutative Diagrams

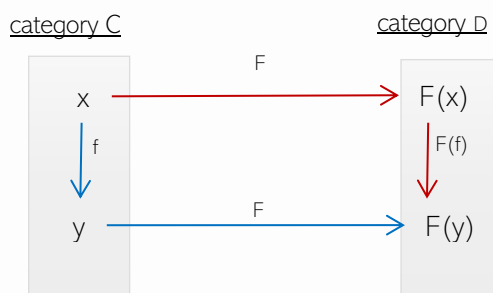
There is one important thing about the mapping function, though. The mapping should get the same result for [4] if we take any of the two possible routes to get there. This means **map (aka lifting of a function from C to D) should preserve the structure of C.**

1. We could first get the Functor and then map it. This is the red path on the diagram.
`functor(y) = functor(x).map(f);`

2. Or first lift x with f and then get the Functor.
`functor(y) = functor(f(x));`

Two objects that were connected with an arrow in category C, should also be connected with an arrow in category D. And reversely, if there was no connection in C there should be no connection in D.

When the red and blue paths give the same result, then we say that the diagram **commutes**



Moreover, that means that the lifting of morphisms (aka arrows, aka functions in programming) preserves the structure of the objects in C.

In practical day to day programming, commuting diagrams means that **we can exchange the order of operations and still get the same result.** It is not something that happens automatically and is something very helpful to have when coding.

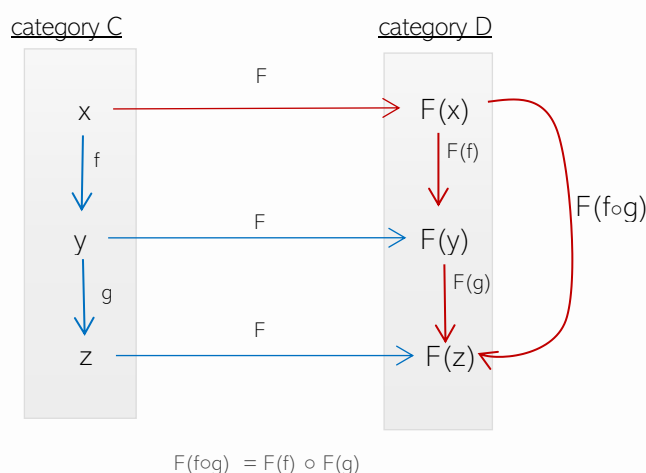
4.3 The Functor Laws

Every data structure has not just a signature, but some expected behaviour. For example, a Stack has a peek, push and a pop operation that provide a Stack functionality, and we expect these operations to behave in a certain way:

1. for all x and s , it holds that $\text{peek}(\text{push } x \text{ } s) = x$
2. for all x , it holds that $\text{pop}(\text{push } x \text{ empty}) = \text{empty}$
3. etc

The laws for a given data structure usually follow from the specifications for its operations, as do the two examples laws given above. Most of the constructions we are going to discuss in this book come with laws. Any functor must obey two laws when lifting the morphisms:

1. $F(\text{id}_X) = \text{id}_{F(X)}$. This means that the functor **preserves the Identity**.
The functor should map the id function $x \Rightarrow x$ correctly
2. $F(f \circ g) = F(f) \circ F(g)$ This means that the functor **preserves the composition of functions**.



Again, those two laws seem that are mathematically strict, but in real software development, those two conditions express a **well-designed map method that behaves as someone would expect it to behave**.

As we have seen in the section discussing currying, we expected that those two expressions should give the same result:

```
[10, 20, 30].map(compose(curry(discountedPrice)(0.1), toDiv)) ;
[10, 20, 30].map(curry(discountedPrice)(0.1)).map(toDiv);
```

More generally for all f and g the following should hold:

```
// Law 1-identity preserving fmap id = id
Id(value).map(x=>x).value ≡ (value)

// Law 2-composition of functions is preserved fmap (f . g) == fmap f . fmap g
functor(value).map(x=>f(g(x))).value ≡ functor(value).map(f).map(g).value
```

Run This: [Js Fiddle](#)

imagine for the following example your surprise if you got different results for these two statements for our identity functor:

```
var f = x => x.toUpperCase();
var g = x => `Title: ` + x;

console.log(functor(value).map(x => g(f(x))).value); // "Title: FUNCTOR LAWS"
console.log(functor(value).map(f).map(g).value); // "Title: FUNCTOR LAWS"
```

Run This: [Js Fiddle](#)

This is a very simplistic example, but it can become easy to get something wrong when code gets complex, especially for functors that are not as familiar as the Array.

How easy is to make a lawless functor? Just add for examples a validation inside the map:

```
const Id = (v) => ({
  value: v,
  map: (f) => Id (f(v>2 ?0: v))
});
```

← This map does not preserve structure anymore

Run This: [Js Fiddle](#)

by the way, I hope the composition law $F(f \circ_f g) = F(f) \circ_F F(g)$ reminds you of the Homomorphism equation $\phi(a_1 * a_1) = \phi(a_1) \bullet \phi(a_1)$ because that is what it is. Homomorphisms are everywhere

4.4 A brief mentioning of *Catamorphisms*

Here we will make a stop to have a brief look at **Catmorphisms** in order to use them in our examples. There will be an in-depth examination of the concept of catamorphisms in a later chapter.

After we wrap a value in a functor and manipulate it with the map, **eventually, we want to get the value out of the functor** elegantly and consistently. Until now, we could directly access the value because we had stored it explicitly in our object literal:

```
var Id = v => ({ v: v, ...});
```

Not very pretty. Especially when everyone can access it and mutate it. As object-oriented programmers, we do not see any problem with that, but it could drive a functional programmer mad.

Nonetheless, exposing the value is always an option **if we do it consciously**. However, there are some more orthodox ways to extract a value from a Functor, and one of them is catamorphisms.

A catamorphism is “collapsing the Functor (container) into a single value.” Some catamorphisms are very easy to implement. For example, how can we extract the value out of an `Id` functor? Provide a method (`cata`) that takes a callback (`alg`) [`alg` is a short name for algebra, hinting the maths behind catamorphisms], to which we pass the value:

```
var Id = v => ({
  v: v,
  map: f => Id(f(v)),
  bind: f => f(v),
  cata: alg => alg(v)
});
```

Cata allows us to do

```
Id(5).cata(console.log) //5
```

`console.log(Id(5).v)` // Instead of this

Run This: [Js Fiddle](#)

that allow us to just get the value, we can pass the identity function inside

```
var value = Id(3).cata(x=>x);
```

or we can just create an “overload” for `cata` (but we cannot have an actual overloads in Js)

```
cata: () => v
```

We could leave it at that or rename to `Fold` which is the most common convention

```
fold: () => v
```

```
var value = new Id(3).fold();
```

[the standard convention for a fold is to have a type signature like this (using TypeScript in order to be able to see the types of the various parts) :

```
fold<TM>(acc: TM, reducer: (v: T, acc: TM) => TM){ return reducer(this.Value, acc); }
```

which is an overkill for the simple `Id` functor]

Because the `Id` is very simple almost all folding concepts pretty much coincide. We will see the difference between them at the Traversable chapter where we will have to deal with more complex data structures.

The implementation of `cata` is usually quite straightforward for the functors that we are going to see in this chapter.

4.5 Id Functor on the Fly

We can use extension methods to allow the creation an `Id<T>` from a value `T` and use the `Map` immediately and then `Fold`.

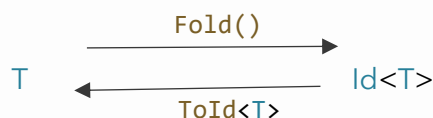
```
Object.prototype.toId = function () { return Id(this);}
```

And thus write:

```
(5).toId().map(x=>x+2).cata(console.log);
```

Is this useful? maybe, maybe not. Libraries like `loadsh` became famous for just wrapping objects and then allowing for chaining capabilities. Nonetheless it signifies a way of writing, that is very common in Functional programming by using chaining and transformation methods like `.toId()` in order to move from one data structure into another.

The `Id` and the `T` are **isomorphic**. They contain the exact same information and thus we can go back and forth without losing any information or needing any additional information by using the methods `ToId<T>` and `Fold()` (which are inverses in a sense)



throughout this book we will see many isomorphisms. Those **isomorphisms in the case of Functors are called Natural Transformations**. We will see them in more detail in a latter chapter. For now, we are going to see four more isomorphisms (`Task`, `Func`, `Lazy`, `IO`) with the plain value `T` and thus provide you with a unifying abstraction when dealing with them.

4.6 Extending Promise to Functor

In the following sections, we will see some examples of other popular Functors in javascript. We will start by extending `Promise`, by providing a **map** method that obeys the functor laws and thus promote native `Promise` into a functor.

[**Sidenote:** Native implementation of promises (based on the Promises/A+ specification) supports a kind of **map** achieved by overloading the `then` in order to transform a resolved value into a new promise:

```
var thenableMapResult = new Promise((resolve,reject)=>resolve(5))
  .then(x=>x+3)
  .then(console.log)//8
```

Run This: [Js Fiddle](#)

This overload of `then` often creates confusion, and problems.]

We have said that the usual metaphor for a functor is “a container.” A Promise can be seen as a container that takes a value and wraps it, until it is resolved. In order to promote Promise to Functor, there must be a **mapping** function that would be able to lift any function and give a new Promise with the lifted value.

Here is one possible mapping function that preserves structure:

```
Promise.prototype.map = function(f) {
  var initialPromise = this;
  return new Promise(function(resolve, reject) {
    initialPromise.then(result => resolve(f(result)))
      .catch(reject);
  });
}
```

Run This: [Js Fiddle](#)

The mapping function that lifts the function $f: \text{int} \rightarrow \text{int}$ It follows the steps:

1. Waits for the result of the Promise (so in a way unwraps the contained value)
`initialPromise.then()`
2. Applying the function $f : \text{result} \Rightarrow \text{resolve}(f(\text{result}))$
3. wraps the resulting value again into a new Promise : `return new Promise()` because When we implement a **map**, we always return something of the same type in order to belong to the same category (in this case type) or Promise.

Keep those steps in mind because they are common in the implementation of many Functors and monads.

! Something worth pointing out here is the `.catch(reject)` **this part belongs to the path of failure and ignores the f mapping**. We are going to come back to this observation many times throughout this book.

We can use our map function now:

```
var fetchClientsMock = () => new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(
      [{ id: 1, name: 'jim', age: 29 },
        { id: 2, name: 'jane', age: 25 }])
  }, 1000);
})
```

```
var clientRepository = ({
  getById: (id) =>
    fetchClientsMock()
      .map(clients =>
        clients.filter(c => c.id == id))//promise of a filtered array matching id
});
```

```
var clientNameById=id => clientRepository
  .getById(id)
  .map(x => x[0].name); //might throw exception if array is empty
                        we will deal with this in the next section
```

```
clientNameById(1)
  .then(x => console.log(`the client name is ${x}`))
  .catch(e => console.log(`error getting clients ${e}`))
```

Run This: [Js Fiddle](#)

The cata for the Promise is the `(then, catch)` set. Each takes a callback and applies the value of success or failure respectively. Compare this to the `Id` `cata` method implementation of the previous section.

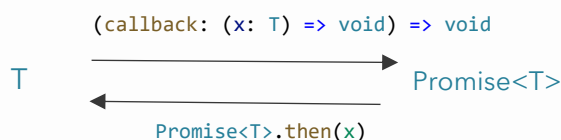
Sidenote:

Obviously, the `Promise<T>` is isomorphic with the `T`, we have `Promise.resolve(5)` which means every value can be a promise `T→Promise<T>`. The reverse morphism from `Promise<T>→T` is different because `Promise <T>` and `T` live in different time frames. But we know from a theorem called Yoneda lemma that those must be considered isomorphic . The Yoneda lemma for the simple case of the `Id` functor says that a value `T` is the same as a call-back that takes a `T` for example a value 5 is the same as this expression

```
type Continuation<T> = (callback: (x: T) => void) => void;
```

```
var continuation: Continuation<number> = (resolve => resolve (5));
continuation (x => console.log(x) );
```

The `Promise<T>` in its core is the same with this `(x: T) => void => void`



4.6.1 Optional The Promise - functor laws

Here we only display the laws for the resolve path and not the reject (but we will discuss this at the section of the `either` functor, but the failed path is like a constant Functor that just maps everything to the same object the exception)

```
var id = x=>x;
var value = 5
// Law 1 - the identity is preserved
new Promise((resolve) => resolve(value)).map(id)
  .then(x=>console.log(x===id(value))) // true
```

```
// Law 2 composition of functions is preserved
var f =x=>2*x;
var g =x=>3*x;

new Promise((resolve) => resolve(value)).map(x=>f(g(x))).then(console.log);
new Promise((resolve) => resolve(value)).map(f).map(g).then(console.log);
```

Run This: [Js Fiddle](#)

4.7 IO Functor, a Lazy Id Functor

Another simple functor which extends the identity functor by adding Laziness is the IO functor. The IO is a lazy Id functor. IO comes handy when we want to Objectify a function, pass it around and use it later to produce a side effect as we will see.

FP

Side effects

Side effects are operations that **change the global state of a computation**. Formally, all assignments and all input/output operations are considered side-effects. The *functional programming* style tries to avoid /reduce side effects.

4.7.1 Lazy as Functor

4.7.1.1 Thunks

A **thunk** `()=>{...}` is a subroutine used to inject an additional calculation into another subroutine. Thunks are primarily used to delay a calculation until its result is needed, or to insert operations at the beginning or end of the other subroutine. For example, a very common pattern is to initialize an object only when it is needed. This is called Lazy initialization because if we have a very costly computation that may not be needed, we should not eagerly evaluate it.

```
var bigArrayComputation = [...Array(10e6)].map((_, i) => i).reduce((a, i) => a + i)
MayOrnMayNotDisplay(bigArrayComputation)
```

Instead of eager initialization, we can the creation or evaluation with a factory method that is only performed once if/and when it is needed.

```
var Lazy = function (fn) {
  this.value = () => {
    if (!this.v) { this.v = fn(); }
    return this.v;
  }
}
```

```

    }
  }

  var v = new Lazy(() => [...Array(10e6)].map((_, i) => i).reduce((a, i) => a + i))
  var s = v.value();           //if we use value then the calculation is performed and cached
  var s1 = v.value();          //we use the cached result or memoized as usually called in FP

```

Run This: [Js Fiddle](#)

This Lazy structure is known as Lazy loading design pattern and can be extended to a functor by providing the following method

```

var Lazy = function (fn) {
  this.value = () => {
    if (!this.v) {
      this.v = fn();
      console.log("evaluating the factory " + this.v)
    }
    return this.v;
  }
  this.map = (f) => {
    return new Lazy(() => f(this.value()))
  }
}

var lazy = new Lazy(() => 2).map(x => x + 3);
//the value was not accessed when we used the map method
console.log("still not accessed the value")

var result = lazy.value();// only now the sequence is evaluated
console.log(result)

```

Run This: [Js Fiddle](#)

Another very common design pattern that reifies a function and stores it in an object is the command pattern. The command pattern is often used to store the history of actions and provide undo functionality. Look at the following code

```

var context = {
  value: 0,
  history: [],

  execute: (command) => {
    context.history.push(command);
    command.execute(context)
  },
  undo: () => context.history.shift().undo(context)
};

var addComand = (number) => ({
  execute: (context) => context.value = + number,
  undo: (context) => context.value = - number
})

context.execute(addComand(5)); //context.value==5
context.execute(addComand(5)); //context.value==10

```

```
context.undo();           //context.value==5
context.undo();           //context.value==0
```

Run This: [Js Fiddle](#)

4.7.2 IO Functor

Functional programming in the same lines provides the IO functor

```
var IO = fn => ({
  map: f => IO(() => f(fn())),
  run: () => fn(),
})
```

the map implementation should remind you slightly of the *promise map* line of reasoning while implementing it. We first have to “run” the previous computation `fn()` then use the function `f` to get the lifted value `f(fn())` and then finally rewrap this new value in a new `IO(() => f(fn()))`.

An example application coming from the frontend web design universe uses IO, to isolate the effects of accessing the DOM elements with jQuery

```
var getjQueryElementIO = IO(() => $("#container"));
```

```
var title = getHtmlElementIO
  .map(x=>x.text())
  .map(x.toUpperCase())
  .run()
```

Run This: [Js Fiddle](#)

It's the same as the Id functor but encloses the value that we provide in a thunk `() => { }` just to make it lazy. We would have to collapse the function `fn()` in order to get back the value, most IO implementations around usually provide a `run ()` function that does just that. You can take a look at [monet.js](#) library implementation of IO. Also, with a prototype extension on the Object, we can create IOs out of everything on the fly

```
Object.prototype.toIO = function () { return IO(() => this); }
```

And use it like this:

```
var getjQueryElementIO = $("#container").toIO ();
```

Run This: [Js Fiddle](#)

The **Lazy loading design pattern** (with the map method) is **Isomorphic** to the **IO Functor**

4.8 Reader Functor

The Reader Functor represents a computation, which can read values from a shared environment. The reader functor conceptualizes the idea of having stored expression that we can evaluate in different configurations at a later time. We can represent this like that $((\rightarrow) \text{ env})$, which means we give an environment, and we get out the thing that we want.

```
Reader = (expr) => ({
  expr: expr,
  map: f => Reader (env=>f(Reader (expr).run(env))) ,
  run: env => expr(env),
});
```

```
var reader = Reader (g => g.x1);
var result = reader.map(x=>x*3).run({x1:20});
```

Run This: [Js Fiddle](#)

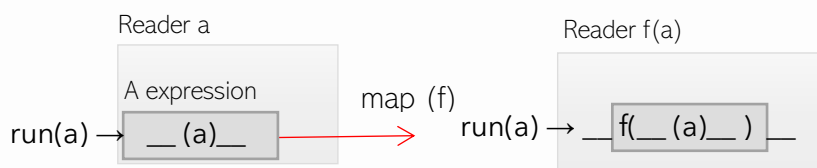
the map here takes a function $(a \rightarrow b)$ and then lifts the initial reader that contains an a $((\rightarrow) e) a$ to a reader that contains a b $((\rightarrow) e) b$

map: $(a \rightarrow b) \rightarrow ((\rightarrow) e) a \rightarrow ((\rightarrow) e) b$

the implementation is more convoluted but again follows the lines of promise and IO

```
map: f => Reader(env=>f(Reader(expr). run(env)))
```

1. Evaluate the inner reader `Reader(expr). run(env)` (so in a way unwraps the contained value)
2. Applying the function $f : f(\text{Reader}(\text{expr}). \text{run}(\text{env}))$
3. wraps the resulting value again into a new `Reader(env=>f(Reader(expr). run(env)))`.



As an example, for the reader taken again from the front-end universe we can represent html templates as reader that will be evaluated when we provide them a model which represents the shared environment

```
const titleViewTemplate =
  Reader(({ firstName, lastName }) => `

${firstName} - ${lastName} </div>` )

var titleView = titleViewTemplate
  .run({firstName:"dimitris", lastName:"papadim"});

console.log(titleView); //<div>dimitris - papadim </div>


```

Run This: [Js Fiddle](#)

We can use map to decorate the resulted html

```
var addToSpan = titleViewTemplate.map( x=>`<span>${x}</span>`).run({firstName:"dimitris",
  lastName:"papadim"});
console.log(addToSpan); //<span><div>dimitris - papadim </div></span>
```

Run This: [Js Fiddle](#)

Also you can use it along with the IO in order to inject the constructed html into an Html element:

```
var appendContent= content=>IO(() => $("#container").append(content));
```

You can see a more extensive examination of the usage of Reader with React at Jack Hsu [blog post](#) (don't mind if you are not familiar with React, just take a brief look).

Now you might have noticed that the IO functor is really a Reader without the context/environment passed to the computation. Which means that we can naturally get a reader out of an IO as we will see later.

4.9 Maybe Functor

Dealing with null - Null object Design pattern

The problem of null or undefined is one of the biggest problems in computation theory. If we want to write meaningful programs, we must accept the fact that some computations might yield no result. The usual way object-oriented languages deal with this is by checking the types for not being undefined in order to use them. Any developer knows the number of bugs that have as source the problem of undefined.

The classical solution is using conditionals everywhere to check for null.

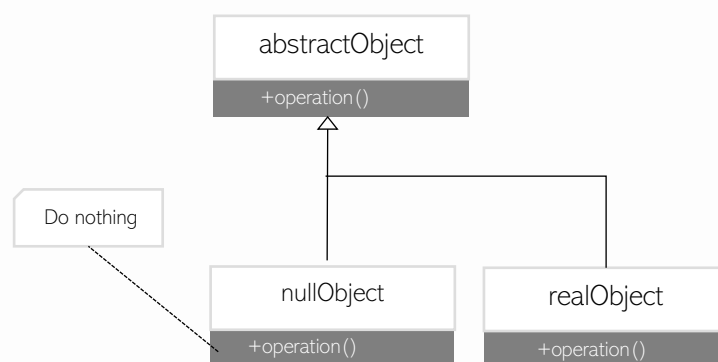
```
if (result) {
  result.operation();
} else {
  //do nothing
}
```

Run This: [Js Fiddle](#)

This reduces the cohesion of the codebase, because we must tightly couple to code relating to the cross-cutting concern of null checking.

4.9.1 The Null Object Design pattern

A more elegant solution to the problem of Null is the “Null Object” design pattern. The null object pattern is a special case of the strategy design pattern. The idea here is to replace conditional with strategy. Instead of setting something as null we can set it as nullObject. Where the nullObject methods are empty, or they do not do anything when



called. nullObject is in fact designed in a way that if it is fed to any method that waits for a realObject, there should not be any unexpected side-effect, or any exceptions thrown.

An object literal implementation of the `nullObject` design pattern with Js without using the ECMAScript Inheritance but just dynamic binding could be something like this :

```
var realObject = () => ({
  operation: () => console.log("Real Object")
});

var nullObject = () => ({
  operation: () => { }
});
```

Run This: [Js Fiddle](#)

This implementation is unobtrusive. It should not affect the rest of the code and should remove the need to check for null. In essence we moved the effect for null checking to our sum type `abstractObject = Either <nullObject, realObject>`.

The major problem with this pattern is that for each object, we need to create a `nullObject` with the operations that we are going to use inside the rest of the code. **This will force us to duplicate all the domain objects in our model. It is a big trade-off.** There is no easy way to abstract the mechanics of this implementation in an object-oriented world, mainly because we cannot have specific information about the operations that we want to isolate and create a `nullObject` that provides those operations.

4.9.2 The Functional equivalent - Maybe as Functor

Now the Maybe functor idea takes this line of reasoning one step further, by abstracting the null Object mechanism inside a functor. Thus, instead of applying the objects on functions, we reverse the flow by applying the functions onto objects. Now, we can isolate the effect inside a single point; the "map."

After all this discussion, hopefully, the implementation of maybe functor would be apparent

```
var some = (v) => ({
  map: (f) => some(f(v)),
  cata: alg => alg.some(v),
});

var none = () => ({
  map: (f) => none(),
  cata: alg => alg.none(v),
});
```

Run This: [Js Fiddle](#)

Maybe Cata implementation for Maybe is straightforward. Since it is a coproduct, we have to implement for each subtype the `cata`, and we are going to call different callbacks, so we can distinguish (pattern match) between `some` and `none` .

And we can use it like this

```
some('Jim')
  .map(x => x.toUpperCase())
  .cata({
    some: value => {
      /*do your work here safely*/
      console.log(value)
    },
    none: () => {
      console.log("nothing")
      /*deal with Maybe.none cases here*/
    }
  });
```

Run This: [Js Fiddle](#)

Now let us revisit the problem of fetching a client asynchronously with a certain Id from a repository.

```
var clientRepository = ({
  getById: (id) =>
    fetchClientsMock()
    .map(clients =>
      clients.filter(c => c.id == id))//promise of a filtered array matching id
});

var clientNameById=id => clientRepository
  .getById(id)
  .map(x => x[0].name);//might throw exception we will deal with this in the next section

clientNameById(1)
  .then(x => console.log(`the client name is ${x}`))
  .catch(e => console.log(`error getting clients ${e}`))
```

Run This: [Js Fiddle](#)

Here we want to replace the result of the array filtering

```
fetchClientsMock().map(clients =>clients.filter(c => c.id == id))
```

with something that will return a `Maybe<client>`. If there is no client for a specific id, we should return `none()`. If there is a client, we will return `some(client)`. We will define an array extension that will do just that :

```
Array.prototype.safeHead = function () {
  return this.length > 0 ?
    some(this[0]) :
    none()
}
```

Now we can write

```
var clientRepository = ({
  getById: (id) =>
```

```

    fetchClientsMock()
      .map(clients =>
        clients.filter(c => c.id == id)
          .safeHead())
  });

var clientNameById = id => clientRepository
  .getId(id)
  .map(response => response.map(client=>client.name)); //we have to map through promise
                                                    //and through Maybe to access client object
                                                    //we will later see how to simplify this

clientNameById(1)
  .then(response => response.cata({
    some: value => console.log(`the client name is ${value}`)
  }))
  .catch(e => console.log(`error getting clients ${e}`))

```

Run This: [Js Fiddle](#)

The **Null object design pattern** (with the map method) is **Isomorphic** to the **Maybe Functor**

Libraries

Maybe in Functional libraries

[monet.js](#) : [Maybe](#) use :Some(5) and None ()
[Sanctuary.js](#) : [Maybe](#) use :S.Just(5) and S.Nothing

4.10 Either Functor

Either Functor is an extension of Maybe. The **right** is equal to the Maybe.some, and the **left** is the equivalent of Maybe.none, **but now we can allow it to carry a value** **left** **=(v)=>{}**. This value can be viewed as a **message** (usually, it represents an error message).

```

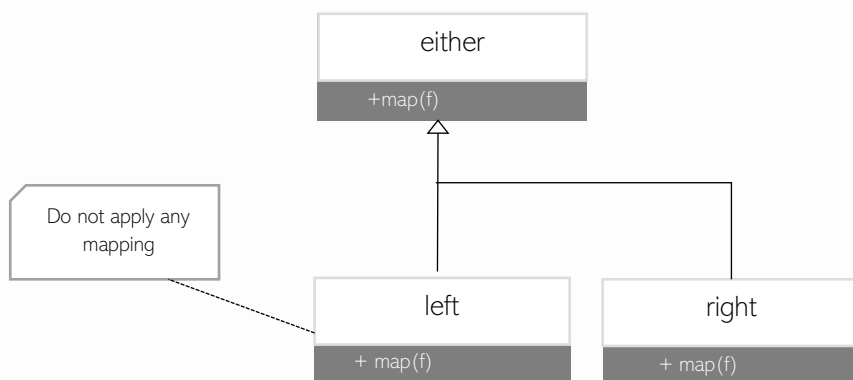
const right = (v) => ({
  map: (f) => right(f(v)),
});

const left = (v) => ({
  map: (f) => left(f(v)),
});

```

Run This: [Js Fiddle](#)

The thing we must pay attention to here is that the *Left* is a functor that **dismisses the mapped function f** and returns itself - `map:(f)=>left(v)`. In essence, it preserves the value that it holds. After we reach a *Left* **all the subsequent transformation through the `map(f)` are ignored**



[Side note: One can verify that the left side also obeys the functor laws

```

// Law 1 - identity preserving      fmap id = id
left(value).map(id).v === id(value)
// Law 2 composition of functions is preserved  fmap (f . g) == fmap f . fmap g
left(value).map(x => g(f(x))).v == left(value).map(f).map(g).v ;
  
```

Run This: [Js Fiddle](#)]

The Cata implementation for Either is similar to Maybe.

```

var right = (v) => ({
  map: (f) => right(f(v)),
  cata: alg => alg.right(v)
});
var left = (v) => ({
  map: (f) => left(v),
  cata: alg => alg.left (v)
});

left("error").cata({
  right: value => { console.log(value) ;},
  left: value => { console.log(value) ;}
})
  
```

Run This: [Js Fiddle](#)

4.10.1 Using Either for exception handling

As we saw the Either can be reduced to maybe if we discard the value passed on the left side. **The left side of either represents a faulty path where the Error** is kept and does not participate in any other computation; it just passes along the Error information. Let us refactor a try/catch to its's Either equivalent. We will start with this try-catch block:

```
var finalPrice;
try {
  var discount = 0.1;
  finalPrice = 10 - discount * 10;
} catch (e) {
  console.log(e);
}

if (finalPrice) {
  console.log(finalPrice);
}
```

The equivalent of the map, in this case, is this:

```
try {
  var computation = () => {
    var discount = 0.1;
    var finalPrice = 10 - discount * 10;
    return finalPrice;
  }
} catch (e) {
  console.log(e);
}
```

then we extract a method and pass the computation as a parameter

```
var Try = (f) => {
  try {
    return f();
  } catch (e) {
    console.log(e);
  }
}

var finalPrice = Try(() => {
  var discount = 0.1;
  var finalPrice = 10 - discount * 10;
  return finalPrice;
});
```

Run This: [Js Fiddle](#)

Finally, we return an Either:

```
var finalPrice = Try(() => {
  var discount = 0.1;
  var finalPrice = 10 - discount * 10;
  return finalPrice;
});

finalPrice.cata({
  right: v => console.log(v),
  left: v => console.log("left" + v)
})
```

Run This: [Js Fiddle](#)

Here the type of the result is `Either<Error, Maybe<string>>` even if there might not be any errors, we might not have any results available.

Monet.js has this `Either.fromTry` function wraps a function execution around a try and returns an Either :

```
!
Either.fromTry(() => {
  var discount = 0.1;
  var finalPrice = 10 - discount * 10;
  return finalPrice;
})
```

Run This: [Js Fiddle](#)

Sanctuary.js has the `S.encase` that does the same thing

`encase :: (a → b) → a → Either Error b`

```
//https://sanctuary.js.org/#encase
S.encase (() => {
  var discount = 0.1;
  var finalPrice = 10 - discount * 10;
  return finalPrice;
}) ( )
```

Run This: [Js Fiddle](#)

Ramda has `tryCatch`

Now that we have our Try in place, we can modify some of the functors that represent delayed computations like the IO and reader to Execute the contained computation in a safe manner that would yield an Either:

```
var IO = fn =>({
  map:f=> IO(()=>f(fn())),
  runIO:()=>fn(),
  runSafe:()=>Try(fn),
})

IO(()=>$('#container`')
.map(x=>x.toUpperCase())
.runSafe()
.cata({
  right: x => console.log(`right: ${x}`),
  left: x => console.log(`left: ${x}`)
}));
```

Run This: [Js Fiddle](#)

We will see later that we can even use either on delayed computations like promises.

4.10.2 Either and Promises

A Promise can be used as an Either because a Promise has a Successful path that applies the map and a failed path that ignores the map exactly like the two components of an Either.

```
Promise.prototype.map = function(f) {
  var initialPromise = this;
  return new Promise(function(resolve, reject) {
    initialPromise.then(result => resolve(f(result)))
    .catch(error=>reject(error));
  });
}
```

Same as **Right** - applies f

Same as **Left** - ignores f

You can take a look at those two code snippets that exhibit this equivalence. In the first one we return a Promise.resolve with the array of clients and in the second snippet, we return an Either.right of the clients without affecting the rest of the program.

```
//Promise implementation
var fetchClientsMockSuccess = () => new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(
      [{ id: 1, name: 'jim', age: 29 },
       { id: 2, name: 'jane', age: 25 }])
  }, 1000);
});
```

```

    }, 1000);
  })

  var clientRepository = ({
    getById: (id) => fetchClientsMockSuccess()
      .map(clients => clients.filter(c => c.id == id)[0])
  });

  var clientNameById = id => clientRepository
    .getById(id)
    .map(client => client.name);

  clientNameById(1)
    .then(x => console.log(`the client name is ${x}`))
    .catch(e => console.log(`error getting clients ${e}`))

```

Run This: [Js Fiddle](#)

// Either implementation

```

var fetchClientsMockSuccess = () =>
  right(
    [{ id: 1, name: 'jim', age: 29 },
     { id: 2, name: 'jane', age: 25 }])

var clientRepository = ({
  getById: (id) => fetchClientsMockSuccess()
    .map(clients => clients.filter(c => c.id == id)[0])
});

var clientNameById = id => clientRepository
  .getById(id)
  .map(client => client.name);

clientNameById(1)
  .cata({
    right: x => console.log(`the client name is ${x}`),
    left: e => console.log(`error getting clients ${e}`)
  });

```

Run This: [Js Fiddle](#)

Of course, **we can also normalize the catamorphism operations** of the promise by defining

```

Promise.prototype.cata = function (alg) {
  this.then(alg.right).catch(alg.left);
}

```

This will allow us to use the same syntax for promises also, instead of the (then, catch)


```

Promise.resolve("jim")
  .cata({
    right: x => console.log(`the client name is ${x}`),
    left: e => console.log(`error getting clients ${e}`)
  });

```

Run This: [Js Fiddle](#)

Either and Promise start to converge. The one big difference, of course, is that Promises are asynchronous and Either is synchronous, we will finally overcome this difference by using a fundamental idea in category theory ([Co-Yoneda lemma](#)) that assure us that having a callback of value like in a promise is the same as having the value itself.

That's all about Either, for now.

Libraries

Either

Either is another Functor that is included in almost every functional library out there.

```

monet.js    : Either
sanctuary.js: Either Use S.Right(42) S.Left ('Cannot divide by zero')
folktale    : result
pratica     : result
crocks      : Either

```

4.11 Functors from Algebraic Data Types

We have seen an implementation of the map function for our algebraic definition of the list data structure in [section 3.4](#). I will restate the implementation here:

```
var Cons = (value, rest) => ({
  value: value,
  rest: rest,
  map: f => Cons(f(value), rest.map(f))
})

var Empty = () => ({ map: f => Empty() })

var list = Cons(2, Cons(2, Empty()));
var liftedList = list.map(x=>x+2)
```

Now that we know much more about functors, we can extrapolate the implementation of the map for a binary tree. So, let us go from Linear to quadratic.

The tree is also **a recursive data structure**. The definition of a binary tree is this:

```
Tree (a) = leaf(a) + Node(Tree(a) , Tree (a))
```

Which states **that a Tree(a) can be a Leaf (a) or a Node of two Trees of type a**.

[Sidenote: The list can be viewed as a tree, which has its Right side of all nodes are always leaves: $\text{Tree } a = \text{Leaf } a + \text{Node}(\text{Leaf } a) * (\text{Tree } a) \cong \text{List } a = a + a * (\text{List } a)$]

With the ES6 syntax, this algebraic definition would be (we have already seen this in [section 3.4](#) table.)

```
class Tree {}

class Leaf extends Tree {
  constructor(value) {
    super()
    this.value = value;
  }
}

class Node extends Tree {
  constructor(left, right) {
    super()
    this.left = left;
    this.right = right;
  }
}
```

In order to create a functor based on this algebraic data type we define a valid **map** method. We will again follow our **method of structural induction** in order to define the **map**.

First, because it is a **coproduct**, we must define the map method in *both subtypes* :*Leaf* and *Node* of the coproduct.

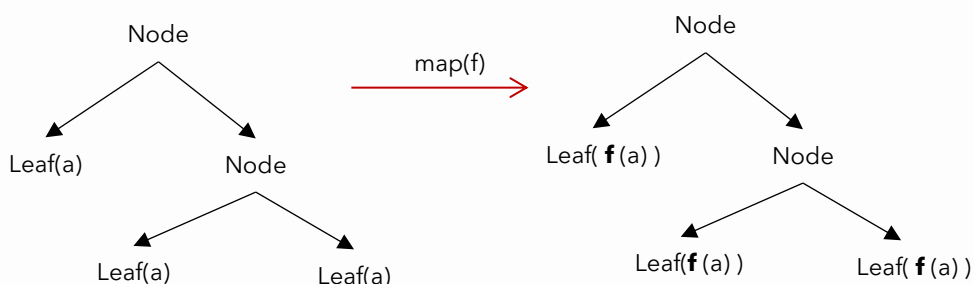
1. We start with the *Leaf* which is the base case of the structural induction. We must return something of the same type. That's why we return a new *Leaf* with the function applied to the value

```
map(f) {
  return new Leaf(f(this.value));
}
```

A leaf is just an identity functor. Hopefully, you recognized it.

2. The *Node* part is a product of two trees (*left*, *right*) both are of the same type as the one for which we want to define the map. The only way to deal with it is to assume that both must have a map function already in place that works as expected (this is the inductive hypothesis again),

```
map(f) {
  return new Node(this.left.map(f), this.right.map(f));
}
```



The whole tree, with the map method, now becomes:

```
class Tree {}
class Leaf extends Tree {
  constructor(value) {
    super()
    this.value = value;
  }
  map(f) {
    return new Leaf(f(this.value));
  }
}

class Node extends Tree {
  constructor(left, right) {
    super()
    this.left = left;
    this.right = right;
  }
}
```

```

    }
    map(f) {
      return new Node(this.left.map(f), this.right.map(f));
    }
  }
}

```

Run This: [Js Fiddle](#)

One usual variation of the tree data structure is to **include an additional value at the node level**

```

class Tree {}

class Leaf extends Tree { ... }

class Node extends Tree {
  constructor(left, value, right) {
    super()
    this.left = left;
    this.right = right;
    this.value = value;
  }
}

```

```
Tree (a)= leaf(a) + Node(Tree(a) ,a, Tree (a))
```

Can you extend the previous definition of map to also lift the value of the node?

Can you write a symbolic equation for the definition of the map for this tree version?

4.12 Functor Composition

Functors are closed under composition. The term closed means that if we have two functors and we compose them, the resulting structure is also a functor. This means that we should be able to construct a well-behaved map function for the composition, **using only the map functions of the composing functors** and nothing more.

```

var composeF = composition => ({
  value: () => composition,
  map: fab => composeF(composition.map(x => x.map(fab))),
  cata: alg => composition.map(x => x.cata(alg))
});

```

```
Object.prototype.compose = function () { return composeF(this); }
```

```

Id(Id(3)). compose()
  .map(x => 3 * x)
  .map(x => 3 * x)
  .cata(console.log)//27

```

Run This: [Js Fiddle](#)

It is easy to understand how this works. You first pass the function inside with the map of the first functor, and then you map the resulting value again using the map of the inner (second) functor.

This works for all the functors. If we compose for example the Id with the reader, we get still a valid functor.

```
var result = Reader(g => Id(g.x1))
  .compose ()
  .map(x => 3 * x)
  .map(x => 3 * x)
  .value()
  .run({x1:20}); //Id(180)
```

Run This: [Js Fiddle](#)

We can revisit the repository example from the Maybe section where we have an asynchronous operation of a repository that returns a maybe of an array:

```
var clientRepository = {
  getById: () => Promise.resolve(some([
    { name: 'jim', age: 29 },
    { name: 'jane', age: 25 }]))
};

clientRepository.getById().then(response => {
  response
    .compose() //skip the some(_) and map to the array directly
    .map(x => x.name)
    .value()
    .cata({
      Ok: result => console.log("client names: " + result),
      Error: error => console.log("error: " + error)
    });
});
```

Run This: [Js Fiddle](#)

As we have seen If we do not use the array composition to map through maybe to the array, we must write explicitly the nested map

```
clientRepository.getById().then(response => {
  response
    .map(maybe => maybe.map(employee => employee.name))
    .cata({
      Ok: result => console.log("client names: " + result),
      Error: error => console.log("error: " + error)
    });
});
```

4.13 Applicative Functor

Applicative is a relatively easy concept to grasp. It is a **functor F that contains a function** $(a \rightarrow b)$ instead of just a value, and we must provide a valid method that can take another functor with a value $F(a)$ and give an $F(b)$. We call this **apply** (usually the method is named **ap**):

$ap: F(a \rightarrow b) \rightarrow F(a) \rightarrow F(b)$

you can compare this with the map signature $map: (a \rightarrow b) \rightarrow F(a) \rightarrow F(b)$ the only difference is the initial F

If we take the Identity functor as an example, we can implement the apply as follows:

```
var Id = f => ({
  v: v,
  map: f => Id(f(v)),
  ap: fa => fa.map(a => f(a)) //f is (a -> b)
});
```

[run this fiddle](#)

if we use currying we can chain multiple applicatives like that:

```
var chain = Id(x =>y=> x +y).ap(Id(5)).app(Id(5));
```

[run this fiddle](#)

! What is happening here? Well the first app applies the `Id(5)` to the `Id (x =>y=> x +y)` this gives us an `Id (y=> 5+y)`, and since we still have a function inside a functor, we can use `ap` again `Id (y=> 5+y).ap(Id(5))` which eventually gives us `Id(10)`.

Brace yourselves because we are going to use this extensively throughout this book. You must appreciate how nice this property of the applicative is. *In a way, we can chain multiple operations without passing any values around but binding them through a partial application.*

Obviously if there was a curried function of arity 10 we could chain 10 applicatives

```
Id(a =>b=>c=>d=>e=>...=>{ }).ap(Id(_)).ap(Id(_)) .ap(Id(_))... .ap(Id(_))
```

[run this fiddle](#)

We will represent this as `{f _ _ _ ..._ }`

So let's take this one step further than the simple `Id` functor and define applicative on a reader functor.

4.14 Reader Applicative Functor

A reader applicative is a reader that has a function stored as the expression `expr`. The implementation is a bit complex. Try to imagine that you want to combine a reader that has a function and a reader that has just a value. What would you do? Take some time and think about it, or try to fork the [jsfiddle](#) and play around.

```
Reader = (expr) => ({
  expr: expr,
  map: f => Reader (env=>f(Reader(expr).run(env))) ,
  run: env => expr(env),
  ap: reader=>Reader(env=> Reader(expr).run(env)(reader.run(env)))
});
```

```
var res = Reader(g=>x=>g.x2+x).ap(Reader(g => g.x1));
```

```
var r =res.run({x1:10, x2:20}); //30
```

Run This: [Js Fiddle](#)

I would suggest reading this section a couple of times until it becomes clearer.

if we take the applicative operation

```
ap: f(a → b) → f(a) → f(b) (this is the name for ⟨⊗ ...⟩)
```

for the reader specifically by substituting the f with the $((\rightarrow) \text{ env})$ [that's $(\text{env} \rightarrow _)$] we get

```
⟨⊗⟩: ((→) env) (a → b) → ((→) env) a → ((→) env) b
```

With some reductions we get the form

```
(env → (a → b)) → (env → a) → (env → b)
```

! If for example, we have a curried function of arity 2 like $x \Rightarrow y \Rightarrow y + x$ we can do that

$\langle \text{add} \ \otimes \ 1 \ \otimes \ 2 \rangle \rightarrow \langle 1 + 2 \rangle$ we will get a reader of the addition

```
var res = Reader(g=>x=>y=>y+x).ap(Reader(g => 1)).ap(Reader(g => 2)).run({}); //3
```

Run This: [Js Fiddle](#)

I have mentioned that the S combinator has something to do with the Reader. Well we have seen that $Sxyz = xz(yz)$ which corresponds in js code to `const S = f => g => x => f(x)(g(x));`

```
(r → (a → b)) → (r → a) → (r → b) this is our S combinator
```

We have seen [at the higher-order functions section](#) that we could use S combinator to write a simple strategy like code for product discounting:

```
var Product = (price) => ({
  getFinalPrice(pricingStrategy) {
```

```

    return pricingStrategy(price)
  }
})

const S = f => g => x => f(x)(g(x));

var config = ({ discount: 0.1, productPrice: 100 });

var discountedPrice = S
  (config => product => product.getFinalPrice(price => price - config.discount * price))
  (config => Product(config.productPrice))
  (config);

console.log(discountedPrice)//90

```

now we **can rewrite it using the Reader functor as applicative**, just to connect the dots with a practical application, between S and the applicative reader.

```

var discountedPrice =
  Reader(config => product => product.getFinalPrice(price => price*(1-config.discount)))
  .ap(Reader(config => Product(config.productPrice)))
  .run({ discount: 0.1, productPrice: 100 });//90

```

Run This: [Js Fiddle](#)

Category Theory

Monoidal Functor



Let $(C, \otimes_C, 1_C)$ and $(D, \otimes_D, 1_D)$ be two monoidal categories. A lax monoidal functor between them is

1. a functor $F: C \rightarrow D$,
2. a morphism $\epsilon: 1_D \rightarrow F(1_C)$
3. a natural transformation $\mu_{x,y}: F(x) \otimes_D F(y) \rightarrow F(x \otimes_C y)$ for all $x, y \in C$

$\epsilon: 1_D \rightarrow F(1_C)$ represents the pure operation of the applicative, that allows us to get into an applicative

$F(x) \otimes_D F(y) \rightarrow F(x \otimes_C y)$ is the applicative operation of **ap**. The interesting thing with this definition is that it connects the "multiplication" \otimes_D of one category with "multiplication" \otimes_C of the other category.

In our example above the equivalence is that the multiplication $\otimes_{\text{int}}: (+)$ of the integers is now the same as the \otimes_{Reader} , applicative operation $\text{ap} \langle \rangle$ of the reader :

$(+) \xrightarrow{\text{is the same}} \langle \rangle$

```

  Reader(g=>x=>y=>y+x).ap(Reader(g => 1)).ap(Reader(g => 2)).run({});

```


Two completely different domains but actually, `ap` is equivalent with `+` in a different level of abstraction. Cool right.

Also as we will see in the traversable section the Arrays “multiplication” \otimes_{Array} `concat` $(:)$ will be lifted to the now the same as the \otimes_{Reader} , applicative operation `ap` $\langle \rangle$ of the reader

$$(:) \rightarrow \langle \rangle$$

```
Reader(g=>x=>y=>[y].concat(x)).ap(...).ap(Reader(...)) ;
```

(Hopefully you would recognized that $\mu_{x,y}:F(x) \otimes_D F(y) \rightarrow F(x \otimes_C y)$ is a monoid homomorphism)

4.15 Optional Composing Applicatives

“The composition of applicative functors is always applicative, but the composition of monads is not always a monad.”
- [hackage.haskell](#)

Applicatives are also closed under composition. Like functors. This means that if we have two applicatives `F` and `G`, and we compose them, the resulting structure `FG` is also applicative. This means that we can construct a well-behaved `ap` function for `FG`, using only the `map` functions of the functor composition (`FG.map`) and the applicative operations of the composing applicatives (`F.ap`, `G.ap`). If there is no way to construct something like that, we have to say that the structure is not closed under the operation.

```
var composeF = FG => ({
  unComp: FG,
  map: fab => composeF(FG.map(x => x.map(fab))),
  ap: FGa => composeF(FG.map(Gfab => Gfab.app).app(FGa.unComp))
});

var result = composeF(Id(Id(x => y => x * y)))
  .app(composeF(Id(Id(3))))
  .app(composeF(Id(Id(3))));
```

Run This: [Js Fiddle](#)

4.16 Decorator design pattern functional idiom with Functor

We already have seen how the functional composition resembles the decorator design patterns mainly because it does not affect the type of the decorated function when

the composing function has a type $x \rightarrow x$. The Functor can provide some additional convenience due to the map function. Let us look again the decorator example from [section 2.1.1.1](#)

```
var createProduct = (price) => ({ price:()=>price})
var addPackaging = (product) => ({ price:()=>product.price() + 0.5, })
var addRibbons = (product) => ({price:()=>product.price() + 0.3, })

var finalProduct = addRibbons (addPackaging (createProduct (20)))
```

now we can replace composition with map chaining which is equivalent

```
//with the help of List as a functor that provides map
var finalProduct= [20]
  .map(createProduct)
  .map(addPackaging)
  .map(addRibbons);
```

Run This: [Js Fiddle](#)

The true reason that map is equivalent to compose, is the second functor law $F(f \circ g) = F(f) \circ F(g)$, which states that we can replace the map of a composition with the composition of maps. (by the way, I hope the composition law $F(f \circ_f g) = F(f) \circ_F F(g)$ reminds you of the Homomorphism equation $\phi(a_1 * a_1) = \phi(a_1) \bullet \phi(a_1)$ because that is what it is. Homomorphisms are everywhere).

Because of the functor laws, we can use **any functor** in order to compose the underling functions in a decorator style.

```
//with our identity functor just to provide us with map support
const Identity = (v) => ({
  value: v,
  map: (f) => Identity (f(v))
});

var finalProduct = Identity (20)
  .map(createProduct)
  .map(addPackaging)
  .map(addRibbons)
```

Run This: [Js Fiddle](#)

In a next section, we will also see the [coyoned](#) method which allows us to create a functor on the fly.

This concludes the functor section of this book. After we familiarized ourselves with the concept of a functor, we will go on to explore some generalizations of the Array.reduce.

Traversing Functors

5 Catamorphisms Again

This Chapter is about Folding, Decomposing, various structures it's all about **getting values out of data structures** in an elegant and consistent manner. The path that we will follow will involve many concepts from category theory, like the concept of **catamorphism** (from the Greek: κατά "downwards" and μορφή "form, shape") its dual anamorphism (from the Greek: ανά "upwards" and shape), and their combination hylomorphism (from the Greek: ὕλη *hyle* "matter" and shape) and probably is the most difficult part of this book. Feel free to skip any part you don't like or find interesting and maybe revisit in a later time.

5.1 A brief mentioning of F-algebras

If F is a Functor, then an **F-algebra** is a pair (A, α) , where A is called carrier type and α is a function $F(A) \rightarrow A$ called **structure map**. The F-Algebra is a way to get an **A out of a F(A)**. It is an *evaluation*.

For our Peano Numerals functor

```
var Succ = (rest) => ({ rest: rest, })
var Zero = () => ({})
```

```
var fourNumeral = Succ(Succ(Succ(Succ(Zero())))); //Example numeral
```

one algebra with carrier type the integers would be this:

```
var algInt = {
  succ: (n) => n + 1,
  zero: () => 0
};
```

That's called the **structure map** but when we say algebra, we will mean this structure

This algebra tells us how to *interpret* a `Numeral` Expression into an integer. This algebra is the *intended interpretation* (also called a *standard model* a term introduced by Abraham Robinson in 1960) because it gives us the intended "meaning" of the Expression, which in this case was to represent numbers using a formal language. Nevertheless, we can have many algebras based on this Functor.

Here some more:

```
var algVonNeuman = {                                     //Von Neumann universe
  succ: (n) => n.concat([n]) ,
  zero: () => []
};
```

This is the Set-theoretic definition of natural numbers given by the great Von Neuman and look like this

```
0 = { } = ∅
1 = { 0 } = {∅}
2 = { 0, 1 } = { ∅, {∅} }
3 = { 0, 1, 2 } = { ∅, {∅}, {∅, {∅}} }
```

The algebra below needs no further explanation

```
var selfExplained = {
  succ: (n) => `add one(${n})` ,
  zero: () => "zero"
};
```

The thing is that we cannot apply the algebra just yet to an expression like `Succ(Succ(Succ(Succ(Zero()))))` for example. The algebra is for evaluating only one **“step” or “layer”** of the expression. We must somehow push the algebra downwards, perform an evaluation, and then compose the results. **The catamorphism is this exact process.**

If C is a category, and $F : C \rightarrow C$ is an endofunctor of C , then an **F-algebra** is a pair (A, α) , where A is an object of C and α is a morphism $F(A) \rightarrow A$ the a often called *structure map*.

5.2 Catamorphisms

which we will write `Numeral` in a class notation to simplify some things along the way and also define a matching method on the `Numeral`

```
class Numeral { }

class Succ extends Numeral {
  constructor(rest) {
    super();
    this.Rest = rest;
  }
  matchWith(pattern) {
    return pattern.succ(this.Rest)
  }
}
```

```
class Zero extends Numeral {
  matchWith(pattern) {
    return pattern.zero()
  }
}
```

Now we are going to define a `cata` method for the `Numeral` union type, which will take an algebra and evaluate the `Numeral`. Here the implementation will be ad hoc-specifically for the `Numeral` structure.

```
Numeral.prototype.cata = function (algebra) {
  return this.matchWith({
    zero: () => algebra.zero(),
    succ: (rest) => algebra.succ(rest.cata(algebra))
  });
}
```

Run This: [Js Fiddle](#)

[SideNote: We have seen Recursive definitions before using `matchWith`.

For the first time we are creating a method that takes as an argument `{zero:()=>{}, succ:(_)=>{}}` which is of the same type as the `matchWith` argument. **Inception.** The `cata` is a generalised `matchWith` for composite Algebraic types. Or `matchWith` is just the `cata` for elementary Types]

This definition follows *the structural induction* and should be easy to follow by now. Now we can evaluate a numeral using our algebras above.

```
var fourNumeral = new Succ(new Succ(new Succ(new Succ(new Zero()))))
var fourInt= fourNumeral.cata(algInt)//4
var fourVonNeuman= fourNumeral.cata(algVonNeuman) //[[],[[.....]]]
var selfExplained= fourNumeral.cata(selfExplained) //"add one(add one(add one
(add one(zero))))"
```

Run This: [Js Fiddle](#)

The `Numeral` functor has two parts the type of `Zero` is $() \rightarrow \mathbf{1}$ this means that we can get an initial object $((), [], 0$ etc) depending on A , and the type of `Succ` is $A \rightarrow A$ since the `Numeral` is a union we should add them $\text{Numeral}: A \rightarrow 1 + A$

You might already have seen the `ToString()` implementation for `Numeral`

```
Numeral.prototype.Show = function () {
  return this.matchWith({
    zero: () => "0",
    succ: (r) => `(${r.show()})`
  });
}
```

The `cata` just generalize this by extracting the Recursion mechanics and allow `cata` to be used with different algebras without defining any additional extension methods.

For example, We could provide a different `ToString` by providing custom pattern matching algebras. For example, this is a different mapping from `Natural` \rightarrow `string`

```
var justExclamationmarks = ({
  zero: () => "",
  succ: (r) => `!(${r})`
});
```

Run This: [TS Fiddle](#)

I don't want to confuse you but we could define `cata` directly without the `matchWith` but I wanted to make the distinction clear.

```
var Succ = (rest) => ({
  rest: rest,
  cata: (alg) => alg.succ(rest.cata(alg)),
});

var Zero = () => ({
  cata: (alg) => alg.zero()
});
```

pass the algebra to the rest

Use the algebra to evaluate

Run This: [Js Fiddle](#)

TypeScript

Look at the types of the arguments of `matchWith` pattern and the `cata` algebra. Almost identical can you spot the difference ?

```
matchWith<T1>(pattern: { Zero: () => T1; Succ: (rest: Natural) => T1 }): T1
```

```
cata<T1> (algebra: { Zero: () => T1; Succ: (rest: T1) => T1 }): T1;
```

```
Natural.prototype.Cata = function <T, T1>(
  algebra: ({ Zero: () => T1, Succ: (rest: T) => T1 }))
{
  return this.MatchWith({
    Zero: () => algebra.Zero(),
    Succ: (rest) => algebra.Succ(rest.Cata(algebra))
  });
}
```

pass the algebra to the rest

Use the algebra to evaluate

Run This: [TS Fiddle](#)

```
var algebraInt: ({ Zero: () => number, Succ: (rest: number) => number }) = ({
  Zero: () => 0,
  Succ: (r) => r + 1
});
```

```
var fourNumeral: Numeral = new Succ(new Succ(new Succ(new Succ(new Zero()))));

var four = fourNumeral.Cata<number>(algebraInt); //"4"
```

Run This: [JS Fiddle](#)

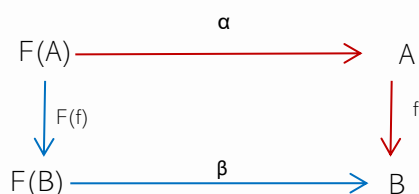
5.3 Optional Initial algebra

This section is optional but **highly recommended**. You might want to revisit the whole 5th section even if you don't feel yet that everything "connects".

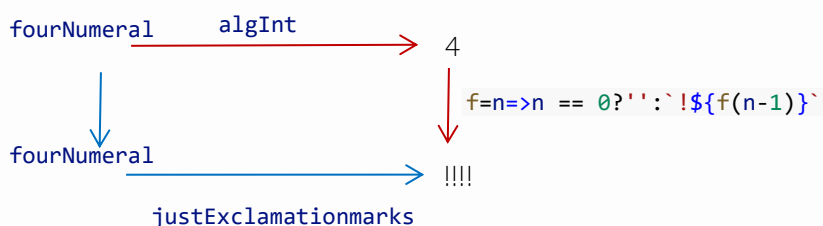
5.3.1 F-Algebras Homomorphisms

Category theorists will form a category out of anything so why not get a category of F-Algebras. So, let's take two F-Algebras and see the relationships.

A homomorphism from an F-algebra (A, α) to an F-algebra (B, β) is a morphism $f: A \rightarrow B$ such that $f \circ \alpha = \beta \circ F(f)$, according to the following diagram:



What that practically means is that for example we had two algebras for the `Numeral` functor



```
var justExclamationmarks = {succ: (n) => `!${n}`, zero: () => ""};
var algInt = {succ: (n) => n + 1, zero: () => 0};
```

Run This: [JS Fiddle](#)

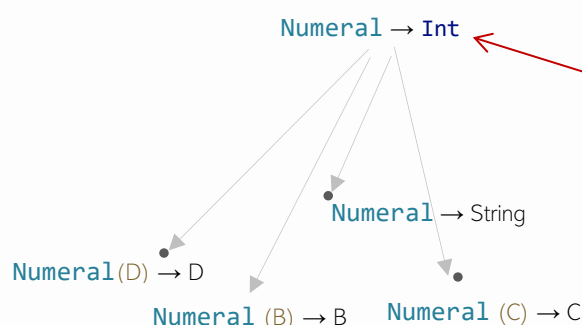
we can go from the `Succ(Succ(Succ(Succ(Zero()))))` to 4 obviously, but also we can get `!!!!` by using the `justExclamationmarks` to consume this `fourNumeral`. And then use this function `var f = n => n == 0 ? '' : `!${f(n - 1)}`` to go from 4 to `!!!!`

The truth is that we can go from the 4 to any other possible interpretation of any functor with type $1+A$ for any A because the integers and the `algInt` is an initial algebra for our `Natural` functor. Initial algebra means that for any other algebra there is an arrow (which represents a Homomorphism in the category of F-algebras) to that object from the Initial algebra. In Essence this proves that we **can represent Iteration** with this

```
var recursion = n => n == 0 ? algebra.e : (algebra.f(recursion(n - 1)));
```

```
for any algebra var algebra = ({ e: "", f: n => `!${n}` });
```

Run This: [Js Fiddle](#)



The natural numbers together with the tuple $(0, n+1)$ form the **initial object in the category of F-Algebras** for the `Natural` Functor

The point of all this is **that the initial algebra of the Peano functor `Natural` is semantically equivalent to a `for()` loop.**

Category Theory

Natural Number System

This generalization of the simple Peano arithmetic to other domains, first presented by Lawvere and its known as the Peano-Lawvere axiom. This triple `(Natural, Succ, Zero)` because it is "Equivalent" to the natural numbers $(\mathbb{N}, S, 0)$ is what is called a **natural number system** (in TS Type category in our case), and `Natural` a natural number object.

We have seen another natural number system residing in lambda calculus; the church encodings:

$$0 \cong \lambda f. \lambda x. x$$

$$1 \cong \lambda f. \lambda x. f(x)$$

$$2 \cong \lambda f. \lambda x. f(f(x))$$

5.4 Catamorphisms for Trees

We already have seen the Peano Numerals as the simplest recursive Algebraic types. We will also look at the binary tree. Let us say we have a simple algebra for this tree. This algebra sums up the values of a node or returns the value of a leaf if it is a leaf.

```
var algSumInt = {
  node: (l, v, r) => l + v + r,
  leaf: (l) => l
};
```

we could evaluate it if we added a simple cata method to our tree, like this:

```
Tree.prototype.cata = function (algebra) {
  return this.matchWith({
    leaf: v => algebra.leaf(v),
    node: (l, v, r) => algebra.node(l.cata(algebra), v, r.cata(algebra))
  });
};
```

Run This: [Js Fiddle](#)

Hopefully, the implementation of a method, like cata by structural induction on a tree, should be familiar by now. In case you don't still feel confident I will go through with the construction one more time below.

TypeScript

Our cata method would look like the following :

```
Tree.prototype.cata = function <T, T1>(algebra: TreePattern<T, T1>) {
  return this.matchWith({
    leaf: (v) => algebra.leaf(v),
    node: (l, v, r) => algebra.node(l.cata(algebra), v, r.cata(algebra))
  });
}
```

Run This: [TS Fiddle](#)

Again pay attention to the types of the matchWith pattern and the cata algebra

```
matchWith<T1>(pattern: { leaf: (v: T) => T1; node: (left: Tree<T>, v: T, right: Tree<T>) =>
T1; }): T1
```

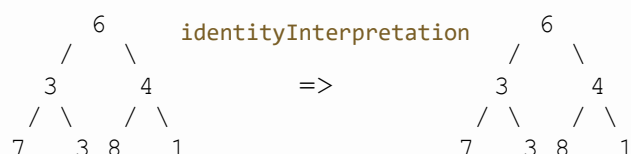
```
Cata(algebra: { leaf: (v: T) => T1; node:(left: T1, v: T, right: T1) => T1; }): T1
```

5.5 Reversing a Tree

Since we can find an algebra to map a Tree to string and numbers or anything we want, we might want to transform it back to a new Tree.

We can have for example an algebra that just maps the Tree to a same tree:

```
var identityInterpretation = () => ({
  leaf: v => new Leaf(v),
  node: (l, v, r) => new Node1(l, v, r)
});
```



A way to reverse a tree is by this recursive function

```
Reverse(){
  return this.matchWith({
    leaf: (v) => new Leaf(v),
    node: (l, v, r) => new Node1(r.Reverse(), v, l.Reverse())
  });
}
```

if we have a tree that look like this through the function **Reverse** that



Here we can use our cata implementation to define the reversing

```
var reverseInterpretation =({
  leaf: v => new Leaf(v),
  node: (l, v, r) => new Node1(r, v, l)
});
```

We swapped l and r

```
const tree = new Node1(new Node1(new Leaf(1), 2, new Leaf(3)),
4, new Node1(new Leaf(5), 6, new Leaf(7)));

var reverse = tree.cata(reverseInterpretation);
console.log(reverse.show());
```

Run This: [jsFiddle](#)

5.6 Catamorphisms with the Visitor Design pattern

We can also use our algebra on a tree with the visitor design pattern. If we define an `accept` method on the tree structure. The visitor offers one more degree of freedom because we can use different kinds of visitors.

```
class Tree {
  Show() {
    var visitor = {
      VisitLeaf: l => `${l.Value}`,
      VisitNode: n =>
        `${n.Left.Accept(visitor)}, ${n.Value}, ${n.Right.Accept(visitor)}`
    };
    return this.Accept(visitor);
  }
}

class Node1 extends Tree {
  Accept(visitor) {
    return visitor.VisitNode(this);
  }
}

class Leaf extends Tree {
  Accept(visitor) {
    return visitor.VisitLeaf(this);
  }
}

const tree = new Node1(
  new Node1(new Leaf(1), 2, new Leaf(3)),
  4,
  new Node1(new Leaf(5), 6, new Leaf(7))
);

console.log(tree.Show());
```

Run This: [Js Fiddle](#)

As you can see the Visitor is pretty much isomorphic to the catamorphism implementation. Visitors use the **double dispatch** to dispatch a function call to different concrete functions depending on the runtime types of two objects involved in the call. Double Dispatch is a Kind of discrimination between the Members of the Union (aka Inheritance)

The visitor methods are almost identical with the algebra of `matchWith` mechanism

```
visitor = {
  VisitLeaf: l => {},
  VisitNode: n => {}
}

matchWith({
  leaf: (v) => {},
  node: (l, v, r) => {}
});
```

The **Visitor design pattern** can be **Isomorphic** to **Catamorphisms**

5.6.1 F-Coalgebra

A F-coalgebra is the dual structure of a F-algebra. If an algebra is something like $\mathbf{F}(\mathbf{A}) \rightarrow \mathbf{A}$ where F is a Functor then by reversing the arrows (in order to get the dual) a coalgebra is something of this form $\mathbf{A} \rightarrow \mathbf{F}(\mathbf{A})$. We start from an object and we get a Functor of an object.

Look again at this coalgeabr `coAlg= n => n <= 0 ? new None() : new Some({...})` from an int we get a `Maybe`.

5.6.2 Optional A brief mentioning of Anamorphisms

Its easy to use recursion in order to construct instances of this type `new Some({ value: 2, rest: new Some({ value: 1, rest: new None() }) })`

```
var ana = n => n == 0 ? new None() : new Some({ value: n, rest: ana(n - 1) });
(ana(2))//{"v":{"value":2,"rest":{"v":{"value":1,"rest":{}}}}}
```

Run This: [Js Fiddle](#)

This is an example of an unfold or anamorphisms. Starting from a single seed integer value we gradually build a Maybe functor instance.

The above definition has two entangled concepts that can be separated. We can isolate the recursive part. Writing something like this

```
var coAlgListBase = n => n <= 0 ? new None() : new Some({
  head: n - 1,
  tail: n - 1
});

var ana = (colAg) => oldState => colAg(oldState).cata({
  some: (v) => new Some({value:v.head, rest:(ana(colAg)(v.tail))}),
```

```

    none: (_) => new None()
  })

  var nat = ana(coAlgListBase)(2); //{"v":{"value":1,"rest":{"v":{"value":0,"rest":{}}}}}

```

Run This: [Js Fiddle](#)

I know that this way more complex and the only thing that we here is to be able to use different `colAg` for example using the following we get only the even numbers:

```

var coAlgListBaseEvens = n => n <= 0 ? new None() : new Some({
  head: n - 2,
  tail: n - 2
});

```

We can also change the `ana` function and generate lists by using instructions made of `Maybe's`

```

var ana = (colAg) => oldState => colAg(oldState).cata({
  some: (v) => [v.head].concat(ana(colAg)(v.tail)),
  none: (_) => []
});

```

```

var nat = ana(coAlgListBaseEvens)(10); // [8,6,4,2,0]

```

Run This: [Js Fiddle](#)

this can be further simplified using the definition of anamorphisms from the perspective of category theory, but we are not going into this here [You can find more on the [extended version of this book at Leanpub](#)]. We indirectly introduced the definition of F-Coalgebras.

5.6.3 Corecursion

We all know that this definition is recursive:

```

var ana = n => n == 0 ? new None() : new Some({ value: n, rest: ana(n - 1) }); (!)

```

the characteristics that make it recursive are three:

1. There is a base **terminal** condition `n == 0 ? new None()`.
2. There is an either/union/coproduct involved. In our case the conditional operator (`n => n == 0 ? _ : _`) plays this role
3. We call the function again **with a reduced** argument that ensures that we will eventually reach the base condition (1) and this process will finally stop. So, we **deconstruct** the **initial** value step by step.

But mathematicians came up with the dual concept to recursion, which is called, co-recursion. Here instead of working top-to-bottom we build things bottom-up. A corecursive definition of definition (!) would be :

```
var coana = (n, rest) => coana(n + 1, new Some({ value: n, rest: rest }));
```

```
var stream = coana(0, new None());
```

the resulting structure `stream` is **infinite** and gives us **all** the natural numbers! Corecursion often gives us infinite data structures we call co-data.

The thing is that we cannot call `coana` because JavaScript will try to evaluate everything at once. But we can use a `()=>` to make it lazy

```
var coana = (n, rest) => () => coana(n + 1, new Some({ value: n, rest: rest })); (1)
```

and we can lazily call it:

```
var stream = coana(0, new None());
```

```
stream = stream(); //{  
stream = stream(); //{ "v": { "value": 1, "rest": { "v": { "value": 0, "rest": {} } } } }  
stream = stream(); //{ "v": { "value": 2, "rest": { "v": { "value": 1, "rest": { "v": { "value": 0, "rest": {} } } } } }  
}}}}}
```

Run This: [Js Fiddle](#)

the characteristics that make it co-recursive are three:

1. There is a base **initial** condition (`0, new None()`).
2. There is a **product** involved for value accumulation. In our case the tuple (`n, rest`) plays this role
3. We call the function again **with a increased** argument. There is no assurance that this process is going to stop eventually. We **construct** the **final** value step by step.

5.6.3.1 Optional JavaScript Generators

ES6 brought a some new functions to deal with asynchronous data streams that one of them is called Generator functions. A generator function is a special kind of iterator which can return the iteration values by using **yield** keyword multiple times. Generators is a simple concept but also can become complex fast. If you look at the following example what a generator does is when the `next()` is called, then we get the **yield 1** value and move to the next step

```
function* gen() {  
  yield 1;  
  yield 2;  
  yield 3;  
}
```

```
var g = gen(); //calls the generator
console.log(g.next().value); //==1
console.log(g.next().value); //==2
console.log(g.next().value); //==3
```

Run This: [Js Fiddle](#)

I am not going to get into the details here, but we can use generators to generate co-data. The following script does exactly what the `coana(0, new None())` does.

```
function* Stream(n, rest) {
  while (true) {
    n = n + 1;
    yield rest;
    rest = new Some({ value: n, rest: rest })
  }
}
```

```
var stream = Stream(0, new None());
stream.next().value.log()
stream.next().value.log()
stream.next().value.log()
```

Run This: [Js Fiddle](#)

the `.next().value` is essentially the same with the one step decompression of the lazy `()=>` in our corecursion stream definition `stream = stream()`.

5.6.4 Optional A brief mentioning of Hylomorphisms

"Hylomorphism (or hylemorphism) is a philosophical theory developed by Aristotle, which conceives being (ousia) as a compound of matter and form"

-Wikipedia

We can now combine `ana` and `cata`, by first using an anamorphisms `ana` to generate an expression like `new Some({ value: 2, rest: new Some({ value: 1, rest: new None() }) })` and then use a catamorphism `cata` to compress this expression it into a single value. This process is called Hylomorphism

```
var ana = n => n <= 0 ?
  new None()
  : new Some({ value: n, rest: ana(n - 1) });

var cata = list => list.cata({
  some: ({ value, rest }) => value * cata(rest),
  none: _ => 1,
})
```

```
var hyllo = n => cata(ana(n));
```

Run This: [Js Fiddle](#)

the mathematical formulation behind hylomorphism is a recursive function

$$h(a) = \begin{cases} c & \text{if } p(a) == \text{true} \\ b \otimes h(a') & \text{where } g(a) = (a', b) \end{cases}$$

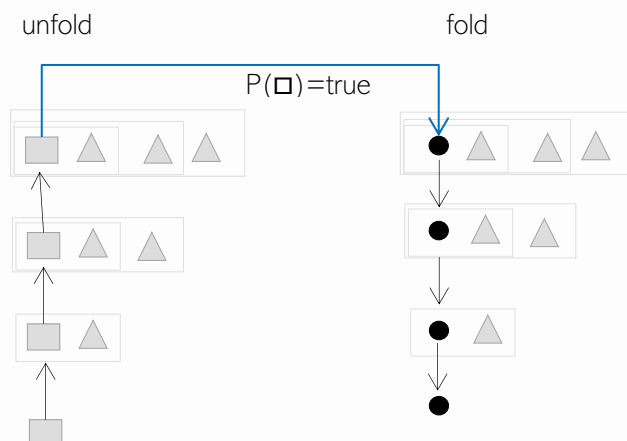
where h is a function $h: A \rightarrow C$. This can be decomposed in an anamorphic part $g: A \rightarrow A \times B$ that generates a pair of $(a: A, b: B)$ and a catamorphic part represented by the operator

$\otimes: C \times B \rightarrow C$ that folds a pair $(c: C, b: B)$ into an $a: A$ and also a conditional $P(a)$ that determines if the anamorphisms has to stop and start accumulating the generated items on a base c element of type C . In order to fully grasp the idea, you can look at the following graphic representation and then revisit the mathematical notation.

suppose that we have a co-algebra $g: \square \rightarrow [\square \triangle]$ and an algebra $h: [\bullet \triangle] \rightarrow \bullet$ the process starts with the g replacing squares \square with pairs of $[\square \triangle]$

$\square \rightarrow [\square \triangle] \rightarrow [[\square \triangle] \triangle] \rightarrow \dots$ at some point when a condition is reached, we replace the most inner square \square with a \bullet this starts the collapsing(catamorphism)

$\dots \rightarrow [[[\bullet \triangle] \triangle] \triangle] \rightarrow [[\bullet \triangle] \triangle] \rightarrow \bullet$ until we get a single $\bullet: C$



[For a practical realistic application of Hylomorphisms take a look at this [medium](#) article: [Solving the Domino problem using Hylomorphisms in Javascript](#)]

5.6.5 Hylomorphism example: Mergesort

“In any change, there must be three things:
 (1) something which underlies and persists through the change;
 (2) a “lack”, which is one of a pair of opposites, the other of which is
 (3) a form acquired during the course of the change”
 Aristoteles -*Physics* »

In this section we will see a functional approach on mergesorting an array using hylomorphism (you can see in the [Wikipedia page](#) how ugly the imperative code can become when we try to solve problems that are use divide and conquer in which functional style excels). First, we will transform an array to a binary tree using an anamorphisms, then we will gradually fold the leaves of the tree while preserving a property of ordering using a catamorphism.

1) Unfolding to a tree

We recursively break the array in half until we reach single elements that we are going to use a leaf of Nodes.

```
var ana = a => a.length == 1 ? new Leaf(a) :
  new Node(ana(a.slice(0, a.length / 2)),
    ana(a.slice(a.length / 2, a.length)));
```

```
ana([3, 7,10, 4, 1, 11, 5]) → ((3,(7,10)),((4,1),(11,5)))
```

Run This: [Js Fiddle](#)

If we wanted to use the analogy of Aristoteles, then the Underlying set of Integers that are contained in the list is the requirement (1) “something which underlies and persists through the change”

2) Folding the tree while preserving the order

Here there are two steps. Let’s say that we have as an inductive hypothesis a function `zipOrdered` that when given two ordered lists this gives us the zipped list preserving the ordering. We will define `zipOrdered` later. For now, we can assume that we have already defined it. Now we can define a recursive sort function on the Tree using pattern matching with `cata`:

```
Tree.prototype.sort = function () {
  return this.cata({
    Leaf: v => v,
    Node: (l, r) => zipOrdered(l.sort(), r.sort())
  })
}
```

This says if we are in **Leaf** just return the value **v** which is ordered because its single. Now for the **Node** we first recursively use `sort` to get the sorted lists (`l.sort()`, `r.sort()`) and then we zip them using `zipOrdered`.

Now we should define `zipOrdered` which also has a recursive definition:

```
var zipOrdered = (a1, a2) => a1.cata({
  empty: _ => a2,
  concat: (x, xs) => a2.cata({
    empty: _ => xs,
    concat: (y, ys) => x > y ?
      [x].concat(zipOrdered(xs, [y].concat(ys)))
      : [y].concat(zipOrdered(ys, [x].concat(xs)))
  })
})
```

Run This: [Js Fiddle](#)

This definition is the same as the `zip` definition but also uses **a comparison at the node level in order to decide which array has the next element in order**. You can run the jsfiddle and look

```
var ana = a => a.length == 1 ? new Leaf(a) :
  new Node(ana(a.slice(0, a.length / 2)),
    ana(a.slice(a.length / 2, a.length)));

Tree.prototype.sort = function () {
  return this.cata({
    Leaf: v => v,
    Node: (l, r) => zipOrdered(l.sort(), r.sort())
  })
}
```

```
var sorted = ana([11, 6, 5, 4, 3, 2, 2]).sort()//
```

Run This: [Js Fiddle](#)

The concept of “order-ness” was lacking from the **Set** and we had to introduce it using comparisons. The best structure to represent “comparison” as a binary relation is obviously the Tree with its left and right sides.

5.7 Fold relation with Cata

Folds are also ways to collapse a structure. Catamorphisms are more general in the sense that we have more freedom on how we access the values. We will see Folds in more detail in a later chapter. The point of this section is to highlight the similarity between `cata` and `fold` and highlight the *fact that some libraries provide fold functions that we can use in order to extract values out of certain data structures*.

we can create a fold for example for our `Id` functor (which is useless but its only for display purposes and to prove that `array.reduce` is just a subcase) In the same way that `Array.reduce` is defined

```
[0, 1, 2, 3].reduce((accumulator, currentValue) => accumulator + currentValue, 0)
```

By analogy for the `Id` Functor

```
var Id = v => ({
  map: f => Id(f(v)),
  cata: alg => alg(v),
  fold: (accumulator, reducer) => reducer(accumulator, v)
});
```

And use it like this:

```
var r = Id(5).fold([], (accumulator, currentValue) => accumulator.concat([currentValue]))
//[5]
var r = Id(5).fold(5, (accumulator, currentValue) => accumulator+currentValue)
//10
```

Run This: [Js Fiddle](#)

Alternatively, if we had a simple Product structure, we can provide a fold like this:

```
var Pair = (a, b) => ({
  map: f => pair(f(a), f(b)),
  fold: (accumulator, reducer) => reducer(reducer(accumulator, a), b)
});
```

```
var r = Pair(2,3).fold(0, (accumulator, currentValue) => accumulator + currentValue) ;
```

Run This: [Js Fiddle](#)

Notice that this is the same as the monoidal folding idea in the monoids section.

Also you can see that if we add more members to a product structure (a,b,c,d,...) then the fold would be something like that :

```
fold: (accumulator, reducer) => reducer(reducer(reducer(reducer(accumulator, a), b), c, d, ...))
```

which approximates the list definition of the fold, because in essence, an arbitrary product (a,b,c,d,...) is a list . We can also define folds for `Either`, `IO` and the `Reader` functors (we will skip this part for now)

6 Traversable

Let's say that we have a monoid for example the `Array` (`[]`, `concat`) and the `Identity` functor as an applicative. We can use the applicative chaining to merge functors that contain arrays:

```
Id(x => y => x.concat(y)).app(Id([3])).app(Id([3])); // Id([3, 3])
```

we can do that repeatedly:

```
Id(x => y => x.concat(y)).app(Id([3]))
  .app(Id(x => y => x.concat(y))
    .app(Id([5])).app(Id([4]))); // Id([3, 5, 4])
```

we can use recursion to use this in any list:

```
Array.prototype.distribute = function() {
  if (this.length === 0) return Id([]) ;
  else {
    var head = this.shift();
    return Id(x => y => x.concat(y))
      .app(head.map(x=>[x]))
      .app(this.distribute());
  }
};
```

```
var s = [1,2,3].distribute(); //Id([1, 2, 3])
```

Run This: [Js Fiddle](#)

Inside the paper that introduced the concept of applicatives "[Applicative Programming with Effects](#)" Conor McBride and Ross Paterson called this **applicative distributor** for lists:

```
distribute:: Applicative f : [f a] → f [a]    // the f from inside the list goes to the outside
distribute ([]) = ⟨ [] ⟩
distribute ([value: rest]) = ⟨ (:) value (distribute (rest)) ⟩
```

compare this to the map

```
map f ([]) = [ ]
map f ([value: rest]) = [ f(value): map f (rest)]
```

this is a loose representation where the $\langle \dots \rangle$ represents the applicative operation.

the $\langle (\text{concat}) _ _ \rangle$ represents the following expression when our **f** is the Id applicative functor :

```
Id(x => y => x.concat(y)).app(_).app( _ );
```

It is passing a lambda inside the constructor of our data structure

```

      ↙      ↘
    ⟨ _ _ _ ⟩  (x=>y=>_ concat _)
  ↙      ↘
Id(x => y => x.concat(y)).app(_).app( _ );
```

Here we have placed the curried concat function (`_=>_=>_.concat_`) Inside an applicative in order to be able to chain the applicative calls. In Haskell, for example, this is called sequenceA $:: \text{Applicative } f \Rightarrow t \ (f \ a) \rightarrow f \ (t \ a)$, and what does is to take a **traversable t that contains f(a) items and evaluates them and collect the results.**

Sanctuary.js: sequence $:: (\text{Applicative } f, \text{Traversable } t) \Rightarrow \text{TypeRep } f \rightarrow t \ (f \ a) \rightarrow f \ (t \ a)$

Ramda.js: sequence

Why get into all this trouble just to get an `Id` (`[1, 2, 3]`)? Well, we can abstract the `Id` and have a function `a → Id(b)` that will do the generation of the `Id` (`[_]`). We get this implementation:

```
Array.prototype.traverse= function(f, action) {
  return (this.length === 0) ? f ([]):
    f (x => y => [x].concat(y))
      .app(f(this.shift()))
      .app(this.traverse(TypeRep, f));
};
var result = [2, 3, 5]. Traverse( Idf, v => Idf(v + 1));
```

Run This: [Js Fiddle](#)

Here the `f` is the applicative functor constructor, aka `pure` (`_`) in the JsFiddle is written as TypeRep for Type Representable, (**TypeRep is the constructor of the Functor**). In a symbolic notation, it could be described as follows:

`traverse: Applicative f : (a → f(b)) → [a] → f [b]`

`traverse (f , []) = { [] }`

`traverse (f ,[value, rest]) = { (:) f(value) (traverse(f, rest)) }`

now if you compare the signatures of the `distribute` and `traverse` you can see that we can create `traverse` from `distribute` by first mapping the `(a → f(b))` on the array elements so we get:

`[a]. map(a → f(b)) → [f(b)]` and then apply the `distribute` to get the final `f [b]` So :

`list .traverse (g) = list. map(g). distribute ()` where `g : a → f(b)` in js :

```
var TraverseEquivalent= (list,g)=>list.map(g).distribute();
```

Run This: [Js Fiddle](#)

6.1 Traversable Array with Either applicative for validation

Let us take the Either applicative functor instead of the Identity Functor as applicative. The only difference is that the `left` side does not do anything except holding a single value

(which is usually seen as an error message). You can see below a validator that only takes values above 3 as valid if we traverse an array of ints with this validator. If there is an invalid element, this will yield an invalid(left) state else a valid(right) state containing the array

```
const right = (v) =>({ map:(f)=>right(f(v)), app: fa =>fa.map(a => v(a))});
```

```
const left =(v)=>({ map:(f)=>left(v),app:fa =>left(v)});
```

```
var validation = v =>(v > 3 ? right(v) : left("invalid " + v));
```

```
var allSuccesfull = traverse([10, 4, 5], right, validation);
```

```
var oneInvalid = traverse([2, 4, 5], right, validation);
```

Run This: [Js Fiddle](#)

The major drawback of this idea is that the traverse stops at the first false validation and it does not accumulate the validation errors. In most functional libraries, there is a Validation Type that is used for validation error accumulation.

6.2 Traversable Algebraic data structures

Let us now define traverse for Algebraic data structures, taking as an example again the simple Tree.

```
tree (a)= leaf(a) + node(tree(a) , a , tree (a))
```

This is again an inductive definition. The symbolic definition would be

```
traverse (f , leaf(a)) = leaf (f(a))
```

```
traverse (f , node(l, x, r)) = { (node) traverse (f , r) f(x) traverse (f, r) }
```

this translates to the following implementation.

```
var node = (left, v, right) => ({
  left: left,
  v: v,
  right: right,
  map: f => node(left.map(f), f(v), right.map(f)),
  traverse: (TyperRep, f) =>                                     < (node) _ _ >
    typerRep(li => vi => ri => node(li, vi, ri))
      .app(left.traverse(TyperRep, f))
      .app(f(v))
      .app(right.traverse(TyperRep, f)),
});

var leaf = v => ({
```

```

v: v,
map: f => leaf(f(v)),
traverse: (TyperRep, f) => TyperRep(vi => leaf(vi)).app(f(v)),
});

```

↖ (leaf) _

Run This: [Js Fiddle](#)

This is a bit tricky if you try to implement it for the first time. Here again in the node we have the applicative operation $\langle (node) _ _ \rangle$ where the node constructor `node = (left, v, right) => ({...})` was curried, in order to use applicative chaining. This is a repetitive theme. Take a moment to compare this with the Array equivalent $\langle (concat) _ _ \rangle$ and try to make sense of the mechanics involved in this concept by yourself.

I want to make sure that you will see the monoid homomorphism that is at play here. **We use the applicative as a monoid to fold the List.**

In our example above the equivalence is that the multiplication $\otimes_{node} : Node1(,)$ represented by the node constructor becomes equivalent with the applicative `Ap` operation of the `Id`:

$(Node(,)) \rightarrow \text{is the same} \langle (node) _ _ \rangle$

```

Id(x => y => new Node<int>(x, y)).ap(l.distribute())...;

```

↙ ↘

Two completely different domains but actually. `ap` is equivalent with the Node constructor `Node1(,)` in a different level of abstraction.

Also as we will see in the traversable section the Arrays “multiplication” $\otimes_{Array} \text{concat} (:)$ will be lifted to the now the same as the \otimes_{Reader} , applicative operation `ap` $\langle _ \rangle$ of the reader

$(:) \rightarrow \langle _ \rangle$

```

Reader(g=>x=>y=>[y].concat(x)).ap(...).ap(Reader(...)) ;

```

6.3 Identity Functor Traversable

Having implemented a tree traversable, we can take the special case where the tree only has one node without any branches. This would be our Identity Traversal:

```

var Id = v => ({
  map: f => Id(f(v)),
  traverse: (TyperRep, f) => TyperRep(x => Id(x)).app(f(v))
});

```

Here we have pruned the `left.traverse(TyperRep,f)` and `right.traverse(TyperRep,f)` branches from our tree Traversable.

6.4 Traversing with The Promise - Parallel

Now let's traverse some structures with a task. First we have to make Task applicative by adding this extension method, that allows us to use Tasks that contain functions :

```
Promise.prototype.ap = function (p1) {
  return new Promise((resolve, reject) => {
    this.then(f => p1.then(x => resolve(f(x))).catch(reject)).catch(reject);
  });
}
```

Now we can define a Distribute. This method takes a `List<Task<T>>` a list of tasks and after executing all of them collects the results in a list as a Task `Task<List<T>>` which is pretty amazing

```
Array.prototype.distribute = function() {
  if (this.length === 0) return Promise.resolve([]) ;
  else {
    var head = this.shift();
    return Promise.resolve(x => y => x.concat(y))
      .ap(head.map(x=>[x]))
      .ap(this.distribute());
  }
};

var promiseOfList=[ Promise.resolve(1), Promise.resolve(2), Promise.resolve(3)].distribute();
//Promise ([1, 2, 3])

promiseOfList.then(x=>{
  console.log(x)
})
```

Run This: [JS Fiddle](#)

we can also Define a traverse that will start from a list apply some function that return a Task and then await for all and return the results:

```
Array.prototype.traverse = function (trypeRep, f) {
  return this.matchWith({
    empty: () => trypeRep([]),
    cons: (v, r) => trypeRep(x => y => [x].concat(y))
  })
}
```



```

        .ap(f(v))
        .ap(r.traverse(trypeRep, f))
    });
}

```

```

var traversed = [1,2,3].traverse(x=>Promise.resolve(x),x=>Promise.resolve(x+1)); //Promise
e.resolve([1, 2, 3])

```

```

traversed.then(x => {
    console.log(x)
})

```

Run This: [JS Fiddle](#)

Now with this we can for example start with a list of urls that contain the Github api urls for <https://api.github.com/users> some user details

```

var userUnis: Array<string> = [
    "https://api.github.com/users/mojombo",
    "https://api.github.com/users/defunkt"];

```

Now if I have an async method that makes a Http call to get the details from the url

<https://api.github.com/users/{userName}>

```

fetch(uri).then(res => res. json ())

```

I can get in parallel for multiple users the details by using Traverse

```

var gitUserDetails = [
    "https://api.github.com/users/mojombo",
    "https://api.github.com/users/defunkt"]
    .traverse(x=>Promise.resolve(x),uri=>fetch(uri).then(x=>x.json()));

gitUserDetails.then(x => {
    console.log(x)
})

```

Run This: [JS Fiddle](#)

6.5 Applicative Reader Isomorphism with the Interpreter

Design pattern

This is another isomorphism between classic design patterns of the functional and Object-oriented world that display the interconnectedness of the two paradigms beautifully.

Let us say we have a simple data structure that represents expressions:

```
Exp v = Val i + Add (Exp v) (Exp v) + Sub (Exp v) (Exp v)
```

We only have a Val expression that contains a number, an Add that can contain two expressions and a subtract Sub. This forms again a recursive algebraic data structure.

We want to create a function eval that will evaluate an expression Exp formed by combinations of (Val, Add, Sub). The type of eval should be in this case:

```
Eval : (u:Expr, env:Env) → result:Int
```

Eval should take an expression u:Expr and a context (env) and return a result of type result:Int. In our example the context will be empty for simplicity, but we will not omit it in the code.

The idea here is to create a reader Applicative which will traverse each expression, apply the context and accumulate the result. I rewrite here the applicative reader, just to look at it again.

```
var Reader = (expr) => ({
  run: env => expr(env),
  ap: reader=>Reader(env=> Reader(expr).run(env)(reader.run(env)))
});

var res = Reader(g=>x=>g.x2+x). ap(Reader(g => g.x1));
var r =res.run({x1:10,x2:20}); //30
```

below is my implementation, try to play around with the fiddle

```

var AddExpr = (leftExpr, rightExpr) => ({
  leftExpr: leftExpr,
  rightExpr: rightExpr,
  eval: (env) => Reader(g=>x=>y=>x+y)
    .ap(Reader(g => leftExpr .eval(g)))
    .ap(Reader(g => rightExpr.eval(g)))
    .run(env)
});

var valueExpr = v => ({
  eval: (env)=>Reader(g=>v).run(env)
});

var resultAdd = AddExpr(valueExpr(4), valueExpr(5)).eval({}) ;//9

```

Run This: [Js Fiddle](#)

The algebra of eval using our applicative notation $\langle \rangle$ is

$\text{eval} :: (\text{Expr } u) \rightarrow \text{Env } v \rightarrow b$

$\text{eval } (\text{Val}(i), g) = \langle i \rangle$

$\text{eval } (\text{Add}(x, y), g) = \langle (+) \text{ eval } (x, g) \text{ eval } (y, g) \rangle$

Take a moment to appreciate the use of the *reader applicative* to evaluate expressions.

6.5.1 A Catamorphism implementation

6.5.2 Object Oriented Interpreter Pattern implementation

In the object-oriented world exists something isomorphic to the above implementation: the **Interpreter design pattern**. I am stating bellow an implementation based on ECS6 in order to emphasize the object-oriented characteristics:

```

class Expr { }

class ValExpr extends Expr {
  constructor(value) {
    super()
    this.value = value;
  }
  eval(g) {
    return this.value;
  }
}

```

```

    }
  }

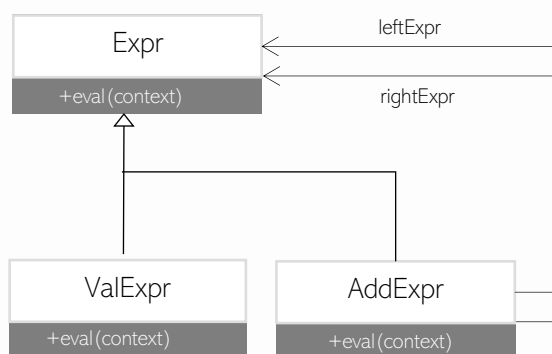
  class AddExpr extends Expr {
    constructor(leftExpr, rightExpr) {
      super()
      this.leftExpr = leftExpr;
      this.rightExpr = rightExpr;
    }
    eval(g) {
      return this.leftExpr.eval(g) + this.rightExpr.eval(g);
    }
  }

  var addition = new AddExpr( new Val(4),new Val(5));

  var result = addition.eval({}); //9

```

Run This: [Js Fiddle](#)



As you can see, those two implementations are pretty much equivalent. We can argue which one is better, but the object-oriented implementation looks easier. Easy, intuitive implementation is one of the advantages of object-oriented programming, even if the interpreter design pattern is not used often in development. On the opposite side, functional applicative implementation in my opinion looks prettier.

6.6 Composing Traversables

Traversables are closed under composition. We can formulate a valid **traverse** operation for our two functor composition by just pushing the traverse operation to the inner functor and then traverse with this function the inner Functor

```
var composeF = FG => ({
  value:()=>FG,
  map: fab => composeF(FG.map(x => x.map(fab))),
  traverse:(TyperRep,f)=>FG.traverse(TyperRep, x =>x.traverse(TyperRep,f) )
});
```

Run This: [Js Fiddle](#)

The type of `x.traverse(TyperRep,f)` is applicative of type `TyperRep` because after the traversal with `traverse` the applicative type is now at the outside of the G functor. Look at this simple example where we compose two identity Functors.

```
var result = composeF(Id(Id(4))).traverse(a => [a], a => [a + 1]);//[Id(Id(5))]
```

Alternatively, maybe we choose to try the opposite and traverse composite Arrays `[[[]]` with an `Id`.

```
var result = composeF([[1, 2, 3], [5, 6]]).traverse(a => Id(a), a => Id(a + 1));
//Id( [[2,3,4],[6,7]])
```

Run This: [Js Fiddle](#)

You can try other variations like composing trees etc. by forking some of the previous jsFiddles. I think that traversable is an underrated concept and there should be used more often in real situations.

6.7 Foldable

In this section, we are going to extend traversable structures to foldables. Foldables are the practical implementation of the Traversable. The main motivating idea behind Foldables **is to combine traversing with a monoid that will accumulate the values**. So Monoids come again to the front scene. We will begin with the Sum monoid on integers:

```
var Sum = { concat: (a, b) => a + b, mempty: 0 };
```

we would like to **create an applicative functor based on a monoid**, in order to use it with a traversable, to perform the accumulation.

Within the same paper of "[Applicative Programming with Effects](#)" Conor McBride and Ross Paterson used a constant functor

```
var Acc = m => v => ({
  return: v,
  pure: () => Acc(m)(m.empty),
  map: f   => Acc(m)(f(v)),
  app: fa  => fa.map(a => m.concat(a, v))
});
```

Run This: [Js Fiddle](#)

Acc represents a constant functor `Const` such that, the value of type `Const(a, b)` holds a value `a` which is unaffected by `map`. We only need this `Const` functor to store a monoid `m` and only use the applicative operations, which in turn only utilize the monoid.

Monoid `a => Applicative (Acc a)` where

```
pure ( _ ) = Acc(empty)
Acc (o1 ) ⊗ Acc (o2 ) = Acc (o1.concat(o2).)
```

```
Array.prototype.traverse = function(TyperRep, f) {
  return this.length === 0
    ? TyperRep([])
    : TyperRep(x => y => [x].concat( y ))
      .app(f(this.shift()))
      .app(this.traverse(TyperRep, f));
};

var accSum = Acc(Sum);
var f = x => x*x;
var result = [2, 3, 5].traverse(accSum()).pure , x=>accSum(f(x)).v;
```

Run This: [Js Fiddle](#)

You can try some other monoids to verify that this is correct

```
Accumulate: (a→m) → t(a) → m //where t : traversable and m: monoid
accumulate f = return ◦ traverse (Acc ◦ f)
```

The `accumulate` also is known as [FoldMap](#) in Haskell. Now if we use as an `f` above the `id` `:x→x` then we get the definition of `fold` or `reduce` from `traverse`

```
fold: t(a) → m // where t : traversable and m: monoid
fold = return ◦ traverse (Acc ◦ id)
```

```
Array.prototype.fold = function(acc) {
  return this.traverse (acc().pure, acc).v
};
```

Run This: [Js Fiddle](#)

This way of definition was followed in order to display the connection between traversable, monoid and foldable

6.7.1 FoldMap

Let's discuss foldMap a bit more. FoldMap takes a function that return a monoid ($a \rightarrow m$) and apply it on a traversable in order to get a single value of type m . This is actually a variation of Fold.

Foldmap: $(a \rightarrow m) \rightarrow t(a) \rightarrow m$ where m is a monoid

So we define foldMap directly from fold like below (I used here the native js reduce)

```
Array.prototype.foldMap = function(m,f) {
    return this.reduce((acc,i)=>m.concat(acc,f(i)), m.empty )
};
```

Run This: [Js Fiddle](#)

With the help of foldMap we can define some operations more intuitively, for example for this conjunction monoid:

```
var AndBoolM = { concat: (a, b) => a && b, empty: true };
```

we can define All That return true if all the members of a traversable satisfy a condition

```
Array.prototype.All = function (f) {
    return this.foldMap(AndBoolM, f);
};
```

```
var allEvens = [1, 4, 5]. All(x => x % 2 == 0) ;//false
```

Run This: [Js Fiddle](#)

We can also use a direct implementation of a foldMap without traverse or fold

```
Array.prototype.foldMap = function (empty, f) {
    return this.cata({
        empty: _ => {
            return empty;
        },
        concat: ({ value, rest }) => {
            return f(value).concat(rest.foldMap(empty,f))
        }
    });
}

var Sum = v => ({ v:v, concat: (b) => Sum(v + b.v) });
var sum = [1, 4, 5, 67, 8].foldMap(Sum(0), x => Sum(x))
```

Directly chaining
with the concat that
monoid exposes

Run This: [Js Fiddle](#)

If we provide an **f that return something that have a concat method** that we could use in order to accumulate over.

Look this tree foldMap implementation as well .

```
Tree.prototype.foldMap = function (empty,f) {
  return this.cata({
    Leaf: v => f(v),
    Node: (left, v, right) => {
      return left.foldMap(empty,f).concat(f(v)).concat(right.foldMap(empty,f))
    }
  });
}
```

```
var Sum = v => ({ v:v, concat: (b) => Sum(v + b.v) });
```

```
var sum = new Node(new Leaf(3), 4, new Node(new Leaf(3), 4, new Leaf(3)))
.foldMap(Sum(0), x => Sum(x))
console.log(sum.v)
```

Run This: [Js Fiddle](#)

6.7.2 Filterable structures

We can use recursion to implement filter functions on an array like this:

```
Array.prototype.filter2 = function (predicate) {
  return this.cata({
    empty: _ => []
    concat: ({ value, rest }) => {
      return ( predicate(value)? [value] : [] ).concat(rest.filter(predicate))
    }
  });
}
var sum = [1, 4, 5, 67, 8].filter( x=>x>5);
```

Run This: [Js Fiddle](#)

here we just accumulate values over **an Array [] only if predicate(value) is true**. But we can also use foldMap to implement filter functions over Traversable structures

```
Array.prototype.filter3 = function (predicate) {
  return this.foldMap([], x => predicate(x) ? [x] : []);
}
```

Run This: [Js Fiddle](#)

6.8 Fold and FoldMap Derivation from Catamorphism

Finally, we could use our cata definition to define Fold without rewriting the recursive parts. If we have any monoid `monoid` we can just return an algebra that uses this monoid to concatenate all the values of the Traversable.

```
Array.prototype.cata = function (algebra) {
  return this.matchWith({
    empty: () => algebra.empty(),
    concat: (v, r) => algebra.concat(v, r.cata(algebra))
  })
}
```

```
var foldAlgebra = monoid => ({
  empty: () => monoid.empty(),
  concat: (v, r) => monoid.concat(v, r)
});
```

```
var foldAlgebraSum = foldAlgebra({ empty: () => 0, concat: (x, y) => x + y })
var sum = [1, 2, 3, 4, 5].cata(foldAlgebraSum);
console.log(sum);
```

Run This: [Js Fiddle](#)

that's pretty much the definition of the fold on a List.

TypeScript

```
type ListPatternAlg<T, T1> = ({ empty: () => T1, cons: (v: T, rest: T) => T1 });

function FoldAlgebra<T>(monoid: monoid<T>): ListPatternAlg<T, T> {
  return ({
    empty: () => monoid.empty(),
    cons: (v, r) => monoid.concat(v, r)
  });
}
```

```
var array: Array<number> = [1, 2, 3];
console.log(array.Cata(FoldAlgebra(Sum)));
```

Run This: [TS Fiddle](#)

Or if we want to use the accumulation form of the monoid then we get the foldMap equivalent

```
class Sum {
```

```

    constructor(v) { this.v = v; }
    static empty() { return new Sum(0); }
    concat(b) { return new Sum(this.v + b.v); }
  };

  var monoidAccumulation = {
    empty: () => Sum.empty(),
    concat: (x, y) => new Sum(x).concat(y)
  };

  var foldAlgebra = monoid => ({
    empty: () => monoid.empty(),
    concat: (v, r) => monoid.concat(v, r)
  });

  var foldAlgebraSum = foldAlgebra(monoidAccumulation)
  var sum = [1, 2, 3, 4, 5].cata(foldAlgebraSum);
  console.log(sum.v);

```

Run This: [Js Fiddle](#)

6.9 Traverse Derivation from Catamorphism

The applicative is a monoid so we can use this to generate an algebra to feed our cata for example for the Id functor (as applicative) we have this monoid

```

  var applicativeMonoid = (typeRep, f) => ({
    empty: () => typeRep([]),
    concat: (x, r) => typeRep(v => r => [v].concat(r)).ap(f(x)).ap(r)
  });

  var foldAlgebra = monoid => ({
    empty: () => monoid.empty(),
    concat: (v, r) => monoid.concat(v, r)
  });

  var traverseAlgebra = applicativeMonoid(Id, x => Id(x + 1))
  var foldAlgebra = foldAlgebra(traverseAlgebra)

  var sum = [1, 2, 3, 4, 5].cata(foldAlgebra);
  console.log(sum.v);

```

Run This: [Js Fiddle](#)

6.9.1 Mixins

Because for every **foldable** derivation we have a **traversable** as a base, we can use mixins to provide a default implementation of foldable if a functor is traversable. In object-oriented programming languages, a **mixin** is a class that contains methods for use by other classes without having to be the parent class of those other classes. Mixins are a form of adding functionality on classes. Our implementation just Copy all the properties of a Mixin class to another class.

```
Object.prototype.extend = function (mixin) {
  Object.keys(mixin).forEach(key => this[key] = mixin[key](this));
  return this;
};
```

We can define for example a **foldableMixin** Mixin which is just an object with some method that we want to copy on another class.

```
var foldableMixin = {
  fold: self => acc => self.traverse(acc().pure, acc).v
};
```

Run This: [Js Fiddle](#)

And we can use it like this

```
var SumM = { concat: (a, b) => a + b, empty: 0 };

var foldArray = [1, 2, 3].extend(foldableMixin).fold(Acc(SumM));
```

6.9.2 Composing Foldables

Now that we have the implementations of traversable of **composeF** and the **foldableMixin** we can fold any composition of traversables.

```
composeF([[1, 2, 3], [5, 6]]).extend(foldableMixin).fold(Acc(SumM)); //17
composeF(Id([1, 2, 3])).extend(foldableMixin).fold(Acc(SumM)); //6
composeF([Id(1), Id(2), Id(3)]).extend(foldableMixin).fold(Acc(SumM)); //6
```

Run This: [Js Fiddle](#)

Or why not try tree of lists:

```
var tree = node(leaf([1, 3, 5]), [7], node(leaf([9]), [11, 13], leaf([15])));
var result = composeF(tree).extend(foldableMixin).fold(Acc(SumM)); //64
```

Run This: [Js Fiddle](#)

Also, we can use other monoids to fold. For example, with the help of the *Array monoid* we can fold the previous structure in this manner:

```
var ListM = { concat: (a, b) => a.concat(b), empty: []};
```

```
var r = composeF(Id([1, 2, 3])).map(x=>[x])
.extend(foldableMixin).fold(Acc(ListM)); //[3, 2, 1]
```

```
var r = composeF([Id(1), Id(2), Id(3)]).map(x=>[x])
.extend(foldableMixin).fold(Acc(ListM)); //[3, 2, 1]
```

Run This: [Js Fiddle](#)

it's like a [flatMap](#) but between two different functors. You can try to fold compositions using different monoids like Min, Max, Or etc.

6.9.3 Iterators

Since a *Functors* can be viewed as a container of values, it is reasonable to be able to get the values out of a functor in a specified linear manner and use it directly in a loop, for example. The iterator pattern is so integrated into programming that we usually assume it.

```
var Id = v => ({
  map: f => Id(f(v)),
  *[Symbol.iterator]() {
    yield v;
  }
});
```

```
for (let value of Idf(3)) {
  console.log(value);
}
```

Run This: [Js Fiddle](#)

Javascript defines an object as **iterable** if it defines its iteration behaviour, such as what values are looped over in a [for...of](#) construct. From the discussion in the previous sections I hope it's apparent that we can provide on the fly a generic Iterable object for any traversable, since traversable is a much stronger operation. It's easy to collect the values in an array which we will return as an iterable.

We should be able to get an iterable implementation that is independent of any specific implementation of traversable and foldable.

```
Object.prototype.getIterator = function () {
  var self = this;
  return {
    *[Symbol.iterator]() {
      {
        var content = self
          .map(x => [x])
          .extend(foldableMixin)
          .fold(Acc({concat: (a, b) => a.concat(b), empty: [] }));
        for (var element of content) yield element;
      }
    }
  }
}
```

Run This: [Js Fiddle](#)

We could define iterators in a more ad hoc way in order to optimize it, but in complex structures, we can get the derivation from *traversable* and avoid any implementation confusion or possible bugs.

```
var tree = node(leaf(1), 2, node(leaf(3), 4, leaf(5)))
for (let value of tree.getIterator()) {
  //...
}
```

Run This: [Js Fiddle](#)

Basic Folds summary

Name	Signature
Sequence	Applicative $f : [f\ a] \rightarrow f\ [a]$
Traverse	Applicative $f : (a \rightarrow f(b)) \rightarrow [a] \rightarrow f\ [b]$
FoldMap	Monoid $m : (a \rightarrow m) \rightarrow t(a) \rightarrow m$
Fold	Monoid $m : t(a) \rightarrow m$

7 The Yoneda Lemma

The Yoneda Lemma is arguably one of the most important results of category theory. The [nlab](#) calls it "an elementary but deep and central result in category theory" and [Dan Piponi](#), calls it "the hardest trivial thing in mathematics.". Fortunately, its consequences are completely embedded in Programming in a way that there is no reason to understand it at all. We are going to immediately define the Yoneda lemma in programming terms and then follow with some examples.

So, Yoneda lemma for the purposes of programming it states that:

$\forall x. (a \rightarrow x) \rightarrow F(x) \cong F(a)$ where **F is a Functor**

if for some fixed a we have a function from $(a \rightarrow x)$ for any x and we have a $F(x)$ then this is the same as having the $F(a)$ itself. If we take F as the array functor $[]$ we can write:

$\forall b. (a \rightarrow b) \rightarrow [b] \cong [a]$

When the functor F is the Identity functor then we have:

$(a \rightarrow x) \rightarrow x \cong a$

This is the famous **continuation passing style** and feel free to come back here again and revisit this section when we discuss the continuation monad and the connection of logic - Type theory and Category theory. This means that if we have a callback of a , is as good as having an a itself

```
var callback = action => a => action(a)
```

we can transform every function that takes an element to an equivalent one that takes a callback

```
var f = function (x) {return x+5;} //type of: int →int
```

```
var f = function (callback) {callback(x+5);} // type of : (int →void)→ void
```

this is called continuation passing style transformation.

If we interpret this $(a \rightarrow b) \rightarrow b$ it says if you give me an function from $a \rightarrow b$ and from this we can get somehow a b for any b [$\forall b. (a \rightarrow b) \rightarrow b$], then I must surely have an a somewhere hidden, in order to be able to get **always** a b .

7.1 Co-Yoneda

The Co-Yoneda lemma is the dual of the Yoneda lemma:

$F(x) \rightarrow (a \rightarrow x) \cong F(a)$ where F is **not necessarily** a functor

With this coyoneda interpretation if we use as $F = (_ \rightarrow c)$ we get this equivalence

$$(x \rightarrow c) \rightarrow (a \rightarrow x) \cong (a \rightarrow c)$$

This is the default definition of composition. The usual implementation of Coyoneda for javascript is the following:

```
const Coyoneda = (x, f) => ({
  map: g => Coyoneda(x, x => f(g(x))),
  lower: () => f(x) ,
});
```

Run This: [Js Fiddle](#)

what co-yoneda does is stores all the functions by composition $x \Rightarrow f(g(x))$ that are applied through the map method. **In order to initialize it we use the identity function**

```
Coyoneda.lift = x => Coyoneda(x, x => x)
```

And when we want to collapse the structure and actually get back a result, we have to use the final function composition f with the argument x with `lower()`.

```
var result = Coyoneda.lift(5)
  .map(x=>x*2)
  .map(x=>x-1);
```

```
console.log(result.lower());
```

Run This: [Js Fiddle](#)

We can do that any way we want but we assume here that a simple Type x has been used so we can just apply $f(x)$ and get a final value. **This means that the initial x that was lifted doesn't need to be a functor** if we finally know how to apply f to x . One application of coyoneda is when we want to provide a Functor, but we don't have one yet so `Coyoneda.lift` gives us a functor on any arbitrary type x .

With Coyoneda we can create Functros on the fly just to use the map method to chain computation in an optimal way. For example, we can rewrite the idiomatic decorator design pattern with the use of CoYoneda like this

```
var createProduct = (price) => ({ price: () => price, })
var addPackaging = (product) => ({ price: () => product.price() + 0.5, })
var addRibons = (product) => ({ price: () => product.price() + 0.3, })

//with our identity functor just to provide us with map support
var finalProduct = Coyoneda.lift (20)
  .map(createProduct)
  .map(addPackaging)
  .map(addRibons)
  .lower();
```

Run This: [Js Fiddle](#)

A nice property of Coyneda is the Loop fusion that you get on the map. When we say Loop fusion we mean the merging of separate iterations in a single loop.

```
for (i = 0; i < n; i++)
  console.log(a[i]) ;
for (i = 0; i < n; i++)
  console.log(b[i])

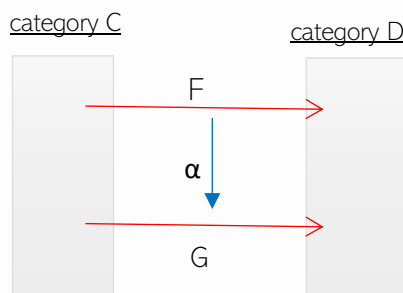
for (i = 0; i < n; i++)
{
  console.log(a[i])
  console.log(b[i])
}
```


8 Natural Transformations

"I didn't invent categories to study functors;
I invented them to study natural transformations."
-Saunders Mac Lane

In category theory, a **natural transformation** provides a way of transforming one functor into another while respecting the internal structure (i.e., the composition of morphisms) of the categories involved. Hence, a natural transformation can be considered as a "morphism of functors". Natural transformations are, one of the most fundamental concepts in category theory, after categories and functors.

If we have two functors F and G between two categories sometimes we may have a consistent transformation between them this is called natural transformation in the language of category theory. And we represent it with an arrow from one functor to the other



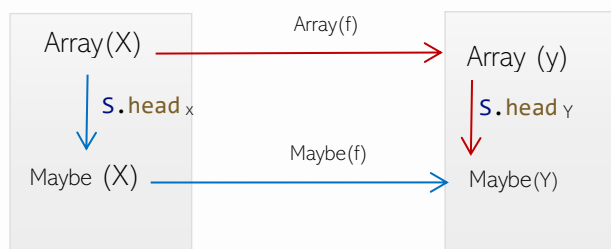
Let's see a simple example if we use `list[0]` to get the first element/head of a list we may get an undefined in case of the empty list `[]`. But we could implement a `safeHead` function that would return a `Maybe[a]`. This is a transformation between the List and Maybe Functors

```
Array.prototype.safeHead = function () {
  var head = [...this].shift();
  if (head )
    return some(head );
  else
    return none();
}
```

Sanctuary.js has an implementation of the head function that works like this

```
S.head ([1, 2, 3])    //Just (1)
S.head ([])           //Nothing
```

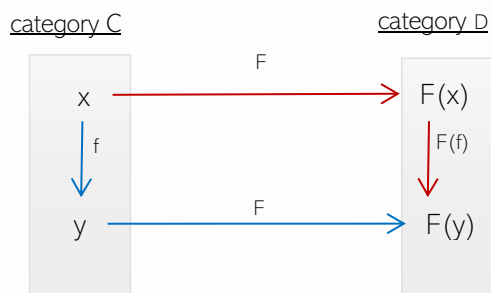
we can represent this in a diagram



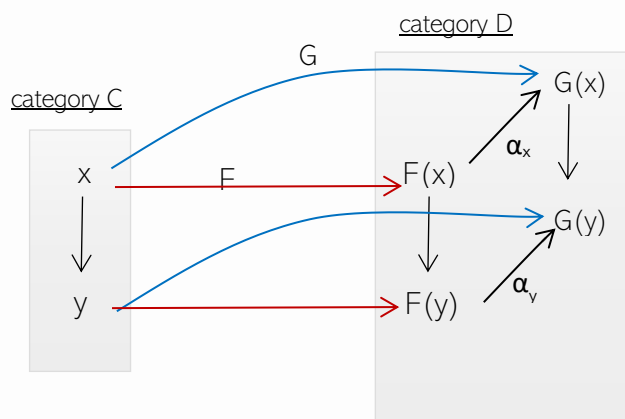
For the head definition to be a valid natural transformation the diagram above should commute. This means that the following must hold for all arrays x:

```
s.head( X.map(f) ) == s.head (X).map(f)
```

Run This: [Js Fiddle](#)



if you take the diagram of a functor F between C and D and you try to add another functor diagram in the same picture then you can visualize the natural transformation like this



α is denoted with an index x . The α_x it's called the component of α at x .

8.1.1 Natural Transformation Between Maybe and Either

Another common example of natural transformation that is very common and practical is between Either and Maybe functors. If we have a Maybe.None, we can assume it is equivalent with an Either.Left that contains an arbitrary value depending on the situation.

```
var some = (v) => ({
  map: (f) => some(f(v)),
  toEither: (defaultLeft) => right(v),
  cata: alg => alg.Ok(v)
});

var none = () => ({
  map: (f) => none(),
  toEither: (defaultLeft) => left(defaultLeft),
  cata: alg => alg.Error()
});
```

Run This: [Js Fiddle](#)

Let's say we have a repository that returns a Maybe type for demonstrative reasons. We can add a default error message in case there were no data found.

```
var clientRepositoryMock = {
  getById: (id) => Promise.resolve(some({ name: 'jim', age: 29 }))
};

clientRepositoryMock.getById().then(response => {
  response
    .toEither("could not find any client") // enhance the response by providing a
    .map(x => x.name)                     //default error message in case of none
    .cata({
      Ok: result => console.log("client name: " + result),
      Error: error => console.log("error: " + error)
    })
});
```

Run This: [Js Fiddle](#)

Similarly, we can go from Either to Maybe, if we drop the value of the Either.Left.

```
const right = (v) => ({
  map: (f) => right(f(v)),
  toMaybe: () => some(v),
  cata: alg => alg.Ok(v)
});

const left = (v) => ({
  map: (f) => left(v),
  toMaybe: () => none(),
});
```

```

    cata: alg => alg.Error(v)
  });

right(5).map(x => x * 5).toMaybe()
.cata({
  Ok: v => console.log(v),
  Error: v => console.log("error: " + v)
});

right(5).toMaybe().map(x => x * 5)
.cata({
  Ok: v => console.log(v),
  Error: v => console.log("error: " + v)
});

left("error occured").toMaybe().map(x => x * 5)
.cata({
  Ok: v => console.log(v),
  Error: () => console.log("none")
});

```

Run This: [Js Fiddle](#)

You can find this implementation in most of functional libraries for example sanctuary has [S.maybeToEither](#), [S.fromEither](#). Some other obvious and very trivial transformations would be for example from `Id` to `IO`, or `IO` to reader and state. You can try as an exercise to create *IO to either by running safely the IO and returning a Right if there is no exception*.

```

var IO = fn => ({
  map: f => IO(() => f(fn())),
  run: () => fn(),
  toId: () => Id(fn())
});

var Id = v => ({
  map: f => Id(f(v)),
  cata: alg => alg(v),
  toIO: () => IO(() => v)
});

Id(5).toIO().map(x => x + 2).run() == IO(() => 5).map(x => x + 2).run()

```

Run This: [Js Fiddle](#)

Libraries

Natural transformations

Crocks: has a collection of transformations between different structures they call [Transformation functions](#) many of them targeting usual suspects like maybe, either and arrays.

Monads

9 Monads

The Idea:

When we use **map** on a function $f: A \rightarrow B$ we transform the value A inside the functor $T(A)$ to a new type $T(B)$. Sometimes we may happen **that type B is itself a $T(A)$** . In those situations we end up with something like **$T(T(A))$** if T is a Monad it has a method **Join** that allows us to go back to the initial state **$T(A)$** . join operation is idempotent which is a nice property to have as we saw, because it leads to uniformity. This Join operation is what makes a monad a monad.

Monad is an abstraction that allows us to use common characteristics while different monads providing different effects.

We will now extend our Identity functor to make it into an identity monad.

```
const Id = (v) => ({
  value: v,
  bind: (f) => f(v),
  map: (f) => Id (f(v))
})
```

The new thing here is the bind method. Sometimes we might find ourselves in a situation where when we use map we have to map to the same type as the functor:

```
Id(5).map(x=>Id(x+1))
```

Then we end up with this `Id(Id(6))` how would you get out of this ? well if we drop the `Id` on the map `map: (f) => Id (f(v))` would be fine. So just for those times when we want to map to a functor we use another method instead of map called bind `bind: (f) => f(v)`,

```
Id(5).bind(x=>Id(x+1)) == Id(6)
```

Run This: [Js Fiddle](#)

The same reasoning goes for all functors , here some other examples leading to the same situation :

```
IO(()=>5).map(x=>IO(()=> x+1)) === IO(()=>IO(()=> 6))
```

```
[5].map(x=>[x+1]) === [[5]]
```

```
maybe(5).map(x=>x+1) === maybe(maybe(5))
```

we can also define a join operation that flattens the $T(T())$ situation `join: () => v`,

```
Id(Id(5)).join() == Id(5)
```

Because `join() \cong bind(x=>x)` (can you verify that this is the case?) usually we define only the more generic which is `bind`.

So that's the monad. Now let's look a more rigorous definition just for completeness

Definition : A monad is just a monoid in the category of endofunctors

This definition has become an internal joke in the functional community. Because it mystifies a simple concept. We already have seen all the components of this definition. But you must think of all of them at the same time to get monad.

1. **A functor** is a container that can be mapped.
2. **Endofunctor** is a functor from a category to itself.
3. **Monoid** is a structure with a single binary operation and an identity element.

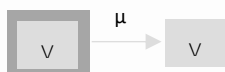
Monads have two operations called return and join (those are the **monoid** operations):

1. $\eta: v \rightarrow T(v)$ //Return (this is the empty of the monoid)
2. $\mu: T(T(v)) \rightarrow T(v)$ //Join (this is the concat of the monoid)

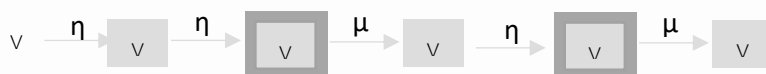
1) In this first operation $v \rightarrow T(v)$ we can start from an object v and get a container $T(v)$. This is our way to get monads. This is just a fancy way to say **Monad constructor** in programming terms



2) The second one $T(T(v)) \rightarrow T(v)$ means that we can join/flatten the two containers in one.



So, no matter how many applications of enclosures we have we can always get the Type of $T(v)$.



We can keep applying the η operation to get a wrapped object and then μ to flatten the two containers. The join operation comes from the more generic `bind` that we use in everyday development called **bind** or chain if we pass inside the *identity function* $x \Rightarrow x$

```
bind: T(v) → (v → T(u)) → T(u)
```

in the rest of this chapter we are going to implement this bind method for all the functors we have already seen until now.

Category Theory

Endofunctor

A Functor from a category to itself is called an endofunctor. Given any category C , we can create a new category $\text{End}(C) = C \rightarrow C$ that is called the endofunctor category of C . The objects of $\text{End}(C)$ are endofunctors $F:C \rightarrow C$, and the morphisms are **natural transformation** between such endofunctors.

A monoid in this endofunctor category is called a monad on C .

9.1 The List Monad

If we have a list of integers and try to get a list of the squares, cubes and quads, we might try to go about it by using map:

```
var list = [2, 3].map(x => [x * x, x * x * x, x * x * x * x])
```

Run This: [Js Fiddle](#)

this would create a list- in-a-list `var list = [[4,8,16], [9,27,91]]`

It would be great if there was a way to just get the union of those sub-list elements. Well that's exactly the purpose of the list as a monad. Let's revisit one by one the operations of a monad specifically for the list.

1. $\eta : c \rightarrow T(c)$ represents the array construction by the element `[]`. If we apply list constructor once more we get a list in a list `[[]]`
2. $\mu : T(T(c)) \rightarrow T(c)$ is the join operation that reverses the one level of containment that was added and gives us the $T(c)$ again `[]`

```
[[ ]].flatMap(x=>x) = [ ]
```

3. Run This: [Js Fiddle](#)

9.2 The Identity Monad

Identity Monad is the simplest form of a monad.

```
const Id = (v) => ({
  value: v,
  bind: (f) => f(v),
  map: (f) => Id (f(v))
})
```

It does not do much. It doesn't have any computational value. The new thing with the Identity Monad is the bind method:

```
this.bind = f => f(v);
```

The type of the bind operation is

$$\text{bind}: (T \rightarrow M [T1]) \rightarrow M [T1]$$

That takes a function $T \rightarrow M [T1]$, which, when provided with the contained value v it returns a new monad $M [T1]$ of type $T1$.

```
const Id = (v) => ({
  value: v,
  bind: (f) => f(v),
  map: (f) => Id (f(v))
})

var result = Id (2)
               .bind(x => Id (2))
               .bind(y => Id (x + y));
```

Run This: [Js Fiddle](#)

Let's put a monad in a monad and Bind with the identity $x \Rightarrow x$ to get the initial Monad back. This should remind you the Array with the [flatMap](#).

```
const Id = (v) => ({
  value: v,
  bind: (f) => f(v),
  map: (f) => Id(f(v))
})

var s = Id (Id (2)).bind(x => x)
```

Run This: [Js Fiddle](#)

The difference between bind and map in the identity monad is that map passes the lifted value to a monad constructor $\text{Id}(f(v))$ while bind does not $f(v)$. In other monads the map will be significantly different than the bind.

9.2.1 Optional Monad laws for Identity Monad

A monad has 3 straightforward laws, which are:

1. Left identity, applying the unit function to a value and then binding the resulting monad to a function f is the same as calling f on the same value.

```
var value = 2;
var f = x => Id (x*x);
var m1= Id (value).bind(f);
var m2 = f(value);
console.log(m1.value===m2.value)//true
```

Run This: [Js Fiddle](#)

2. Right identity, binding the unit function to a monad doesn't change the monad

```
var value = 2;
var m1 = Id (value).bind(Id);
var m2 = Id (value);
console.log(m1.value === m2.value)
```

Run This: [Js Fiddle](#)

3. Associativity, if we have a chain of monadic function applications, it doesn't matter how they are nested: `m.bind(f).bind(g) == m.bind(x => f(x).bind(g));`

```
var m = Id (1);
var f = x => Id (x * 2);
var g = x => Id (x * 5);

var m1 = m.bind(f).bind(g);
var m2 = m.bind(x => f(x).bind(g));

console.log(m1.value === m2.value)//true
```

Run This: [Js Fiddle](#)

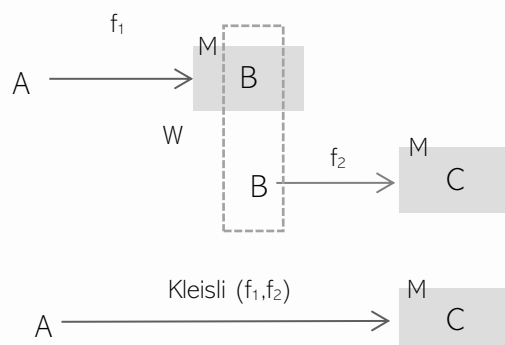
Both left and right identity guarantee that applying a monad to a value will just wrap it: the value won't change, nor the monad will be altered. The last law guarantees that monadic composition is associative. All laws together make code more resilient, preventing counter-intuitive program behaviour that depends on how and when you create a monad and how and in which order you compose functions.

9.3 Monads -Kleisli Composition in Javascript

if we have a two functions $f_1 : A \rightarrow B$, $f_2 : B \rightarrow C$ we know how to compose them:

```
var composition = x=>f2(f1(x))
```

But if we have two functions $f_1 : A \rightarrow M[B]$ and $f_2 : B \rightarrow M[C]$ that return a monad, we cannot directly compose them. Because in order to use the second function, we need B instead of $M[B]$ which is the result of f_1 .



this kind of monadic composition is called Kleisli composition. if we have again the Identity Monad for simplicity let's see how would implement the Kleisli composition. if we have two functions:

```
var f1 = x=> Id(x*x);
var f2 = x=> Id(2*x);
```

then their composition can be done a couple of ways. We could apply A to $f1 : A \rightarrow M[B]$ and get $M[B]$ the monad of B then we could apply $f2: B \rightarrow M[C]$ by mapping on $f(a)$ `f1(a).bind(f2)` in order to let f2 go underneath $M[C]$ and thus get $M[M[C]]$, which then flatMap with the Bind operation and get The $M[C]$ result.

```
var composition= a => f1(a). map(b=>f2(b)). bind(x=>x);
console.log(composition (4).value); //prints 32
```

Run This: [Js Fiddle](#)

Another way to do that would be this:

```
var composition = a => f1(a). bind(f2);
console.log (composition (4).value);
```

Run This: [Js Fiddle](#)

9.4 Optional Monad laws following Kleisli formulation

Let's get back to the monadic laws from the previous section. We can use the kleisli composition to rewrite them in a more intuitively way.

```
var compose = (f, g) => a => f(a).bind(g);
```

Using this definition, the three monadic laws can be expressed like this:

Left identity : `compose(id, f) \equiv f`

Right identity: `compose(f, id) \equiv f`

Associativity: `compose(f, compose(g, h)) \equiv compose(compose(f, g), h)`

Run This: [Js Fiddle](#)

it's now easy to see that monad composition is an associative operator with left and right identities.

This definition is not restricted in the identity Monad. Category theory in this case provides an intuition and a formal proof, for Kleisli definition assuring us that this definition will hold regardless the specific Monad. In the latter monad chapters, there would be usually a jsfiddle that will contain a proof of monadic laws for each monad following the Kleisli formulation. In any case, is a nice exercise to try to prove the monadic laws by yourself using kleisli.

This is another important way to express the three monad laws, because they are precisely the laws that are required for monads to form a mathematical category. This category is called Kleisli category C_T that arises from any Category C and a Monad T

1. The class $\text{ob}(C_T)$ of objects is the same with C ;
2. The class $\text{hom}(C_T)$ of functions are all the functions $f: X \rightarrow T Y$

as we just saw in the first chapter in order to form a category the morphisms should also follow Associativity and have an Identity, those are exactly the left and right identity and associativity laws coming from Kleisli compose function above.

9.5 Maybe Monad

Maybe monad is extension of the Maybe Functor. In order to make this maybe functor implementation into a Monad we must provide a bind method that combines two Maybe monads into one. **There are 4 different ways to combine the 2 possible states of maybe (some,none)** one can see that a valid implementation would be the following :

```
const some = (v) => ({
  map: (f) => some(f(v)),
  bind: (f) => f(v),
});
```

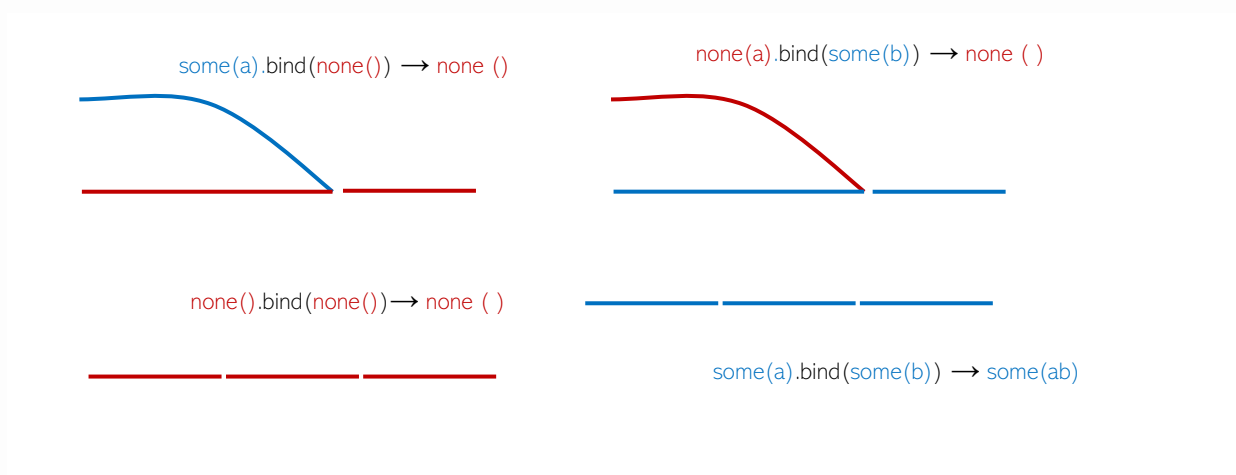
```
const none = () => ({
  map: (f) => none(),
  bind: (f) => none(),
});
```

Run This: [Js Fiddle](#)

the only combination that leads to a new some (__) is when the two paths that both have values are being combined.

```
some(x).bind(none)
none().bind(some)
some(x).bind(some) //only this gives a some result
none().bind(none)
```

Run This: [Js Fiddle](#)



Here is a realistic example where we retrieve from the server the information of a client and then try to get the related employee.

We use `bind` `.bind(client => employeeRepository.getById(client.employeeId))` to capture the client value from the maybe and pass it to the `employeeRepository.getById` which returns a maybe, so the final result of the bind is in fact a Maybe.

```
Array.prototype.safeHead = function () {
  return this.length > 0 ?
    some(this[0]) :
    none()
}

var clientRepository = ({
  getById: (id) =>
    [{ id: 1, name: 'Joan', age: 29, employeeId: 1 },
     { id: 2, name: 'Rick', age: 25, employeeId: 2 }]
    .filter(c => c.id == id)
    .safeHead()
});

var employeeRepository = ({
  getById: (id) =>
    [{ id: 1, name: 'jim', age: 29 },
     { id: 2, name: 'jane', age: 25 }]
    .filter(employee => employee.id == id)
    .safeHead()
});

var displayAssignedEmployeeForClientId = (clientId) =>
  clientRepository
    .getById(clientId)
    .bind(client => employeeRepository.getById(client.employeeId))
    .map(e => e.name)
    .cata({
```

```

    some: name => console.log("assigned Employee name: " + name),
    none: _ => console.log("client is not assigned ")
  });

```

```
displayAssignedEmployeeForClientId(1)
```

Run This: [Js Fiddle](#)

if we add promises into the mix then suddenly we cannot use the `Maybe.bind` (please play around with the [following fiddle](#) where the repositories return Promises to see the difficulties that arise when we try to use `Maybe.bind` [jsFiddle](#) or debug the source)

9.6 Either Monad

In the same way we can extend the Either functor to a monad by providing a valid `bind` method:

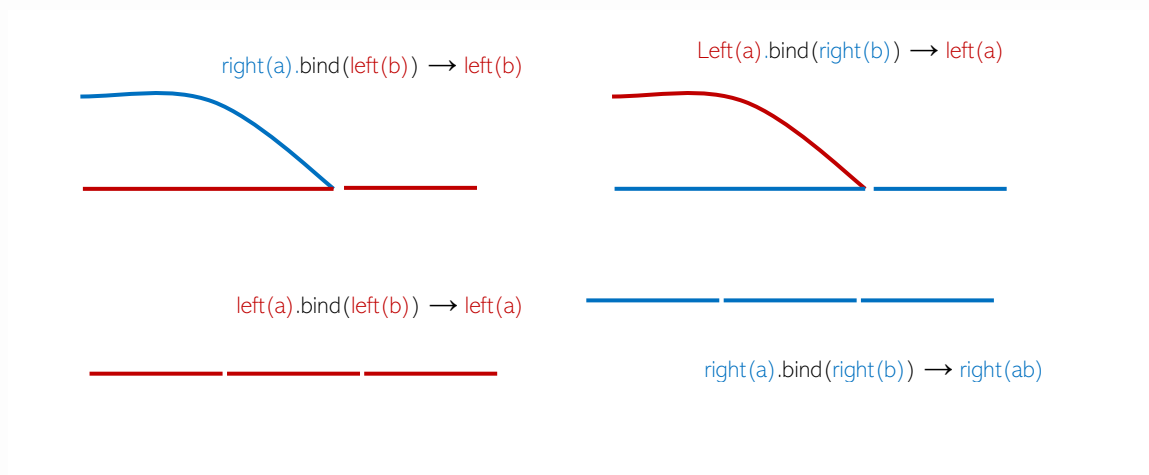
```

const right = (v) => ({
  map: (f) => right(f(v)),
  bind: f => f(v),
  cata: (alg) => alg.right(v),
});

const left = (v) => ({
  map: (_) => left(v),
  bind: f => left(v),
  cata: (alg) => alg.left(v),
});

```

The left always stops the computation and returns itself `left(v)`. This means that for the four possible bind combinations between `right` and `left` again only in the case of `right.bind(right)` we get a `right`. In all other cases the computation stops, and the first `left` is return



9.6.1 Using Either Monad exception handling

In this section I want to build up, on our previous discussion on how to use either to address the issue of error handling. This is suitable for cases that we want to return the first failure signified by a `left(_)`

With a minor refactoring from Maybe to Either we can modify our previous section example to display a more specific error message if we could not find an employee to whom the `client` is assigned. Here we have used the **Maybe**. `toEither`.

```
var clientRepository = ({
  getById: (id) =>
    [{ id: 1, name: 'Joan', age: 29, employeeId: 1 },
     { id: 2, name: 'Rick', age: 25, employeeId: 2 }]
    .filter(c => c.id == id)
    .safeHead()
    .toEither(`there is no client with id ${id}`)
});
```

Using `toEither` in order to convert the possible `None`, to an `Either`. `Left` with a more specific message

```
var employeeRepository = ({
  getById: (id) =>
    [{ id: 1, name: 'jim', age: 29 },
     { id: 2, name: 'jane', age: 25 }]
    .filter(employee => employee.id == id)
    .safeHead()
    .toEither(`there is no employee with id ${id}`)
});
```

```
var displayAssignedEmployeeForClientId = (clientId) =>
  clientRepository
    .getById(clientId)
    .bind(client => employeeRepository.getById(client.employeeId))
```

```

.map(e=>e.name)
.cata({
  right: name => console.log("assigned Employee name: " + name),
  left: error => console.log("error: " +error)
});

```

```
displayAssignedEmployeeForClientId(1)
```

Run This: [Js Fiddle](#)

let's make it more realistic. Let's say that our actual repository can throw an exception when communicating with the server

```
var fetchFailed = () => { throw 'could not connect to the server' }
```

```

var clientRepository = ({
  getById: (id) => {
    return safe(() => fetchFailed())
      .bind(response => response.filter(c => c.id == id))
      .safeHead()
      .toEither(`there is no client with id ${id}`))
  }
});

```

1.Safe fetch returns Either

2.Bind Eithers

Run This: [Js Fiddle](#)

Here we enclose the fetch into a `safe = fn => { }` that return an Either

```
var safe = fn => { try { return right(fn()) } catch (e) { return left(e) } }
```

Even if there are values the id still might not be in the array, which is another Either, and we have used `bind` to combine those two eithers and return a final either as a result. This can be a `Either.right` if there was no connection problem *and* there was a client with this Id. **In any other case the result will be an `Either.left` containing the appropriate error.**

Libraries

Either

Either is also one type that exists in almost every functional library out there. And that is great.

```

monet.js    : Either
sanctuary.js : Either
folktale    : result
pratica     : result

```

9.6.1.1 Validation type

One drawback with either is that always holds by convention of the left the first value on a left no matter how many more lefts or right we bind. **The first left is considered as the termination of the whole execution.** To overcome this and accumulate multiple errors of most libraries like monet.js have the validation type. I am not going to get into more details here.

9.7 State Monad

The idea behind state monad **is to pass around an immutable state in order to preserve the purity of the code.**

Let's say for example we have this simple object-oriented style program:

```
var CyclicAdding = function () {
  this.counter = 0;
  this.add = function () {

    if (this.counter > 10) {
      this.counter = 0;
      this.sum = 0;
    }
    else
      this.sum = this.sum + 4;
    return this.sum;
  }
}
```

This object has a counter that is increased every time we use the `add`. We could also rewrite the `CyclicAdding` function by passing around the counter:

```
var CyclicAddingUsingState = (x) =>
  x.b.counter > 10 ?
    ({a:0, b:{ counter: 0 }}) :
    ({a:x.a + 2, b: { counter: x.b.counter+1 }});

var addition = CyclicAddingUsingState(CyclicAddingUsingState({a:0, b:{ counter: 0 }}});
```

Run This: [Js Fiddle](#)

doesn't look nice, but it works in the same way. It just passes along the counter. Here I used the array as a product structure. This is a function with type $f: (X, S) \rightarrow (Y, S)$ where the S is the state that has an int counter in this example. If we curry this type we will get this usual definition $f: X \rightarrow S \rightarrow (Y, S)$ this is the idea behind the state monad:


```

var State = (expression) => ({
  map: f => State(environment => {
    var evaluation = expression(environment);
    return ({ value: f(evaluation.value), state: evaluation.state });
  }),
  bind: f => State(environment => {
    var newState = f(expression(environment).value).run(environment);
    return newState;
  }),
  run: environment => expression(environment),
});

```

Run This: [Js Fiddle](#)

It takes an `expression` which is a function $S \rightarrow (Y, S)$ that takes a state and returns a product of a new value and a new state (Y, S) in our implementation $(\{ \text{value}, \text{state} \})$. In order to execute the state we use the `run` that just passes the state `environment` into the `expression` and thus evaluates it.

The map is easy to understand

```

map: f => State(environment => {
  var evaluation = expression(environment);
  return ({ value: f(evaluation.value), state: evaluation.state });
})

```

1. We first return a new state `State(environment=>`
2. we then evaluate the previous `expression` using the `environment`
3. then apply the function `f` to the value of the evaluation - `f(evaluation.value)`
4. and just return the same state without altering it `state: evaluation.state`

Let me display the idea behind state monad with another example having to do with monetary transactions. Let's say you have a portfolio of stocks and their prices

```

var portfolio = {};
portfolio["AKL"] = 7;
portfolio["MSR"] = 4;

```

```

prices = {};
prices["AKL"] = 100;
prices["MSR"] = 20;

```

if we want to make a transaction for a fixed stock to start with, we should write

```

var sell = function (numberOfStocks) {

```

```

    var revenue = (numberOfStocks) * prices["AKL"];
    portfolio["AKL"] = portfolio["AKL"] - numberOfStocks;
    return revenue;
}

```

If we want to abstract the portfolio using State monad (where our state here is the actual portfolio)

```

var sell = (stockName, numberOfStocks) => State((portfolio) => {
    var revenue = (numberOfStocks) * prices[stockName];

    portfolio[stockName] = portfolio[stockName] - numberOfStocks;
    var newPortfolio = portfolio;

    return ({
        value: revenue,
        state: newPortfolio
    })
});

```

Run This: [Js Fiddle](#)

If we create a buy state also, we can pass the portfolio between them without explicitly referring to it

```

var buy = (stockName) => valueOfStocks => State((portfolio) => {

    var numberOfStocks = Math.round(valueOfStocks / prices[stockName]);
    var residualMoney = valueOfStocks - numberOfStocks * prices[stockName];

    portfolio[stockName] = portfolio[stockName] + numberOfStocks;
    var newPortfolio = portfolio;

    return ({
        value: residualMoney,
        state: newPortfolio
    })
});

```

Run This: [Js Fiddle](#)

Having those two definitions one could write

```

var moveTransaction = sell("AKL", 3).bind(buy("MSR")).run(portfolio);

```

Run This: [Js Fiddle](#)

does all this trouble worth the hiding of the state passed around, in a pure way? Object oriented programmers probably won't see the benefits of this style of writing especially when we used to have global object properties accessed directly in all our life. This is the reason why you won't see many references of the state monad in any functional javascript forums.

9.7.1 Traversing with State

Here I rewrite the state using `[,]` as a product so the state is $s \rightarrow [a, s]$ in order to be more readable when I define the applicative for state. By now you will know that in applicative the value inside a monad is a function, so an implementation would be this [also feel free to try and derive it by yourself as an exercise]:

```
var State = (expression) => ({
  app: state => {
    return State(s => {
      const [f, r] = expression(s);    // we evaluate and f is a function by definition
                                         // of applicative

      const [a, u] = state.run(r);
      return [f(a), u];                // we apply the f on the value of the applied
                                         // state
    })
  },
  run: enviroment => expression(enviroment),
});
```

```
State.of = (x) => State(s => [x, s])
```

```
var result = State.of(x => x + 1).app(State(s => [5, s + 1])).run(7)//[6,8]
```

```
var result = State.of(x => y => x.concat(y)).app(State(s => [[5], s + 1]))
  .app(State(s => [[5], s + 1]))
  .run(0); //[[5,5],2]
```

```
var result = traverse([2, 3, 5], State.of, x=>State(s => [x, s + 1]));
console.log(result.run(0));//[[2,3,5], 3]
```

Run This: [Js Fiddle](#)

we can also do that obviously with any kind of traversable, if we try the same state function `x=>State(s => [x, s + 1])` we can count the nodes of a tree.

```
var n = (l, v, r) => ({
  traverse: (TyperRep, f) =>
    TyperRep(li => vi => ri => n(li, vi, ri))
    .app(l.traverse(TyperRep, f))
    .app(f(v))
    .app(r.traverse(TyperRep, f))
});
```

```
var lf = v => traverse: (TyperRep, f) => f(v));

var result = n(lf(3), 4, n(lf(3), 4, lf(3)))
.traverse(State.of, x=>State(s => [x, s + 1]))
.run(0); // [ n(lf(3), 4, n(lf(3), 4, lf(3))) , 5]
```

Run This: [Js Fiddle](#)

For another example you we can use state to accumulate validation errors, in this simple scenario we want to validate a name so that the `name`, `surname` values are not empty.

```
var validations = [
  { validation: n => n.name.length > 0, message: "invalid name" },
  { validation: n => n.surname.length > 0, message: "invalid surname" }
]

var collectValidations =
  validations.traverse(
    State.of,
    v => State(s => [[v[0].validation(s) ? "" : v[0].message], s])
  );
```

```
console.log(collectValidations.run({ name: "jim", surname: "" }));
// [["", "invalid surname"], _]
```

Run This: [Js Fiddle](#)

I haven't seen this implementation anywhere else before, but it is something that works and is brief.

Libraries

State

state-monad: [State](#)
 ramda-fantasy :[state](#)
 crocks:[state](#)

9.8 Reader Monad

We have seen in our Reader Functor example that we could provide a model `{firstName:"dimitris", lastName:"papadim"}` on a `Reader` template to generate a Html

```
const titleViewTemplate =
  Reader(({ firstName, lastName }) => `

${firstName} - ${lastName} </div>` )

var titleView = titleViewTemplate
  .run({firstName:"dimitris", lastName:"papadim"});

console.log(titleView); //<div>dimitris - papadim </div>


```

Run This: [Js Fiddle](#)

And we have used `map` to decorate the resulting html

```
var addToSpan = titleViewTemplate.map( x=>`<span>${x}</span>`).run({firstName:"dimitris"
, lastName:"papadim"});
console.log(addToSpan); //<span><div>dimitris - papadim </div></span>
```

Run This: [Js Fiddle](#)

Let's now try instead of mapping to a `x=>`${x}`` let's say we have another template for the footer that we want to combine with the main template of the content

```
const addfooterTemplate = content => Reader(({ footer }) =>
  `${content} \n<div>c reated:${footer.year} </div>`)
```

Run This: [Js Fiddle](#)

If we run the following we get back a reader monad

```
var html = titleViewTemplate
  .map(addfooterTemplate)
  .run({
    content: { firstName: "dimitris", lastName: "papadim" },
    footer: { year: "2019" }
  })
//reader monad not string
```

Run This: [Js Fiddle](#)

and so, we must use `reader.run` again in order finally get the overall result

instead of doing that we can implement the usual `bind` operation in order to deal with the situations where we end up with a Reader-in-a-Reader. As usual this will allow us to use `bind` instead of `map` when we know that our mapping function returns a new Reader.

```
var Reader = (expr) => ({
  map: f => Reader(env => f(Reader(expr).run(env))),
  run: env => expr(env),
  bind: f => Reader(env => f(expr(env)).run(env)),
});
```

Run This: Js Fiddle

Now we can write

```
const titleViewTemplate = Reader(({content}) =>
  `<div>${content.firstName} - ${content.lastName} </div>`)

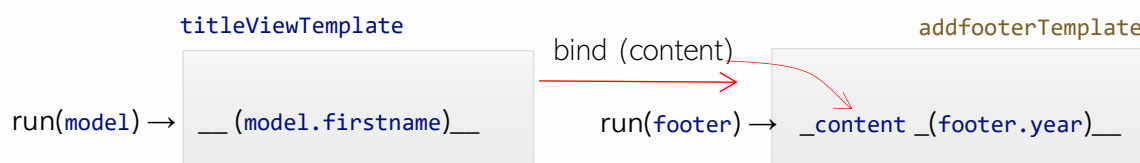
const addfooterTemplate = content => Reader(({footer}) =>
  `${content} \n<div>created:${footer.year} </div>`)

var htmlModel = {
  content: { firstName: "dimitris", lastName: "papadim" },
  footer: { year: "2019" }
};

var html = titleViewTemplate.bind(addfooterTemplate).run(htmlModel)
//<div>dimitris - papadim </div>
//<div>created:2019 </div>
```

Run This: Js Fiddle

Take a look at a graphical representation of what is happening when we use bind



9.9 IO monad

The IO monad that I present here for completeness it's a simpler form of the reader and the state Monads and the implementation should be obvious by now.

It follows the usual steps:

4. evaluating the inner object: `fn()`
5. applying the function `f : f(fn()).run()`
6. create a new `IO` that contains the result: `IO(() => f(fn()).run())`,

```
var IO = fn => ({
  map: f => IO(() => f(fn())),
  bind: f => IO(() => f(fn()).run()),
  run: () => fn(),
  cata: alg => alg(fn())
});
```

Run This: [Js Fiddle](#)

The usual metaphor for the IO monad is this of “**a recipe**”. An `IO(()=>x)` **is a recipe to get a x**. In this way we are not concerned about x in itself, but about the recipe of the x which is the IO. The truth is that all the state monads (reader, state, IO) usually are not taken seriously in the JavaScript world yet, even if they are gaining ground. A realistic example of usage taken from the Frontend world arises when you try to access UI controls. For example:

```
<div id="container"> </div>
```

```
var capitalizedText = $("#container").text().toUpperCase();
$("#container").text(capitalizedText)
```

Instead of going this way the functional recommendation would be to use IO to isolate the side-effects

```
var getContainerTextIO = IO(() => $("#container").text());
var setContainerTextIO = x => IO(() => $("#container").text(x));
```

```
setContainerTextIO("text").run();
```

```
getContainerTextIO
  .map(x => x.toUpperCase())
  .bind(setContainerTextIO)
  .run();
```

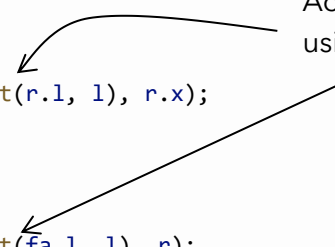
Run This: [Js Fiddle](#)

I hope you can recognize how this part can be easily tested, since it is easier to reason about the program (sequence of commands) `get->map->set`. It would be very easy to mock the `getContainerTextIO` and `setContainerTextIO`.

9.10 Optional Writer Monad

I didn't present Writer up until now because it is not that important concept. Writer takes a tuple `(l, x)` and process the second item `x` normally but accumulating "values" on the first argument `l`. I have used `l` to signify that usually this argument can be thought as a **log** on which we append "stuff" on each operation. The following implementation `WriterM = m =>` takes a monoid `m` and returns a specialized writer that uses this specific monoid to accumulate "stuff"

```
var WriterM = m => (l, x) => ({
  l: l,
  x: x,
  map: f => WriterM(m)(l, f(x)),
  chain: f => {
    var r = f(x);
    return WriterM(m)(m.concat(r.l, l), r.x);
  },
  app: fa => {
    var r = x(fa.x);
    return WriterM(m)(m.concat(fa.l, l), r);
  },
});
```



Accumulating logs using the monoid m

In the first example we accumulate logs as string that describe what is happening on each step :

```
var Writer = WriterM({ concat: (x, y) => y + " " + x });

Writer.of = x => Writer(x, x);

var bindResult = Writer(`start with 5`, 5)
  .chain(x => Writer(`-and then mutluply ${x} by 6`, x * 6))
//{l: "start with 5 -and then mutluply 5 by 6", x: 30,}
```

Run This: [Js Fiddle](#)

Or we can capture the strings as an array and maybe process them latter

```
var Writer = WriterM({ concat: (x, y) => y.concat(x) });

Writer.of = x => Writer(x, x);
```



```
var bindResult = Writer(['start with 5'], 5)
    .chain(x => Writer(['-and then mutluply ${x} by 6'], x * 6))
//{1: "start with 5 -and then mutluply 5 by 6", x: 30,}
console.log(bindResult);
```

Run This: [Js Fiddle](#)

You can come up with any other ideas? Just fork the jsfiddle and play around with the concept of the writer for a bit.

9.10.1 Optional Traversing with Writer

Since Writer can accumulate as a log the first argument under a monoid operation, we might use it together with the concept of traversal. Here we will define the applicative operation for the writer in order to be able to use it on a traversal.

```
var WriterM = m => (l, x) => ({
  app: fa => {
    var r = x(fa.x);
    return WriterM(m)(m.concat(fa.l, l), r);
  },
});
```

if we use the concat as a monoid on strings and traverse the array, we get a string of the array.

```
var Writer = WriterM({ concat: (x, y) => x + " " + y });

Writer.of = x => Writer(x, x);

var result = traverse([2, 3, 5], x => Writer("", x), x => Writer(x, x));
console.log(result);//{1: " 5  3  2 ", x: [2, 3, 5]}
```

Run This: [Js Fiddle](#)

similarly if we try the + monoid and log 1 for each element we can actually count the amount of elements which we traversed.

```
var Writer = WriterM({ concat: (x, y) => x + y });

var result = traverse([2, 3, 5], x => Writer(0, x), x => Writer(1, x));
console.log(result);//{1: 3, x: [2, 3, 5]}
```

Run This: [Js Fiddle](#)

9.11 Deriving the Continuation Monad

Starting from this section we are going to see the Continuation monad and explore some ideas on how to use monads in asynchronous situations. The continuation monad in a way is equivalent to Js Promises but also provide a way to compose them using a bind method. Here we will derive the Continuation monad starting from scratch in order to uncover the specific underlying concepts.

Let's start from a simple function:

```
const square = x => x * x;
console.log(square(4));    // prints 16
```

Run This: [Js Fiddle](#)

1) Use callback and remove return type

We can pass a callback as an argument and remove the return type. This the famous Continuation-passing style

```
const square= (x, callback)=>callback(x*x);
square(4,console.log);    // prints 16
```

Run This: [Js Fiddle](#)

and you can compose functions with callbacks like this:

```
const square = (x,callback)=>callback(x*x);
square(2,
  x=>square(x,console.log));    // prints 16
```

Run This: [Js Fiddle](#)

which of course leads to callback hell which is special case of the arrow anti-pattern...

2) Curry the callback

What if we curry the action and return it as a result ?! we could do this :

```
const square = (x)=>callback=>callback(x*x);
```

Run This: [Js Fiddle](#)

if you look at this for a long time and rename callback to resolve you will see that this is actually resembles a **promise**. We can use it like this:

```
square(4)(y=>console.log(y));    // prints 4
```

we could also make it into a Promise if we enclose the callback into an object that exposes a **then** function :

```
const square = (x)=>callback=>callback(x*x); // transform it into
const square = (x)=>({then:resolve=>resolve(x*x)});
square(2).then(r=>console.log(r))
```

Run This: [Js Fiddle](#)

it looks a bit better, but we can do a few more refactoring moves. This return type of $(x)=>callback=>callback(_)$ is actually $(\text{int} \rightarrow \perp) \rightarrow \perp$ where \perp is the initial object /empty type /false. This says, if you give me a function that takes an int i and returns nothing ($\text{int} \rightarrow \perp$), i will do something with it and then return nothing \perp .

3) Generalize the returned continuation

what if we wanted to generalize this and instead of \perp return an arbitrary type T. Then we should have:

$$(\text{int} \rightarrow T) \rightarrow T:$$

This says, if you give me a function that takes an int i and gives a T i will do something with it and return T which is an extension of

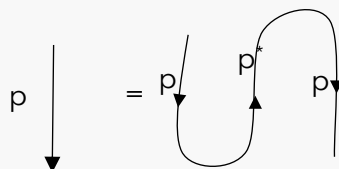
$$(\text{int} \rightarrow \perp) \rightarrow \perp$$

Because we use untyped JavaScript this is not of concern but with a strong type js language extension like Typescript, we would need to actively pay attention to this point.

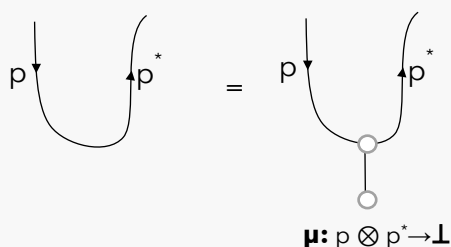
This **function type $(U \rightarrow T) \rightarrow T$** is kind of famous, and we can call it **Continuation**.

In logic, **double negation** is the theorem that states that "If a statement is true, then it is not the case that the statement is not true." That is: $\neg \neg P \rightarrow P$.

Double negation concept has a profound relation with the concept of continuation through the **Curry-Howard correspondence**. We saw that our initial object \perp represents false. The $P \rightarrow \perp$ becomes $\neg P$. This means we have a function that return the initial object is the equivalent with a proposition being false. In a way we the first say if you could give me a P then I could give you the initial element which is not possible, so you cannot have P in Logic this is the equivalent of P being false. If we apply one more time the negation we will get $((P \rightarrow \perp) \rightarrow \perp) \equiv P$



which is actually what continuation passing style says: if you give me a continuation that gets a P it's the same as if you gave me P .



Another way to interpret the U turn on the diagram is by representing the missing elements visually.

$p \otimes p^* \rightarrow \perp$ here means that p and p^* should give us false so p should be the logical opposite of p^* .

If we want to go a step further, then Monads come in. We can form a Monad from this Continuation type $(\mathbf{U} \rightarrow \mathbf{T}) \rightarrow \mathbf{T}$.

4) Extract the continuation structure

Now we only need to extract the Continuation mechanism from the square function so we could reuse it. So, we can have the first part of the Monad, the Return $c \rightarrow T(c)$, which is the wrapping of the value with a continuation, like this:

```
const toCont = v => callback => callback(v);
```

in order to finally get the Continuation monad, we must add a Bind Method: $T(A) \rightarrow (A \rightarrow T(B)) \rightarrow T(B)$ that composes continuations. if we replace T that is the Monad with $(A \rightarrow T) \rightarrow T$ in the definition of bind,

```
T (A) → (A → T (B)) → T (B)
```

we get

```
((A → T) → T) → (A → ((B → T) → T)) → ((B → T) → T)
```

That is : from a continuation of A $[(A \rightarrow T) \rightarrow T]$ and a function that takes an A and gives a continuation for B $[A \rightarrow ((B \rightarrow T) \rightarrow T)]$ return a continuation of B $[(B \rightarrow T) \rightarrow T]$. From which follows the implementation:

```
var bind = (cont1, func) => (callback => cont1(contResult => func(contResult))(callback));
```

ok we have a `cont1` of type $(A \rightarrow T) \rightarrow T$ and a function `func` of type $A \rightarrow ((B \rightarrow T) \rightarrow T)$ we want to return $(B \rightarrow T) \rightarrow T$. We first write `callback=>` which is $(B \rightarrow T)$, the rest of this should return a value T . How are we going to get this value? well we can take it from `cont1 = (A → T) → T` just pass a function inside that takes an argument of type A named `contResult`. Up until now we have

```
bind = (cont1, func) => (callback => cont1(contResult =>...
```

then we can write something like this

```
var contComposition = bind(toCont(x => x * x), y => toCont(y(4)));
```

Run This: [Js Fiddle](#)

9.12 The Continuation monad

If we wanted to rewrite the above derivation of continuation monad in our usual object literal notation the result would be the following:

```
var Continuation = (cont) => ({
  bind: (f) => {
    return Continuation(resolve => {
      cont(res => f(res).run(res2 => resolve(res2)));
    })
  },
  map: (f) => Continuation(resolve => cont(res => resolve(f(res)))),
  run: (alg) => cont(alg)
});

Continuation.of=x=>Continuation(resolve => resolve(x));
```

This is almost equivalent to a simple promise without the (rejection path and explicit state management) and we can use this in the same way we use a Promise.

```
Continuation.of(5).run(console.log)//5
```

```
Continuation(resolve => resolve(5)).map(x => x + 2).run(console.log)//7
```

```
Continuation(resolve => resolve(5))
```

```
.bind(x => Continuation(resolve => resolve(x + 2)))
.run(console.log)//7

//this is equivalent
Continuation.of(5).bind(x=>Continuation.of(x+2)).run(console.log)
```

Run This: [Js Fiddle](#)

Because promises are natively supported in JavaScript there is no reason to use `Continuation`. But we can instead provide some extension on Promises that will provide a monadic way of usage.

9.13 Extending Promises to Monads

In the [ES6 javascript specification](#) the promises as build in object support a changeable then operation. This means that Here's the magic: the `then()` function returns a **new promise**, different from the original:

```
const promise = doSomething();
const promise2 = promise.then(successCallback, failureCallback);
```

this behavior try's to create a fluent interface to allow to sequence promises in an elegant way. Unfortunately, this implementation creates a lot of confusion when you add the possibility of failure. This has led to allot of criticism and the attempt to come up with alternative solutions for asynchronicity like [Fluture.js](#), that are aiming to create an "asynchronous" monad. You can find more to my upcoming book "[asynchronous functional javascript](#)".

Well the actual implementation of a Promise does not provide a monadic bind support but it's quite easy to extend it and promote it to a Monad. We must implement

bind: $\text{promise}(A) \rightarrow (A \rightarrow \text{promise}(B)) \rightarrow \text{promise}(B)$

if you give me a promise that will give A [$\text{Promise}(A)$] and then you give me a function that when it takes A gives a Promise of B [$A \rightarrow \text{Promise}(B)$] then we must return a Promise of B.

```
Promise.prototype.bind = function(func) {
  return new Promise(function(resolve) {
    this.then(result => func(result).then(x => resolve(x)))
  });
}
```

If you look at this implementation is the exact same as the bind of the continuation monad of the previous section. If we extend it to include the possibility of the failure

```

Promise.prototype.bind = function (func) {
  var initialPromise = this;
  return new Promise(function (resolve, reject) {
    initialPromise.then(result => func(result)).then(x => resolve(x))
                      .catch(reject);
  });
};

```

```

Promise.resolve("resolve promise 4")
  .bind(x=>Promise.reject(x.toUpperCase()))
  .cata({
    ok: x => console.log(`success: ${x}`),
    error: x => console.log(`error: ${x}`) //error: RESOLVE PROMISE 4
  });

```

```

Promise.resolve("resolve promise 4")
  .bind(x=>Promise.resolve(x.toUpperCase()))
  .cata({
    ok: x => console.log(`success: ${x}`), //success: RESOLVE PROMISE 4
    error: x => console.log(`error: ${x}`)
  });

```

Run This: [Js Fiddle](#)

9.14 Async Error handling with either

We have a promise that have two possible execution paths the successful and the rejected. It **would make sense if we could get an Either structure out of a promise** and allow to use the Either pattern for the rest of our computation. We want something that looks like this:

```

fetch("https://api.github.com/users")
  .map(response => response.json())
  .map(users=>users.map(u=>u.login))
  .toEither()
  .cata({
    ok: v => console.log(v),
    error: v => console.log("left" + v)
  });

```

Run This: [Js Fiddle](#)

Here we use the [fetch api](#) in order to retrieve the user list from github. Fetch return a promise.

The suggested way to use fetch is from [mozilla](#) something like this:

```

fetch("https://api.github.com/users")
  .then(response => response.json())
  .then(response => console.log('Success:', JSON.stringify(response)))
  .catch(error => console.error('Error:', error));

```

This looks nice but we want to use the consistency and using either seems appropriate. The thing with promise is that we cannot know in advance the outcome of the promise. What if we could delay our decision while providing an interface Either- like and as soon as we know the result complete the appropriate actions.

We have seen that Coyoneda allows us to delay our computation by accumulating the operations of map and allow us to execute the computation when we have more information. In this way I would initially create a new type (or extend Either if available) that will provide a Coyoneda implementation for the Promise

```

var EitherCoyo = function (actions, g) {
  this.g = g;

  this.map = (f) => new EitherCoyo(actions, x => f(this.g(x)));
  this.cata = alg => actions(x => alg.ok(this.lower(x)), alg.error);
  this.lower = (v) => this.g(v);
}

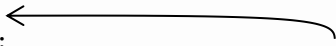
Promise.prototype.toEither = function () {
  var initialPromise = this;
  var either = new EitherCoyo(function (resolve, reject) {
    initialPromise.then(resolve).catch(reject)
  }, x => x);
  return either;
}

```

Use Coyoneda to delay the map



We initialize with the identity $x \Rightarrow x$



Run This: [Js Fiddle](#)

The map of Either only maps the resolved successful part and not the faulty part. That's why in cata we only lower (execute the computations of Coyoneda) and apply the mapping function for the ok part. For the error we just return the error.

9.14.1 Implementing Bind

we also want to be able to chain computations with other Either to make it a monad. We can implement that by knowing that only when the successful path is completed, we combine the right sides of the two Eithers.


```

this.bind = f => {
  var initialPromise = this;
  return new EitherCoyoAsync(function(resolve, reject) {
    actions (x => f(initialPromise.lower(x)). cata({ok:resolve,error:reject}),
      x => reject(x))
    }, x => x);
}

```

The tricky part with implementing bind is that we don't know what the second either would be `f(initialPromise.lower(x))` so we use `cata` to make the correct delegation of the continuation of the successful path. Now we can use promises in a functional syntactic style, without worrying about the error handling

```

var getUser = () => fetch("https://api.github.com/users");

var getUserFollowers = (name) => fetch("https://api.github.com/users/" + name + "/followers");

var toJson = response => response.json()

getUser()
  .map(toJson)
  .toEither()
  .map(users => users[0].login) //or use safeHead
  .bind((v) => getUserFollowers(v).map(toJson).toEither())
  .map(followers => followers.map(follower => follower.login)) //compose functor map can
  .cata({ // be used here
    ok: v => console.log(v),
    error: v => console.log("error: " + v)
  });

```

Run This: [Js Fiddle](#)

where we combine promises in a functional syntax style, using either, without going to the nested try/catch thing. If any of those fetches fail, then the final result would contain the error.

Comonads

10 Comonads

The categorical definition is the dual of the Monad. Comonads are a comonoid in the category of endofunctors. The operations of a Comonad are the same as the ones of monad but with the arrows reversed.

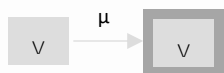
```

 $\eta: W(v) \rightarrow v$            //Extract
 $\mu: W(v) \rightarrow W(W(v))$     //Co-Join or Duplicate
  
```

1) In this first operation $W(v) \rightarrow v$ we can start from a container object $W(v)$ and extract the contained object. This is the inverse of the monadic return where we enclose a value in a monad.



2) The second one: $W(v) \rightarrow W(W(v))$ means that we can expand and duplicate the container.



Intuitively duplicate mean that if we have a comonad, we can apply the duplicate and **add more layers of containment to our value.**

10.1 The Identity Comonad

In the same way with the monad we can have the simplest structure in JavaScript that displays the properties of a comonad. We will call this minimal structure Identity comonad.

```

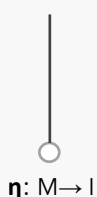
const Id = (v) => ({
  extract: () => v,
  duplicate: ( ) => Id (Id (v)),
  map: (mapping) => Id (mapping(v)),
  extend: (f) => Id (v).duplicate().map(f),
})
  
```

Run This: [Js Fiddle](#)

The duplicate and extend are interdependent (like join and bind are dependent for monads) in the sense that if you define one of the two the other is defined in order to be consistent. Here we defined extend relying in duplicate and map.

Category & Type Theory

In type theory if we wanted to represent the types of the two operations we would say that a monoid in \mathbf{C} is an object \mathbf{M} equipped with a co-multiplication $\mu: M \rightarrow M \otimes M$ called split and a counit $\eta: M \rightarrow I$ called destroy. The string diagrams of the operations make the



Co-Monoids

naming obvious

Comonoids are not that famous in programming because they are somewhat trivial in most type systems. But there are some applications that are not trivial and Comonads is one of them.

10.2 Optional Co-Monad laws for Identity Co-Monad

Like the monad, comonad has its own laws

1. Left identity, extend and extract cancel out

```
//extend extract = id
var id = Id (2);
var extendextract= id.extend(c=>c.extract()) ;
extendextract.extract() ≡ id
```

Run This: [Js Fiddle](#)

2. Right identity,

```
var f = c => c.extract() *3 ;
var extractAfterExtend = Id (2).extend(f) ;
extractextend.extract() ≡ 2 *3
```

Run This: [Js Fiddle](#)

3. Associativity,

```
//extend f . extend g = extend (f . extend g)
var f = c =>c.extract()*3;
var g = c =>c.extract()*2;

var extendf_After_Extendg= Id (2).extend(g).extend(f) ;
```

```
var extend_fAfter_Extendg = Id (2).extend(x=>f(x.extend(g))) ;
extendf_After_Extendg ≡ extend_fAfter_Extendg
```

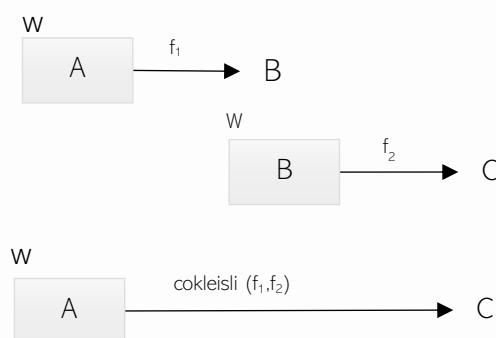
Run This: [Js Fiddle](#)

I won't expand on the comonad laws since we already have seen the Functor and monad laws. Comonad laws should be easy to grasp given the examples above.

10.3 Optional CoKleisli Composition

In the same way we did for monads we get the definition of co-kleisli composition for comonads. In co-kleisli we want to compose two functions that get a comonads as arguments. But if we have two functions that take a comonad, we cannot directly compose them.

Because the first function returns a type B but we need a type W in order to apply f_2



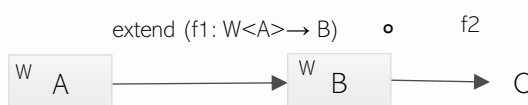
if we have again the Identity comonad for simplicity let's see how would implement the co-Kleisli composition

if we have two functions that take as an argument a comonad:

```
var f1 = c=>c.extract()*2;
var f2 = c=>c.extract()*3;
```

in order to use the f2 we want a comonad<int> but instead we have an **int**. The way to get the $W(B)$ is use the extend operation on comonad<int>.extend(f1) this will give us a comonad<int> by definition and then we can compose directly after with f2 and finally this will get us an Int.

```
cokleisli = (f1,f2)=>c=> f2(c.extend(f1));
```



one can verify that this definition of co-kleisli composition is associative. That is if we have three co-monadic functions then the following holds:

```
var f = c => c.extract() * 2;
var g = c => c.extract() * 3;
var h = c => c.extract() * 5;

console.log(Id (3).extend(cokleisli(f, cokleisli(g, h))).extract());//90
console.log(Id (3).extend(cokleisli(cokleisli(f, g), h)).extract());//90
```

Run This: [Js Fiddle](#)

10.3.1 **Optional** Co-Monad laws following Co-Kleisli formulation

Using this definition, the three monadic laws can be expressed like this:

Left identity : $\text{compose}(\text{id}, f) \equiv f$

Right identity: $\text{compose}(f, \text{id}) \equiv f$

Associativity: $\text{compose}(f, \text{compose}(g, h)) \equiv \text{compose}(\text{compose}(f, g), h)$

Where $\text{id} = c \Rightarrow c.\text{extract}()$;

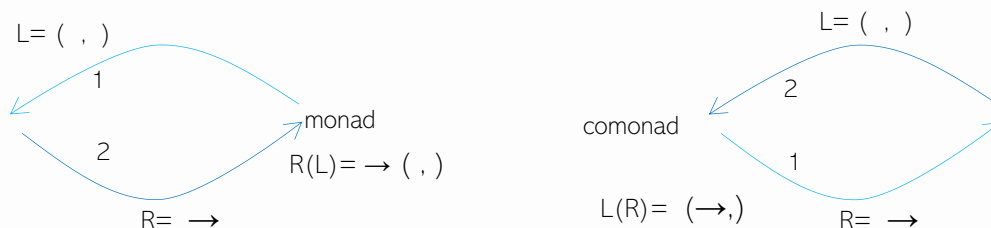
Run This: [Js Fiddle](#)

it's now easy to see that monad composition is an associative operator with left and right identities.

10.4 Store Comonad

Here we will follow a gradual definition of some of the comonads. The store comonad would probably be the logical step after the identity comonad. This comonad arises from the product $R = (,)$ and $L = \rightarrow$ Functor adjunction (those are the same that give the currying transformation. I will not discuss adjunctions in this book because it's an advance topic but will be in the extended version of this book coming soon on Leanpub)

State monad came from $R(L())$ first applying a product and then the \rightarrow that is $_ \rightarrow (_, _)$ and **StoreComonad** is the dual $L(R())$ structure $(_ \rightarrow _, _)$.



```
var StoreComonad = (lookup, index) => ({
  map: f => StoreComonad(x => f(lookup(x)), index),
  extract: () => lookup(index),
  duplicate: () => StoreComonad(x => StoreComonad(lookup, x), index),
  extend: (f) => StoreComonad(lookup, index).duplicate().map(f),
})
```

Run This: [Js Fiddle](#)

The store comonad takes as constructor arguments a lookup this is any function $\text{lookup}: S \rightarrow A$ and an index of type S - $\text{index}: S$. This is directly depicted in the $(_ \rightarrow _, _)$ structure of the constructor of store comonad: $(S \rightarrow A, A)$. The implementation of the extract is straightforward we just apply the index to the lookup to get an $a: A$. the duplicate method puts a StoreComonad at the place of the lookup. $(S \rightarrow (S \rightarrow A, A), A)$.

the extend method is defined again based on duplicate and map.

```
extend: (f) => StoreComonad(lookup, index).duplicate().map(f)
```

10.5 Lazy

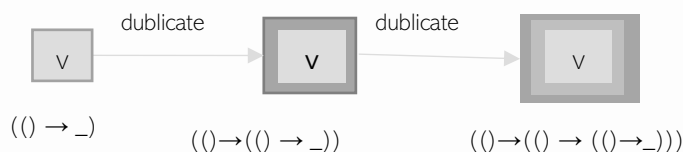
A straightforward variation of the store Comonad is a comonadic implementation of Lazy where we replace the lookup with a factory function - $() \Rightarrow _$ and remove the index. This is the same as setting the index as the unit object $()$. One may also regard the unit type as the type of 0-tuples

```
var LazyComonad = (lookup) => ({
  map: f => LazyComonad(() => f(lookup())),
  extract: () => lookup(),
  duplicate: () => LazyComonad(() => LazyComonad(lookup)),
  extend: (f) => LazyComonad(lookup).duplicate().map(f),
})
```

Run This: [Js Fiddle](#)

So now we basically reduced the store comonad to $(() \rightarrow _)$. Duplicate becomes $((()) \rightarrow ((() \rightarrow _))$. This adds more "laziness" with each duplicate application. This reveals to us that the

Lazy has a natural inclination towards comonadic nature. One can verify that lazy comonad does not violate comonadic laws



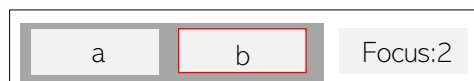
For example, if we use `extend`, we are deferring evaluation of the `lazyComonad`, until the outmost `LazyComonad` uses `extract` to force the evaluation of the whole structure.

```
var lazy = LazyComonad(() => 5);
var lazyMultiplyFive = lazy.extend(s=>s.extract()*5)//no evaluation
console.log("lazy was not evaluated yet")
var result = lazyMultiplyFive.extract();//now both lazyComonads forced in evaluation here
```

Run This: [Js Fiddle](#)

10.6 Pair Comonad

This is another simple version of the **store comonad** we have a `pair(a,b)` and an `focus` (that represents the index) that might signify which one of the two objects is the focused. This is not a very useful structure, but we will use it to gain an understanding of another characteristic of the comonadic nature: **comonads as contextual computations**.



A simple implementation in literal functional form would be the following:

```
const pairComonad = (a, b, focus) => ({
  a: a,
  b: b,
  focus: focus,

  extract: () => (focus === 1 ? a : b),
  duplicate: () => pairComonad(pairComonad(a, b, 1), pairComonad(a, b, 2), focus),

  map: (f) => pairComonad(f(a), f(b), focus),
  extend: (f) => pairComonad(a, b, focus).duplicate().map(f),
})
```

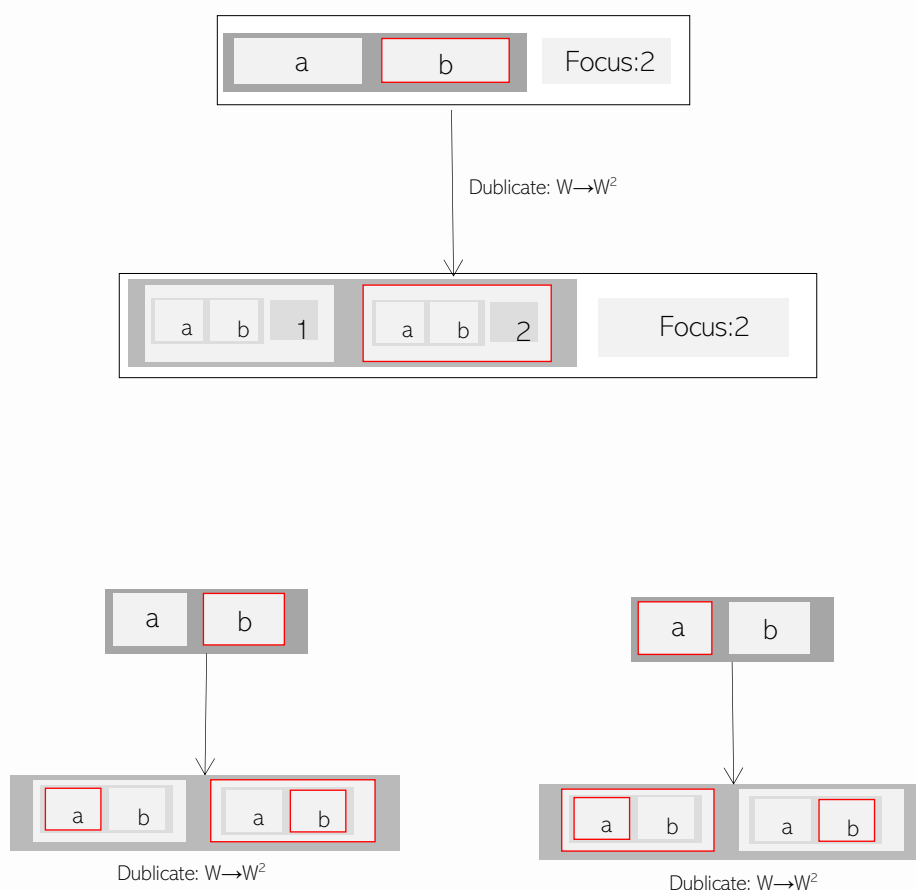
Run This: [Js Fiddle](#)

One can play around with the implementation and verify that this implementation follows the comonadic laws.

So, in a way this convention that focus means the element of the pair, **is an equivalent of the lookup of the Store comonad** where the lookup $S \rightarrow A$ is and the focus represent the index to an element of the pair.

The **extract** has the same interpretation: we can get an element of the pair from the pair and the focus $(A, (A, A) \rightarrow A) \rightarrow A$.

With the duplicate we replace the pair with a pair of comonads that have different focused element. In this way after the duplicate we create a two possible comonads that are part of a comonad. This is like having all the possible (in this case just 2) **states or futures of our initial comonad**.

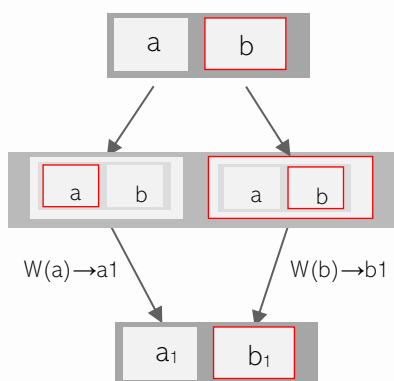


The extend in this structure becomes more interesting, because you can have a whole comonad and you can use it to return a single value. This is often seen as a contextual computation that allows us to access the context and the position information - index and return a single value.

For example, we can have a function f that averages the pair by the focus value depending on the position.

```
var f = c => (c.a + c.b) / c.focus; //we can use the information of c
pairComonad(3, 2, 2).extend(f).extract();//2.5
```

Run this: [Js Fiddle](#)



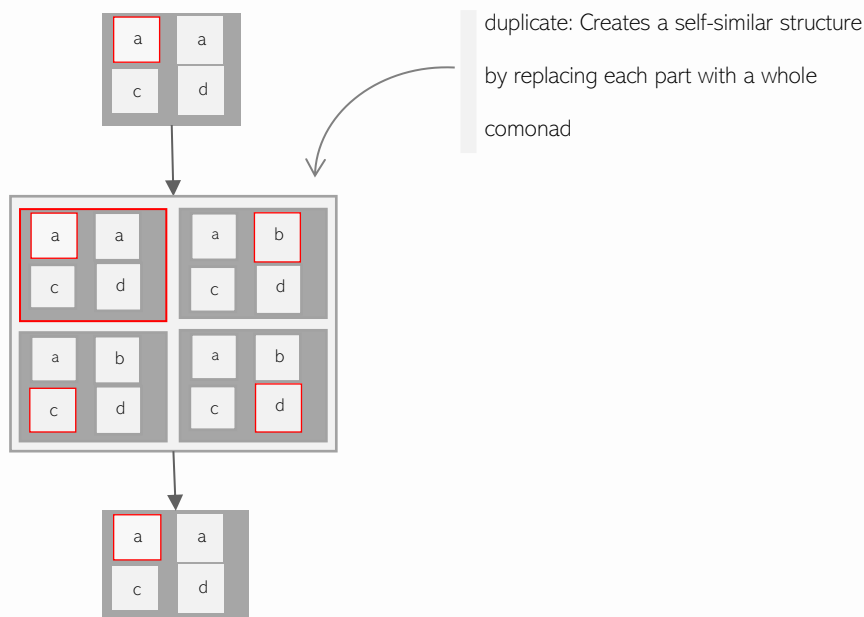
duplicate: Creates a self-similar structure by replacing each part with a whole comonad

map f : f goes under W and is applied to each comonad reducing them. Now the structure is again simplified

Obviously, the geometric representation just for visualization purposes is irrelevant from the actual algebra of this transformation. All the instances have the same form $([], \text{focus})$ to $([([], \text{focus})], \text{focus})$

10.7 Spatial Comonad and the *Game of Life*

If we can create a comonad with a pair of elements why not add more elements. We can keep adding elements to our list of arguments. Let's make them four. The next step would be to make them arbitrary large.



We are going to define for this implementation a grid of Booleans

```
var grid = [
  [false, false, false, false, false, false, false, false, false],
  [false, false, false, false, false, false, false, false, false],
  [false, false, false, false, true, false, false, false, false]
];
```

And we will define a grid comonad that will take a grid as first argument and a focused cell of the grid as second argument called `index`.

```
var gridComonad = (g, index) => ({
  g: g,
  index: index,
  map: f =>
    gridComonad(g.map(x => x.map(y => f(y))),
    { e: f(index.e), x: index.x, y: index.y }),

  duplicate: () => gridComonad(
    g.map((x, i) => x.map((y, j) => gridComonad(g, { e: y, x: i, y: j }))),
    { e: gridComonad(g, index), x: index.x, y: index.y }
```

```

    ),

    extend: f => gridComonad(g, index).map(f),
    extract: () => index,
  });

```

Run this: [Js Fiddle](#)

The duplicate method just places in each cell a whole copy of the grid but with a different index { `e: y, x: i, y: j` } we implemented that using the `g.map((x, i)` that also gives us the index `i` of the element on the array.

```

duplicate: () => gridComonad(
  g.map((x, i) => x.map((y, j) => gridComonad(g, { e: y, x: i, y: j }))),
  { e: gridComonad(g, index), x: index.x, y: index.y }
),

```

we are going now to use this comonad in order **to count how many cells have the value true around each cell** and return a new grid with integers. Each integer is the number of neighbouring squares with true as a value.

The idea here is to pass to our comonad a function that will count all the cells with a certain property in a region of any cell. First, we define a mask of offsets that we will apply to each square.

```

var mask = [{ x: -1, y: -1 }, { x: -1, y: 0 }, { x: -1, y: 1 }, { x: 0, y: 1 },
  { x: 0, y: -1 }, { x: 1, y: -1 }, { x: 1, y: 0 }, { x: 1, y: 1 }
];

```

So for any position (x,y) the squares around would be

```

var filterMaskAtPosition = position =>
  mask
    .map(offset => ({ x: position.x + offset.x, y: position.y + offset.y }));

```

In order to find which coordinates from the grid are permissible within bounds we will filter with the following predicate

```

var bounds = xs => ys => e => e.x < xs && e.y < ys && e.y > 0 && e.x > 0;

var box = bounds(3)(9);

```

now we have in

```

var filterMaskAtPosition = position =>
  mask

```

```
.map(offset => ({ x: position.x + offset.x, y: position.y + offset.y })))
.filter(box);
```

if we pass this function into the extend function of the comonad

```
var f = x =>
  filterMaskAtPosition({ x: x.index.x, y: x.index.y })
    .map(i => x.getElementAt(i.x, i.y))
```

We will get a list of the surrounding values around x cell. The next step is to fold the array to a number using reduce.

```
var countIfTrue = i => (i ? 1 : 0);

var f = x =>
  filterMaskAtPosition({ x: x.index.x, y: x.index.y })
    .map(i => x.getElementAt(i.x, i.y))
    .reduce((accumulation, element) => accumulation + countIfTrue (element), 0);
```

Run this: [Js Fiddle](#)

And we can finally see it running:

```
gridComonad(gr1, { e: true, x: 1, y: 1 })
  .duplicate()
  .map(f)
  .cata({
    e: x => console.log(`${x} `),
    i: x => console.log(x)
  });
```

Run this: [Js Fiddle](#)

Feel free to play around with the [jsFiddle](#) and debug the source code of this section in order to get a better grasp of how this works. You might want to try to use different monoids to fold the region around a position, for example you can fold it with the Any monoid and return an updated grid where each cell has a true value only if there is at least 2 neighbouring cells with true value. You might already recognize that this is the conways game of life. The rules of the game of life are those

The universe of the *Game of Life* is an infinite, two-dimensional orthogonal grid of square *cells*, each of which is in one of two possible states, *alive* or *dead*, (or *populated* and *unpopulated*, respectively). Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step-in time, the following transitions occur:

1. Any live cell with fewer than two live neighbours dies, because of isolation.

2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, by overpopulation.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

As an exercise, based on the existing formulation you can try to create a loop that starts from the previous grid and applies a function that implements those rules.

10.8 Optional Stream Comonad

Stream comonad is another variation of our store Comonad.



we want a data structure that has this form:

```
var stream = Stream([1, 3, 2, 3, 4, 5, 6], 5, [7, 9, 4, 5, 6, 7, 8, 9]);
```

There is a left array, a focus element and a right array. Let's first add a map function. That's easy. Is a recursive product algebraic data structure of the three elements. we have to first lift all three elements and return the product which in this case is a new stream of the lifted subparts.

```
var Stream = (left, focus, right) => ({
  left: left, focus: focus, right: right,
});

var Stream = (left, focus, right) => ({
  left: left, focus: focus, right: right,
  map: (f) => Stream(left.map(f), f(focus), right.map(f)),
});
```

Run this: [Js Fiddle](#)

We are going to define an extension method on the Array in order to be able to convert an array to a stream

```
Array.prototype.toStream = function () { return [...this].map((e, i) =>
Stream([...this].slice(0, i), e, [...this].slice(i + 1, this.length))); }
```

Run this: [Js Fiddle](#)

For example `[1, 2, 3, 4, 5, 6].toStream()` will generate

```
[Stream([],1,[2,3,4,5,6]),Stream([1],2,[3,4,5,6]),Stream([1,2],3,[4,5,6]),
Stream([1,2,3],4,[5,6]),Stream([1,2,3,4],5,[6]),Stream([1,2,3,4,5],6,[])]
```

We can now create the `duplicate` and `extract`

```
var Stream = (left, focus, right) => ({
  left: left, focus: focus, right: right,
  duplicate: () => Stream(left.toStream().reverse(), Stream(left, focus, right), right.toStream()),
  extract: () => focus,
  extend: (f) => Stream(left, focus, right).duplicate().map(f),
  map: (f) => Stream(left.map(f), f(focus), right.map(f)),
});
```

One can verify that the comonadic laws hold for our implementation.

```
//for any f,g,h and stream1
var f = c=>c.extract()*2;
var g = c=>c.extract()*3;
var h = c=>c.extract()*5;
var stream1 = Stream([1,3,4,5], 5, [ 7, 9]);

cokleisli = (f,g)=>c=> g(c.extend(f));
var id = c=>c.extract();
//right identity
stream1.extend(cokleisli(f, id)).extract() ==stream1.extend(f ).extract()
//left identity
stream1.extend(cokleisli(id,f )).extract() ==stream1.extend(f ).extract()
//composition
stream1.extend(cokleisli(f,cokleisli(g,h))).extract()==
stream1.extend(cokleisli(cokleisli(f,g),h)).extract()
```

Run this: [Js Fiddle](#)

10.9 Tree Comonad

We can see our default Tree Algebraic structure as a comonad. The way to create the `duplicate` method is to use the current tree instance as a new value `duplicate:() => node(_, (left, value, right), _)` and then use for the `left` and `right` the recursive definitions as usual `left.duplicate()` and `right.duplicate()`

```

var node = (left, value, right) => ({

  map: f => node(left.map(f), f(value), right.map(f)),
  fold: (acc, f) => left.fold(right.fold(f(acc, value), f), f),
  duplicate: () => node(left.duplicate(), (left, value, right), right.duplicate()),
  extend: f => node (left, value, right).duplicate().map(f),
  extract: () => value,
});

var leaf = value => ({

  map: f => leaf(f(value)),
  fold: (acc, reducer) => reducer(acc, value),
  duplicate: () => leaf(leaf(value)),
  extend: f => leaf(value).duplicate().map(f),
  extract: () => value,

});

```

Run This: [Js Fiddle](#)

10.9.1 Tree annotation example

In this section we will define a comonad on a tree structure ***that is not the same as the standard binary tree implementation that we saw in the previous section.*** Here we will add in each tree node a Boolean value that will signify if there were any changes made to the specific tree node. What we are going to implement here using comonads, **is the scenario that we want to update all the upstream nodes of a node that have changes.**

We will define a new comonad. But this time the important value that we want to include to our comonad that is embedded into the **tree is the annotation only.**

```

var node = (left, right, annotation) => ({

  foldAnn: (acc, f) => right.foldAnn(left.foldAnn(f(acc, annotation), f), f),

  extendAnn: f => node(left.extendAnn(f), right.extendAnn(f),
    f(node(left, right, annotation))),
  extract: () => annotation,
});

var leaf = (value, annotation) => ({

  foldAnn: (acc, f) => f(acc, annotation),

```

```

extendAnn: f => leaf(value, f(leaf(value, annotation))),
extract: () => annotation

});

```

Run this: [Js Fiddle](#)

The foldAnn only accumulates recursively over the annotations of all the nodes. The extend here applies a function over the whole node and return the result as the annotation of the node. So, if we apply a fold to the top node this will take into account all the downstream sub nodes.

```

var hasChanges = node(node(leaf(2, false),
    node(node(leaf(2, false),
        leaf(3, false), false),
        leaf(3, false), false), false),
    leaf(3, false), false)
    .foldAnn(false, (hasOverallChanged, nodeHasChanged) => hasOverallChanged || nodeHas
Changed)); //false

```

Run this: [Js Fiddle](#)

If **any one** of the annotations is true, this will yield a true.

If we combine the Extend with this fold, we get a coflatMap/extend that applies this to all nodes.

```

var tree = node(node(leaf(2, false),
    node(node(leaf(2, false),
        leaf(3, false), false),
        leaf(3, false), false), false),
    leaf(3, false), false);

var hasChanges = tree
    .extendAnn(node =>
        node.foldAnn(false, (hasOverallChanged, nodeHasChanged) => hasOverallChanged ||
nodeHasChanged))
    .extract(); //this is a tree with all annotations updated base on the subnodes;

```

Run this: [Js Fiddle](#)

F- Algebras

11 F-Algebras

If C is a category, and $F: C \rightarrow C$ is an endofunctor of C , then an **F-algebra** is a pair (A, α) , where A is an object of C and α is a morphism $F(A) \rightarrow A$ the a often called *structuremap*. The F-Algebra is a way to get an A out of a $F(A)$.

```
class Expr { }

class ValExpr extends Expr {
  constructor(value) {
    super()
    this.value = value;
  }
  map(f) {
    return new ValExpr(f(this.value));
  }
}

class AddExpr extends Expr {
  constructor(leftExpr, rightExpr) {
    super()
    this.leftExpr = leftExpr;
    this.rightExpr = rightExpr;
  }
  map(f) {
    return new AddExpr(this.leftExpr.map(f), this.rightExpr.map(f));
  }
}
```

Then one algebra with a carrier type of Integers it would be the trivial one (this is the structuremap or α in the above definition)

```
var AlgebraInt = (expression) => {
  if (expression instanceof AddExpr) {
    return expression.leftExpr + expression.rightExpr
  }
  if (expression instanceof ValExpr) {
```

```

        return expression.value
    }
}

```

```
var valExprEval = AlgebraInt(new ValExpr(5)); //5
```

the thing now is to be able to evaluate recursively a more complex expression like this

```
var addExpression = new AddExpr(new ValExpr(4), new ValExpr(5));
```

obviously `var valExprEval = AlgebraInt(addExpression)` does not work. Our first idea is to use `map` and pass the algebra down to the leaf nodes then evaluate those and then work our way up.

An idea with recursion would be this:

```
var evaluateExpression = expression => alg => expression.map(x=>evaluateExpression(x)(alg))
```

```
var addExpression = new AddExpr(new ValExpr(4), new ValExpr(5));
```

```
var evaluation = evaluateExpression(resultAdd)(AlgebraInt);
```

looks great but does not work. Believe me i tried dozens of times. The problem here is that `ValExpr` also try to pass the algebra down to the value. What we need here is the notion of **Base Functor** which unfortunately is too complex to be used in javascript whereas in Haskell there is a whole library for this. The base functor of a functor whould have another `map` we will call it here `mapF` that behaves like below:

```
class Expr { }
```

```

class ValExpr extends Expr {
    constructor(value) {
        super()
        this.value = value;
    }
    map(f) {
        return new ValExpr(f(this.value));
    }
    mapF(f) {
        return new ValExpr(this.value);
    }
}

```

```

class AddExpr extends Expr {
    constructor(leftExpr, rightExpr) {
        super()
        this.leftExpr = leftExpr;
        this.rightExpr = rightExpr;
    }
    map(f) {

```

```

        return new AddExpr(this.leftExpr.map(f), this.rightExpr.map(f));
    }
    mapF(f) {
        return new AddExpr(f(this.leftExpr), f(this.rightExpr));
    }
}

```

Now we have

```
var valExprEval = new ValExpr(5).mapF(AlgebraInt); //5
```

perfect and also it seem to work for Add Expressions

```
var addExpression = new AddExpr(new ValExpr(4), new ValExpr(5));
```

```
var addExpressionResult = AlgebraInt(addExpression.mapF(AlgebraInt)); //9
```

now we feel optimistic that we could possibly write a recursive definition of an evaluator.

```
var evaluateBase = expression => alg => alg(expression.mapF(x => evaluateBase(x)(alg)));
```

```
var evaluation = evaluateBase(addExpression)(AlgebraInt);
```

and this one gives the correct result. We will see this again in the Catmorphism chapter of the recursion schemes.

We can also evaluate the expression with a different algebra

```

var AlgebraArray = (expression) => {
    if (expression instanceof AddExpr) {
        return expression.leftExpr.concat(expression.rightExpr)
    }
    if (expression instanceof ValExpr) {
        return expression.value
    }
}

```

You can guess that this return an array

```
var addExpressionArray = new AddExpr(new ValExpr([4,5,6]), new ValExpr([7,8,9]));
```

```
var evaluation = evaluateBase(addExpressionArray)(AlgebraArray); //Array(6) [4, 5, 6, ..., ...]
```

here the interpretation of the AddExpr means concatenation of arrays (the carrier A here is list[int]) we could Create an algebra on the same Functor that give different interpretations. You can try some for yourself. This functor above is so general because it looks like a monoid. it has an AddExpr that can “add two stuff” like monoidal concat, and a leaf node that has a value (or we can trim the value) which is like the empty.

side note

Base Functor

$$\text{Expr}(a) = \text{ValExpr}(a) + \text{AddExpr}(\text{Expr}(a), \text{Expr}(a))$$

We can get rid of recursion by replacing the `Expr(a)` in the right side with a new type `e`

$$\text{ExprF}(a, E) = \text{ValExprF}(a) + \text{AddExprF}(E, E)$$

That's the base functor. *The functor that we get when we replace recursive occurrences with a new type.* Now the map function with respect to **E** for this new Base Functor are the ones that we create previously.

```
class ValExpr extends Expr {
  mapF(f) {
    return new ValExpr(this.value); //this is like a constant in respect to E
  }
}

class AddExpr extends Expr {
  mapF(f) {
    return new AddExpr(f(this.leftExpr), f(this.rightExpr));
  }
}
```

As we saw I always prefer instead of using this switch statement in order to discriminate the Expression

```
if (expression instanceof AddExpr) { }
if (expression instanceof ValExpr) { }
```

to replace it with plain polymorphism. So, I usually give an eval method to each subclass.

```
class Expr { }

class ValExpr extends Expr {
  eval(alg) {
    return alg.valExpr(this.value);
  }
}

class AddExpr extends Expr {
  eval(alg) {
    return alg.addExpr(this.leftExpr, this.rightExpr);
  }
}
```

```

var AlgebraInt = {
  valExpr: v => v,
  addExpr: (l, r) => l + r,
}

var evaluateBase = alg => expression => expression.mapF(evaluateBase(alg)).eval(alg);

var addExpression = new AddExpr(new ValExpr(4), new ValExpr(5));

var addResult = evaluateBase(AlgebraInt)(addExpression);

```

or we can partially apply the algebra and get an evaluator that we can use with different expressions.

```

var evaluator = evaluateBase(AlgebraInt); //partial appication of the algebra

var addResult =evaluator(addExpression);

```

11.1 F-Algebras Homomorphisms

Category theorists will form a category out of anything so why not get a category of F-Algebras. So, let's take two F-Algebras and see the relationships.

A homomorphism from an F -algebra (A, α) to an F -algebra (B, β) is a morphism $f: A \rightarrow B$ such that $f \circ \alpha = \beta \circ F(f)$, according to the following diagram:

$$\begin{array}{ccc}
 F(A) & \xrightarrow{\alpha} & A \\
 \downarrow F(f) & & \downarrow f \\
 F(B) & \xrightarrow{\beta} & B
 \end{array}$$

What that practically means is that for example we had the above two algebras

```

var AlgebraInt = (expression) => {
  if (expression instanceof AddExpr) {
    return expression.leftExpr + expression.rightExpr
  }
  if (expression instanceof ValExpr) {
    return expression.value
  }
}

```

```

var AlgebraArray = (expression) => {
  if (expression instanceof AddExpr) {
    return expression.leftExpr.concat(expression.rightExpr)
  }
  if (expression instanceof ValExpr) {
    return expression.value
  }
}

```

If we try the mult function from mult: array→int we get that those two evaluations do not match

```

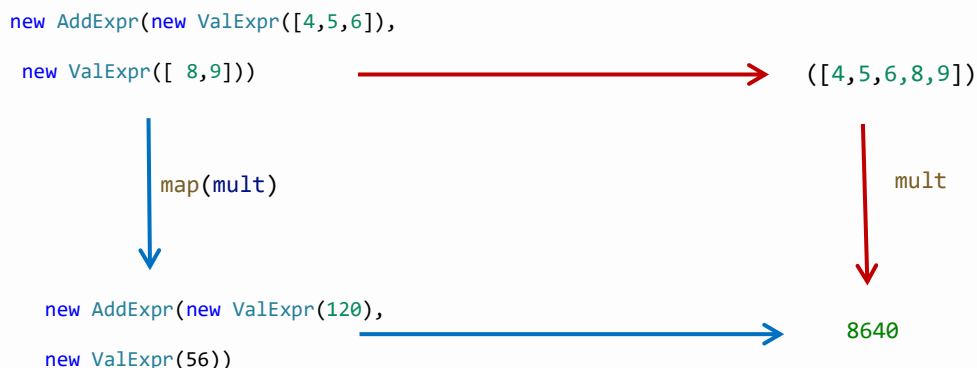
var addExpressionArray = new AddExpr(new ValExpr([4,5,6]), new ValExpr([ 8,9]));

var mult = array=>array.reduce((x,y)=>x*y, 1);

var evaluationAfterMap = evaluateBase(addExpressionArray.map(mult))(AlgebraInt); //192
var mapAfterEvaluation = mult(evaluateBase(addExpressionArray)(AlgebraArray)) //8640

```

In a diagrammatic form it looks like this



so our mult is not a Homomorphism for those two algebras but if we try with a summing

```

var sum = array=>array.reduce((x,y)=>x+y, 0);

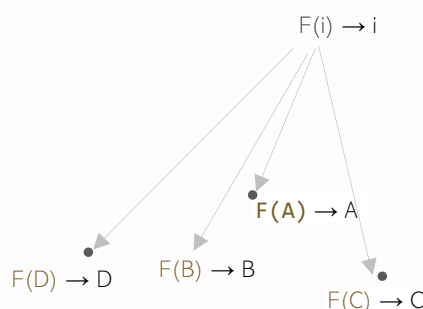
var evaluationAfterMap = evaluateBase(addExpressionArray.map(sum))(AlgebraInt); //32
var mapAfterEvaluation = sum(evaluateBase(addExpressionArray)(AlgebraArray)) //32

```

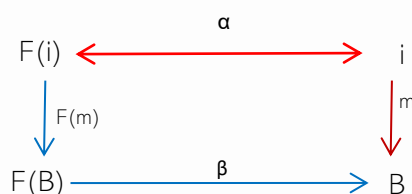
which makes it a F-Algebra Homomorphism for those two specific algebras.

11.2 Initial Algebras

Now in this whole category of F-Algebras there might be this algebra that is the initial object of the category. Any other f-algebra should have an arrow from this initial object by definition to any other object.



Anyways leibek proven that for an initial algebra the structuremap/ evaluation must be an isomorphism. There should be an arrow from the i back to the $F(i)$ not only the $F(i) \rightarrow i$ but $i \rightarrow F(i)$ *also*.



The initial algebra means that has the most information is lossless in the same sense also is the minimal description of something. there would be no other way to be able to reduce this to any other arbitrary algebra if it didn't have all the available information to describe the Algebra. The only general way that we seem to be able to do that is by using the Fix.

```
var Fix = a => ({ unfix: a });
```

12 F-Coalgebra

F-Colagebra is the dual of F-Algebra. So i co-copy the definition reversing the arrows. If C is a category, and $F: C \rightarrow C$ is an endofunctor of C , then an **F-coalgebra** is a pair (A, α) , where

A is an object of C and α is a morphism $A \rightarrow F(A)$ the a often called co-structuremap. The F -coAlgebra is a way to get an $F(A)$ out of a A .

Lets see an example based on the `Expr` Functor used in F -Algebras.

How about this coalgebra:

```
var coAlgebraInt = value => new AddExpr(new ValExpr(value-1), new ValExpr(1));
```

this takes an integer and return a functor `coAlgebraInt: int → Expr`

obviously, we could return whatever expression. Should not have to make any sense

```
var coAlgebraInt = value => new AddExpr(new ValExpr(value *3), new ValExpr(1));
```

```
var coAlgebraInt = value => new AddExpr(new ValExpr(value *3), new ValExpr(value *( value - 1)));
```

```
var coAlgebraInt = value => new ValExpr(1) ;
```

all of them are perfectly valid. Now, are they useful? It depends. For the algebras we also gave an example of catamorphisms. Here we will give an example of an anamorphism of unfold. Where we keep applying a coalgebra to progressively more complex expressions. Here my idea is to replace the second expression recursively with a new `AddExpr` until a get a terminal condition where I will return a terminating expression of `ValExpr`

```
var unfold = value => value == 0 ?
    new ValExpr(0) :
    new AddExpr(unfold(value-1), new ValExpr(value))

var expression = unfold(5);
```

Here I cheated for now because i entangled the coalgebra with the recursion. We will see in more detail in the anamorphism section of the recursion schemes how to get a simple recursive scheme for anamorphisms.

One can test that this expression works as expected by providing it to a catamorphism with the simple `AlgebraInt` where we expect to get back a 5.

```
evaluateBase(AlgebraInt)(unfold (x)) = x
```

the reverse does not hold necessarily as the unfold can only recreate a right expression that is right biased.

```
unfold (evaluateBase(AlgebraInt)(x)) not necessarily x
```

12.1 Catmorphisms

Generalization

```
var zeroF = () => ({mapF: f => zeroF(), })

var succF = (v) => ({v: v, mapF: f => succF(f(v)) });
```

Run this: [Js Fiddle](#)

We can get a coalgebra from int to this functor

```
var coalgNat = n => n > 0 ? succF(n - 1) : zeroF()

var ana = coalg => n => Fix(coalg(n).mapF(v => ana(coalg)(v)))

var natFour = ana(coalgNat)(4)
//Fix(succF(Fix(succF(Fix(succF(Fix(succF(Fix(zeroF))))))))))
```

Run this: [Js Fiddle](#)

Now we will create a catamorphisms for this `Fix[Nat]` object

```
var zeroF = () => ({ eval: () => 0, mapF: f => zeroF() })

var succF = (v) => ({ v: v, eval: () => v + 1, mapF: f => succF(f(v)) });
```

Run this: [Js Fiddle](#)

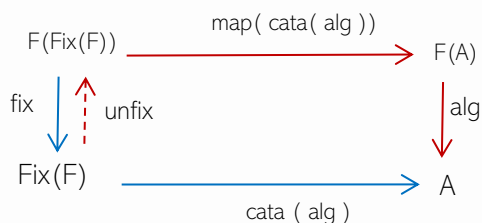
Here the algebra evaluation of a term is embedded in the term, so we can extract this by passing the algebra in the evaluation.

This is another way part of the concept of `Fix` that I would recommend you keep in mind until it makes sense. Because `Fix` is lossless this means that $F(\text{Fix}(F))$ and $\text{Fix}(F)$ are isomorphic having any of them is the same as having the other. You cannot say that with any algebra because in general the $F A \rightarrow A$ transformation seem one way if you evaluate $F a$ to an a then you cannot find exactly what was $F a$. But `Fix` has the simplest possible structure that is `var Fix = a => ({ unfix: a })` you can go in and then out again. Basically `Fix` is the “object composition” in the simplest form just to avoid the recursion through inheritance.

A catamorphisms is a transformation from a `FixF` for example

```
Fix(succF(Fix(succF(Fix(succF(Fix(succF(Fix(zeroF))))))))
```

to an A for example 4 in this case. Look at the following diagram: here we have at the left the $\text{fix}(F)$ which is the initial algebra for F. and in the right we have our algebra. This should commute. Here we want the bottom arrow $\text{Fix}(F) \rightarrow A$ this is the catamorphisms because we can deconstruct the structure by using unfixing and if you follow the red arrows should be equal with the blue arrow so we can write



$$\text{cata}(\text{alg}) = \text{alg} \circ \text{map}(\text{cata}(\text{alg})) \circ \text{unfix}$$

in a symbolic form its not easy to comprehend. I was perplexed when I first saw this without any explanation.

this means:

1. first, we apply the unfix and get the inner structure unfix
2. Then (\circ) we map cata to the children of the structure map(cata(alg)) \circ
3. Then (\circ) apply the algebra to evaluate the childrens alg \circ

```
const cata = fix => fix.unfix.mapF(cata).eval();
```

```
cata(ana(coalgNat)(4)) ;//4
```

Run this: [Js Fiddle](#)

```
var zeroF = ( ) => ({ eval:(alg)=>alg.zeroF(this), mapF: f => zeroF(), })
```

```
var succF = (v) => ({v:v, eval:(alg)=>alg.succF(succF(v)), mapF: f => succF(f(v)), });
```

Run this: [Js Fiddle](#)

And we modify the cata function in order to add the algebra as a parameter

```
var algNat = {
  succF: (x) => x.v + 1,
  zeroF: () => 0,
}
```

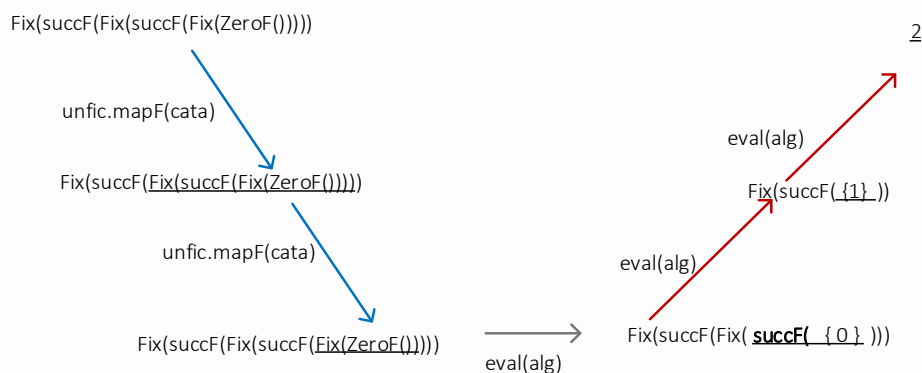
```
const cata = alg => fix => fix.unfix.mapF(cata(alg)).eval(alg);

cata(algNat)(ana(4)).1();//4
```

Run this: [Js Fiddle](#)

We can try another algebra now for example from $\text{alg}: \text{Fix}[\text{Nat}] \rightarrow \text{string}$

```
var algStep = {
  succF: (x) => x.v + '/add one ',
  zeroF: () => 'zero',
}
cata(algStep)(ana(4)).1();
// "zero/add one /add one /add one /add one /add one "
```

Run this: [Js Fiddle](#)

Buy the Book

If you liked this book you perhaps, you'd like to have a paper copy of the book.

Also, there is a **video course** on the way on [Leanpub](#) .I think it could help a lot witnessing someone actually implementing the various concepts.

Contact

Feel free to contact me at:

<https://leanpub.com/u/dimitrispapadim>

<https://medium.com/@dimpapadim3>

<https://github.com/dimitris-papadimitriou-chr>

dimitrispapadim@live.com

<https://github.com/dimitris-papadimitriou-chr/functional-resources>

© 2019 Dimitris Papadimitriou