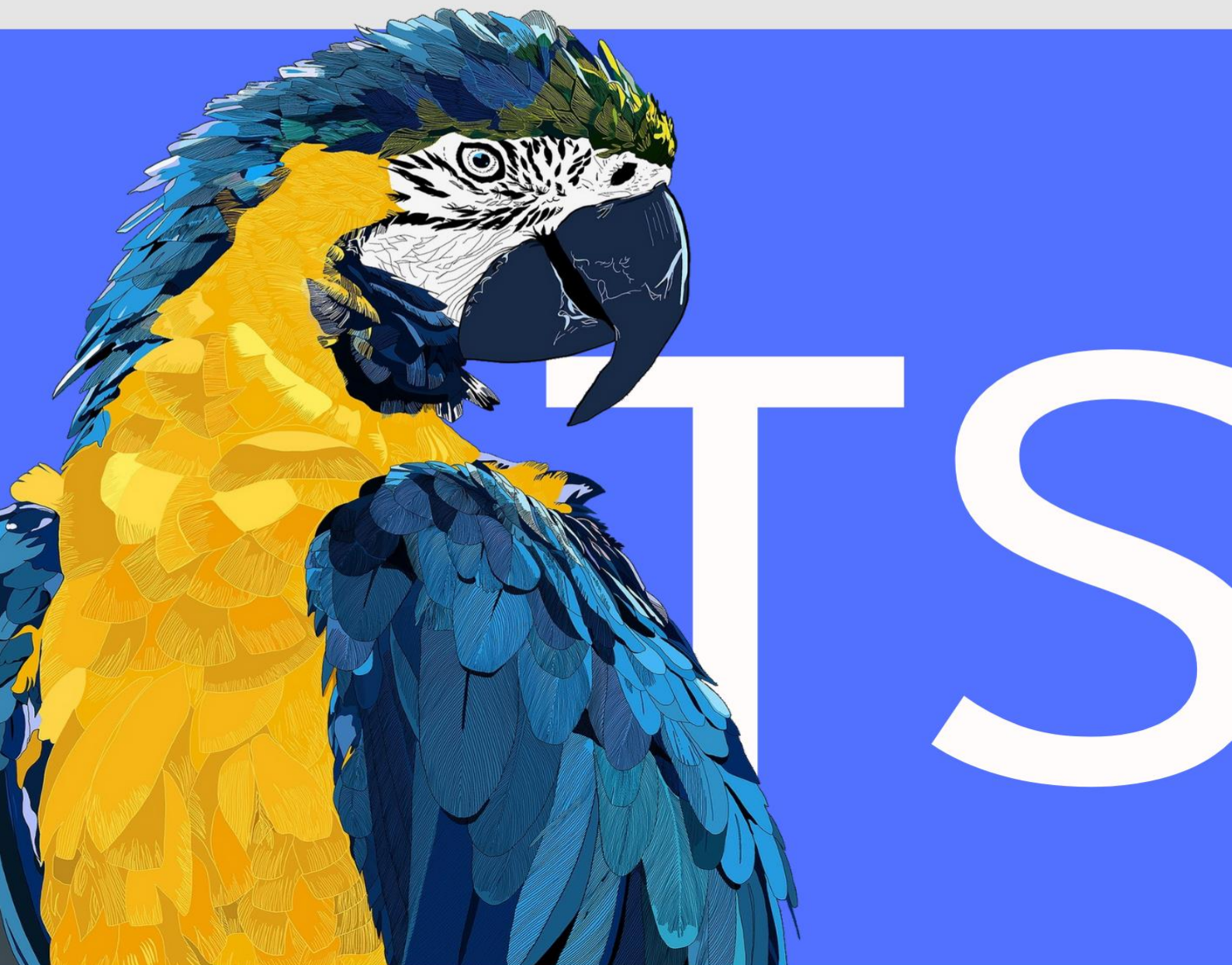


# Functional Programming in **TypeScript**

{ with: categories }

Gain advanced understanding of the mathematics behind  
modern functional programming



**Dimitris Papadimitriou**

# FUNCTIONAL PROGRAMMING IN TYPESCRIPT WITH CATEGORIES

Gain advanced understanding of the mathematics behind modern functional programming

Dimitris Papadimitriou

This version was published on 02/01/2020

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](mailto:dimitrispapadim@live.com)

## Resources

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

## Acknowledgments

<https://pixabay.com/illustrations/tiger-animal-stare-teeth-wild-2823181/>

© 2019 Dimitris Papadimitriou

# Contents

About this book coding conventions .....	5
1 Algebras of Programming .....	6
1.1 Categories .....	6
1.2 Monoids .....	8
1.2.1 Folding monoids .....	11
1.2.2 Monoid homomorphisms and Parallelism.....	11
2 Algebraic Data Types.....	15
2.1 The product structure:.....	17
2.1.1 Introduction / Elimination.....	19
2.2 The Co-Product (aka Union) structure.....	19
2.2.1 Introduction / Elimination.....	22
2.3 Extending Union Types.....	22
2.3.1 Adding Pattern Matching extensions to Union Types .....	23
2.3.2 Rewriting Union Type methods with matchWith .....	24
2.4 One.....	25
2.5 Recursive Algebraic Types .....	26
2.5.1 Rewriting Map with matchWith.....	27
2.5.2 On the value of the symbolic representation .....	28
2.5.3 Adding Pattern Matching extension to Native List<T> .....	28
3 Functors.....	33
3.1 The Identity Functor .....	33
3.2 Commutative Diagrams .....	35
3.3 The Functor Laws .....	36
3.4 A brief mentioning of <i>Catamorphisms</i> .....	38
3.5 Extending Promise<T> to Functor.....	39
3.6 IO Functor, a Lazy Functor .....	41
3.6.1 Lazy<T> as Functor .....	41
3.6.2 IO Functor.....	43
3.7 Reader Functor.....	44
3.8 Maybe Functor.....	45
Dealing with null - Null object Design pattern .....	45
3.8.1 The Null Object Design pattern .....	45
3.8.2 The Functional equivalent - Maybe as Functor.....	47
3.9 Either Functor .....	50
3.9.1 Using Either for exception handling.....	52

3.9.2	Either and the Task<T> functor.....	54
3.10	Functors from Algebraic Data Types.....	57
3.11	Applicative Functors.....	59
3.12	Reader Applicative Functor.....	61
4	Catamorphisms Again.....	63
4.1	A brief mentioning of F-algebras .....	63
4.2	Catamorphisms.....	64
4.3	Initial algebra.....	66
4.3.1	F-Algebras Homomorphisms .....	66
4.4	Catamorphisms for Trees.....	67
4.5	Reversing a Tree .....	69
4.6	Catamorphisms with the Visitor Design pattern.....	70
4.6.1	The Base Functor of the List<T>.....	71
4.6.2	F-Coalgebra.....	75
4.6.3	A brief mentioning of Anamorphisms.....	76
4.6.4	Corecursion .....	77
4.6.5	A brief mentioning of Hylomorphisms.....	78
4.6.6	Hylomorphism example: Mergesort.....	80
4.7	Fold's relation to Cata .....	81
5	Traversable.....	83
5.1	Traversable Algebraic data structures.....	86
5.2	Traversing with The Task<T> aka Parallel .....	87
5.3	Applicative Reader Isomorphism with the Interpreter Design pattern .....	89
5.3.1	A Catamorphism implementation .....	90
5.3.2	Reader applicative implementation .....	92
5.3.3	Object Oriented Interpreter Pattern implementation .....	94
5.4	Foldable.....	95
5.4.1	FoldMap.....	96
5.5	Fold and FoldMap Derivation from Catamorphism .....	98
5.6	Traverse Derivation from Catamorphism.....	99
6	Monads.....	100
7	Comonads .....	100

# 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<sub>2</sub> 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 TS, 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.

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.

## Functional Libraries in TS

Fortunately, or unfortunately there is only a few TS functional libraries at this point. The two most mature are [purify](#) and [fp-ts](#). **Those are complete libraries** with a very good quality codebase. Also, they are mature enough to be used in TS production code. I will not use them in code samples, but I will point out the respective implementations of the Functors that we are going to see in this book.

# Algebras of Programming

---

## 1 Algebras of Programming

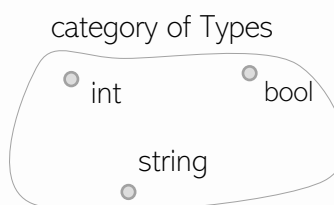
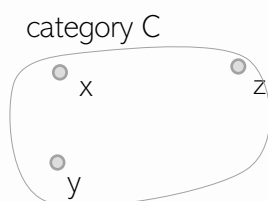
### 1.1 Categories

« Thinking is necessary in order to understand the empirically given, and concepts and “categories” are necessary as indispensable elements of thinking »  
A. Einstein

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** number, boolean, string, etc. There are many interesting categories in programming and in this book, we will explore some of them.

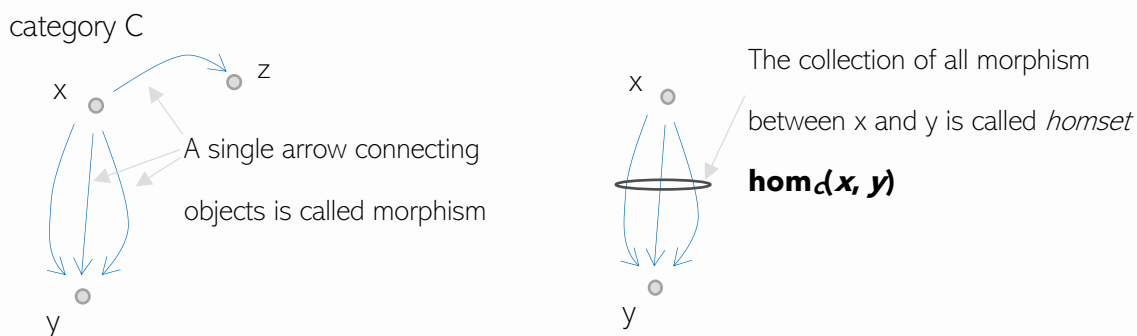
2. A class  $\text{hom}(C)$ , whose elements are called **morphisms** or *arrows*. Each morphism  $f$  has a *source object*  $a$  and *target object*  $b$ . For our type category, any arrow from  $\text{int} \rightarrow \text{bool}$  is a function. For example, this one:

```
var isEven: (v: number) => boolean = a => (a % 2 === 0);
```

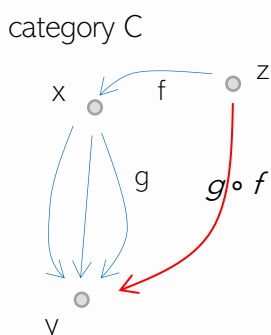
Or this one:

```
var isLessThan10: (v: number) => boolean = a => a < 10;
```

the set of all those morphisms is called HomSet, and we write **hom(int, bool)**. In general, in any category  $C$  the expression **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  $\circ$  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 Monoids

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

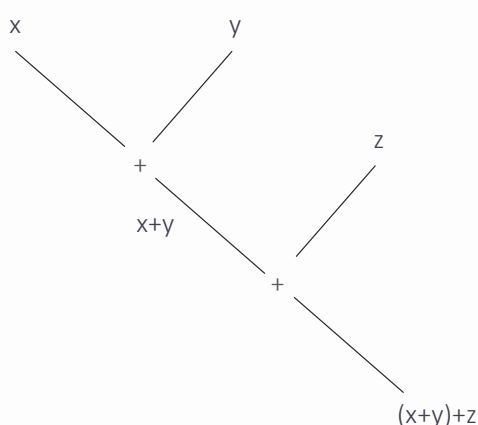
– Categories for the Working Mathematician

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 TS have a default monoid, which is formed by the concatenation and the empty string ( '', **concat** ). String type in TS 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 TS already uses concat for strings and arrays.]

```
interface monoid<T> {
  empty: T;
  concat: (u: T, v: T) => T
}
```

We can define for example:

```
const Sum: monoid<number> = ({
  empty: 0,
  concat: (u: number, v: number) => u + v
})
```

which we will use it like this:

```
var total: number = Sum.concat(Sum.empty, Sum.concat(3, 4));
```

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

```
total = [ 1, 3, 4, 5 ].reduce( sum.concat ,sum.empty);
```

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.

```
interface IMonoidAcc<T> {
  Identity: T;
  concat: (v: IMonoidAcc<T>) => IMonoidAcc<T>
}
```

```
class SumAcc implements IMonoidAcc<number> {
  concat(v: IMonoidAcc<number>): IMonoidAcc<number> {
    return new SumAcc(v.Identity + this.Identity);
  }
  constructor(value: number) {
    this.Identity = value;
  }
  Identity: number = 0;
  //concat: (v: SumAcc) => SumAcc = (v: SumAcc) => new SumAcc(v.Identity + this.Identity);
}
```

The return type is `IMonoidAcc` allowing for fluent chaining

```
var sum :IMonoidAcc<number>= new SumAcc(0).concat(new SumAcc(1)).concat(new SumAcc(2)).concat(new SumAcc(3));
```

Run This: [Fiddle](#)

This formulation is nice in the sense that allows for fluent chaining `var total = new SumAcc(0).concat(new SumAcc(1)).concat(new SumAcc(2))` 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.

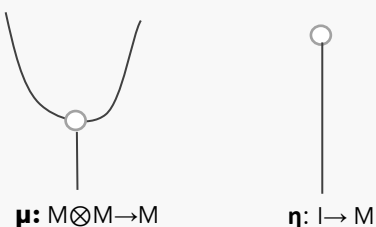
Unfortunately we don't have the pattern matching capabilities of Haskell for example that would allow us to declare something like this `Sum(x).concat(Sum(y))=Sum(x+y)` so we have to expose the value as Identity which makes sense in a categorical point of view also in order to make it work.

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

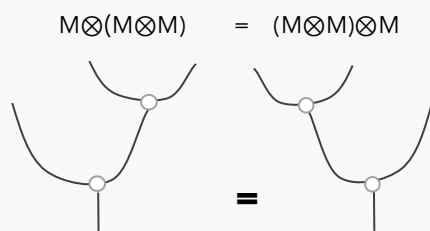
## Category Theory

## 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 and identity have the structure of the following graphical rules in string diagrams (don't worry about the rigorous definition) :



## 1.2.1 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 TS 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.

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

```
var list = [ 1, 2, 3, 4 ];
var total = 0;
foreach (var item in list)
    total = total + item;
```

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.

```
function fold<T>(list: Array<T>, acumulate: T, concat: (u: T, v: T) => T): T {
    list.forEach(item => {
        acumulate = concat(acumulate, item);
    })
    return acumulate;
}
```

Run This: [Fiddle](#)

Now we can use this function with all kind of monoids

```
var sum = fold([2, 3, 4], Sum.empty, Sum.concat);
var product = fold([2, 3, 4], 1, (x, y) => x * y);
var max = fold([2, 3, 4], Number.MIN_SAFE_INTEGER, (x, y) => x > y ? x : y);
```

Run This: [Fiddle](#)

## 1.2.2 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" + "homomorphism ").length == ("monoid".length + "homomorphism ".length);
```

Or this one

```
var totalSum:number = 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 monoid  $(0, (x, y) \Rightarrow x + y)$  and the 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.

```
function slice<T>(list: Array<T>, size: number): Array<Array<T>> {
  var accumulation: Array<Array<T>> = [];
  for (var i = 0; i < list.length; i += size) {
    var chunk: (Array<T>) = list.slice(i, i + size);
    accumulation.push(chunk);
  }
}
```

```

    return accumulation;
}

function mapReduce<T, T1>(
  list: Array<Array<T>>,
  f: (v: T) => T1,
  m: monoid<T1>): T1 {
  var accumulation: Array<T1> = [];

  for (var i = 0; i < list.length; i++) {
    var chunk = list[i];
    var reduction: T1 = chunk.map(f).reduce(m.concat, m.empty);
    accumulation.push(reduction);
  }

  return accumulation.reduce(m.concat, m.empty);
}

var bigList = [...Array(1000)].map((_, i) => i);

var reduced = mapReduce<number, number>(slice(bigList, 100), x => x + 1,
  ({ empty: 0, concat: (x, y) => x + y }));

```

Run This: [Fiddle](#)

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  $(e_1 \bullet \dots (e_2 \bullet (e_3 \bullet e_n)))$  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 Dictionary. We can prove that if we have a *VariableStore*  $\langle T\text{Value} \rangle$  with keys of type *string* and values of type *TValue* and *TValue* **is a monoid** then we can create a *VariableStore monoid*.

```

type VariableStore<T> = { [name: string]: T; };

var dictionary1: VariableStore<number> = {};
dictionary1["and"] = 5;
dictionary1["the"] = 3;

var dictionary2: VariableStore<number> = {};
dictionary2["and"] = 7;
dictionary2["or"] = 4;

var mergeMonoid = new DictionaryMonoid(({ empty: 0, concat: (x, y) => x + y }))
var mergedWordCount: VariableStore<number> = mergeMonoid.concat(dictionary1, dictionary2);

class DictionaryMonoid<TValue> implements monoid<VariableStore<TValue>>
{
    ValueMonoid: monoid<TValue>;
    constructor(valueMonoid: monoid<TValue>) {
        this.ValueMonoid = valueMonoid;
    }

    empty: { [name: string]: TValue; } = {};
    concat: (u: VariableStore<TValue>, v: VariableStore<TValue>) => VariableStore<TValue>
        = (u, v) => {
            var merge = { ...v }; //copy v
            for (var key in u)
                if (merge[key])
                    merge[key] = this.ValueMonoid.concat(merge[key], u[key])
                else
                    merge[key] = u[key]
            return merge;
        };
}

```

Run This: [Fiddle](#)

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

## 2 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) =  $[] + a * \text{List}(a)$

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

```
abstract class ListBase<T> { }

class Cons<T> extends ListBase<T> {
  Rest: ListBase<T>
  Value: T
  constructor(value: T, rest: ListBase<T>) {
    super();
    this.Rest = rest;
    this.Value = value;
  }
}

class empty<T> extends ListBase<T> { }
```

remember in the folding monoids section I briefly mentioned that we could store a bunch of integers [1, 2, 3] in a structure like this as well

```
const list = new Cons(1, new Cons(2, new Cons(3, new empty())));
```

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.

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.

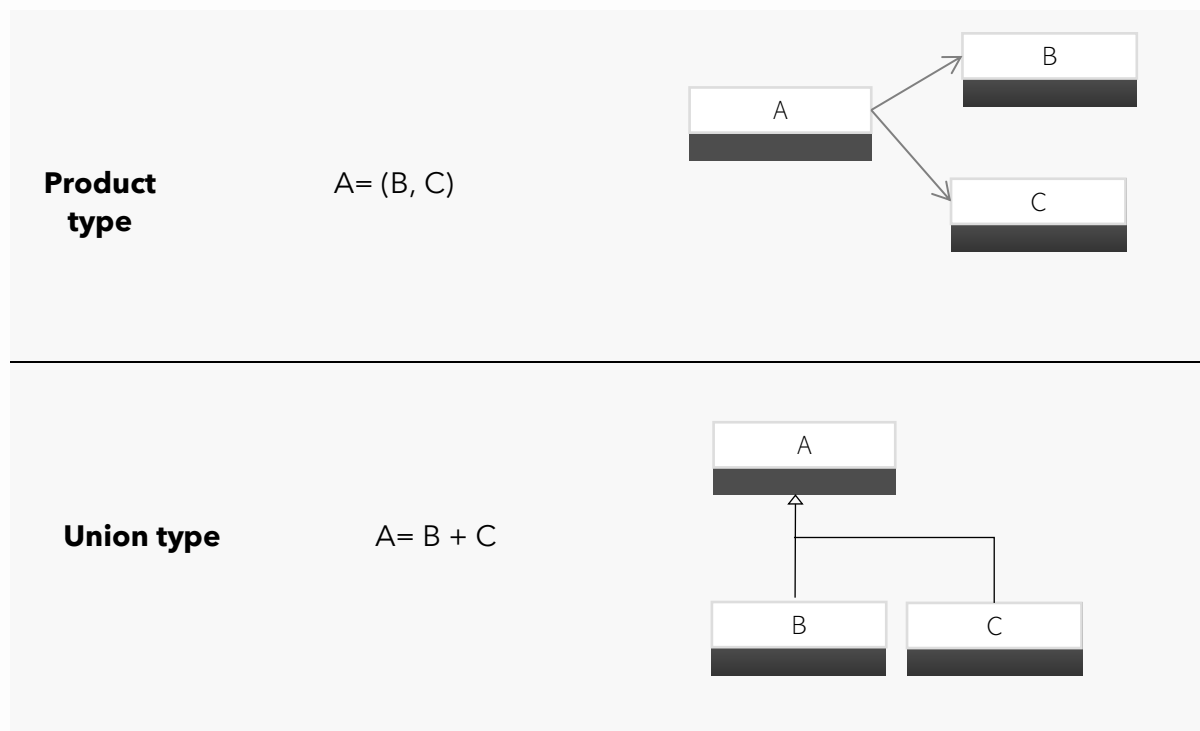


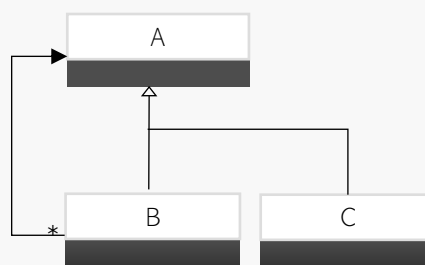
1. **Product** [object-oriented equivalent: **composition**]: In our definition of the list the object literal that has two properties is a product  
`Cons(T value, ListBase<T> rest)`
2. **Coproduct** [object-oriented equivalent: **inheritance**]: In our definition of the list, `Cons<T> extends ListBase<T>`, and `empty<T> extends ListBase<T>` 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<T> extends ListBase<T> Rest: ListBase<T>; Value: T;`  
 the `Rest` is a `ListBase<T>` 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.



**Recursive Type** $A = (B, A) + C$ 

## 2.1 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 TS 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

**Tuples are products**

```
[A, B]
```

```
var create: <TA, TB> (A: TA, B: TB) => [TA, TB] = (A, B) => [A, B];
```

**Objects are products**

```
class Product<TA, TB> {
  A: TA
  B: TB
  constructor(a: TA, b: TB) {
    this.A = a;
    this.B = b;
  }
}
```

```
var create: <TA, TB> (A: TA, B: TB) => Product<TA, TB> = (A, B) => new Product(A, B)
```

**Object Literals are products**

```
({ A: TA, B: TB })
```

```
var create: <TA, TB> (A: TA, B: TB) => ({ A: TA, B: TB }) = (A, B) => ({ A, B });
```

List also are products. In Typescript lists and Tuples are interconnected to point out this fact.

all those are isomorphic /contain the exact same information. In fact, the Typescript “compiler” does not distinguish between `Product<TA, TB>` and `{ A: TA, B: TB }`

```
function f<TA, TB>(product: Product<TA, TB>) { }
```

```
f({ A: 1, B: 2 }) // no error by the compiler. The naming of the properties must agree
```

That’s why we can replace for example the argument list on a method with a Tuple or an object. The following are all equivalent:

```
function f<TA, TB>(A: TA, B: TB) { }
```

```
function f<TA, TB>(product: Product<TA, TB>) { }
```

```
function f<TA, TB>(product: { A: TA, B: TB }) { }
```

```
function f<TA, TB>(product: [TA, TB]) { }
```

This transformation commonly called **Introduce Parameter Object** refactoring move. And it is based on the fact that both `(,,,)` and Tuples `((,,))` 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

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

called **projections**.

For example, for a tuple definition of a product `[,]` the projections would be

```
var productTuples: <TA, TB> (A: TA, B: TB) => [TA, TB] = (A, B) => [A, B];
```

```
var first: <TA, TB> (p: [TA, TB]) => TA = (product) => product[0];
```

```
var second: <TA, TB> (p: [TA, TB]) => TB = (product) => product[1];
```

for a `{ A: TA, B: TB }` definition of a product the projections would be

```
var productLiteral: <TA, TB> (A: TA, B: TB) => ({ A: TA, B: TB }) = (A, B) => ({ A, B });
```

```
var first: <TA, TB> (p: { A: TA, B: TB }) => TA = (product) => product.A;
```

```
var second: <TA, TB> (p: { A: TA, B: TB }) => TB = (product) => product.B;
```

I hope you got the idea.

## 2.1.1 Optional Introduction / Elimination

The Product is the equivalent to a logical AND or a conjunction  $A \wedge B$  of two propositions A and B. In order to conclude that the proposition  $A \wedge B$  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  $A \wedge B$ . This rule is commonly named **introduction** in mathematical logic because it introduces a conjunction  $\wedge$  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  $A \wedge B$  the following hold  $\frac{A \wedge B}{A}$  and  $\frac{A \wedge B}{B}$  or the propositions  $A \wedge B \rightarrow A$ ,  $A \wedge B \rightarrow B$  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.

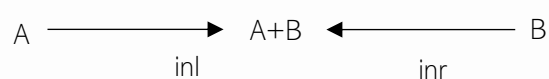
```
var first = product => product.first;
var second = product => product.second;
```

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.

## 2.2 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  $S=A+B$ . In a strong typed language 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.  
**Injections are just the Constructors of the subclasses.**

```
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();
```

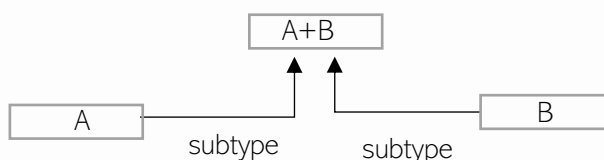
```
class B extends Coproduct {
  constructor() {
    super();
  }
}
```

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 TS 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  $n < m$  than 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

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

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

For example, the Typescript compiler does not allow us to make this assignment

```
function f<TA, TB, TC>(p: { A: TA, B: TB, C: TC }) { }
var t = f({ A: 1, B: 2 });
```

With error:

Argument of type '{ A: number; B: number; }' is not assignable to parameter of type '{ A: number; B: number; C: {}; }'.

Property 'C' is missing in type '{ A: number; B: number; }' but required in type '{ A: number; B: number; C: {}; }'

The reverse also is not recognized from the compiler even if its safe

```
function f<TA, TB, TC>(p: { A: TA, B: TB }) { }
var t = f({ A: 1, B: 2, C: 4 });
```

Argument of type '{ A: number; B: number; C: number; }' is not assignable to parameter of type '{ A: number; B: number; }'.

Object literal may only specify known properties, and 'C' does not exist in type '{ A: number; B: number; }'

The following though is fine obviously, even it does not differ from the previous case

```
interface Union<TA, TB> { A: TA, B: TB }

class Product<TA, TB> implements Union<TA, TB> {
  A: TA; B: TB; C: TB
}
```

```
function f<TA, TB>(p: Union<TA, TB>) { }
var t = f(new Product());
```

Another way to discriminate would be to add a tag to each type and we are not going to do that here. There was always a demand in the TS community especially while functional programming gaining ground for Pattern Matching. Finally, in TS 8.0 there would be pattern matching capabilities that will allow to write more elegant TS for coproduct types.

## 2.2.1 Optional Introduction / Elimination

Co-Product is the equivalent to a logical OR or disjunction **A v B** of two propositions A, B. In order to conclude that the proposition **A v 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 v B**. This rule is commonly named disjunction v **introduction** in mathematical logic because it introduces a

disjunction v 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 v B if we have an AvB 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 → C**) **and** a rule that gives us a C from a B (**B → C**), **and** we have either of A or B (**A v 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.

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.

## 2.3 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 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:

```
abstract class ListBase<T>
```

```

{
  abstract Fold(monoid: { empty: () => T; concat: (x: T, y: T) => T; }): T;
}

class Cons<T> extends ListBase<T> {
  Fold(monoid: { empty: () => T; concat: (x: T, y: T) => T; }): T {
    return monoid.concat(this.Value, this.Rest.Fold(monoid));
  }
}

class empty<T> extends ListBase<T> {
  Fold(monoid: { empty: () => T; concat: (x: T, y: T) => T; }): T {
    return monoid.empty()
  }
}

const list = new Cons(1, new Cons(2, new Cons(3, new empty())));
var fold = list.Fold({ empty: () => 1, concat: (x, y) => x * y });

```

Run This: [TS Fiddle](#)

**Here I have added `Fold` in both parts of the union.** Don't worry about the implementation which is recursive we will see examine it in the next section closely.

## 2.3.1 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`.

```

abstract class ListBase<T> //implements Showable
{
  abstract matchWith<T1>(pattern: ({ empty: () => T1, cons: (v: T, rest: ListBase<T>) => T1 }
)): T1;
}

class Cons<T> extends ListBase<T> {
  matchWith<T1>(pattern: { empty: () => T1; cons: (v: T, rest: ListBase<T>) => T1; }): T1 {
    return pattern.cons(this.Value, this.Rest);
  }

  Rest: ListBase<T>
  Value: T
  constructor(value: T, rest: ListBase<T>) {
    super();
    this.Rest = rest;
    this.Value = value;
  }
}

```



```
class empty<T> extends ListBase<T> {
  matchWith<T1>(pattern: { empty: () => T1; cons: (v: T, rest: ListBase<T>) => T1; }): T1 {
    return pattern.empty();
  }
}
```

Run This: [TS Fiddle](#)

## 2.3.2 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 `ListBase<T>` Union Type using just `matchWith`. Here I rewrite the `Fold` using an extension method:

```
interface ListBase<T> {
  Fold(monoid: { empty: () => T; concat: (x: T, y: T) => T; }): T
}

ListBase.prototype.Fold = function <T>(monoid:
  { empty: () => T; concat: (x: T, y: T) => T; }): T
{
  return this.MatchWith({
    empty: () => monoid.empty(),
    cons: (v, r) => monoid.concat(v, r.Multiply(monoid))
  });
}
```

Run This: [TS Fiddle](#)

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

```
abstract class ListBase<T>
{
  Fold(monoid: { empty: () => T; concat: (x: T, y: T) => T; }): T {
    return this.MatchWith({
      empty: () => monoid.empty(),
      cons: (v, r) => monoid.concat(v, r.Fold(monoid))
    });
  }
}
```

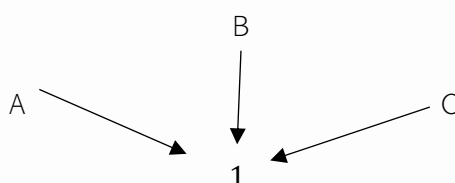
Run This: [TS Fiddle](#)

We are going to use this way of extending Union Types extensively in this book, as it is one of the functional best practices.

## 2.4 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 TS 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:

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

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.

## 2.5 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:

```
abstract class ListBase<T>
{
  abstract Map<T1>(f: (y: T) => T1): ListBase<T1>;
}

class Cons<T> extends ListBase<T> {
  Rest: ListBase<T>
  Value: T
  constructor(value: T, rest: ListBase<T>) {
    super();
    this.Rest = rest;
    this.Value = value;
  }
  Map<T1>(f: (y: T) => T1): ListBase<T1> {
    ???
  }
}

class empty<T> extends ListBase<T> {
  Map<T1>(f: (y: T) => T1): ListBase<T1> {
    ???
  }
}
```

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  $f$ .

```
Map<T1>(f: (y: T) => T1): ListBase<T1> {
  return new empty<T1>();
}
```

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

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<T1>(f: (y: T) => T1): ListBase<T1> {
  return new Cons<T1>(f(this.Value), this.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.

```
abstract class ListBase<T>
{
  abstract Map<T1>(f: (y: T) => T1): ListBase<T1>;
}

class Cons<T> extends ListBase<T> {
  Rest: ListBase<T>
  Value: T
  constructor(value: T, rest: ListBase<T>) {
    super();
    this.Rest = rest;
    this.Value = value;
  }
  Map<T1>(f: (y: T) => T1): ListBase<T1> {
    return new Cons<T1>(f(this.Value), this.Rest.Map(f))
  }
}

class empty<T> extends ListBase<T> {
  Map<T1>(f: (y: T) => T1): ListBase<T1> {
    return new empty<T1>();
  }
}
```

Run This: [TS Fiddle](#)

## 2.5.1 Rewriting Map with matchWith

And since we have a `matchWith` method in place we can rewrite `Map` as an extension method:

```
abstract class ListBase<T>
{
  map<T1>(f: (y: T) => T1): ListBase<T1> {
    return this.MatchWith({
      empty: () => new empty<T1>(),

```

```

    cons: (v, r) => new Cons<T1>(f(v), r.map(f))
  });
}
}

```

## 2.5.2 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.

## 2.5.3 Adding Pattern Matching extension to Native List<T>

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 <T1>(pattern: ListPattern<any, T1>) {
  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<T>`

```
function Fold<T>(array: Array<T>, monoid: { empty: () => T; concat: (x: T, y: T) => T; }):
T {
  return this.MatchWith({
    empty: () => monoid.empty(),
    cons: (v, r) => monoid.concat(v, r.Fold(monoid))
  });
}
```

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`

```
function map2<T, T1>(array: Array<T>, f: (y: T) => T1): Array<T1> {
  return array.MatchWith({
    empty: () => [],
    cons: (v, r) => [f(v), ...map2(r, f)]
  });
}
```

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)]`

```

public static Zip<T>(a1: Array<T>, a2: Array<T>): Array<T> {
    return a1.MatchWith({
        empty: () => a2,
        cons: (x, xs) => a2.MatchWith({
            empty: () => a1,
            cons: (y, ys) => [x, y].concat(F.Zip(xs, ys))
        })
    });
}

console.log(F.Show(F.Zip([1, 2, 3], [6, 7, 8, 7,9,10,11]))); //[1,6,2,7,3,8,9,10,11]

```

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

```

public static ZipMap<T, T1>(a1: Array<T>, a2: Array<T>,
    f: (u: T | null, v: T | null) => T1): Array<T1> {
    return a1.MatchWith({
        empty: () => a2.map(x => f(null, x)),
        cons: (x, xs) => a2.MatchWith({
            empty: () => a1.map(x => f(x, null)),
            cons: (y, ys) => [f(x, y)].concat(F.ZipMap(xs, ys, f))
        })
    });
}

```

Run This: [TS Fiddle](#)

For example `F.ZipMap([1, 2, 3], [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=>..`

## Summary of Algebraic data

Name

Signature

### Natural numbers

[Peano numerals]

```
abstract class Numeral { }

class Succ extends Numeral {
  Rest: Numeral
  constructor(rest: Numeral) {
    super();
    this.Rest = rest;
  }
}

class Zero extends Numeral { }

var numeral = new Succ(new Succ(new Succ(new Succ(new Zero()))))
```

### List

```
abstract class ListBase<T> { }

class Cons<T> extends ListBase<T> {
  Rest: ListBase<T>
  Value: T
  constructor(value: T, rest: ListBase<T>) {
    super();
    this.Rest = rest;
    this.Value = value;
  }
}

class empty<T> extends ListBase<T> { }

var list = new Cons(2, new Cons(1, new empty()))
```

### Tree

```
abstract class Tree<T>{ }
class Node1<T> extends Tree<T> {
  Left: Tree<T>;
  Value: T;
  Right: Tree<T>;
  constructor(left: Tree<T>, value: T, right: Tree<T>) {
    super();
    this.Left = left;
    this.Value = value;
    this.Right = right;
  }
}

class Leaf<T> extends Tree<T> {
  Value: T;
  constructor(value: T) {
    super();
    this.Value = value;
  }
}
```



```

    }

    var tree:Tree<number> = new Node1(new Leaf(2),2, new Node1(new
    Leaf(2),4,new Leaf(2) ))

```

### **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 terminal 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

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.

$0 \cong \lambda f. \lambda x. x \quad \Rightarrow x \quad \Rightarrow \text{Zero}()$

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

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

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.»

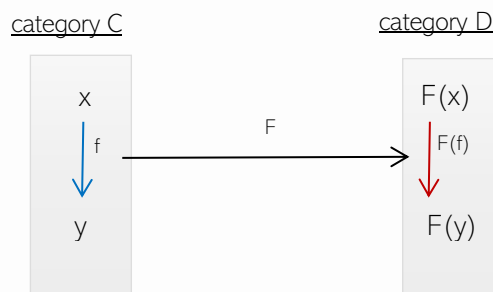
S. Eilenberg and S. MacLane »

## The Idea:

In TS the most famous functional programming idea is to use `Array<T>.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.

## 3 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  $f$ .

### 3.1 The Identity Functor

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

```

class Id<T>
{
  Value: T
  constructor(value: T) {
    this.Value = value;
  }
  map<T1>(f: (y: T) => T1): Id<T1> {
    return new Id<T1>(f(this.Value))
  };
}

```

that's requirement 1

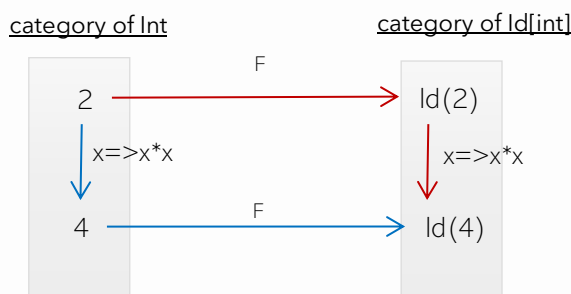
that's requirement 2

Run This: [TS Fiddle](#)

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

1. A “**constructor**” that lifts an object `T` to `Id<T>`
2. and it has a **mapping** method `map<T1>(f: (y: T) => T1)` 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:

```

map<T1>(f: (y: T) => T1): Id<T1> {
  return new Id<T1>(f(this.Value))
};

```

Because:

```

var square = x=>x*x;

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

```

The type of the functor map method is

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  $f(b)$

#### Libraries

lifting objects with. of(\_)

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. In many functional libraries online you may find the explicit definition of an `.of()` that does the same thing

```
class Id<T>
{
  public static of<T>(v: T) { return new Id<T>(v); }
}
```

## 3.2 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.

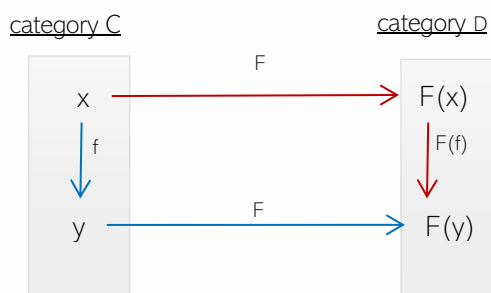
`Id(y) = Id(x).map(f);`

2. Or first lift  $x$  with  $f$  and then get the Functor.

`Id(y) = Id(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.

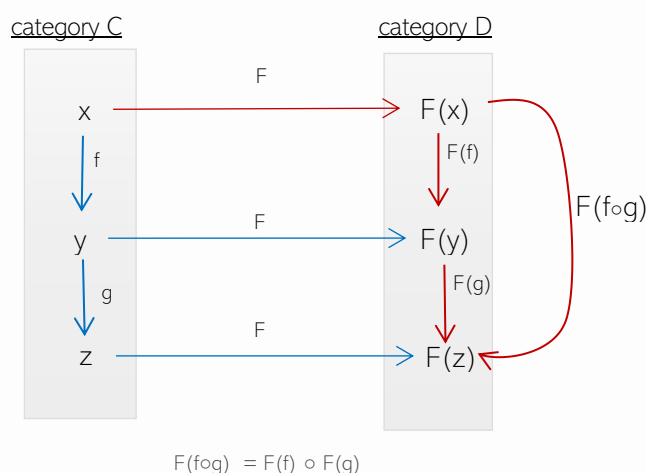
### 3.3 The Functor Laws

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**.

we expect for example that those two expressions should give the same result:

```
[2.0, 3, 4, 5 ].map(discountedPrice.compose(addTitle));

[2.0, 3, 4, 5 ].map(discountedPrice).map(addTitle);
```

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

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

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

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.

### SideNote:

Another way to represent the `Id` functor would be using the object literal notation

```
var Id = <T>(x: T) => ({
  map: <U>(f: (y: T) => U) => Id<U>(f(x))
})
```

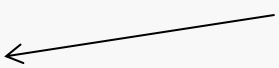
In plain vanilla JavaScript this my preferred way to represent functors. In this book I will be using mostly the class notation.

### 3.4 A brief mentioning of *Catamorphisms*

Here we will make a stop to have a brief look at ***Catamorphisms*** 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:

```
class Id<T>
{
  public Value: T
  constructor(value: T) {
    this.Value = value;
  }
}
```

 We want to not allow direct access


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 or folds.

**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:

```
class Id<T>
{
  private Value: T
  constructor(value: T) { this.Value = value; }
  map<T1>(f: (y: T) => T1): Id<T1> { return new Id<T1>(f(this.Value)); }

  cata<T1>(alg: (y: T) => void): void { alg(this.Value); }
}
```

 **cata** allows us to do

```
new Id(3).cata(console.log);

console.log( new Id(3).Value) // Instead of this
```

Run This: [TS Fiddle](#)

But we can use the more flexible implementation

```
cata<T1>(alg: (y: T) => T1): T1 { return alg(this.Value); }
```

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

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

or we can just create an “overload” for cata (we cannot have overloads in Js and in TS there are constraints because the code transpiles in JS )

```
T cata() { return this.Value ; };
```

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

```
T fold() { return this.Value ; };
```

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

[the standard convention for a fold is to have a type signature like this :

```
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.

## 3.5 Extending Promise<T> to Functor

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

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 <T1>(f: (v: any) => T1) {
  return new Promise<T1>((resolve, reject) => {
    this.then(x => resolve(f(x))).catch(reject);
  });
};
```

Run This: [TS Fiddle](#)



```
new Promise<number>((resolve, reject) => { reject(1) })
  .map(x => x + 3)
```

The mapping function that lifts the function  $f: T \rightarrow T_1$  It follows the steps:

1. Waits for the result of the `Promise` (so in a way unwraps the contained value) `this.then(x => {...})`
2. Applying the function `f(x)`
3. wraps the resulting value again into a new `Promise`: `this.then(x => resolve(f(x)))` 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 possibility of exception **this part belongs to the path of failure and ignores the f mapping** (one could argue that if there is an exception then this does not behave like a functor we will deal with this issue when we talk about the Either functor) We are going to come back to this observation many times throughout this book.

We can use our map function now:

```
class MockClientRepository {
  GetById(id: number): Promise<Client> {
    let clients: Array<Client> = [
      new Client(1, "jim"),
      new Client(2, "john")];

    var client = clients.find(c => c.Id === id);
    return new Promise<Client>((resolve, reject) => {
      if (!client)
        reject('no client found');
      else
        resolve(client)
    });
  }
}

new MockClientRepository()
  .GetById(4)
  .map(x => x.Name)
  .then(x=>console.log(`client Name: ${x}`))
  .catch(x=>console.log(`error : ${x}`))
```

Run This: [TS Fiddle](#)

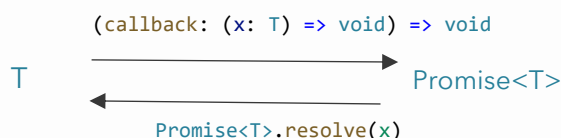
Obviously, the `Promise<T>` is isomorphic with the `T`, we have `Promise.resolve(5)` from `T→Promise<T>` The reverse morphism from `Promise<T>→T` is different because `Task<T>` and `T` live in different time frames. But we know from a theorem called Yoneda

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`



## 3.6 IO Functor, a Lazy 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.

### 3.6.1 Lazy<T> as Functor

#### 3.6.1.1 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.

```
class Lazy<T>
{
    private Value?: T;
    get value(): T {
        if (!this.Value) {
            this.Value = this.Fn();
        }
        return this.Value;
    }

    private Fn: () => T;
    constructor(fn: () => T) {
        this.Fn = fn;
    }
}

var lazy = new Lazy(() => 3);
console.log(lazy.value) //if we use value then the calculation is performed and cached
console.log(lazy.value) //we use the cached result or memoized as usually called in FP
```

Run This: [TS Fiddle](#)

We can extend this lazy type in order to provide a Map method

```
map<T1>(f: (y: T) => T1): Lazy<T1> {
    return new Lazy<T1>(() => f(this.Fn()))
};
```

Run This: [TS Fiddle](#)

The mechanics behind the mapping function that lifts the function  $f: T \rightarrow T1$  follows the same stapes as Task<T>:

1. Waits for the result of the **Lazy** (so in a way unwraps the contained value) `this.Fn()`
2. Applying the function  $f$  : `f(this.Fn())`
3. wraps the resulting value again into a new **Lazy**: `Lazy<T1>(() => f(this.Fn()))`  
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.

You must appreciate that this preserves the Laziness of the **Lazy<T>** at no point the Lazy must collapse without trying to access the outer **Value**. At the step 3 we apply a lazy wrapper `() =>` around what seems as evaluation `this.Value` which thus is not executed.

## 3.6.2 IO Functor

Functional programming in the same lines provides the IO functor

```
class IO<T>
{
  Fn: () => T;
  constructor(fn: () => T) {
    this.Fn = fn;
  }
  map<T1>(f: (y: T) => T1): IO<T1> {
    return new IO<T1>(() => f(this.Fn()))
  };
  matchWith<T1>(f: (y: T) => T1): T1 {
    return f(this.Fn());
  };
  run<T1>(): T {
    return this.Fn();
  };
}
```

```
new IO<int>(() => 3)
.map(x => x + 3)
.matchWith(console.log);
```

Run This: [TS Fiddle](#)

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 `new IO<T1>(() => f(Fn()))`.

We also define a Run method usually that just executes explicitly the computation

```
public T Run() => matchWith (x => x);
```

An example application coming from the frontend web design universe uses IO, to isolate the effects of accessing the DOM elements with jQuery. This is considered Side effect because there is no predictability on the Input. It's not Referential Transparent.

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

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

Run This: [TS 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.

## 3.7 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.

```
class Reader<Env, T>
{
  Fn: (e: Env) => T;
  constructor(fn: (e: Env) => T) {
    this.Fn = fn;
  }
  map<T1>(f: (y: T) => T1): Reader<Env, T1> {
    return new Reader<Env, T1>((env) => f(this.Fn(env)))
  };
  matchWith<T1>(f: (y: T) => T1): (env: Env) => T1 {
    return (env: Env) => f(this.Fn(env));
  };
  run<T1>(env: Env): T {
    return this.Fn(env);
  };
}

public class Config
{
  public Name:string
}

var getName = new Reader<Config, string>(environment => environment.Name).
  map(name => $"Name: {name }").
  run(new Config ( "Sql" ));
```

Run This: [TS 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 `Task<T>` and `IO`

```
new Reader<Env, T1>((env) => f(Fn(env)))
```

1. Evaluate the inner reader `Fn(env)` (so in a way unwraps the contained value)
2. Applying the function `f : f(Fn(env))`
3. wraps the resulting value again into a new `new Reader<Env, T1>((env) => f(Fn(env)))`

## 3.8 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.

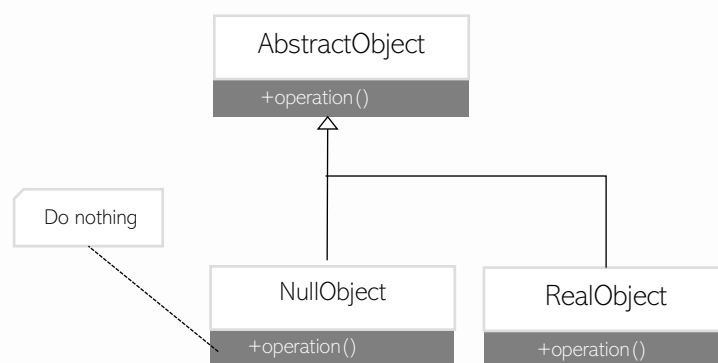
The classical solution is using conditionals everywhere to check for null.

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

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

#### 3.8.1 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**. Were 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.

A simple implementation of the **NullObject** design pattern with TS could be something like this :

```
abstract class AbstractObject<T> {
```

```

    abstract Operation(): void;
}

class RealObject<T> extends AbstractObject<T> {
    Operation(): void {
        // do something with value
    }
    Value: T;
    constructor(value: T) {
        super();
        this.Value = value;
    }
}

class NullObject<T> extends AbstractObject<T> {
    Operation(): void {
        //Do nothing here
    }
}

```

Run This: [TS 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 = NullObject + RealObject`.

Take as an example the more realistic problem that we are going to build upon. Let's say we want to get a `Client` from a repository by Id

```

class Client {
    Id: number;
    Name: string;
    constructor(id: number, name: string) {
        this.Id = id;
        this.Name = name;
    }
}

class MockClientRepository {
    GetById(id: number): AbstractObject<Client> {
        let clients: Array<Client> = [
            new Client(1, "jim"),
            new Client(2, "john")];

        var client = clients.find(c => c.Id == id);

        if (!client)
            return new NullObject<Client>();
        else
            return new RealObject<Client>(client);
    }
}

```

Run This: [TS Fiddle](#)

We want to display the name without having to check for nulls :

```
console.log(new MockClientRepository().GetById(1).Display(v => v.Name));
```

if there was no client we would still use the same syntax, without exceptions

```
console.log(new MockClientRepository().GetById(5).Display(v => v.Name));
```

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.

## 3.8.2 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

```
abstract class Maybe<T>
{
    abstract MatchWith<T1>(pattern: ({ none: () => T1, some: (v: T) => T1 })): T1;
    abstract Map<T1>(f: (v: T) => T1): Maybe<T1>;
}

class Some<T> extends Maybe<T> {
    Value: T
    constructor(value: T) {
        super();
        this.Value = value;
    }

    Map<T1>(f: (v: T) => T1): Maybe<T1> {
        return new Some<T1>(f(this.Value));
    }

    MatchWith<T1>(pattern: ({ none: () => T1, some: (v: T) => T1 })): T1 {
        return pattern.some(this.Value);
    }
}

class None<T> extends Maybe<T> {
    Map<T1>(f: (v: T) => T1): Maybe<T1> {
        return new None<T1>();
    }
}
```



```

    }
    MatchWith<T1>(pattern: ({ none: () => T1, some: (v: T) => T1 })): T1 {
        return pattern.none();
    }
}

```

Run This: [TS Fiddle](#)

Maybe `matchWith` implementation for `Maybe` is straightforward. Since it is a coproduct, we have to implement for each subtype the `MatchWith`, 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

```

var d = new None<number>()
    .map(x => x + 1)
    .matchWith({ none: () => "nothing", some: v => v.toString() });

d = new Some<number>(2)
    .map(x => x + 1)
    .matchWith({ none: () => "nothing", some: v => v.toString() });

```

Run This: [TS Fiddle](#)

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

```

class MockClientRepository {
    GetById(id: number): Promise<Client> {
        let clients: Array<Client> = [
            new Client(1, "jim"),
            new Client(2, "john")];

        var client = clients.find(c => c.Id === id);
        return new Promise<Client>((resolve, reject) => {
            if (!client)
                reject('no client found');
            else
                resolve(client)
        });
    }
}

```

We want to replace this  
with a `Maybe`

Run This: [TS Fiddle](#)

Here we want to replace the result of the array filtering

```
.find(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 :

```

public static FirstOrNone<T>(array: Array<T>, predicate: (v: T) => boolean): Maybe<T> {
    return F.MatchWith(

```

```

    array,
    {
      empty: () => new None<T>(),
      cons: (v, r) => predicate(v) ?
        new Some<T>(v) :
        this.FirstOrNone(r, predicate)
    });
  }
}

```

Where I used the recursive extension of the list that we have seen earlier in order to provide a `matchWith` functionality. Now we can write:

```

class MockClientRepository {
  GetById(id: number): Promise<Maybe<Client>> {
    return new Promise<Maybe<Client>>>((resolve, reject) => {
      let clients: Array<Client> = [
        new Client(1, "jim"),
        new Client(2, "john")];

      var maybeClient: Maybe<Client> = F.FirstOrNone(clients, c => c.Id == id);
      resolve(maybeClient)
    });
  }
}

new MockClientRepository()
  .GetById(5)
  .map(x => x.matchWith({
    none: () => "no client found",
    some: client => client.Name
  }))
  .then(x => {
    console.log(`client Name: ${x}`)
  })

```

We now return a `Maybe<Client>`

Run This: [TS Fiddle](#)

Libraries

Maybe

purify: <https://gigobyte.github.io/purify/adts/Maybe>

fp-ts: <https://gcanti.github.io/fp-ts/modules/Option.ts.html>

### 3.9 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).

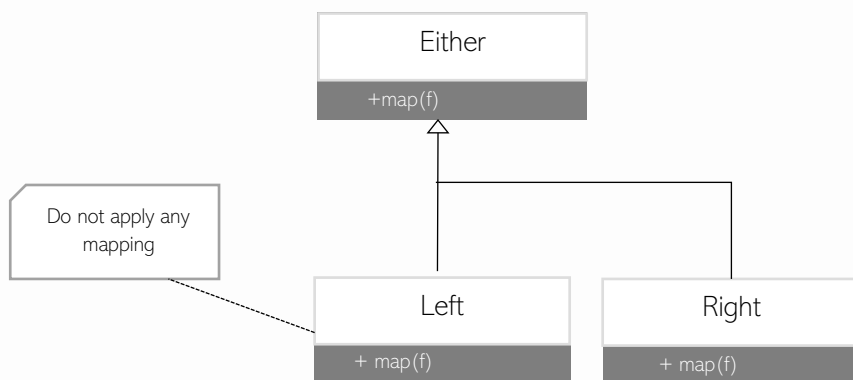
```
abstract class Either<TL, TR>
{
  abstract Map<TR1>(f: (v: TR) => TR1): Either<TL, TR1>;
}
```

```
class Right<TL, TR> extends Either<TL, TR>{
  Value: TR
  constructor(value: TR) {
    super();
    this.Value = value;
  }
  Map<TR1>(f: (v: TR) => TR1): Either<TL, TR1> {
    return new Right<TL, TR1>(f(this.Value));
  }
}
```

```
class Left<TL, TR> extends Either<TL, TR>{
  Value: TL
  constructor(value: TL) {
    super();
    this.Value = value;
  }
  Map<TR1>(f: (v: TR) => TR1): Either<TL, TR1> {
    return new Left<TL, TR1>(this.Value);
  }
}
```

Run This: TS 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(Value)`. It preserves the value that it holds. After we reach a `Left` **all the subsequent transformation through the `map(f)` are ignored**



The `matchWith` implementation for `Either` is similar to `Maybe`:

```

abstract class Either<TL, TR>
{
  abstract MatchWith<TR1>(pattern: ({ Left: (v: TL) => TR1, Right: (v: TR) => TR1 })): TR1;
}

class Right<TL, TR> extends Either<TL, TR>{
  MatchWith<TR1>(pattern: ({ Left: (v: TL) => TR1, Right: (v: TR) => TR1 })): TR1 {
    return pattern.Right(this.Value);
  }
}

class Left<TL, TR> extends Either<TL, TR>{
  MatchWith<TR1>(pattern: ({ Left: (v: TL) => TR1, Right: (v: TR) => TR1 })): TR1 {
    return pattern.Left(this.Value);
  }
}
  
```

Run This: [TS Fiddle](#)

And we can use it like this to represent different paths of computation

```

var d = new left<string, number>("invalid operation")
  .map(x => x + 1)
  .matchWith({
    left: v => "error :" + v,
    right: v => v.toString()
  });

var d = new right<string, number>(4)
  .map(x => x + 1)
  .matchWith({
    left: v => "error :" + v,
    right: v => v.toString()
  })
  
```

Run This: [TS Fiddle](#)

### 3.9.1 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: number;

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

we could extract the outline of the try catch and inject the two segments of try... and catch, with delegates [this is a functional flavored version of the **template method design pattern**]. So, we can rewrite it like bellow, in a more general way.

```
class Try<T>
{
  Result: T | null
  constructor(fn: () => T, exceptionHandler: (error: any) => void) {
    try {
      this.Result = fn();
    }
    catch (e) {
      exceptionHandler(e);
    }
  }
}
```

and use it like that :

```
var try1 =
  new Try<number>(() => {
    var discount = 0.1;
    finalPrice = 10 - discount * 10;
    return finalPrice = 10 - discount * 10;
  }, e => console.log(e));
```

Run This: [TS Fiddle](#)

Finally, we return an Either:

```
class Try<T>
{
  Result: Either<T, any>
```

```

    constructor(fn: () => T) {
      try {
        this.Result = new Right(fn());
      }
      catch (e) {
        this.Result = new Left(e);
      }
    }
  }
}

```

Run This: [TS Fiddle](#)

And we can either use it by exposing the result

```

var try =
  new Try<number>(() => {
    var discount = 0.1;
    finalPrice = 10 - discount * 10;
    return finalPrice = 10 - discount * 10;
  });

var result = try.Result.MatchWith({
  Left: e => "error",
  Right: v => "result : " + v
})

```

Run This: [TS Fiddle](#)

Or we could have one more Functor named Try Functor by just hiding the result and exposing a Map and a MatchWith. Here my quick implementation:

```

class Try<T>
{
  private Result: Either<any, T>
  constructor(fn: () => T) {
    try {
      this.Result = new Right(fn());
    }
    catch (e) {
      this.Result = new Left(e);
    }
  }

  map<T1>(f: (v: T) => T1): Either<any, T1> {
    return this.Result.matchWith({
      Left: e => new Left<any, T1>(e),
      Right: v => new Right<any, T1>(f(v))
    })
  }

  matchWith<T1>(pattern: ({ Left: (v: any) => T1, Right: (v: T) => T1 })): T1 {
    return this.Result.matchWith(pattern);
  }
}

```

```

var try =
  new Try<number>(() => {
    var discount = 0.1;
    var finalPrice = 10 - discount * 10;
    return finalPrice = 10 - discount * 10;
  });

var result = try.matchWith({
  Left: e => "error",
  Right: v => "result :" + v
})

```

We could also postpone execution instead of running the `fn:()=>T` immediately `new Right(fn())`.

### 3.9.2 Either and the Task<T> functor

A `Promise<T>` 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 <T1>(f: (v: any) => T1) {
  return new Promise<T1>((resolve, reject) => {
    this.then(x => resolve(f(x))).catch(reject);
  });
};

```

Same as `Right` - applies f

Same as `Left` - ignores f

We could further replace the `.then(...).catch(...)` pattern and replace it with an Either like pattern `.MatchWith({`

```

  ok: x => { ... },
  error: x => { ... }
})

```

We can define it directly by capturing the resolved or rejected values and pass them into the different matching pattern call-backs, take a look at the following implementation:

```

Promise.prototype.matchWith =
function (pattern: ({ ok: (v: any) => void, error: (v: any) => void })) {
  this.then(pattern.ok).catch(pattern.error);
};

```

The constructor of the `Promise<T>` in the Microsoft's typescript ECMAScript library

```

/**
 * Creates a new Promise.
 * @param executor A callback used to initialize the promise. This callback is passed t
wo arguments:

```

```

    * a resolve callback used to resolve the promise with a value or the result of another
    promise,
    * and a reject callback used to reject the promise with a provided reason or error.
    */
new <T>(executor: (resolve: (value?: T | PromiseLike<T>) => void, reject: (reason?: any) => void) => void): Promise<T>;

```

what the constructor takes the usual executor `(resolve, reject)=>{ ... }` that contains two callbacks in effect. The `(resolve: (value?: T | PromiseLike<T>) => void` which I will simplify for this section `resolve: (value: T) => void` this is a common call-back that takes a `value` of type `T`

```

type PromiseCallbackActions<T> = (resolve: (value: T) => void, reject: (reason?: any) => void) => void;

```

here we are create a new Type to wrap those callbacks, that will allow us to extend the Promise behavior.

```

class EitherAsync<T, T1>
{
    private executor: PromiseCallbackActions<T>

    constructor(executor: PromiseCallbackActions<T>) {
        this.executor = executor;
    }

    matchWith(pattern: ({ ok: (v: T1) => void, error: (v: any) => void })): void {
        this.executor(x => pattern.ok(x), x => pattern.error);
    }
}

```

Run This: [TS Fiddle](#)

We can straight away provide a `matchWith` extension in order to normalize the syntax of the Promise in order to conform to an Either-like pattern. This will allow us to use the same syntax for `Promise<T>` and `Either<T>`, instead of the using external try/catch blocks

```

var eitherAsync= new EitherAsync((resolve ,reject )=>{resolve(5)});

eitherAsync.matchWith({
    ok: x => {
        console.log("ok" + x)
    },
    error: x => {
        console.log("error" + x)
    }
})

```

Run This: [TS Fiddle](#)

This implementation looks so trivial, yet it actually allows us to write Either-like style syntax



Here I rewrote the example of the previous section, but I replaced the promise with EitherAsync transformation:

```
new MockClientRepository()
  .GetById(2)
  .map(x => x.matchWith({
    none: () => "no client found",
    some: client => client.Name
  }))
  .toEither()
  .matchWith({
    ok: x => {
      console.log("client name: " + x)
    },
    error: x => {
      console.log("error" + x)
    }
  })
})
```

Run This: [TS Fiddle](#)

That's all about Either and Promise for now.

Libraries

EitherAsync

### **Purify : EitherAsync**

### **Fp-ts : TaskEither**

"TaskEither<E, A> represents an asynchronous computation that either yields a value of type A or fails yielding an error of type E"

## 3.10 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](#). 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:

$$\text{Tree}(a) = \text{leaf}(a) + \text{Node}(\text{Tree}(a), \text{Tree}(a))$$

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. )

```
abstract class Tree<T> { }

class Node1<T> extends Tree<T> {
  Left: Tree<T>;
  Value: T;
  Right: Tree<T>;
  constructor(left: Tree<T>, value: T, right: Tree<T>) {
    super();
    this.Left = left;
    this.Value = value;
    this.Right = right;
  }
}

class Leaf<T> extends Tree<T> {
  Value: T;
  constructor(value: T) {
    super();
    this.Value = value;
  }
}
```

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.

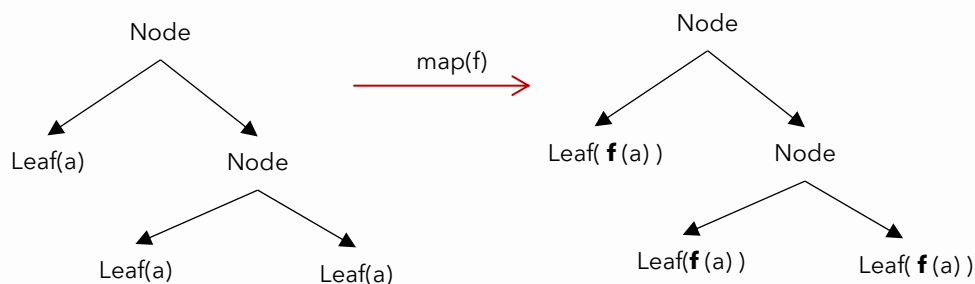
1. We start with the `Leaf` which is the base case of the structural induction. We must return something of the same type. That's why we return a new `Leaf` with the function applied to the value

```
Tree<T1> Map<T1>(Func<T, T1> f) => new Leaf<T1>(f(V))
```

**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),

```
Tree<T1> Map<T1>(Func<T, T1> f) => new Node<T1>(Left.Map(f), Right.Map(f))
```



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

```
abstract class Tree<T>
{
  abstract MatchWith<T1>(pattern: ({ leaf: (v: T) => T1, node: (left: Tree<T>,
    right: Tree<T>) => T1 })): T1;

  map<T1>(f: (v: T) => T1): Tree<T1> {
    return this.MatchWith({
      leaf: (v) => new Leaf(f(v)),
      node: (l, r) => new Node(r.map(f), f(v), l.map(f))
    });
  }
}
```

Run This: [TS Fiddle](#)

## 3.11 Applicative Functors

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  $\text{map} : (a \rightarrow b) \rightarrow F(a) \rightarrow F(b)$  the only difference is the initial F

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

```
export class Id<T>
{
    Value: T

    constructor(value: T) { this.Value = value; }

    public static of<T>(v: T) { return new Id<T>(v); }

    map<T1>(f: (y: T) => T1): Id<T1> { return new Id<T1>(f(this.Value)); }

    apply<T1>(applicative: Id<(v: T) => T1>): Id<T1> {
        return applicative.map(f => f(this.Value)); }

    public static ap<T, T1>(applicative: Id<(v: T) => T1>, fa: Id<T>) {
        return applicative.map(f => f(fa.Value)); }
}
```

run this: [fiddle](#)

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

```
var addition: (v: number) => (u: number) => number = (v) => u => u + v;

var idAddition: Id<(v: number) => (u: number) => number> = new Id<(v: number) => (u: number)
=> number>(addition);
//Id(u=>v=>u+v)

var idAdd3: Id<(v: number) => number> = Id.ap(idAddition, new Id<number>(3))
//Id(env=>(v) =>v+3)

var idAdd3And5: Id<number> = Id.ap(idAdd3, new Id<number>(5))
//Id(      5+3)

var result = idAdd3And5.Value;
//8
```

run this [fiddle](#)

unfortunately typescript is making it difficult to define the ap on the Id in order to be able to chain in this direction `Id(y=>x=>...).ap(Id(1))` so we are going to use the static method on Id `ap<T, T1>(applicative: Id<(v: T) => T1>, fa: Id<T>)`. Equivalently we may use the apply function where we have to chain with the Id(v) before the `Id(y=>x=>...)` `Id(1).apply(Id(y=>x=>...))`

**So we will use any of those:**

```
Id.ap(Id.ap(new Id(u => u + v), new Id(3)) , new Id(3))

new Id(3).apply(new Id(3).apply(new Id(u => u + v)))
```

! 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 => 1+y)`, and since we still have a function inside a functor, we can use `Ap` again `Id ( y => 1+y).ap(Id(1))` which eventually gives us `Id(2)`.

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.

## 3.12 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. This is my implementation:

```
export class Reader<Env, T>
{
  apply<T1>(applicative: Reader<Env, (v: T) => T1>): Reader<Env, T1> {
    return new Reader<Env, T1>((env) => applicative.run(env)(this.Fn(env)));
  };

}

var readerAddFirst = new Reader<[number, number], (v: number) => number>((env: [number, number]) => (v: number) => env[0] + v); //Reader(env=>(u,v)=>u+v)

var readerAddFirstAnd5 = new Reader<[number, number], number>((env: [number, number]) => 5).apply(readerAddFirst) //Reader(env=>5+3)

var result = readerAddFirstAnd5.run([4, 5]); //9
```

Run This: [TS Fiddle](#)

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

if we take the applicative operation

$Ap: f(a \rightarrow b) \rightarrow f(a) \rightarrow f(b)$  (this is the name for  $\langle \otimes \dots \rangle$ )

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

$\langle \otimes \rangle: ((\rightarrow) \text{ env}) (a \rightarrow b) \rightarrow ((\rightarrow) \text{ env}) a \rightarrow ((\rightarrow) \text{ env}) b$

With some reductions we get the form

$(\text{env} \rightarrow (a \rightarrow b)) \rightarrow (\text{env} \rightarrow a) \rightarrow (\text{env} \rightarrow b)$

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 addition: (v: number) => (u: number) => number = (v) => u => u + v;

var readerAdd = new Reader<any, (v: number) => (u: number) => number>((env: any) => addition); //Reader(env=>(u,v)=>u+v)
var readerAdd3 = new Reader<any, number>((env: any) => 3).apply(readerAdd) //Reader(env=>(v) =>v+3)
var readerAdd3And5 = new Reader<any, number>((env: any) => 5).apply(readerAdd3) //Reader(env=>5+3)
```

```
var result = readerAdd3And5.run(null); //8
```

Run This: [TS Fiddle](#)

## Category Theory

## Monoidal Functor

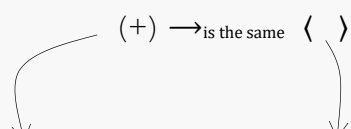
Let  $(C, \otimes_C, 1_C)$  and  $(D, \otimes_D, 1_D)$  be two monoidal categories. A lax monoidal functor between them is

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  $\otimes_{\text{Reader}}$ , applicative operation  $\text{ap} \langle \rangle$  of the reader :

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


```
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"  $\otimes_{\text{Array}} \text{concat} (:)$  will be lifted to the now the same as the  $\otimes_{\text{Reader}}$ , applicative operation  $\text{ap} \langle \rangle$  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)

# Traversing Functors

## ! 4 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 *hylē* "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.

### 4.1 A brief mentioning of F-algebras

If  $F$  is a Functor, then an **F-algebra** is a pair  $(A, \alpha)$ , where  $A$  is called carrier type and  $\alpha$  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 one algebra with carrier type the integers would be this

```
public abstract class Numeral { }

public class Succ : Numeral
{
    public Numeral Rest { get; set; }
    public Succ(Numeral rest)
    {
        this.Rest = rest;
    }
}

public class Zero : Numeral { }
```

```
var algebraInt: ({ Zero: () => number, Succ: (rest: number) => number }) = ({
    Zero: () => 0,
    Succ: (r) => r + 1
});
```

← That's called the **structure map** but when we say algebra, we will mean this structure



This algebra tells us how to *interpret* a `Natural` 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<sup>1</sup>. Nevertheless, we can have many algebras based on this Functor. Here some more:

The algebra below needs no further explanation

```
var algebraString: ({ Zero: () => string, Succ: (rest: string) => string }) = ({
  Zero: () => "zero",
  Succ: (r) => `AddOne(${r})`
});
```

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  $(A, \alpha)$ , where  $A$  is an object of  $C$  and  $\alpha$  is a morphism  $F(A) \rightarrow A$  the  $\alpha$  often called *structure map*.

## 4.2 Catamorphisms

Now we are going to define a `cata` method for the `Natural` union type, which will take an algebra and evaluate the `Natural`. Here the implementation will be ad hoc-specifically for the `Natural` structure.

```
! 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](#)

<sup>1</sup> [Lectures on Semantics: The initial algebra and final coalgebra perspectives](#)

**[SideNote:** We have seen Recursive definitions before using `MatchWith`.

For the first time we are creating a method that takes as an argument `(Func<T1> Zero, Func<T1, T1> Succ)` `algebra` 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 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: [TS 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 `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 patten matching algebras. For example, this is a different mapping from `Numeral  $\rightarrow$  string`

```
var justExclamationmarks: ({ Zero: () => string, Succ: (rest: string) => string }) = ({
  Zero: () => "",
  Succ: (r) => `!(${r})`
});
```

Run This: [TS Fiddle](#)

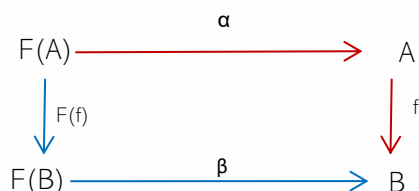
## 4.3 Optional Initial algebra

This section is optional but **highly recommended**. You might want to revisit the whole 5<sup>th</sup> section even if you don't feel yet that everything "connects".

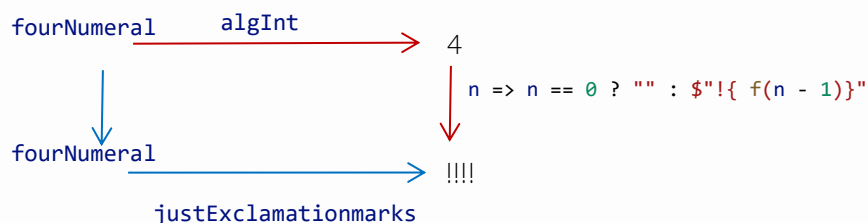
### 4.3.1 F-Algebras Homomorphisms

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  $(A, \alpha)$  to an F-algebra  $(B, \beta)$  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



```
justExclamationmarks = (Zero: () => $"", Succ: (Rest) => $"!{Rest }" );
var algInt = (Zero: () => 0, Succ: (Rest) => Rest + 1 );
```

Run This: TS 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 `f = n => n == 0 ? "" : $"!{ f(n - 1)}"` to go from 4 to `!!!!`

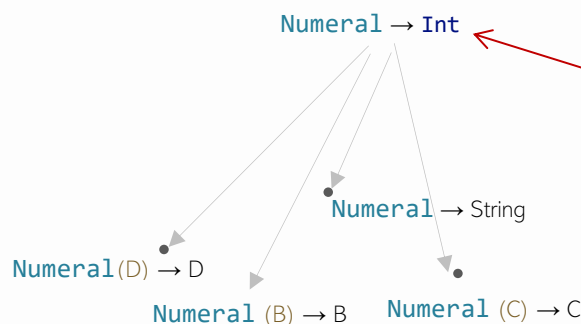
The truth is that we can go from the 4 to any other possible interpretation of any functor with type `1+A` for any A because the integers and the `algInt` is an initial algebra for our `Numeral` functor. Initial algebra means that for any other algebra there is an arrow (which represents a Homomorphism in the category of F-algebras) to that object from the Initial algebra. In Essence this proves that we **can represent Iteration** with this

```
Func<int, string> recursion = null;
```

```
recursion = n => n == 0 ? algebra.E() : `${algebra.F(recursion(n - 1))}`;
```

```
for any algebra : (Func<string> E, Func<string, string> F) algebra = (E: () => ``,  
F: (Rest) => `${Rest}`);
```

Run This: TS Fiddle



The natural numbers together with the tuple (0, n+1) form the **initial object in the category of F-Algebras** for the `Numeral` Functor

The point of all this is **that the initial algebra of the Peano functor `Numeral` 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 (`Numeral` , `Succ` , `Zero`) because it is “Equivalent” to the natural numbers (N,S,0) is what is called a **natural number system** (in TS Type category in our case), and `Numeral` [a natural number object](#).

## 4.4 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 = (  
  Leaf: (v) => v,  
  Node: (l, v, r) => 1 + v + r  
)
```

Our Cata method would look like this:

```
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](#)

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.

```

var tree = new Node<int>(new Node<int>(new Leaf<int>(1), 2, new Leaf<int>(3)),
  4,
  new Node<int>(new Leaf<int>(5), 6, new Leaf<int>(7)));

//(Func<T, T> Leaf, Func<T, T, T, T> Node) algebra
Console.WriteLine(
  tree.Cata(algebra: (
    Leaf: (v) => v,
    Node: (l, v, r) => 1 + v + r))
);

```

Run This: [TS Fiddle](#)

### Optional

This is almost identical with the previous definitions in order to highlight the fact that this is an automated process for simple algebraic structures.

Again, because it is a coproduct, we must define the `cata` method in both subtypes `Leaf` and `Node`.

1. We start with the `Leaf`, because this is the base case of the structural induction. We can apply the Algebra directly to this leaf since there is nothing else to do.

```
Leaf: v => algebra.leaf(v),
```

2. The `Node` is a product of two trees (`left`, `v`, `right`) and a value. The only way to deal with it is to assume that those must have a `cata<T>` function that works as expected (that is is the inductive hypothesis again) if we pass the algebra `algebra` on those nodes the result must be an integer in this way we will have three integers -one from the left node : `l.cata<T>(algebra)` [this returns a T type] the node `v`, and one from the right node: `r.cata<T>(algebra)` [this also returns a T type], and we can use the `algebra` to call the `algebra.Node(____)` In order to evaluate all those three results that are of the same type T now

```
Node: (l, v, r) => algebra.Node(r.cata<T>(algebra), v, l.cata<T>(algebra))
```

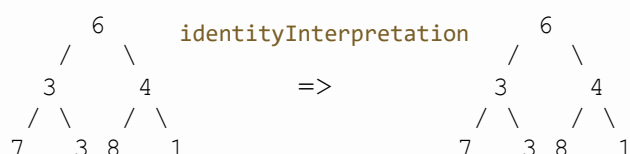
Run This: [TS Fiddle](#)

## 4.5 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: <T>() => TreePattern<T, Tree<T>> = () => ({
  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(): Tree<T> {
  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: <T>() => TreePattern<T, Tree<T>> = () => ({
  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<number> = tree.Cata(reverseInterpretation());
```

```
console.log(reverse.Show());
```

Run This: [TS Fiddle](#)

## 4.6 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.

```
interface Visitor<T, T1> {
    visitLeaf(leaf: Leaf<T>): T1;
    visitNode(node: Node1<T>): T1;
}
```

And we implement the standard accept method in each component

```
abstract class Tree<T>
{
    abstract accept<T1>(visitor: Visitor<T, T1>): T1
}
class Node1<T> extends Tree<T> {
    accept<T1>(visitor: Visitor<T, T1>): T1 {
        return visitor.visitNode(this);
    }
}
class Leaf<T> extends Tree<T> {
    accept<T1>(visitor: Visitor<T, T1>): T1 {
        return visitor.visitLeaf(this);
    }
}
```

Run This: [TS Fiddle](#)

As you can see the Visitor is pretty much isomorphic to the catamorphism implementation. Visitors use the **double dispatch** to dispatches 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

```
interface Visitor<T, T1> {
    visitLeaf(leaf: Leaf<T>): T1;
    visitNode(node: Node1<T>): T1;
}
```

```

abstract MatchWith<T1>(pattern: (
{
    leaf: (v: T) => T1,
    node: (left: Tree<T>, v: T, right: Tree<T>) => T1
})): T1;

```

## 4.6.1 The Base Functor of the List<T>

I was not sure if I wanted to include this section because it only adds unnecessary complexity and probably will leave you more confused because of the TS Types system. This section is partially about Fix points and Base Functors.

Let me introduce you to another Form of the List. the algebraic definition of a list is this

$$\text{List}(a) = [] + a * \text{List}(a)$$

Since we already have the `Maybe` as a Coproduct types that has a subtype `None` (that can represent the concept of Nothing or empty or a Terminal object) and a subtype `Some` that can have a Value we might use this to represent this above definition.

$$\text{List}(a) = \text{None}() + \text{Some}(a * \text{List}(a))$$

This definition is isomorphic to the definition of a list. Let's see another way to prove this. Let's say that we have a class for product

```

class ProductF<T, E>
{
    Value: T
    Rest: E
    constructor(value: T, rest: E) {
        this.Value = value;
        this.Rest = rest;
    }
}

```

If I take the E parameter to be a `Maybe<Product<T>>` this becomes

```

class Product<T> {
    Value: T
    Rest: Maybe<Product<T>>

    constructor(value: T, rest: Maybe<Product<T>>) {
        this.Rest = rest;
        this.Value = value;
    }
}

```

Let's say that we try to write what a `Maybe<Product<T>>` looks like (in a kind of symbolic representation)



```
Maybe<Product<T>> = None<Product<T>> + Some<Product<T>>({ Value=a:T, Rest:
Maybe<Product<T>>})
```

And if we replace `Maybe<Product<T>>` with the name `List<T>` then we get:

```
List<T> = None<Product<T>> + Some<Product<T>>({ Value=a:T, Rest: List<T>}).
```

Which has the correct meaning of a List.

Let me try to explain the same from another perspective. Let's now start from this class

```
class SomeF<T, E>
{
  Value: ProductF<T, E>;
  constructor(value: ProductF<T, E>) {
    this.Value = value;
  }
}
```

Run This: [TS Fiddle](#)

Which looks like the `SomeF<T, E>` where the value is of type `Product<T, E>`

Let's write some instances of this class

In the first one `T` is `number` and `E` is `number`

```
var s: SomeF<number, number> = new SomeF<number, number>(new ProductF<number, number>(1, 1)
);
```

Now let's take `E` as `SomeF< number, number >` This is valid:

```
var s1: SomeF<number, SomeF<number, number>> =
  new SomeF<number, SomeF<number, number>>(new ProductF<number, SomeF<number, number>>
>(2, s));
```

Now if we take `E` as `SomeF< number, SomeF< number, number >>` we get

```
var s2: SomeF<number, SomeF<number, SomeF<number, number>>> =
  new SomeF<number, SomeF<number, SomeF<number, number>>>(
    new ProductF<number, SomeF<number, SomeF<number, number>>>(3, s1));
```

Run This: [TS Fiddle](#)

the thing is we suspect that `s1` should be a Subtype of `s2`, `s1<: s2` which is in the Behavioral sense if for example we were writing in TS it would be safe to replace `s1` with `s2`.

We suspect that if we were to repeat this replacement of `E` with

`SomeF<int, SomeF<int, SomeF<int, ...>>>` this should converge somewhere. Let's call this infinite expression as `X` then it would not make any difference if we iterate one more time (like adding `+1` at infinite is still infinite). This is a usual tactic when reasoning with infinite

structures) so `SomeF<int, x> == x` we say that this X type would be a Fix Type for this Type `SomeF<int, x>` the only thing we can do in TS is write this equivalence this way

```
class SomeFixed<T, E extends SomeFixed<T, E>>
{
  Value: ProductF<T, E>;
  constructor(value: ProductF<T, E>) {
    this.Value = value;
  }
}
```

and finally, completely replace E like this

```
class SomeFix<T>
{
  Value: ProductF<T, SomeFix<T>>;
  constructor(value: ProductF<T, SomeFix<T>>) {
    this.Value = value;
  }
}
```

Run This: [TS Fiddle](#)

The point of all this is that `Maybe<Product<T>>` is equivalent to a `List<T>`

side note

Base Functor and Fix points

In the list definition

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

We can get rid of recursion by replacing the `List(a)` in the right side with a new type `E`

$ListF(a, E) = [] + a * E$

**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.

Imagine this as a type `F(_)` where  $F(_) = ListF(a, _) = [] + a * _$

If we replace `E` with any Type we get a `ListF(a, E)` Type for example we can get `E` as `int` then we get the Type `ListF(a, int) = [] + a * b` where `b` is `int`. We can take as `E` whatever we want but for this example there is one Type `X` that if `E` is of that type then something special happens :

**ListF(a, X) = X (!)**

We want to find this specific `X` because if we replace it in the definition, we get:

$ListF(a, X) = [] + a * X \leftarrow [] + a * ListF(a, X)$  Replace from (!)

And now if we replace `ListF (a, X)` with a name like `list(a)` for example we get

**`list(a) = [] + a * list(a)`**

well we have the list definition. We started from a non-recursive definition `ListF (a, E) = [] + a * E` and we found an `E` that gives us the recursive type that we wanted.

This `X` is called A Fix point of `F(⋅)`. A fix point must satisfy an equation like this:

$F(X) = X$

Fix points arise everywhere and are a thing of beauty. We are not going to get in any more detail here. But in the **TS type systems the Recursive Types are Fix points of other Higher order types that we never see**. *The ability of a language to have Recursive types hides all the complexity we just discussed.* We use Fix points all the time that's why we cannot even see them.

Finally, **the `List<T>` is a Fix point of the Functor `Maybe<Product<T, E>>`**

Lets now use our Base list definition as `Maybe<Product<T>>` where

```
class Product<T> {
  Value: T
  Rest: Maybe<Product<T>>

  constructor(value: T, rest: Maybe<Product<T>>) {
    this.Rest = rest;
    this.Value = value;
  }
}
```

We can define our usual List Methods on this structure

```
function Multiply(list: Maybe<Product<number>>): number {
  return list.MatchWith(({
    none: () => 1,
    some: (product: Product<number>) => product.Value * Multiply(product.Rest)
  })))
}
```

or just generalize to a fold

```
function Fold<T>(list: Maybe<Product<T>>, monoid: monoid<T>): T {
  return list.MatchWith(({
    none: () => monoid.empty,
    some: (product: Product<T>) => monoid.concat(product.Value, Fold(product.Rest, monoid))
  })))
}
```

```
}
```

```
var list = new Some<Product<number>>(new Product<number>(5, new Some<Product<number>>(
new Product<number>(5, new None()))));
```

```
var product = Fold(list, { empty: 1, concat: (x, y) => x * y })
```

Run This: [TS Fiddle](#)

One could also give an alternative product definition with object literals for example:

```
interface IProduct<T> {
  Value: T
  Rest: Maybe<IProduct<T>>
}

var list: IProduct<number> = ({
  Value: 1, Rest: new Some<IProduct<number>>(
    ({
      Value: 2, Rest: new Some<IProduct<number>>(
        ({
          Value: 3, Rest: new Some<IProduct<number>>(
            ({
              Value: 4,
              Rest: new None<IProduct<number>>()
            })
          })
        })
      })
    })
  )
});
```

## 4.6.2 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.

## 4.6.3 Optional A brief mentioning of Anamorphisms

It's 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 anaToListBase: (n: number) => Maybe<Product<number>> =
  (n: number) => n == 0 ?
    new None<Product<number>>() :
    new Some<Product<number>>(new Product(n, anaToListBase(n - 1)));

var twoUnfoldList = anaToListBase(5);
```

Run This: [TS 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

```
type coAlgebraInt = (n: number) => Maybe<ProductF<number, number>>;

var coAlgListBase: coAlgebraInt = (n: number) =>
  n == 0 ?
  new None<ProductF<number, number>>() :
  new Some(new ProductF<number, number>(n - 1, n - 1));
```

which we can use with an more generic recursive scheme to generate Lists

```
var ana: (coalgebra: coAlgebraInt) => (oldState: number) => Maybe<Product<number>> =
  (coalgebra: coAlgebraInt) => (oldState: number) =>
    coalgebra(oldState).MatchWith({
      none: () => new None<Product<number>>(),
      some: (product: ProductF<number, number>) =>
new Some<Product<number>>(new Product<number>(product.Rest, ana(coalgebra)(product.Value)))
    });

var unfoldListFromCoalg = ana(coAlgListBase)(5);
```

Run This: [TS Fiddle](#)

I know that this way more complex and the only thing that we here is to be able to use different `coAlg` for example using the following we get only the even numbers:

```
var coAlgListBaseEvens : coAlgebraInt = (n: number) =>
  n == 0 ?
  new None<ProductF<number, number>>() :
  new Some(new ProductF<number, number>(n - 1, n - 1));
```

We can also change the `ana` function and generate lists by using instructions made of `Maybe's`

```
var anaArray: (coalgebra: coAlgebraInt) => (oldState: number) => Array<number> =
  (coalgebra: coAlgebraInt) => (oldState: number) =>
    coalgebra(oldState).MatchWith({
      none: () => [],
      some: (product: ProductF<number, number>) =>
        [product.Rest, ...anaArray(coalgebra)(product.Value)]
    });
```

```
var nat = anaArray (coAlgListBaseEvens)(10);//[8,6,4,2,0]
```

Run This: [TS 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 imagine that we could create for example Trees or LinkedList or other data structures with the right unfold algorithm.

## 4.6.4 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: number, r: Maybe<Product<number>>) => Maybe<Product<number>>
= (n, r) => coana(n + 1, new Some(new Product(n, r)))
```

```
var stream = coana(0, new None<Product<number>>());
```

Run This: [TS Fiddle](#)

well enjoy your Maximum call stack size exceeded Exception

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 TS will try to evaluate everything at once. But we can use a `()=>` to make it lazy right ? well please try it for yourself forking the TS Fiddle.

I added a gap to just witness the co-iteration the first repetitions give us this

```
//(Value: 0, Rest:{ })
//(Value: 1, Rest: (Value: 0, Rest:{ }))
//(Value: 2, Rest: (Value: 1, Rest: (Value: 0, Rest:{ })))
//(Value: 3, Rest: (Value: 2, Rest: (Value: 1, Rest: (Value: 0, Rest:{ }))))
```

Run This: [TS Fiddle](#)

the characteristics that make it co-recursive are dual to those of recursion as noted above:

1. There is a base **initial** condition `coana(0, new None<Product<number>>())`
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 an increased** argument. There is no assurance that this process is going to stop eventually. We **construct** the **final** value step by step.

[Some advanced but manageable papers on coinduction can be found on the page of Jan Rutten [here](#) where the popular “An introduction to (co)algebra and (co)induction” resides<sup>2</sup>]

## 4.6.5 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. In our case we can get the factorial:

```
public static int Multiply(this Maybe<Product<int>> @this) =>
@this.MatchWith(pattern: (
    None: () => 1,
    Some: (product) => product.Value * product.Rest.Multiply()
));

public static Maybe<Product<int>> AnaToListBase(this int @n) =>
@n == 0 ? (Maybe<Product<int>>)new None<Product<int>>() :
    new Some<Product<int>>(new Product<int>())
```

<sup>2</sup> [https://homepages.cwi.nl/~janr/papers/files-of-papers/2011\\_Jacobs\\_Rutten\\_new.pdf](https://homepages.cwi.nl/~janr/papers/files-of-papers/2011_Jacobs_Rutten_new.pdf)

```

{
    Value = @n,
    Rest = (@n - 1).AnaToListBase()
});

```

```

Func<int, int> factorial = n => n.AnaToListBase().Multiply();

```

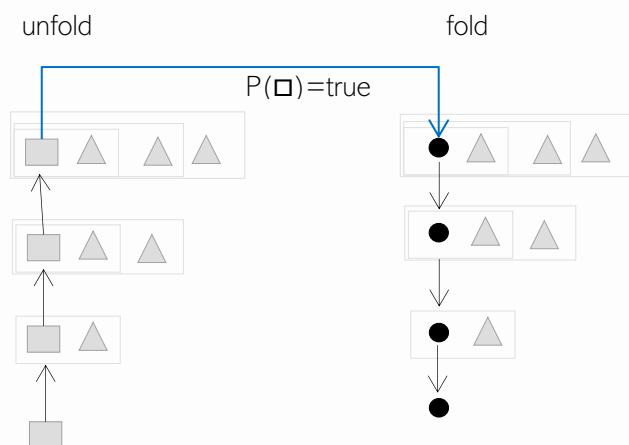
Run This: TS Fiddle

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  $\underline{c}$  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  $g: \square \rightarrow \llbracket \square \triangle \rrbracket$  and an algebra  $h: \llbracket \bullet \triangle \rrbracket \rightarrow \bullet$  the process starts with the  $g$  replacing squares  $\square$  with pairs of  $\llbracket \square \triangle \rrbracket$

$\square \rightarrow \llbracket \square \triangle \rrbracket \rightarrow \llbracket \llbracket \square \triangle \rrbracket \triangle \rrbracket \rightarrow \dots$  at some point when a condition is reached, we replace the most inner square  $\square$  with a  $\bullet$  this starts the collapsing(catamorphism)



## 4.6.6 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 leave to wrap them and Nodes to compose bottom up.

```
function ToTree<T>(array: Array<T>): Tree<T> {
  return array.length == 1 ?
    new Leaf<T>(array[0]) :
    new Node1<T>(ToTree<T>(array.slice(0, array.length / 2)),
      ToTree<T>(array.slice(array.length / 2, array.length)));
}

var array: Array<number> = [1, 2, 3, 4, 5, 6, 7, 8];

var tree = ToTree(array); → (((((1) ,(2)) ,(3) ,(4))) ,((5) ,(6)) ,((7) ,(8))))
```

Run This: [TS 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:

```
interface IComparable<T> {
  compareTo(other: T): number;
}

function Sort<T extends IComparable<T>>(tree: Tree<T>)
```

```

: Array<T> {
  return tree.MatchWith({
    leaf: (v) => [v],
    node: (l, r) => ZipOrdered(Sort(l), Sort(r))
  })
}

```

This says if we are in **Leaf** just return the value **v** which is ordered because its single. Now for the **Node1** we first recursively use sort to get the sorted lists (**Sort(l)**, **Sort(r)**) and then we zip them using **ZipOrdered**.

Now we should define **ZipOrdered** which also has a recursive definition:

```

function ZipOrdered<T extends IComparable<T>>(a1: Array<T>, a2: Array<T>)
: Array<T> {
  return a1.MatchWith({
    empty: () => a2,
    cons: (x, xs) => a2.MatchWith({
      empty: () => xs,
      cons: (y, ys) => x.compareTo(y) > 0 ?
        [x, ...(ZipOrdered(xs, [y, ...ys]))] :
        [y, ...(ZipOrdered(ys, [x, ...xs]))]
    }),
  })
}

```

Run This: [TS 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**.

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.

## 4.7 Fold’s relation to 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 In the same way that **Array**. **reduce** is defined

```

[ 2, 3, 4 ].reduce((accumulate, element) => accumulate + element, 0);

```

the signature of the extension method is pretty standard as you can see from the source code of the `Array<T>` :

```
reduce<U>(callbackfn: (previousValue: U, currentValue: T, currentIndex: number, array: T[])
=> U, initialValue: U): U;
```

The `initialValue: U` and the `callbackfn: (previousValue: U, currentValue: T) => U` are enough to Define a Monoid with the empty being the `initialValue` and the concat is the `(previousValue: U, currentValue: T) => U` (you will notice the different types `T` and `U` well is sufficient for `U` to be a monoid. We will talk more about that in FoldMap section )

By analogy for the `Id` Functor we can define a Fold with the same signature

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

And use it like this, in order to use the value:

```
var result = new Id(3).fold(0, (acc,v)=>acc+v );
```

Run This: [TS Fiddle](#)

Alternatively, if we had a simple Product structure, we can provide a fold like this:

```
var Pair =<T> (a:T, b:T) => ({
  fold<TM>(acc: TM, reducer: (v: T, acc: TM) => TM) { return reducer(b, reducer(a, acc)); }
});
```

```
var r= Pair(2,3).fold(0, (accumulator, currentValue) => accumulator + currentValue) ;
```

Run This: [TS 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){
  return 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)

## 5 Traversable

I expect that you will find the Following section to be a bit difficult. I know this because it took me a while to derive the specific implementations from theory, since there are no similar examples in TS. Non the less in other functional languages (like Haskell and Scala) the concept of Traversables is considered pretty standard. Initially we will try to derive some specialized cases of the `List` being traversed by an `Id` and then generalize over this. After that we are going to reach Traversables from another path: `FoldMap`. Hopefully by looking the same thing from two different perspectives will allow you to recognize the pattern.

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:

```
var concat: (v: Array<number>) => (u: Array<number>) => Array<number> =
    (v) => u => v.concat(u);

var idConcat: Id<(v: Array<number>) => (u: Array<number>) => Array<number>> =
    new Id<(v: Array<number>) => (u: Array<number>) => Array<number>>>(concat);

var idConcatArray12: Id<(u: Array<number>) => Array<number>> = Id.ap(idConcat, new Id<Array<number>>>([1, 2]))

var idConcatArray1234: Id<Array<number>> = Id.ap(idConcatArray12, new Id<Array<number>>>([3, 4]))

var result = idConcatArray1234; //Id([1,2,3,4])
```

Run This: [TS Fiddle](#)

we can do that repeatedly by replacing the last `ap()` with the whole structure since it is of the same type:

```
var distribute =
    new Id<Array<number>>>([1, 2])
    .apply(new Id<Array<number>>>([3, 4])
        .apply(idConcat))
    .apply(new Id<Array<number>>>([5, 6])
        .apply(idConcat))
    .apply(new Id<Array<number>>>([7, 8])
        .apply(idConcat))
```

Run This: [TS Fiddle](#)

we can use recursion to use this in any list:

```
public static Distribute<T>(array: Array<Id<T>>) {
    return array.MatchWith({
        empty: () => new Id<Array<any>>>([]),
```

```

        cons: (v, r) => {
            var a = this.Distribute(r)
            .apply(new Id<(rest: Array<any>) => (v: any) => Array<any>>((u) => (v) => [v, ...u])))
            var b = v.apply(a);
            return b;
        }
    });
}

var arrayOfId: Array<Id<number>>=[new Id<number>(1), new Id<number>(2), new Id<number>(3)];

var idOfArray: Id<Array<number>> = F.Distribute(arrayOfId);

```

Run This: [TS 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)]

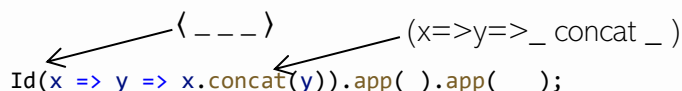
```

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


  
 $\langle \_ \_ \rangle$  (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](#) :: Applicative f => t (f a) → f (t a), and what does is to take a **traversable t that contains f(a) items and evaluates them and collect the results.**

Why get into all this trouble just to get an `Id` (`[1, 2, 3]`) ? Well, we can abstract the `Id` and have a function  $a \rightarrow \text{Id}(b)$  that will do the generation of the `Id` (`[_]`). We get this implementation:

```
Array.prototype.Traverse = function <T1>(f: (v: any) => Id<T1>) {
    return this.MatchWith({
        empty: () => new Id<Array<any>>([]),
        cons: (v, r) => {
            var a =
                r.Traverse(f)
                .apply(new Id<(rest: Array<any>) => (v: any) => Array<any>>((u:
Array<any>) => (v: any) => [v, ...u])))
            var b = f(v).apply(a);
            return b;
        }
    });
}

var array: Array<number> = [1, 2, 3];

var traversed: Id<Array<number>> = array.Traverse<number>(x => new Id<number>(x + 1)
);
```

Run This: [TS Fiddle](#)

Here the `f` is the applicative functor constructor, aka `pure` (`_`) in the fiddle 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  $(a \rightarrow f(b))$  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  $g : a \rightarrow f(b)$  :

```
var TraverseEquivalent= (list,g)=>list.map(g).distribute();
```

Run This: [TS Fiddle](#)

## 5.1 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.

```
function Distribute<T>(tree: Tree<Id<T>>): Id<Tree<T>> {
  return tree.matchWith({
    leaf: (v) => Id.ap(Id.of(x => new Leaf<T>(x)), v),
    node: (l, v, r) => {
      return Id.ap(Id.ap(Id.ap(
        Id.of((left: Tree<T>) => (v: T) => (right: Tree<T>) => new Node1(left,
v, right))),
        Distribute(l)),
        v),
        Distribute(r)
      )
    }
  });
}
```

Run This: [TS Fiddle](#)

Unfortunately, the need to write the Types explicitly makes TS code unreadable pretty fast.

If we strip down the types, we get this:

```
function Distribute<T>(tree: Tree<Id<T>>): Id<Tree<T>> {
  return tree.matchWith({
    leaf: (v) => Id.ap(Id.of(x => new Leaf<T>(x)), v),
    node: (l, v, r) => {
      return
        Id.of(l => v => r => new Node1(l, v, r))
          .ap(Distribute(l))
          .ap(v)
          .ap(Distribute(r))
    }
  });
}
```

Annotations in the original image:

- An arrow points from `<(leaf) _>` to the `leaf` case in the `matchWith` function.
- An arrow points from `<(node) _ _>` to the `node` case in the `matchWith` function.

Run This: [TS 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  $\langle (node) \_ \_ \_ \rangle [Id(x \Rightarrow y \Rightarrow new Node<int>(x, y))]$  where the node constructor `new Node<int>(x, y)` 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.

Run This: [TS Fiddle](#)

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(1.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”  $\otimes_{Array} concat (:)$  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(...)) ;
```

## 5.2 Traversing with The Task<T> aka 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 :

```
public static ap<T, T1>(_this: Promise<(v: T) => T1>, p1: Promise<T>): Promise<T1> {
    return new Promise<T1>((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



```

public static Distribute<T>(array: Array<Promise<T>>) {
    return array.MatchWith({
        empty: () => new Promise<Array<T>>>(r => r([])),
        cons: (v, r) =>
            P.ap(P.ap(P.of<ListPattern<T>>>((v: T) => (u: Array<T>) => [v, ...u]), v),
                this.Distribute(r))
    });
}

var PromiseOfArray: Promise<Array<number>> = P.Distribute(arrayOfPromises);

PromiseOfArray.MatchWith({
    ok: x => {
        console.log(`result: ${x}`)
    },
    error: x => {
        console.log(`error: ${x}`)
    },
})

```

Run This: [TS 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:

```

public static Traverse<T, T1>(array: Array<T>, f: (v: T) => Promise<T>): Promise<Array<T1>>{
    return array.MatchWith({
        empty: () => P.of([]),
        cons: (v, r) => P.ap(P.ap(P.of<ListPattern<T>>>((v: T) => (u: Array<T>) => [v, .
        ..u]), f(v)), this.Traverse(r,f))
    });
}

var array: Array<number> = [1, 2, 3, 4];

var traverseResult: Promise<Array<number>> = P.Traverse(array, x => P.of(x + 1));
traverseResult.MatchWith({
    ok: x => {
        console.log(`result: ${x}`)
    },
    error: x => {
        console.log(`error: ${x}`)
    },
})

```

Run This: [TS 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 userUris: 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.text())
```

I can get in parallel for multiple users the details by using Traverse

```
var gitUserDetailsPromise: Promise<Array<string>> =
  P.Traverse(userUris, uri => fetch(uri).then(res => res.text()));

gitUserDetailsPromise.MatchWith({
  ok: x => {
    console.log(`result: ${x}`)
  },
  error: x => {
    console.log(`error: ${x}`)
  },
})
```

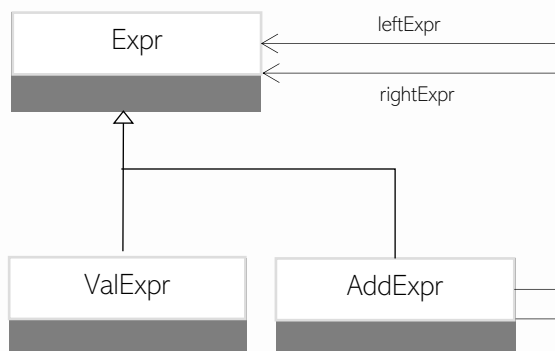
Run This: [TS Fiddle](#)

## 5.3 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)
```



This forms again a recursive algebraic data structure.

```
abstract class Expr<T> {}

class ValExpr<T> extends Expr<T> {
  Value: T;
  constructor(value: T) {
    super();
    this.Value = value;
  }
}

class AddExpr<T> extends Expr<T> {
  X: Expr<T>;
  Y: Expr<T>;
  constructor(x: Expr<T>, y: Expr<T>) {
    super();
    this.X = x;
    this.Y = y;
  }
}
```

We want to create a function eval that will evaluate an expression Exp formed by combinations of (Val, Add, Sub). The type of eval whould 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. But you can imagine that we can pass around a dictionary with values for our variables.

## 5.3.1 A Catamorphism implementation

We have seen many Cata implementations, so hopefully the following is not going to surprise you:

```
type ExprAlgebra<T, T1> = ({ value: (v: T) => T1, add: (x: T1, y: T1) => T1 });
```

```
abstract class Expr<T>
{
  abstract MatchWith<T1>(pattern: ExprPattern<T, T1>): T1;
```

```
  Cata<T1>(algebra: ExprAlgebra<T, T1>): T1 {
    return this.MatchWith({
      value: (v) => algebra.value(v),
      add: (x, y) => algebra.add(x.Cata(algebra), y.Cata(algebra))
    });
  }
```

```

}

class ValExpr<T> extends Expr<T> {

  MatchWith<T1>(pattern: { value: (v: T) => T1; add: (x: Expr<T>, y: Expr<T>) => T1; }): T1 {
    return pattern.value(this.Value);
  }
  Value: T;
  constructor(value: T) {
    super();
    this.Value = value;
  }
}

class AddExpr<T> extends Expr<T> {
  X: Expr<T>;
  Y: Expr<T>;
  constructor(x: Expr<T>, y: Expr<T>) {
    super();
    this.X = x;
    this.Y = y;
  }
  MatchWith<T1>(pattern: { value: (v: T) => T1; add: (x: Expr<T>, y: Expr<T>) => T1; }): T1 {
    return pattern.add(this.X, this.Y);
  }
}

```

Having an expression like the following:

```
var expression = new AddExpr(new ValExpr(5), new AddExpr(new ValExpr(1), new ValExpr(2)));
```

we can interpret it in many ways as we saw using the `Cata` method, of course there is the intended interpretation as addition:

```
var addition = expression.Cata<number>({ value: v => v, add: (x, y) => x + y })
```

but as usually we can interpret it in non-arbitrary, for example nothing preventing us to interpret `AddExpr` as Multiplication, or any binary operation in integers:

```
var multiplication = expression.Cata<number>({ value: v => v, add: (x, y) => x * y })
```

or we could interpret an expression with domain in strings as an Html generator

```

var htmlExpression = new AddExpr(new ValExpr("jim"), new AddExpr(new ValExpr("john"), new ValExpr("george")));

var htmlView = htmlExpression.Cata<string>({
  value: v => `<span>${v}</span>`,
  add: (x, y) => `<ul> <li>${x}</li> <li>${y}</li> </ul>`
})

```

Run This: [TS Fiddle](#)

We could example pass around a environment with variables dictionary:

```
type VariableStore<T> = { [name: string]: T; };

var Eval: (expr: Expr<string>, environment: VariableStore<number>) => number =
  (expr: Expr<string>, environment: VariableStore<number>) => {
    return expr.Cata({
      value: (v) => environment[v],
      add: (x, y) => x + y
    });
  }

var variables: VariableStore<number> = {};

variables["a1"] = 1;
variables["a2"] = 2;
variables["a3"] = 3;

var expression1: Expr<string> = new AddExpr(new ValExpr("a1"), new AddExpr(new ValExpr("a2"),
  new ValExpr("a3")));

var addition1 = Eval(expression1, variables);
```

Run This: [TS Fiddle](#)

## 5.3.2 Reader applicative implementation

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.

```
export class Reader<Env, T>
{
  apply<T1>(applicative: Reader<Env, (v: T) => T1>): Reader<Env, T1> {
    return new Reader<Env, T1>((env) => applicative.run(env)(this.Fn(env)));
  }
}

var readerAddFirst = new Reader<[number, number], (v: number) => number>((env: [number, number]) => (v: number) => env[0] + v); //Reader(env=>(u,v)=>u+v)

var readerAddFirstAnd5 = new Reader<[number, number], number>((env: [number, number]) => 5)
  .apply(readerAddFirst)
  //Reader(env=> 5+3)

var result = readerAddFirstAnd5.run([4, 5]); //9
```

below is my implementation, try to play around with the fiddle

```
abstract class Expr<T>
{

  Traverse<Env, T1>(f: (v: T) => Reader<Env, T1>): Reader<Env, Expr<T1>>
  {
    return this.MatchWith({
      value: (v) => Reader.ap(new Reader<Env, (v: T1) => Expr<T1>>(
        (env: Env) => x => new ValExpr(x)), f(v)),
      add: (x, y) => Reader.ap(
        Reader.ap(new Reader<Env, (u: Expr<T1>) => (v: Expr<T1>) => Expr<T1>>(
          (env: Env) => u => v => new AddExpr(u, v)),
          x.Traverse(f)),
          y.Traverse(f))
        ));
    }
  }
}
```

We can then use a variable dictionary

```
type VariableStore<T> = { [name: string]: T; };

var variables: VariableStore<number> = {};

variables["a1"] = 1;
variables["a2"] = 2;
variables["a3"] = 3;
```

Run This: [TS Fiddle](#)

that we can use to traverse an expression like this

```
var expression1: Expr<string> = new AddExpr(new ValExpr("a1"), new AddExpr(new ValExpr("a2"), new ValExpr("a3")));
```

and replace the variables with specific values

```
var getValue: (v: string) => Reader<VariableStore<number>, number> =
  v => new Reader((env) => env[v]);

var traverse: Expr<number> = expression1.Traverse(getValue).run(variables)

var result: number = traverse.Cata({
  value: (v) => v,
  add: (x, y) => x + y
})
```

The algebra of eval using our applicative notation  $\langle \rangle$  is

$\text{eval} :: (\text{Expr } u) \rightarrow \text{Env } v \rightarrow b$

$\text{eval } (\text{Val}(i), g) = \langle i \rangle$

$\text{eval } (\text{Add}(x, y), g) = \langle (+) \text{ eval } (x, g) \text{ eval } (y, g) \rangle$

### 5.3.3 Object Oriented Interpreter Pattern implementation

Take a moment to appreciate the use of the *reader applicative* to evaluate expressions. 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:

```
interface Context<T> { Accumulation: string; }

class VariablesContext<T> implements Context<T> {
  Variables: VariableStore<T>
  Accumulation: string;
  constructor(variables: VariableStore<T>) {
    this.Variables = variables;
  }
}

abstract class Expr<T> {
  abstract Eval(context: Context<T>)
}

class ValExpr<T> extends Expr<T> {
  Eval(context: Context<T>) {
    context.Accumulation += `${this.Value}`
  }
  Value: T;
  constructor(value: T) {
    super();
    this.Value = value;
  }
}

class AddExpr<T> extends Expr<T> {
  Eval(context: Context<T>) {
    context.Accumulation += `(`
    this.X.Eval(context);
    context.Accumulation += `,`
    this.Y.Eval(context);
  }
}
```

```

    context.Accumulation += ``
  }

  X: Expr<T>;
  Y: Expr<T>;
  constructor(x: Expr<T>, y: Expr<T>) {
    super();
    this.X = x;
    this.Y = y;
  }
}

```

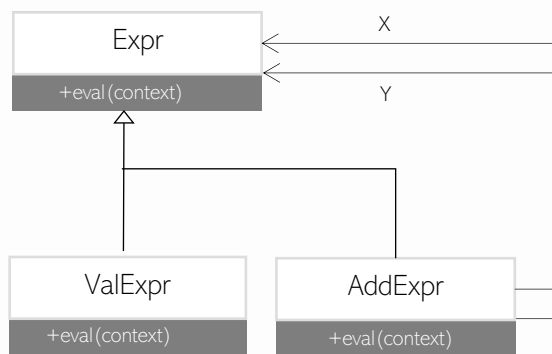
```
var expression = new AddExpr(new ValExpr(5), new AddExpr(new ValExpr(1), new ValExpr(2)))
```

```

var s = expression.Show();
var context = ({ Accumulation: `` })
expression.Eval(context)  //(5,(1,2))

```

Run This: [TS 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.

## 5.4 Foldable

Within the same paper of "[Applicative Programming with Effects](#)" Conor McBride and Ross Paterson used **an applicative constant functor based on a monoid**, in order to use it with a traversable, to perform the accumulation. I was planning to present this way of connecting Traversable with Foldable here but because of TS typing I think it only would make things much more difficult to follow.

Instead, we will go straight to some implementations of Fold on lists and Trees .Let's start by defining Fold on a List (this is basically the [Array.reduce](#))



```
function Fold<T>(array: Array<T>, monoid: { empty: () => T; concat: (x: T, y: T) => T; }):
T {
    return this.MatchWith({
        empty: () => monoid.empty(),
        cons: (v, r) => monoid.concat(v, r.Fold(monoid))
    });
}
```

Here we have a List and a monoid (`Func<T> empty, Func<T, T, T> concat`) in its simple Tuple form and we end up with a single type `T` in the next section of `FoldMap` we will see the more generic case where the `T` does not have to be a monoid but we have to provide a function `Func<T, TM> f` that converts it into a monoid `TM`

The Type definition of `Fold` is this:

```
Fold: t(a) → m      // where t : traversable and m: monoid
```

Now let's define the same thing for a Tree:

```
fold(monoid: { empty: () => T; concat: (x: T, y: T) => T; }): T {
    return this.MatchWith({
        leaf: (v) => monoid.concat(v, monoid.empty()),
        node: (l, v, r) => monoid.concat(monoid.concat(l.fold(monoid), v), r.fold(monoid))
    });
}
```

```
const tree = new Node1(new Node1(new Leaf(1), 2, new Leaf(3)), 4, new Node1(new Leaf(5), 6,
new Leaf(7)));
```

```
var fold = tree.fold({ empty: () => 0, concat:(x,y)=>x+y })
```

Fold with a simple  
(+,0) monoid

alternatively we can use the Array as a monoid and thus convert a tree into a map by folding with the list monoid (`: ,[]`)

```
tree.map(x=>[x]).Fold(monoid: (() => [], (x, y) => x.concat(y)));
```

Run This: TS Fiddle

but we had to map to a list first `Map(x => [x])` in order to use this definition of `Fold`. We could define a new Method to do this Mapping together with the `Fold`. This is called traditionally `FoldMap` and is the topic of the next section.

## 5.4.1 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

```
foldMap<TM>(m: { empty: () => TM; concat: (x: TM, y: TM) => TM; }, f: (x: T) => TM): TM {
  return this.MatchWith({
    leaf: (v) => m.concat(f(v), m.empty()),
    node: (l, v, r) => m.concat(m.concat(l.foldMap(m,f), f(v)), r.foldMap(m,f))
  });
}
```

Run This: [TS Fiddle](#)

In some cases, we can just omit the demand for both

1. m: { empty: () => TM; concat: (x: TM, y: TM) => TM; }
2. f: (x: T) => TM

But is not the case that we can always give the very convenient

(empty: () => TM , reducer: (x: TM, y: TM) => TM)

That we find in the List implementation

To see why can you implement the following for a tree?

```
foldMap<TM>(empty: () => TM, reducer: (x: TM, y: T) => TM): TM {
  return this.MatchWith({
    leaf: (v) => reducer(empty(), v),
    node: (l, v, r) => reducer(l.foldMap(empty, reducer), v)
  });
}
```

There is no way to squeeze  
in the `: r.foldMap`



Let's move on. With the help of FoldMap we can define some operations more intuitively, for example for this conjunction monoid:

(empty: () => true, concat: (x, y) => x && y)

And ask questions about Booleans like

```
var allLargerThanThree = tree.foldMap(
  (empty: () => true, concat: (x, y) => x && y), i => i > 3);
```

we can define an All That return true if all the members of a traversable satisfy a condition

```
All(f: (x: T) => boolean): boolean {
  return this.foldMap1<boolean>(({ empty: () => true, concat: (x, y) => x && y }), f)
}
```

Run This: [TS Fiddle](#)

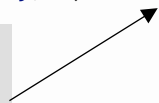
Another Implementation which is equivalent would be to define the Monoid we accumulate over as a different class

```
interface IMonoidAcc<T> {
  Identity: T;
  concat: (v: IMonoidAcc<T>) => IMonoidAcc<T>
}
```

And now have this equivalent formulation of FoldMap

```
foldMap<TM>(empty: IMonoidAcc<TM>, f: (v: T) => IMonoidAcc<TM>): IMonoidAcc<TM> {
  return this.MatchWith({
    leaf: (v) => f(v),
    node: (l, v, r) => l.foldMap(empty, f).concat(f(v)).concat(r.foldMap(empty, f))
  });
}
```

Directly chaining  
with the Concat that  
IMonoidAcc Exposes



Run This: [TS Fiddle](#)

We just provide an **f that return something that have a concat method** that we could use in order to accumulate over.

And use it like this with an Explicit Monoid

```
var foldMapResult = tree.foldMap(new SumAcc(0), x => new SumAcc(x))

class SumAcc implements IMonoidAcc<number> {
  concat(v: IMonoidAcc<number>): IMonoidAcc<number> {
    return new SumAcc(v.Identity + this.Identity);
  }

  constructor(value: number) {
    this.Identity = value;
  }
  Identity: number = 0;
}
```

Run This: [TS Fiddle](#)

## 5.5 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: monoid<T>` we can just return an algebra that uses this monoid to concatenate all the values of the Traversable.

```
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](#)

that's pretty much the definition of the fold on a List.

## 5.6 Traverse Derivation from Catamorphism

In the same spirit if we use any applicative functor as a Monoid to fold a Traversable then what we get is our implementation of Traverse:

```

function TraverseAlgebra<T, T1>(f: (v: any) => Id<T1>): TreePattern<T, Id<Tree<T1>>> {
  return ({
    leaf: v => Id.ap(Id.of(x => new Leaf<T1>(x)), f(v)),

    node: (l, v, r) =>
      Id.ap(Id.ap(Id.ap(
        Id.of((left: Tree<T1>) => (v: T1) => (right: Tree<T1>) => new Node1(left, v, right)),
        l),
        f(v)),
        r)
  });
}

```

Run This: [TS Fiddle](#)

```

var traversedTree = tree.Cata(TraverseAlgebra<number, number>(x => Id.of(x + 1)));

console.log(traversedTree.Value.Show());

```

# Second Instalment

This book is not finished yet the second instalment which is autonomous and builds on the previous sections include two main Chapters. You will be notified when those are ready.

## 6 Monads

## 7 Comonads

# 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](mailto:dimitrispapadim@live.com)

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

© 2019 Dimitris Papadimitriou