

# Functional Programming **TypeScript**

{with: categories}

Gain advanced understanding of the mathematics behind modern functional programming

# TS



dimitris papadimitriou  
Version : 2.0.0

# FUNCTIONAL PROGRAMMING IN TYPESCRIPT WITH CATEGORIES

Gain advanced understanding of the mathematics behind modern functional programming

Dimitris Papadimitriou

This version was published on 08/Nov/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)

© 2021 Dimitris Papadimitriou

# Contents

About this book coding conventions .....	4
1 Basic Concepts .....	5
1.1 Lenses .....	5
2 Algebras of Programming .....	9
2.1 Monoids .....	9
2.2 Parenthesis Balancing using Monoids.....	12
3 Algebraic Data Types.....	14
3.1 The product structure: .....	16
3.2 Introduction / Elimination.....	18
3.3 The Co-Product (aka Union/Sum) structure .....	19
3.4 Extending Union Types.....	20
3.5 Functors.....	22
3.6 The Identity Functor .....	23
3.7 The Basic Functor Mechanics .....	25
4 Natural Transformations.....	27
5 Monads.....	30
5.1 Maybe Monad .....	31

# Purpose

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

— Edsger Dijkstra

OO makes code understandable by encapsulating moving parts.

FP makes code understandable by minimizing moving parts.

—Michael Feathers (Twitter)

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

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

# Basic Concepts

---

## 1 Basic Concepts

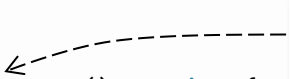
"Classes should be immutable unless there's a very good reason to make them mutable."  
-Joshua Bloch

### 1.1 Lenses

Lenses are basically functional getters and setters. Let us see step by step the evolution of lenses and its Object-oriented equivalence. ES6 has allowed getter and setter functions within classes (and object literals). For a `Client` object created using the `class`

```
class Client implements Named, Identifiable {
  private _id: number;
  private _name: string;

  constructor(id, name) {
    this._id = id;
    this._name = name;
  }
  get name(): string {
    return this._name;
  }
}
```



This `get` keyword used to declare a getter

Run This: [Ts Fiddle](#)

We cannot access the client name directly by `client._name` because it is private, since ES6 we can also add a getter function :

```
get name(): string {
  return this._name;
}
```

And access the `_name` by writing instead `client.name`. Alternatively, we can extract the getter operation as an external function:

```
var getName: (c: Named) => string = client => client.name;
```

this notation comes handy when we use for example the map on an Array of `Clients`

```
var clients: Array<Named> = [
  new Client(1, `jim`),
  new Client(2, `jane`),
  new Client(3, `john`)
];

var clientNames: Array<string> = clients.map(client => client.name);

var clientNames: Array<string> = clients.map(getName);

var clientNames: Array<string> = clients.map(Client.getName);
```

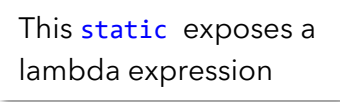
Run This: [Ts Fiddle](#)

this is more convenient because it allows a point free notation, instead of the usual pointed notation `clients.map(client => client.name)`. Finally, we can place the `getName` function as a static method inside the `Client` object:

```
class Client {
  private _id: number;
  private _name: string;

  constructor(id, name) {
    this._id = id;
    this._name = name;
  }

  static get getName(): (c: Named) => string {
    return c => c.name;
  }
}
```



And now we can write `clients.map(Client.getName)`. We have not mentioned the set operation yet. But everything follows the same line of reasoning. We can extract a `setName` function responsible for updating the name value in an immutable way that returns an updated copy of the object

```
type SetterType = (c: Client, v: string) => Client;

var setName: SetterType = (client, newName) => new Client(client.id, newName); //immutable

var updatedClient = setName(client, `Jim Doe`);
console.log(getName(updatedClient));
```

Run This: [Js Fiddle](#)

If we put the `getName` and `setName` functions in one object then we have created a lens. So, what is a lens? just an object that has a get and a set method:

```
export type LensGet<T, A> = (obj: T) => A;
```

```
export type LensSet<T, A> = (obj: T) => (newValue: A) => T;

export interface Lens<T, A> {
  get: LensGet<T, A>;
  set: LensSet<T, A>;
}
```

Run This: [Js Fiddle](#)

We can create a generic lens that takes the name of the property and implements the `get` and `set` functions in a general manner as follows :

```
export let propLens = function <T, A>(key: string): Lens<T, A> {
  return {
    get: (obj: T): A => obj[key],
    set: (obj: T) => (value: A): T => ({ ...obj, [key]: value })
  };
};
```

Run This: [Js Fiddle](#)

The `set` method:

1. Uses the spread operator notation (`{ ...obj }`) in order to copy all the properties of the object into an new literal.
2. And then sets the property named `key` with the new `value` this is done with the `[key]: value` statement.

Now that we have this `lens` we can use this to create a property lens like this `lens(`name`)` and then use this lens to get or set the name of the client:

```
var nameLens: Lens<Client, String> = propLens(`name`);
let client: Client = new Client(1, "Jim");
console.log(nameLens.get(client)); //getter

let updatedClient: Client = nameLens.set(client)(`Jim doe`); //setter -
update object
console.log(nameLens.get(updatedClient)); //getter
```

Run This: [Js Fiddle](#)

It is also very common to use this `lens` to create three functions called `view`, `set` and `over`:

```
export let view = <T, A>(lens: Lens<T, A>, obj: T): A => lens.get(obj);

export let set = <T, A>(lens: Lens<T, A>, obj: T, value: A): T => lens.set(obj)(value);

export let over = <T, A, B>(lens: Lens<T, A>, f: (x: A) => B, obj: T) =>
  lens.set(obj)(f(lens.get(obj)));
```

Run This: [Js Fiddle](#)



the view and set are just the get and set functions of the `lens` whereas the `over` is just a mapping function, which we use in a straightforward manner :

```
var updatedClient: Client = over(nameLens, x => x.toUpperCase(), client);
console.log(view(nameLens, updatedClient));
```

Run This: [Js Fiddle](#)

There is a simple reason that destroys the case for this form of the lens : we should not use strings to represent the property: `lens(`name`)`. If you have to rename the property, you will have to manually find and update all the strings referencing the property, the VS code cannot detect the usages. The possibility of generating new bugs is extremely high. For this reason, I would usually use either "strong type" hardcoded lens :

```
//Immutable
//ad-hoc lens - specifically crafted for the name property
const nameLens = {
  get: client => client.name,
  set: (client, newName) => ({ ...client, [`name`]: newName })
};
```

Or choose the static lambda get, set on the Object class:

```
class Client {
  static get getName() {
    return c => c._name;
  }
  static setName(value) {
    return c => new Client(c._id, value);
  }
}

var updatedClient = Client.setName(`Jim Doe`)(client);
```

# Algebras of Programming

## 2 Algebras of Programming

### 2.1 Monoids

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

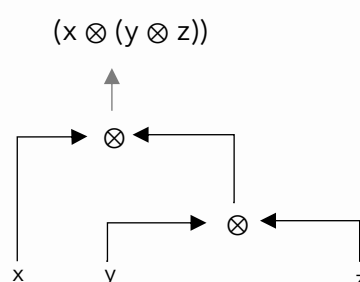
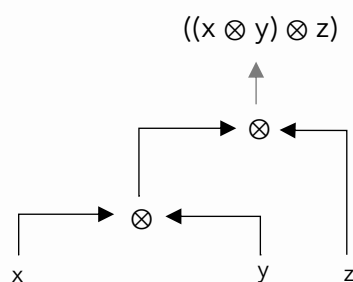
– Categories for the Working Mathematician

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

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

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

1. **Associativity:** For all  $a, b$  and  $c$  in  $S$ , the equation  $(x \otimes y) \otimes z = x \otimes (y \otimes z)$  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);
}

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

The return type is `IMonoidAcc` allowing for fluent chaining

Run This: [Fiddle](#)

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

The use of `reduce` is a bit more tedious thought

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

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.

## 2.2 Parenthesis Balancing using Monoids

As a first example we are going to see the classic Parenthesis Balancing problem is problem that states: **Given an expression string exp, write a program to examine whether the pairs and the orders of "(" , ")" are correct in exp.**

So this : ( ( ) ( ) ( ) ) is balanced but this one: ( ) ) is not. We are going to define a simple balance type to record the number of the left end right values.

```
class Balance {

    static Left = new Balance(0, 1)
    static Right = new Balance(1, 0)
    static Empty = new Balance(0, 0)

    L: number
    R: number
    constructor(l: number, r: number) {
        this.L = l;
        this.R = r;
    }
}
```

We can amazingly form a monoid on this `Balance` type. Since we can concatenate two `Balance` objects and get a new `Balance`.

```
class BalanceMonoid {

    empty: Balance = Balance.Empty;
    concat(x: Balance, y: Balance): Balance {
        if (x.R < y.L)
            return new Balance(x.L + y.L - x.R, y.R);
        else
            return new Balance(x.L, y.R + x.R - y.L);
    }
}
```

Run This: [Ts Fiddle](#)

Hopefully, this makes the rest of the idea obvious.

1. we will split the string in an array of characters
2. then we will convert each character in a balance (parse)
3. and then we will reduce the array with the Balance monoid.

the simple parsing function would be something like this :

```

var parse: (c: string) => Balance = (c: string) => {
  switch (c) {
    case "(": return Balance.Left
    case ")": return Balance.Right
    default: return Balance.Empty
  }
}

```

Run This: [Ts Fiddle](#)

so finally, we can write the computation like this :

```

var weight = new BalanceMonoid();

var getBalance: (input: string) => Balance = input =>
  Array.from(input)
    .map(parse)
    .reduce(weight.concat, weight.empty);

let parsed = getBalance("((()))()"); //{L:0,R:0}

```

Run This: [Ts Fiddle](#)

# Algebraic Data Types

---

## 3 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:

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

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 coproduct, 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
```

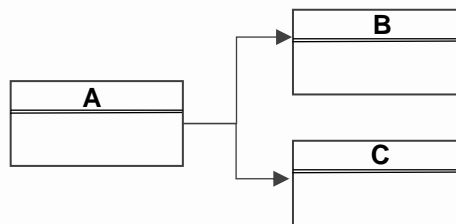
The fact that a data type can be formed by applying some operations between them gives them the name **algebraic**. An **algebraic data type** is a kind of composite type.

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.



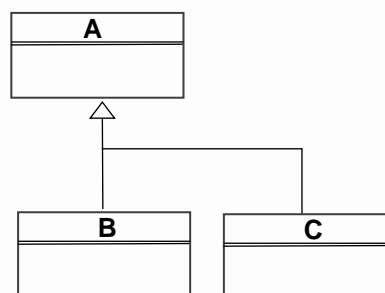
Product type

$$A = (B, C)$$

Has-a: Composition

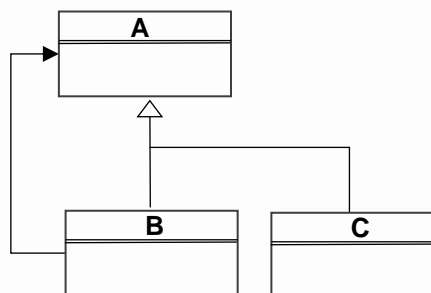
Union type

$$A = B + C$$

Is-a: Inheritance

Recursive Type

$$A = (B, A) + C$$



### 3.1 The product structure:

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

We can form products using the available 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 called **projections**.

$$A \xleftarrow{\text{first}} A \otimes B \xrightarrow{\text{second}} B$$

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.

## 3.2 Introduction / Elimination optional

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.

### 3.3 The Co-Product (aka Union/Sum) structure

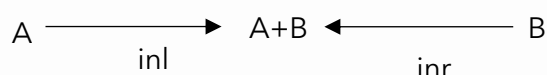
"...If you only have studied conventional programming languages you never heard of **Sum types** before, because there is this gigantic blind spot..."

– Robert Harper

Oregon Programming Languages Summer School

The co-product (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();
```

## 3.4 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 its recursive we will see examine it in the next section closely.

# 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:

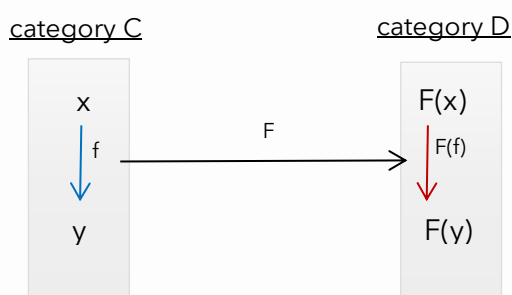
In TS the most famous functional programming idea is to use `Array<T>.map` to replace iterations instead of *for loops* in order to transform the values of the array. That is because an array is a Functor, which is a more abstract idea that we will explore in this section.

“Practically a Functor is anything that has a valid `.map(f)` method”

Functors can be considered the core concept of category theory.

## 3.5 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  $f$ .

## 3.6 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))
  };
}
```

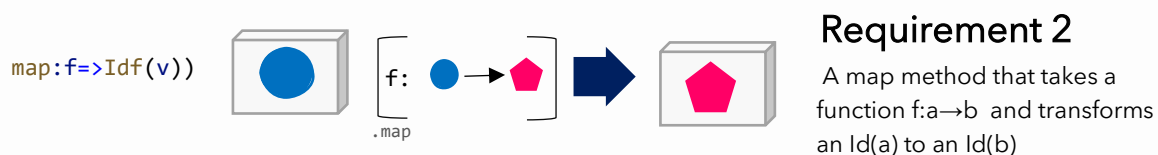
that is requirement 1

that is requirement 2

Run This: [TS Fiddle](#)

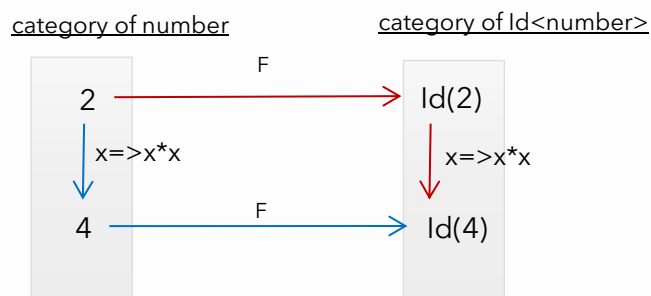
This is the minimum construction that we could call a functor because it has exactly two things

1. A "constructor" that lifts an object  $T$  to  $\text{Id}<T>$
2. And it has a **mapping** method  $\text{map}<T1>(f: (y: T) \Rightarrow T1)$  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<number>` 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 is easy to see intuitively that the correct mapping is:

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

Because if we use this map method with the squaring function `x=>x*x` we will get as a result an `Id (4)`:

```
Id(2).map(x=>x * x) // Id(4)
```

## Discussion on the Types

Usually when one looks at the map method in the context of Pure functional languages is represented like this - **map: (a → b) → f(a) → f(b)** where **f** is a Functor and a, b are types - (eg. fp-ts, [Haskell](#), [Scala\[Cats\]](#), [Sanctuary.js](#), [Ramda.js](#) ).

We can get this type if we completely extract the method as a function that acts on the object like below:

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

```

    }
  }

  let map = function<T, T1>(f: (y: T) => T1): (id: Id<T>) => Id<T1> {
    return (id: Id<T>) => new Id<T1>(f(id.Value));
  };

  // map
  let result1: Id<number> = map<number, number>(x => x + 2)(new Id<number>(5));

```

Run This: [TS Fiddle](#)

Here the function map has the type  $(f: (y: T) \Rightarrow T1) \Rightarrow (id: Id<T>) \Rightarrow Id<T1>$  which is  $(T \rightarrow T1) \rightarrow Id<T> \rightarrow Id<T1>$

if we rearrange the terms into this form  $Id<T> \rightarrow (T \rightarrow T1) \rightarrow Id<T1>$  (this is the form in Fantasy land spec) which is the Object oriented version which we can be interpreted in this way: if you give me a function from a to b ( $a \rightarrow b$ ) and I have an  $Id<T>$ , I can get an  $Id<T1>$ . The  $Id<T>$ , in this case is the object ([this](#)) that contains the method.

## Libraries

### Mapping 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  $Id = (v) \Rightarrow \{ \}$ . In many functional libraries online you may find the explicit definition of an `.of()` that does the same thing

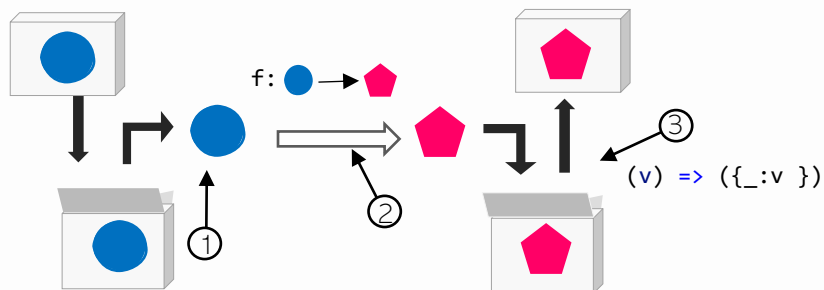
```
Id.of = v=>({ v: v, map: f => Id.of(f(v)) })
```

Functors that provide an `.of` method are called **Pointed Functors**

Run This: [Js Fiddle](#)

## 3.7 The Basic Functor Mechanics

The following picture represents the whole idea behind how we construct the map of a functor. **If we accept the Functor as a container metaphor**, which is an acceptable way to visualize functors especially in an Object-oriented programming setting.



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

1. Open the container  $F(a)$  and access the value  $a$ . We might have direct access or use pattern matching
2. Applying the function  $f$  on the value  $a$  and get a new value  $b$
3. wraps the resulting value again into a new container  $F(b)$  and return this  $F(b)$

in the simplest form of the Identity functor the map follows those steps

```
let Id = <T>(value: T) => ({
  map: <T1>(f: (y: T) => T1) => Id<T1>(f(value)),
  getValue: (): T => value
});
```

Diagram annotations for the code above:

- 1. points to `Id<T1>(f(value))`
- 2. points to `f(value)`
- 3. points to `Id<T1>`

# Transformations

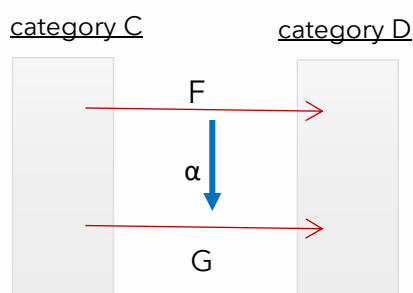
## 4 Natural Transformations

"I didn't invent categories to study functors.  
I invented them to study natural transformations."

-Saunders Mac Lane

In category theory, a **natural transformation** provides a way of transforming one functor into another while respecting the internal structure (i.e., the composition of morphisms) of the categories involved. Hence, a natural transformation can be considered as a "morphism of functors". Natural transformations are, one of the most fundamental concepts in category theory, after categories and functors.

If we have two functors **F** and **G** between two categories sometimes we may have a consistent transformation  $\alpha: \mathbf{F} \rightarrow \mathbf{G}$  between them, which is called natural transformation in the language of category theory. And we represent it with an arrow from one functor to the other:



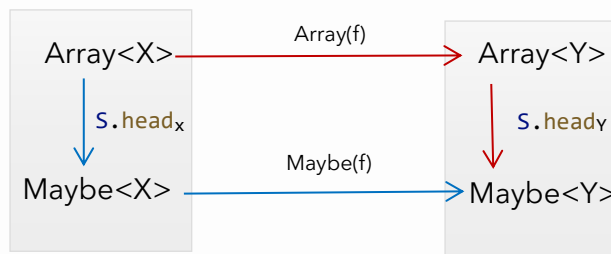
Let's see a simple example if we use `array[0]` to get the first element/head of a list we may get an undefined in case of the empty list `[]`. But we could implement a `firstOrNull` function that would return a `Maybe<A>`. This is a transformation between the List and Maybe Functors:

```
let firstOrNull = function (array) {
  return array.length > 0 ? Some(array[0]) : None()
}
```

Sanctuary.js has an implementation of the head function that works like this

```
S.head ([1, 2, 3])    //Just (1)
S.head ([])           //Nothing
```

we can represent this in a diagram

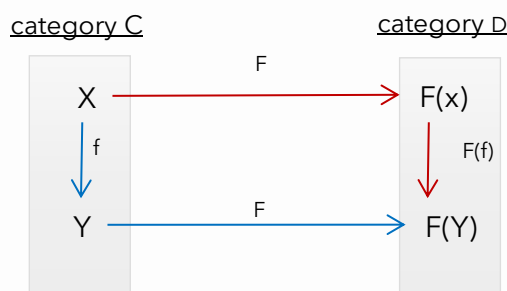


For the `S.head` definition to be a valid natural transformation the diagram above should commute. This means that the following must hold for all arrays `x`:

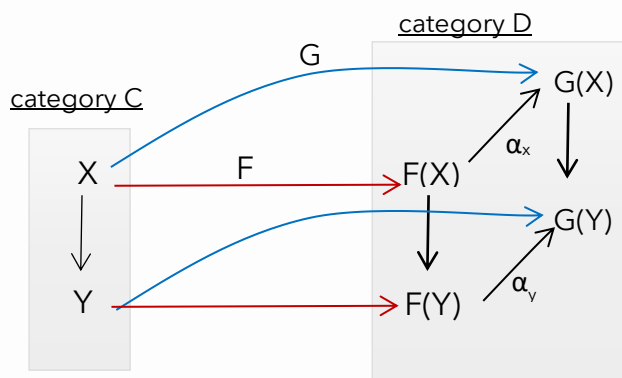
```
firstOrNone(array.map(f)) == firstOrNone(array).map(f)
```

Run This: [Js Fiddle](#)

if you take the diagram of a functor **F** between C and D :



and you try to add another functor diagram **G** in the same picture then you can visualize the natural transformation like this



$\alpha$  is denoted with an index  $X$ . The  $\alpha_x$  it is called the component of  $\alpha$  at  $X$ .

Let us revisit some of the Natural transformations that we have encountered so far.

## Array to Maybe

The maybe type because it represents the null case is of central importance. We already have seen the `firstOrNone: Array<T> → Maybe<T>`

```
let firstOrNone = function (array) {  
  return array.length > 0 ? Some(array[0]) : None()  
}
```

Run This: [Js tests Fiddle](#)

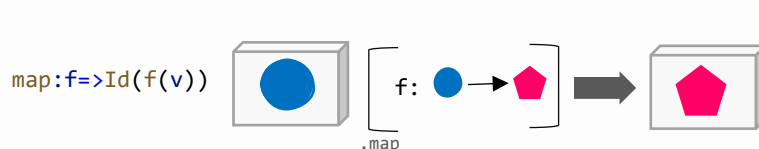
In the same way any filter, or predicate (like All () etc.) function on data structures like Arrays and Trees may return a `Maybe <_>`.

# Monads

"Wadler tries to appease critics by explaining that "a monad is a monoid in the category of endofunctors, what's the problem?"  
*-Brief, Incomplete and Mostly Wrong History of Programming Languages*

## 5 Monads

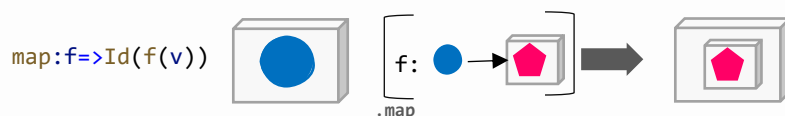
If you remember in the section "Intro to Functors" we stated that when we use the `map` on a function  $f: A \rightarrow B$  we transform the value  $A$  inside the functor  $F<A>$  to a new type  $F<B>$ .



### Requirement 2

A `map` method that takes a function  $f: a \rightarrow b$  and transforms an `Id(a)` to an `Id(b)`

Sometimes it might happen that the type  $B$  is inside a Functor  $F$  is itself a  $F<B>$ , this means that the function  $f$  might look like this  $f: A \rightarrow F(B)$ . In those situations, after the `map` we end up with something like  $F<F<B>>$ . A Functor-In-A-Functor situation.



You can see this in the following example in TypeScript (to witness the types) where the type of the result is `result: Id<Id<number>>`

```
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)); }

  match(pattern: (y: T) => void): void { pattern(this.Value); }
}
```

```

}

let result: Id<Id<number>> = new Id(5).map(x => new Id(x + 2));
//NOTE: we use the bind only when the lambda returns an Id() type

```

Run This: [Ts Fiddle](#)

If  $F$  is a monad then for those situations where the function is of this type  $f: A \rightarrow F(B)$  we can use a method called **flatMap** instead of the **map** [ In the functional world is known as **bind**



or chain sometimes ].

The same reasoning goes for all functors , here some other examples leading to the same situation :

```
IO(()=>5).map(x=>IO(()=> x+1)) === IO(()=>IO(()=> 6))
```

```
[5].map(x=>[x+1]) === [[5]]
```

```
maybe(5).map(x=>x+1) === maybe(maybe(5))
```

## 5.1 Maybe Monad

Maybe monad is extension of the Maybe Functor. In order to make this maybe functor implementation into a Monad we must provide a bind method that combines two Maybe monads into one. **There are 4 different ways to combine the 2 possible states of maybe (Some, None)** one can see that a valid implementation would be the following :

```

abstract class Maybe<T> {
  abstract matchWith<T1>(pattern: { none: () => T1; some: (v: T) => T1 }): T1;
  abstract map<T1>(f: (v: T) => T1): Maybe<T1>;

  bind<T1>(f: (v: T) => Maybe<T1>): Maybe<T1> {
    return this.matchWith({
      none: () => new None<T1>(),
      some: (v: T) => f(v)
    })
  }
}

```



```

    })
  }
}

```

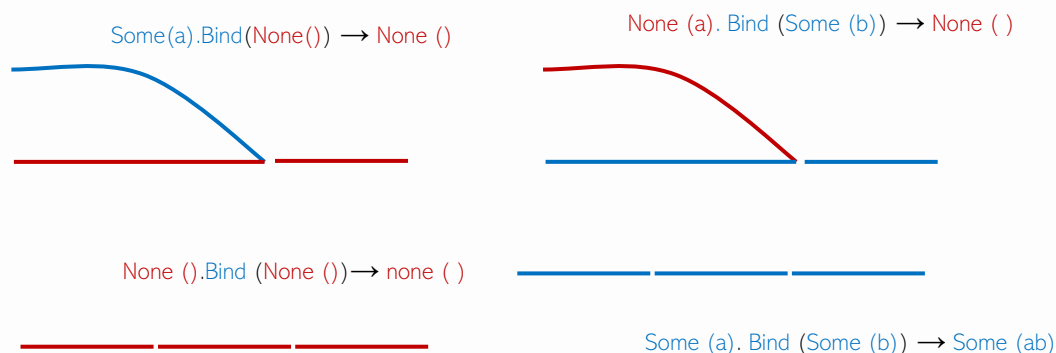
Run This: [TS Fiddle](#)

the only combination that leads to a new some (Some) is when the two paths that both have values are being combined.

```

Some (x). Bind ( None )
None ( ). Bind ( Some )
Some (x). Bind ( Some ) //only this gives some result
None ( ). Bind ( None )

```



Let us say that we have this scenario where we have a list of clients and each client is assigned to an employee. Then let us say get the name of the assigned employee for that client.

This is the standard one-to-many relationships and the way to model it is to add on the client entity a property that will store the value of the Id of the related employee



```

class MockClientRepository {
    GetById(id: number): Maybe<Client> {
        let clients: Array<Client> =
            [new Client(1, "jim", 1),
             new Client(2, "john", 1)];
        var maybeClient: Maybe<Client> = F.FirstOrNone(clients, c => c.Id == i
d);
    }
}

```

```

        return maybeClient;
    }
}

class MockEmployeeRepository {
    GetById(id: number): Maybe<Employee> {
        let employees: Array<Employee> =
            [new Employee(1, "jane"),
             new Employee(2, "rick")];
        var maybeEmployee: Maybe<Employee> = F.FirstOrNone(employees, c => c.Id == id);
        return maybeEmployee;
    }
}

```

We use `bind` `.Bind(employeeRepository.GetById)` to capture the client value from the `maybe` and pass it to the `employees.getById` which returns a `Maybe`, so the final result of the `bind` is in fact a `Maybe <Employee>`.

```

class SearchService {
    getEmployeeNameForClientWithId(clientId: number) {
        let clientRepository = new MockClientRepository();
        let employeeRepository = new MockEmployeeRepository();
        return clientRepository
            .GetById(clientId)
            .bind(client => employeeRepository.GetById(client.EmployeeId))
            .map(e => e.Name)
    }
}

new SearchService()
    .getEmployeeNameForClientWithId(2)
    .matchWith({
        some: name => console.log("assigned Employee name: " + name),
        none: () => console.log("client is not assigned or there was no client ")
    });

```

Run This: [TS Fiddle](#)

As you can see we have used a very generic error message in the case of `none` because using `Maybe` we cannot distinguish between the `None` we merge with the `bind`. The Either monad that we will see in the next section allows us to provide different messages for each case.