FUNCTOR AND APPLICATIVE FUNCTOR USAGE IN TYPESCRIPT

Matija Matović[1]*,
Milan Segedinac[2]

[1]Singidunum University,
Belgrade, Serbia

[2]Faculty of technical sciences at
University of Novi Sad,
Novi Sad, Serbia

Abstract:

Leveraging functional programming concepts to make front-end application development faster and easier, with fewer bugs has been tried with pure functional programming languages such as Elm. Visible improvement in development time and application quality compared to JavaScript has been recorded. TypeScript is a multi-paradigm programming language, so it is possible to implement these concepts without switching languages. Functors and applicative functors are some of the key concepts of functional programming, useful for processing complex objects and collections of data. As TypeScript is often used to visualize lists of data retrieved from a server, functors and applicative functors could be used to process data into a format required for visualization. This paper presents theoretical explanations of the Functor and Applicative Functor concepts in category theory and provides their implementations in TypeScript, focusing on Maybe and List functors. The use cases were then shown, which demonstrate that they could be useful, especially when abstracting complex concepts, so they could be used and invoked on demand, usually with a simple command.

Keywords:

Functional Programming, Functor, Applicative Functor, TypeScript.

INTRODUCTION

Web applications today are used for a number of purposes in every walk of life. One of the largest industries in the IT sector is web programming with 24.5 million people employed and millions of apps deployed annually [1]. With such work volume, it would be convenient to develop tools and technologies that could aid in lessening the amount of time spent in writing code, error correction, testing and validation, and synchronizing ideas between developers working on projects. Concepts of functional programming [2] are highly suitable for solving these problems [3]. The nature of functional programming languages aids in developing highly abstract and declarative code, that is readable and concise. This allows programmers to write and review code faster as declarative code maps better to a programmer's thinking process [3]. Also, since the functional paradigm is based on rigorous mathematical concepts [4], [5], code validation is easier [2]. Another useful feature of functional programming are pure functions which make testing much easier and facilitate code parallelization [3].

Correspondence:
Matija Matović

e-mail:
mmatovic@singidunum.ac.rs

DOI: 10.15308/Sinteza-2023-116-123

TypeScript is a multi-paradigm programming language that adds static typing to JavaScript. In TypeScript, functions are first-class objects, and generic typing is supported, which enables an easy implementation of concepts of functional programming.

This paper will focus on the most common functional programming concepts, functors and applicative functors, how they can be implemented in a multi-paradigm environment of TypeScript, and how they can be leveraged and used in front-end development. The paper is structured in the following way. First, the mathematical background of functors is presented and their implementation in TypeScript. Then, a similar structure is used for discussing applicative functors. Finally, the concluding section discusses if TypeScript achieved the desired improvements and compare them to a scenario where they aren't used.

2. FUNCTORS

After categories, functors are the most important concept of category theory [5]. They are particularly useful when dealing with complex objects or collections of data. Due to the importance of functors, concepts based on them are implemented in many procedural programming languages such as Python, Java, JavaScript, etc.

2.1. FUNCTORS IN CATEGORY THEORY

In category theory, functors are a mapping between two categories [6]. Given two categories, C and D, functor F is a mapping such that:

- associates every object a in *C* to an object *Fa* in *D*, and
- associates each morphism f*: x→y* in *C* to a morphism *Ff: Fx→Fy* in *D*.

These mappings of morphisms must be such that the following two conditions hold:

- It maps identity morphisms in *C* to identity morphisms in *D*. For each object *X* in *C* it must hold that $Fid_x=id_{Fx}$, and
- Mapping of composition of any two morphisms *g* and *h*, such that *h: a→b* and *g: b→c*, must be equal to composition of mappings of those morphisms, or formally: *F(g∘h) = Fg∘Fh*.

If these two conditions are met, the functor preserves the structure of the original category. If a category is pictured as a web where the objects are considered as nodes, and morphisms between them as edges, then functors aren't allowed to introduce tears to the web's fabric [5]. They can merge multiple objects or morphisms together, but they cannot remove any connections. If a path exists between some two objects in the original category, then a path must also exist between their mappings in the resulting category. Categories consist of objects and morphisms, i.e. connections between objects. If these concepts are transferred to programming, then objects would be data types and morphisms would correspond to functions, which map one data type to another [5]. If these categories are pictured as objects, in a category of categories, then functors would be morphisms between categories. Considering that categories in programming correspond to types, functors in programming would then be type constructors. In other words, functors in programming are a type of containers or wrappers on basic data types, that add certain semantics to them, and they would also add functionality according to that semantics to the functions they map and the underlying data types that they wrap [2].

2.2. FUNCTORS IN TYPESCRIPT

A functor implementation[1] needs to implement one function called the map function. This function receives a functor and a function, then applies that function to a value contained inside the functor. Function map receives a function (that receives a value of type a, and returns a value of type b), and a value of type a wrapped inside functor f. It returns a value of type b, also wrapped within f. From the function type, it can be concluded that the map function receives a functor and a function and applies that function to the value wrapped inside the given functor, according to the context of the functor within which it is wrapped [2]. If currying [7] is applied, the function type can be written as in Listing 1.

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)
```

Listing 1 - map function type rewritten so it accepts one parameter (a function) and returns one parameter (also a function)

1 All the code implementations in this paper, and more, can be found at our GitHub page

Function map can also be seen as a function that takes a function of type *a→b*, and returns a function of type *Fa→Fb*. If compared to the mathematical definition of a functor, it can be derived that the definitions are equivalent. The map function, thus, maps objects and morphisms in one category, to objects and morphisms in another category. To be a functor, *f* has to satisfy the two conditions specified above. TypeScript considers functions first-class objects, and supports type variables and generic types, features to be used extensively for functor construction in TypeScript.

2.2.1. Maybe functor

One of the most commonly used functors in functional programming is the Maybe functor. The Maybe functor gives the value a context of "uncertainty". The value may or may not be contained within the functor. This is commonly used for computations that may or may not return a value (like finding an element within a list, or an entry in a database), or for computations that may break due to an error. There are two forms that the Maybe functor can take: Nothing, which represents the absence of the value contained within the functor, and Just val, which represents that a value is present inside. The two options are two separate type constructors, so the Maybe functor has two functions that return different forms of the Maybe functor. In addition to this, value a represents a type parameter which is used to declare which type the Maybe functor will wrap. Mapping of a function (Listing 2) on a Nothing value produces no result, as there is nothing to map the function to. Mapping of a function to Just val applies that function to val, the value contained inside the functor, i.e., passes on the function to the value inside the functor. The map function satisfies both functoriality conditions [5]. As mentioned, Maybe is made from two constructors, and it receives one type parameter. If this is translated into TypeScript, that would mean that there are two types necessary, each with its own constructor. This is best implemented if Maybe is a generic abstract class that takes a type parameter and is inherited by Just and Nothing types (Listing 2). The Maybe class could also have an abstract map method that can be used for polymorphism.

The Maybe functor is particularly useful when chaining multiple operations that may fail, or for element lookup in lists, which may fail (Listing 3).

```
abstract class Maybe<T> {
    abstract mmap<X>(f: (x: T) => X): Maybe<X>
}
class Just<T> extends Maybe<T> {
    value: T;

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

    mmap<X>(f: (x: T) => X) {
        return new Just<X>(f(this.value));
    }
}
class Nothing<T> extends Maybe<T> {
    mmap<X>(f: (x: T) => X) {
        return new Nothing<X>();
    }
}

function maybeMap<T, U>(f: (x: T) => U, m: Maybe<T>): Maybe<U> {
    f = curry(f);
    if (m instanceof Just) {
        let {value} = m;
        return new Just(f(value));
    }
    return new Nothing<U>();
}
```

Listing 2 - Example implementation of Maybe functor in TypeScript.

```
function listLookup(list: Array<number>, x: number): Maybe<number> {
    for (let i = 0; i < list.length; i++) {
        if (list[i] == x)
            return new Just(i);
    }
    return new Nothing<number>;
}

function inc(x: number): number {
    return x + 1;
}

let list1 = [1, 2, 3, 4, 5];
let lookupResult1 = listLookup(list1, 3).mmap(inc).mmap(inc);
let lookupResult2 = listLookup(list1, 6).mmap(inc).mmap(inc);
console.log(lookupResult1);  // Prints Just: { "value": 4 }
console.log(lookupResult2);  // Prints Nothing: { }
// More functional programming way

lookupResult1 = maybeMap(inc, maybeMap(inc, listLookup(list1, 3)));
lookupResult2 = maybeMap(inc, maybeMap(inc, listLookup(list1, 6)));
console.log(lookupResult1)  // Prints Just: { "value": 4 }
console.log(lookupResult2)  // Prints Nothing: { }
```

Listing 3 - Example of Maybe usage in TypeScript.

2.2.2. List functor

When constructing a list, in most programming languages, it is necessary to pass the type of elements it will contain. This is similar to previously mentioned type constructors, which receive a type as their parameter. Also, a list adds a new context to a type, a context of non-determinism [5], since a list isn't a single value, but a collection of multiple values. Thus, if a map function that satisfies both functoriality conditions can be constructed, List can also be considered a functor. Such a function can indeed be constructed, and List is indeed a functor. Both lists, and map function for lists are already implemented in TypeScript, but the List type, however, isn't a recursive type as it is in functional programming. Therefore, we replicated recursive definition in TypeScript (Listing 4).

Map function (Listing 5) can be easily implemented similar to how it was implemented in Haskell [2]. This function receives a list and a function, then applies that function to each element in the list. This makes sense from a mathematical perspective - a list is a collection of values, where each one has an equal probability of being accessed or modified. Because of that, there is no specific subset of elements, but the function is applied to each element.

```
abstract class List<T> {
}
class Empty<T> extends List<T> {
}
class Cons<T> extends List<T> {
    head: T;
    tail: List<T>;

    constructor(value: T, tail: List<T>) {
        super();
        this.head = value;
        this.tail = tail;
    }
}
```

Listing 4 - List type definition in TypeScript.

```
let l = new Cons(5, new Cons(4, new Cons(3, new Cons(2, new Cons(1,
Empty<number>)))))
function listMap<T, U>(f: (x: T) => U, l: List<T>): List<U> {
    if (l instanceof Cons) {
        var {head, tail} = l;
        return new Cons(f(head), listMap(f, tail));
    }
    return new Empty<U>();
}
let listMapResult = listMap(inc, l);
console.log(listMapResult)  // Prints List: {“head”: 6, “tail”: {“head”: 5, “tail”:
{“head”: 4, “tail”: {“head”: 3, “tail”: {“head”: 2, “tail”: {}}}}}}
```

Listing 5 - List map function implementation in TypeScript and sample usage.

3. APPLICATIVE FUNCTORS

Functors are useful when applying a function to an element contained within some wrapper when those functions have only one input argument. To enable function with multiple input arguments, one has to deal with currying and partial application. Currying isn't native to TypeScript as it is to pure functional programming languages, so it needs to be implemented. Still, the currying in TypeScript is easy to implement, and can be particularly useful when partial application is used, especially for concepts from category theory such as applicative functors. It was already discussed that a functor is a sort of wrapper that adds context to a value. It is possible to pass a function to this wrapper, that is, it is possible to enclose a function within a functor. If the function takes multiple parameters, it is also possible to map a function to a value within a functor, and thus receive a partially applied function in a wrapper. Eventually, it will be required to pass the second parameter to the function, so that a result can be evaluated. Let the other value be wrapped in a functor as well. A function that will achieve this has to have a signature f (a -> b) -> f a -> f b. The reason why this function returns a value in a functor is that, if the initial function takes more than two parameters, multiple partial applications will need to be performed. If the function has the signature f (a -> b) -> f a -> f b, then this partial application can be chained and performed easily [8].

Invocation of a partially applied function in a functor and chaining described in the previous section can be achieved using a special concept called applicative functors. Applicative functors need to fulfill two main functionalities [9]. The first is to implement a function with a signature described above, which is usually called “supermap”, and represented with an infix “<*>” operator.

The second functionality is to put a value within a wrapper or to add a certain context to it. In other words, it should be able to put a value inside a functor. This can be described with a typeclass definition shown in Listing 6.

```
class (Functor f) => Applicative f where
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
```

Listing 6 - Applicative typeclass definition.

Type of the pure function somewhat describes its function. It takes a value of type a, and encapsulates it into functor f, producing f a. On the other hand, the “<*>” operator type is similar to map function type, but instead of a -> b, it takes f (a -> b). That's why, in this paper, this function will be designated as a “supermap”

There are a couple of conditions that constructs need to satisfy in order to be considered applicative functors. Those conditions are [8]:

1. Identity: pure id `<*> v = v`;
2. Composition: pure `(.) <*> u <*> v <*> w = u <*> (v <*> w)`;
3. Homomorphism: pure `f <*> pure x = pure (f x)`;
4. Interchange: pure `f <*> pure x = pure (f x)`, and
5. pure `f <*> x = fmap f x`, especially important for mapping functions that receive multiple parameters.

3.1. APPLICATIVE FUNCTORS IN TYPESCRIPT

Implementation of applicative functors isn't much more difficult than their implementation in, for example, Elm [10]. In addition, because of the inability of TypeScript to statically infer the return data types in currying, the code TypeScript interpreter will throw warnings about type mismatch. Still, the code will work and produce the desired outputs.

3.1.1. Maybe applicative functor

The supermap function (Listing 7) returns `Nothing` if the passed value that a function is being mapped to is also `Nothing`. Otherwise, the function is taken out of the wrapper, and applied to the value inside the `Just` wrapper. For a value `x`, the pure function (Listing 7) returns `Just x`, because that is the minimal context that keeps the information about the original value [2]. One important thing to remember is, since applicative functors are largely used together with partial application, the function passed as a parameter must be curried. So, it will be necessary to modify the map function by adding the function currying.

One of the main benefits of applicative functors is applying a function of multiple parameters to values in a wrapper (Listing 8).

```
function maybeMap<T, U>(f: (x: T) => U, m: Maybe<T>): Maybe<U> {
    f = curry(f);  // IMPORTANT: this line is added compared to the implementation in
Listing 8
    if (m instanceof Just) {
        var {value} = m;
        return new Just(f(value));
    }
    return new Nothing<U>();
}
function maybePure<T>(x: T): Maybe<T> {
    return new Just(x);
}

function maybeSuperMap<T, U>(f: Maybe<(x: T) => U>, m: Maybe<T>): Maybe<U> {
    if (f instanceof Just) {
        var {value: func} = f;
        return maybeMap(func, m);
    }
    return new Nothing<U>();
}
```

Listing 7 - Implementation of pure and supermap functions for Maybe and modified map function in TypeScript.

```
function threeNumbers(a: number, b: number, c: number): number {
    return a * b + c**2;
}

let n1 = new Just(3);
let n2 = new Just(4);
let n3 = new Just(2);

let supermap1 = maybeSuperMap(maybePure(threeNumbers), n1);
let supermap2 = maybeSuperMap(supermap1, n2);
let supermap3 = maybeSuperMap(supermap2, n3);
console.log(supermap3);  // Prints Just: {“value”: 16}
```

Listing 8 - Example usage of the Maybe applicative in TypeScript.

3.1.2. List applicative functor

Lists are not only functors, but also applicative functors. Functors wrap regular values, while applicative functors wrap functions. So, a list applicative functor would be just a list of functions. The supermap function should then apply all those functions within a list, to a list of values. The supermap function combines the functions in the first list with the values in the second list by the Cartesian product. If the list of functions has m elements, and the list of values has n elements, then the resulting list must have $m \times n$ elements. This sort of behavior is necessary in order to satisfy the applicative functoriality conditions [5]. Second, if only some of the functions were applied to only some values, how would one choose the functions or the values that should be excluded from the product? Third, in category theory, a list can be considered an object with non-deterministic value (it contains multiple values simultaneously). So, the product of two lists, as they are non-deterministic values, should contain all possible combinations of all the values, which is exactly what Cartesian product is. The pure function, which needs to keep the minimal context of the original value, just puts the value into a singleton list. Implementation of the List applicative in TypeScript is a lot more difficult than the Maybe applicative, because of the `concatMap` function needed in the `supermap` implementation. The `concatMap` function, in turn needs the `concat` function to be implemented, which needs the `fold` function to be implemented, which needs the `joinLists` function to be implemented (Listing 9). All this requires a lot of work and functional programming knowledge, as well as good knowledge of TypeScript peculiarities.

```
function listMap<T, U>(f: (x: T) => U, l: List<T>): List<U> {
    f = curry(f);
    if (l instanceof Cons) {
        let {head, tail} = l;
        return new Cons(f(head), listMap(f, tail));
    }
    return new Empty<U>();
}
function fold<T>(product: (l: T, r: T) => T, accumulator: T, xs: List<T>): T {
    if (xs instanceof Cons<T>) {
        let {head, tail} = xs;
        return product(head, fold(product, accumulator, tail));
    }
    return accumulator;
}
function listAdd<T>(x: T, l: List<T>): List<T> {
    return new Cons(x, l);
}
function join<T>(l1: List<T>, l2: List<T>): List<T> {
    if (l1 instanceof Cons) {
        let {head, tail} = l1;
        return listAdd(head, join(tail, l2))
    }
    return l2;
}
function concat<T>(l: List<List<T>>): List<T> {
    return fold(join, new Empty<T>(), l);
}
function listConcatMap<T>(f: (x: T) => List<T>, l: List<T>): List<T> {
    return concat(listMap(f, l));
}
 listPure<T>(x: T): List<T> {
    return new Cons(x, Empty<T>);
}
function listSuperMap<T, U>(fs: List<(x: T) => U>, xs: List<T>): List<U> {
    fs = listMap(curry, fs);
    return listConcatMap(f => listMap(x => f(x), xs), fs);
}
```

Listing 9 - Implementation of the applicative functions for the List type in TypeScript. Implementation of necessary helper functions is also shown.

The `supermap` function applies all the functions in the first list to the values in the second one. This produces a list of lists, that is, a list of results for each of the functions for a specific value. This list of lists is then flattened to a single list using the `concat` function. The list `supermap` function is very useful for implementing the list comprehension mechanism. List comprehension is a mechanism to process lists in a quick and readable way, and it is common in languages known for their expressiveness and readability like Haskell and Python. TypeScript doesn't have the list comprehension mechanism, but it can be implemented, to an extent, using the `supermap` function for lists (Listing 10).

With all the necessary functions implemented, the List applicative can be used in the same way as it would be in Haskell (Listing 10). It is up to the programmers themselves to decide whether the benefits of the List applicative outweigh the effort necessary to implement it. In general, if there is a combining function f that combines *n* elements from *n* lists, the lists can be combined if it has the following type:

```
function combine<T>(x1: T, x2: T, …, xn: T)
```

The combination would then be achieved by chaining a sequence of *n* `supermap` calls similar to the one in Listing 29.

```
function combine<T>(a: T, b: T) {
    return new Cons(a, new Cons(b, new Empty<T>()));
}

let l_left = new Cons(1, new Cons(2, new Empty<number>()));
let l_right = new Cons(11, new Cons(12, new Empty<number>()));
let _cartesian = listSuperMap(listPure(combine), l_left);
let cartesian = listSuperMap(_cartesian, l_right);
```

Listing 10 - Example implementation of Cartesian product using the List applicative functor in TypeScript.

4. CONCLUSION

This paper provided theoretical explanations of the Functor and Applicative Functor concepts in category theory. It then provided their implementations TypeScript, focusing on Maybe and List functors. Their use-cases were then shown, where it was demonstrated that they could be useful, especially when abstracting complex concepts, so they could be used and invoked on-demand, usually with a simple command. This paper had also shown that, because TypeScript is a multi-paradigm language, the implementation of these concepts is not significantly more difficult, or different, to their implementation in Haskell, and the only drawback of TypeScript is that the return types of some functions cannot be statically type-checked. Thus, for a programmer that is well-versed in functional programming and wants to speed up or improve some of their work, TypeScript provides a good support for implementation of functional programming concepts.

5. REFERENCES

[1] N. Galov, "A Dive Into the Ocean of Web Design Statistics in 2022," 30 March 2023. [Online]. Available:https://webtribunal.net/blog/web-design-statistics/#gref. [Accessed 22 April 2023].

[2] M. Lipovaca, Learn you a Haskell for Great Good: A Begginer's Guide, No Starch Press, 2011.

[3] X. C., "StackOverflow," 22 October 2009. [Online]. Available: https://stackoverflow.com/a/1604828. [Accessed 22 August 2022].

[4] S. MacLane, Categories for the Working Mathematician, 2nd ed., Springer, 1971.

[5] B. Milewski, Category Theory for Programmers, Boston: Self-published, 2018.

[6] J. N., Basic Algebra, 2nd ed., vol. II, Dover Books, 2009.

[7] H. Curry and F. Robert, Combinatory Logic, 2nd ed., vol. I, Amsterdam, Netherlands: North-Holland Publishing Company, 1958.

[8] M. Matovic, Analysis of Category Theory Concepts in Elm framework, Novi Sad: Faculty of Technical Sciences, 2022.

[9] C. McBride and R. Paterson, "Applicative Programming with Effects," *Journal of Functional Programming*, 1 January 2008.

[10] E. Czaplicki, "The Elm Programming Language," [Online]. Available: https://guide.elm-lang.org. [Accessed 5 September 2022].