



THE 21ST EDITION OF THE INTERNATIONAL CONFERENCE
EUROPEAN INTEGRATION
REALITIES AND PERSPECTIVES

Categorical Thinking in Modern Programming: A TypeScript Based Approach

Radu Bucea-Manea-Țoniș¹

Abstract: This paper explores the fundamental connections between category theory and modern programming languages, focusing on how Cartesian Closed Categories (CCC) provide a rigorous semantic foundation for functional computation. We begin by introducing the core categorical notions-objects, morphisms, products, exponentials - and show how untyped lambda calculus and Church numerals naturally embed into CCCs. Building on this theoretical basis, the paper examines how advanced TypeScript features reflect categorical structures. Structural types behave as objects in a category of types, while function types correspond to exponentials. We analyze three central TypeScript mechanisms - Distributive Types, Indexed Access Types, and Mapped Types-and reinterpret them through categorical semantics: distributive types as functorial behavior over coproduct-like structures, indexed access types as evaluation and projection morphisms, and mapped types as endofunctors acting on the category of TypeScript types. By synthesizing these perspectives, we show that TypeScript's type system implicitly forms a CCC like environment, where categorical reasoning clarifies the expressive power and limitations of advanced type level programming. The paper provide a categorical perspective for understanding type transformations and concludes by highlighting how categorical abstractions unify these mechanisms.

Keywords: Category Theory; Cartesian Closed Categories; Lambda Calculus; TypeScript; Type-Level Programming

JEL Classification: C02; C63; C88; O33

1. Introduction

There are fundamental correspondences between notions such as computability (data type, program), logic (proposition, proof), and category theory (objects, morphisms). A basic category is defined by the following, according to (Smith, 2025):

- A. a collection of objects, usually interpreted as types. For example, in TypeScript we have types like: string, number, {a: number}, etc.;
- B. a collection of morphisms between objects, interpreted as functions or transformations between types, for example, $(x: A) \Rightarrow B$;

¹ Associate Professor, Ph.D., Danubius International University of Galati, Romania, Address: 3 Galati Blvd., Galati 800654, Romania, Corresponding author: radumanea@univ-danubius.ro.



Copyright: © 2026 by the authors.
Open access publication under the terms and conditions of the
Creative Commons Attribution-NonCommercial (CC BY NC) license
(<https://creativecommons.org/licenses/by-nc/4.0/>)

C. an operation of composing morphisms, satisfying the laws of associativity and identity.

The first purely logical programming language is the lambda calculus:

- data and programs are the same thing;
- we only have functions and the application operation.

The following term can be read informally as: $f(x, g(x, y))$, taking into account that x is one parameter and $g(x, y)$ has two parameters:

$$\lambda f. \lambda g. \lambda x. \lambda y. f(x)(g(x)(y))$$

According to Scott (1998), a closed Cartesian category (CCC) provides a categorical framework for the semantics of the lambda-calculus with types. Such a category has:

- a terminal object, usually denoted 1 , which models the unit type;
- Cartesian products, represented by pairs $A \times B$ and their canonical morphisms;
- exponential objects B^A (also written $A \rightarrow B$), which model functional types;
- curry and uncurry operations, which establish the fundamental isomorphism $\text{Hom}(A \times B, C) \cong \text{Hom}(A, C^B)$, this means a function receiving a pair of arguments (a function and a variable) in some context, returns another one by applying that function (binding) to that specific variable in a different context.

Church numerals are functional representations of the natural numbers:

- ✓ $\lambda f. \lambda x. x = 0$
- ✓ $\lambda f. \lambda x. f(x) = 1$
- ✓ $\lambda f. \lambda x. f(f(x)) = 2$
- ✓ ...

For example, the following term corresponds to the Church numeral for 3:

$$\lambda f. \lambda x. f(f(f(x)))$$

TypeScript's type system aligns with ideas from categorical mathematics, specifically with CCC (cartesian closed categories) — the mathematical structure that underlies the functional approach to programming languages. A category is Cartesian closed if it has a terminal object and supports the Cartesian product as well as the exponential form of a function's application.

The identity function is $\lambda x. x$ and can also be written as $f(x) = x$. In TypeScript, we have `void` or `{}`, depending on the interpretation.

According to (Milewski, n.d.), for Cartesian products, a morphism $f : C \rightarrow (A \times B)$ is equivalent to a pair of functions: $g : C \rightarrow A$, $h : C \rightarrow B$.

For the explicit construction in the category of sets, we will have the following:

- pairing: $\langle g, h \rangle(c) = (g(c), h(c))$;
- pojections: $g = \pi_1 \circ f(c)$, $h = \pi_2 \circ f(c)$.

For example, in TypeScript, we have object types (record types):

type Pair = { x: number; y: string };

The next isomorphism can summarize this:

$$\text{Hom}(C, A \times B) \cong \text{Hom}(C, A) \times \text{Hom}(C, B).$$

In the case of fundamental isomorphism for functional types, the following standard definitions concern the currying and uncurrying operations, according to (Thompson, 1999):

- $\text{curry}(f) = \lambda a. \lambda b. f(a, b)$
- $\text{uncurry}(g) = \lambda(a, b). g(a)(b)$

For function types, in TypeScript we have:

type $F = (x: A) \Rightarrow B$;

, which means that TypeScript can be interpreted as a closed Cartesian category with some practical limitations.

2. Methodology

Recursive and structural types transform the TypeScript language into a mini-lambda calculus based on types, but interpreted as a flow of operations, according to (TypeScript, 2025).

2.1. Structural Types in TypeScript

TypeScript is structural rather than nominal. This means that two types are compatible if they have the same shape, not the same name.

For example, the following script implements pattern matching based on shape, not identity, after (TypeScript Team, 2025):

```
interface A { x: number }
interface B { x: number }
```

```
let a: A = { x: 1 };
let b: B = a; // OK — they have the same structure
```

2.2. Recursive Types

A recursive type is a type that refers to itself (Fiadeiro, 2005), being useful for constructing trees, JSON files, nested lists, ASTs, and operationally finite types.

For example, the following tree structure can expand infinitely:

```
interface Tree {
  value: number;
  children: Tree[];
}
```

The following example uses a classic recursive type (generic JSON):

```

type JSONValue =
  | string
  | number
  | boolean
  | null
  | JSONValue[]
  | { [key: string]: JSONValue };

```

Here, we can observe recursion applied in two directions:

- array of JSONValue
- JSONValue object

A nested array can be implemented like this:

```
type NestedArray<T> = T | NestedArray<T>[];
```

, and its initialization could be of the type:

```
const xs: NestedArray<number> = [1, [2, [3, 4]], 5];
```

2.3. Distribution by Types

A universal property defines the categorical coproduct:

$$F(A + B) \cong F(A) + F(B)$$

This occurs when TypeScript applies a separate conditional type to each member of a union. According to (Gibbok, 2025), a conditional type is of the form $T \text{ extends } U ? X : Y$. If T is a union, TypeScript “breaks” it into pieces and applies the rule to each part, for example:

```
type ToArray<T> = T extends any ? T[] : never;
```

, and in the following instruction:

```
type R = ToArray<string | number>;
```

The following types are converted according to the rule:

- $\text{string} \rightarrow \text{string}[]$
- $\text{number} \rightarrow \text{number}[]$

Therefore, we obtain the following distribution:

```
type R = string[] | number[];
```

, particularly useful when performing type-level transformations, when applying mappings over union types, and when constructing filters that refine or exclude specific branches of a type.

Flattening a string can be done recursively after applying the adjoint with infer:

```

type Flatten<T> =
  T extends (infer U)[]
    ? Flatten<U>
    : T;

```

In the following example, TypeScript becomes a type-oriented transformation engine:

```
type A = Flatten<number[][][]>; // number
type B = Flatten<[1, [2, [3]]]>; // 1 | 2 | 3
```

2.4. Index Access Types

Type indexing allows extracting the type of a property from an object using the syntax $T[K]$, where K is the key, as described in (Goldberg, 2025). For example:

```
type User = {
  id: number;
  name: string;
};
, where:
type NameType = User["name"]; // string
```

It can also index over unions, for example:

```
type IdOrName = User["id" | "name"]; // number | string
```

It is also possible to recursively extract the key of all nodes in a tree with `infer` (with the categorical meaning of “right adjoint”), as in the following example:

```
type Node = {
  name: string;
  children?: Node[];
};
type AllNames<T> =
  T extends { name: infer N; children?: infer C }
    ? N | (C extends Node[] ? AllNames<C[number]> : never)
    : never;
```

, where the `Names` type will consist of a union of values:

```
type Names = AllNames<{
  name: "root";
  children: [
    { name: "a" },
    { name: "b", children: [{ name: "c" }] }
  ];
}>; // "root" | "a" | "b" | "c"
```

One can create a type that validates whether an object is “partially JSON”, “partially tree”, etc.

Recursive types can be used for pattern matching on structures. For example, the following code extracts all values from a nested object:

```
type Values<T> =
  T extends object
    ? Values<T[keyof T]>
    : T;
type V = Values<{
```

```
a: { x: 1; y: 2 };
  b: { z: 3 };
}>; // 1 | 2 | 3
```

In conclusion, indexing proves valuable in several ways: it enables the extraction of types from complex structures, supports systematic reuse while maintaining consistency, and allows new types to be derived automatically from existing ones.

2.5. Representation of types (Mapped Types)

According to (The Concise TypeScript book, 2025), mapping means creating a new type by iterating over the keys of another type.

The general syntax is `{[K in Keys]: SomeType}` and applies as follows:

```
type User = {
  id: number;
  name: string;
};
type OptionalUser = {
  [K in keyof User]?: User[K];
};
```

, the result being an object with all the optional properties:

```
type OptionalUser = {
  id?: number;
  name?: string;
};
```

The following example converts all fields to a recursively-optional version:

```
type DeepPartial<T> = {
  [K in keyof T]?: T[K] extends object
    ? DeepPartial<T[K]>
    : T[K];
};
```

Therefore, this form of representation becomes particularly valuable when defining utility types such as `Partial`, `Required`, or `Readonly`, when performing automatic structural transformations on existing types, and when deriving typed APIs directly from underlying data models.

3. Results

The following three concepts in TypeScript correspond to categorical constructs:

3.1. Distribution by Types

Unions ($A \mid B$) in TypeScript are the analog of coproducts in theory. Distributive conditional types like $T \text{ extends } U ? X : Y$, behave like a functor that acts on each component of its coproduct, in categorical terms:

$$F(A + B) = F(A) + F(B)$$

, which is exactly what TypeScript does when it casts over unions.

3.2. Type Indexing

The indexing ($T[K]$) corresponds to the idea of projection into a Cartesian product. For a categorical product ($A \times B$), we have two morphisms:

$$\text{I. } \pi_1: A \times B \rightarrow A$$

$$\text{II. } \pi_2: A \times B \rightarrow B$$

In TypeScript:

```
type User = { id: number; name: string };
type Name = User["name"]; //  $\pi_2$  projection
```

3.3. Representation (Mapped Types)

Mapped types represent endomorphic functors on the category of types. For example, the expression:

```
type Mapped<T> = { [K in keyof T]: Something<T[K]> };
```

, exactly reflects the behavior of an endofunctor in the sense that it accepts an object (type), applies a structured transformation, and preserves the form (key structure). In categorical terms, this means:

$$F : C \rightarrow C$$

, where F structurally maps objects and morphisms.

4. Conclusion

TypeScript functions not merely as a pragmatic type system, but as a language whose structural properties align closely with the categorical semantics traditionally associated with functional programming. Although it is not explicitly grounded in the formalism of Cartesian Closed Categories, its type discipline behaves as if types were objects, functions were morphisms, and type-forming operations reproduced the behavior of categorical products and coproducts. Moreover, the expressive power of mapped and conditional types exhibits a form of functorial behavior, including distributivity patterns characteristic of functorial action over coproducts. Consequently, TypeScript proves conceptually compatible with categorical modeling, even in the absence of an explicit categorical foundation.

References

- Fiadeiro, J. L. (2005). *Categories for software engineering*. Springer. <https://doi.org/10.1007/b138392>
- Goldberg, J. (2025). *Learning TypeScript*. <https://www.learningtypescript.com>
- Gibbok. (2025). *The Concise TypeScript book: A concise guide to effective development in TypeScript (Version 6.0)*. GitHub. <https://github.com/gibbok/typescript-book>
- Milewski, B. (n.d.). *Category theory for programmers*. https://ai.dmi.unibas.ch/research/reading_group/milewski-2023-01-30.pdf
- Smith, P. (2025). *Introducing category theory (2nd ed.)*. Logic Matters. <https://www.logicmatters.net/resources/pdfs/SmithCat.pdf>
- Scott, P. J. (1998). *Some aspects of categories in computer science*. Department of Mathematics, University of Ottawa.
- The Concise TypeScript Book. (2025). *Introduction*. <https://gibbok.github.io/typescript-book>
- Thompson, S. (1999). *Type theory and functional programming*. Computing Laboratory, University of Kent.
- TypeScript. (2025). *The starting point for learning TypeScript*. TypeScript Documentation. <https://www.typescriptlang.org>
- TypeScript Team. (2025). *TypeScript handbook: Structural typing and interfaces*. Microsoft. <https://www.typescriptlang.org/docs/>