I. J. Mathematical Sciences and Computing, **2025, 3, 1-18**
Published Online on October 8, 2025 by MECS Press (http://www.mecs-press.org/)
DOI: 10.5815/ijmsc.2025.03.01

Bridging Category Theory and Functional Programming for Enhanced Learning

Fethi Kadhi*
University of Manouba, National School of Computer Science, Manouba, Tunisia
E-mail: fethi.kadhi@ensi.uma.tn
ORCID iD: https://orcid.org//0000-0001-5173-5363
*Corresponding Author

Received: 23 October, 2024; Revised: 26 February, 2025; Accepted: 18 August, 2025; Published: 08 October, 2025

Abstract: Functional programming is frequently taught in isolation from its mathematical roots, particularly category theory, leading to a fragmented understanding for students. Simultaneously, category theory is often perceived as too abstract and difficult to grasp, despite its foundational role in programming. This gap between theory and practice creates barriers for students, preventing them from fully appreciating the deep connections between functional programming and its underlying mathematical structures. Although there are resources aimed at bridging this divide, such as works by Milewski, MacLane, and Leinster, they often either lack practical examples or fail to delve deeply into the mathematical rigor required for a comprehensive understanding of category theory. This paper presents a novel pedagogical approach that integrates category theory with functional programming in a unified and accessible framework. By leveraging monadic programming, particularly through the list and maybe monads, we offer concrete examples of how abstract mathematical concepts can address real-world programming challenges, such as handling missing data. Our approach builds on and generalizes Dayou Jiang's method of using programming to teach partially ordered relations. In doing so, we concurrently teach functional programming and category theory, making the abstract more tangible and applicable. This interdisciplinary method not only enhances comprehension of both fields but also aligns with contemporary educational reforms that prioritize integrated learning across mathematical and computational domains.

Index Terms: Category Theory, Functional Programming, Modern Education, Monads, Interdisciplinary Learning, Computer Science Pedagogy

1. Introduction

In many computer science programs, functional programming is often taught in isolation from its underlying mathematical foundations, leaving students with a fragmented and incomplete understanding of the subject. This lack of a structured, theoretical framework makes it difficult for students to engage with functional programming principles in a meaningful and systematic way. At the same time, category theory, which provides a deep and rigorous mathematical foundation for functional programming, is often viewed as overly abstract and challenging to comprehend. Despite the availability of resources aimed at introducing category theory to learners with a basic mathematical background [1, 2, 3], many students still struggle to grasp its relevance and practical application, particularly in the context of programming.

Works such as [4] offer valuable insights into the intersection of category theory and programming. However, these resources often lack the depth needed to fully explore the mathematical rigor of category theory and its direct application to functional programming. As a result, students are limited in their ability to fully leverage the strengths of both fields, missing out on the profound connections between abstract mathematics and practical programming.

This paper aims to address this gap by presenting a pedagogical approach that integrates category theory with functional programming in a unified and accessible framework. Our goal is to provide students with a structured method for exploring functional programming while simultaneously offering practical insights into the abstract principles of category theory. Through concrete examples, we demonstrate how monadic programming particularly using the list and Maybe monads can be applied to real world programming challenges such as handling missing data. In doing so, this approach generalizes Dayou Jiang’s method of teaching partially ordered relations through programming [5]. Many educational resources have adopted this approach, using programming languages to teach fundamental mathematical and scientific concepts.

Our method can be seen as a generalization of [5]. First, because partially ordered sets are small categories, and second, because we apply the same approach in both directions, teaching category theory and functional programming simultaneously.

The paper is organized as follows: Section 2 outlines a multidisciplinary pedagogical approach designed to bridge the significant epistemological gap between the abstract theory of category theory and the concrete practice of functional programming. It identifies a dual challenge: mathematics students struggle with the motivation, verification, and application of abstract concepts, while programming students face hurdles in understanding the mathematical abstractions and laws behind functional constructs.

The proposed solution uses Haskell as a tool for instrumental genesis, transforming passive learning into active construction. This approach is grounded in precedents from computational mathematics (e.g., Mathematica, R) and directly addresses the core challenges by providing immediate, practical motivation, objective verification through the compiler, and inherent application of theory. The method is designed to be accessible to students with basic mathematical maturity and varying backgrounds, leveraging the synergy between the two domains to foster a deep, integrated understanding.

In Section 3, we cover the essential concepts of category theory, including the definition of categories, functors, natural transformations, monads, and the Yoneda Lemma. Each of these topics is introduced with examples to make the abstract theory more accessible. Section 4 shifts the focus to functional programming, particularly the Haskell language. We explore how category theory applies to Haskell, treating it as a category (Hask), and examine how functors and monads manifest in this context. We also address practical issues, such as handling missing data using monadic structures. By the end of this section, we further deepen our understanding of category theory through the application of the Yoneda Lemma to the category Hask.

Section 5 offers a conclusion, emphasizing the potential for this interdisciplinary approach to enhance students' engagement with both functional programming and category theory. By integrating these two domains, students gain a deeper understanding of both mathematical principles and their practical applications in programming.

2. A Multidisciplinary Pedagogical Approach

2.1 The Dual Pedagogical Challenge: Abstract Theory and Concrete Practice

In contemporary education, multidisciplinary approaches are recognized as key drivers of innovation, yet they introduce unique complexities. This work is situated at the intersection of two particularly challenging fields: the abstract mathematics of category theory and the applied practice of functional programming. The core difficulty is a bidirectional **epistemological gap** that hinders effective learning in both domains.

From the **mathematical perspective**, students struggle to form operational intuition for highly abstract concepts (e.g., functors, monads, natural transformations) when taught through traditional, passive, axiomatic methods. This manifests as:

- **The Motivation Problem:** Students perceive definitions as arbitrary formalisms, asking, "Why is this important?"
- **The Verification Problem:** Unlike solving an equation, reasoning with commutative diagrams offers no mechanism for self-correction, leading to reliance on external validation.
- **The Transfer Problem:** Students fail to apply theoretical knowledge to practical computational domains where these ideas are influential.

Conversely, from the **programming perspective**, teaching functional programming through a categorical lens presents its own set of obstacles. Students, especially those with imperative backgrounds, must bridge the gap between concrete code and abstract mathematical structures:

- **The Abstraction Hurdle:** Students must learn to see types not just as data containers but as *objects* in a category, and functions not as procedures but as *morphisms* defined by compositional behavior.
- **The Dual Interpretation Challenge:** Concepts like functors must be understood simultaneously as mathematical constructs that preserve structure and as typeclasses with specific methods and laws.
- **The Lawful Abstraction Problem:** Beyond writing code that compiles, students must learn to create abstractions that obey mathematical laws (e.g., functor laws, monad laws), a requirement unfamiliar to most programmers.

This dual challenge exists because the prevailing pedagogical model for both subjects remains one of **passive reception** rather than **active construction**. Students are shown concepts but are given no tools to build with them. Our interdisciplinary approach integrating Haskell with category theory is specifically designed to bridge this gap. We use each domain to illuminate the other: functional programming provides the concrete, executable environment missing from abstract mathematics, while category theory provides the conceptual framework that reveals the deeper structure

and laws underlying functional code. This creates a pedagogical synergy that transforms the challenge of teaching two difficult subjects into an opportunity for developing profound, integrated understanding.

2.2 Our Solution: Haskell as a Tool for Instrumental Genesis

Our interdisciplinary approach integrating functional programming (Haskell) with category theory is the deliberate solution to this problem. It is not a mere juxtaposition of subjects; it is a pedagogical strategy that uses programming as an **instrumental genesis** tool, providing the missing constructive framework.

2.3 Precedents in Computational Mathematics

This methodology is firmly supported by established practices in other mathematical disciplines:

- Tools like **Mathematica** are indispensable in numerical analysis, enabling interactive exploration of iterative methods and approximations [6, 7].
- The language **R** is pivotal in probability and statistics, allowing students to manipulate data and bring stochastic processes to life [8, 9].

These tools succeed by transforming passive learning into active experimentation. Category theory has lacked a similar widely-adopted computational tool. We argue that **Haskell** fills this void. Inspired by categorical principles, it provides a formal, executable environment where theoretical constructs are not just illustrated but are *necessitated* by the language's structure.

2.4 A Comparative Framework: Bridging Theory and Practice

The following table contrasts the traditional approach with our proposed method, demonstrating how it directly targets each facet of the identified gap:

Table 1. Comparison of traditional and interdisciplinary approaches to teaching category theory

Pedagogical Challenge	Traditional Approach (The Gap)	Our Interdisciplinary Solution
Motivation Problem	Abstract justification. "It's a foundational theory."	**Concrete Necessity.** A Monad is not an option; it is the pattern required to handle I/O, state, and failure in a pure language. Purpose is immediately demonstrated.
Verification Problem	Theoretical exercises graded by an instructor. Feedback is delayed and subjective.	**Immediate, Objective Feedback.** Code compiles and behaves correctly, verifying understanding, or it fails, highlighting a misconception. The compiler becomes an automated tutor.
Transfer Problem	Theory is taught in a vacuum, with hoped-for later application.	**Theory is Taught Through Application.** Students learn natural transformations by writing functions that convert between Haskell functors. Application is inherent.

2.5 Pedagogical Benefits and Observed Outcomes

This approach has a profound impact: it bridges theory and practice, allows students to visualize and simulate abstract ideas in real-time, and sharpens critical thinking by applying mathematical reasoning to concrete problems. Our experience at confirmed its efficacy, resulting in markedly higher attendance and deeper in-class engagement. The resultant dual mastery of both category theory *and* functional programming is not the primary goal but a powerful outcome of an efficient educational intervention.

2.6 Target Audience and Prerequisites

To ensure accessibility and maximum effectiveness, our pedagogical approach is designed for students with a foundational yet flexible background. The primary prerequisite is not advanced expertise in either field, but rather a level of mathematical maturity typically gained through early undergraduate studies in computer science, mathematics, or a related discipline. Specifically, students should be comfortable with basic algebraic structures and logical reasoning; a prior encounter with fundamental concepts like sets, functions, and relations is sufficient. No prior knowledge of category theory itself is assumed or required. In parallel, while experience with any programming language is beneficial, proficiency in functional programming is not a prerequisite. The course is structured to introduce Haskell's syntax and core paradigms from the ground up. This design allows students who are stronger in mathematics to grasp programming concepts through theoretical lenses, and students with a programming background to solidify their practical skills by understanding their formal mathematical underpinnings. Essentially, the course meets students at their individual points of strength and uses the synergy between the two domains to bootstrap understanding in the other, making it accessible to a broad range of learners. Some students naturally gravitate toward theoretical and abstract thinking, while others prefer focusing on practical solutions without delving deeply into the underlying frameworks of the tools they use. What distinguishes this approach is that it fosters a balanced learning environment where each group can benefit from the strengths of the other: abstract thinking enriches practical work with deeper insights, while a practical orientation

brings theoretical ideas to life. In this way, the diversity of orientations is not an obstacle but a source of mutual enrichment that strengthens comprehensive understanding and enhances the dynamics of collective learning.

2.7 Roadmap for Implementation

The following sections extend this approach through concrete Haskell examples, demonstrating how category theory provides elegant solutions to real-world programming challenges, offering students a unique opportunity to deepen their understanding in an engaging and practical way.

3. Some Elements of Category Theory

This section introduces the fundamental concepts of category theory in a clear and accessible manner, suitable for readers with varying levels of mathematical background. We will focus on a foundational understanding of categories, limiting ourselves to the essential elements needed for the subsequent sections. For a more detailed exploration, we recommend *Category Theory for the Working Mathematician* by Saunders MacLane [1].

3.1 Category

3.1.1 Definition

A category $\mathcal{A}$ consists of four main components and satisfies two key axioms:

A collection of objects $ob(\mathcal{A})$. Objects can be anything from sets, numbers, or even more abstract entities, depending on the context of the category.

For $a, b \in ob(\mathcal{A})$, a collection $\mathcal{A}(a, b)$ of arrows (also called morphisms) from a to b. These arrows represent transformations between objects, similar to how functions work in set theory.

A composition law: for $a, b, c \in ob(\mathcal{A})$,

$$\begin{array}{lll} \mathcal{A}(b,c) \times \mathcal{A}(a,b) & \rightarrow & \mathcal{A}(a,c) \\ (g,f) & \mapsto & g \circ f \end{array} \tag{1}$$

This law defines how arrows are composed. Just as functions can be composed in mathematics (applying one function after another), arrows in a category follow this rule.

For each object $a \in \mathcal{A}$, there exists a unique arrow 1_a, called the identity arrow, that relates a to itself. This identity arrow behaves just like the identity function in mathematics, where applying it doesn't change the object.

Categories must satisfy two fundamental axioms:

Associativity: For each $f \in \mathcal{A}(a, b)$, $g \in \mathcal{A}(b, c)$, and $h \in \mathcal{A}(c, d)$, we have

$$(h \circ g) \circ f = h \circ (g \circ f) \tag{2}$$

This means that no matter how arrows are grouped when composing them, the result will be the same.

Identity Law: For each $f \in \mathcal{A}(a, b)$, we have

$$f \circ 1_a = f = 1_b \circ f \tag{3}$$

The identity law ensures that applying the identity arrow to any arrow leaves the arrow unchanged, much like multiplying by 1 in arithmetic.

3.1.2 Examples

1. The smallest category is $\emptyset$, which contains no objects or arrows. This is a trivial case of a category with no structure.
2. A category with a single object and only its identity arrow is denoted **1**:

$$\bullet \tag{4}$$

3. A category with two objects linked by an arrow can be depicted as:

$$\bullet \rightarrow \bullet \tag{5}$$

4. The fundamental example of a category is **Set**, whose objects are sets and whose arrows are functions between sets. This is a foundational category in mathematics, where the objects are familiar entities (sets), and the arrows are transformations (functions) between them.

3.2 Functor

3.2.1 Definition

Let $\mathcal{A}$ and $\mathcal{B}$ be two categories. A functor $F: \mathcal{A} \to \mathcal{B}$ is a mapping between categories that preserves their structure:

- It maps each object in $\mathcal{A}$ to an object in $\mathcal{B}$:

$$\begin{array}{ccc} ob(\mathcal{A}) & \to & ob(\mathcal{B}) \\ a & \mapsto & F(a) \end{array} \tag{6}$$

- It also maps each arrow (or morphism) in $\mathcal{A}$ to an arrow in $\mathcal{B}$:

$$\begin{array}{ccc} \mathcal{A}(a, a') & \to & \mathcal{B}(F(a), F(a')) \\ f & \mapsto & F(f) \end{array} \tag{7}$$

This ensures that arrows between objects in category $\mathcal{A}$ are mapped to arrows between the corresponding objects in category $\mathcal{B}$. For instance, if there is a morphism f from object a to object a', then $F(f)$ is a morphism from $F(a)$ to $F(a')$.

A functor must satisfy the following two properties to preserve the structure of the categories:

- For any arrows $f: a \to a'$ and $g: a' \to a''$ in $\mathcal{A}$, we must have:

$$F(g \circ f) = F(g) \circ F(f) \tag{8}$$

This property ensures that functors respect composition of arrows. In other words, if you compose two morphisms in $\mathcal{A}$, the result must be the same as applying the functor to each morphism individually and then composing the results in $\mathcal{B}$.

- For each object a in $\mathcal{A}$, the identity arrow 1_a must be mapped to the identity arrow of $F(a)$ in $\mathcal{B}$:

$$F(1_a) = 1_{F(a)} \tag{9}$$

This guarantees that the identity arrow in $\mathcal{A}$, which does nothing to its object, is mapped to the identity arrow in $\mathcal{B}$, maintaining the identity property across categories.

3.3 Natural Transformation

3.3.1 Definition

Let F and G be two functors from a category $\mathcal{A}$ to a category $\mathcal{B}$. A natural transformation $\alpha: F \Longrightarrow G$ is a way of transforming one functor into another, while preserving the structure of the categories involved.

A natural transformation assigns to each object $a \in \mathcal{A}$ an arrow $\alpha_a: F(a) \to G(a)$ in $\mathcal{B}$. This must satisfy the following condition: for every morphism $f: a \to a'$ in $\mathcal{A}$, we have

$$\alpha_{a'} \circ F(f) = G(f) \circ \alpha_a \tag{10}$$

In other words, applying the functor F to f and then applying $\alpha_{a'}$ must give the same result as applying α_a and then applying the functor G to f. This ensures that natural transformations respect the structure of the functors and the morphisms in the category.

3.3.2 Examples

1. Consider a morphism $f: a \to a'$ in $\mathcal{A}$. This induces a natural transformation $H_f: H^{a'} \Rightarrow H^a$, where H^a and $H^{a'}$ are hom-functors. For each object $b \in \mathcal{C}$, the component $(H_f)_b$ maps an arrow $g \in \mathcal{C}(a', b)$ to $g \circ f \in \mathcal{C}(a, b)$. The commutativity of the naturality square follows from the associativity of arrow composition in categories. This example shows how a natural transformation relates two hom-functors by "shifting" arrows via composition.

2. Another example of a natural transformation is the inclusion map from a set X to its power set $P(X)$, denoted by $\eta_X: X \to P(X)$. For each element $x \in X$, $\eta_X(x)$ is the singleton set $\{x\}$. This defines a natural transformation $\eta: 1_{\mathbf{Set}} \Longrightarrow P$, where $1_{\mathbf{Set}}$ is the identity functor on $\mathbf{Set}$ and P is the power set functor. For any function $f: X \to Y$,
we have:

$$P(f)(\{x\}) = \{f(x)\} \tag{11}$$

3.4 Monad

3.4.1 Definition

A monad on a category $\mathcal{C}$ consists of a triple (T, η, μ), where:

- $T: \mathcal{C} \to \mathcal{C}$ is an endofunctor, meaning that T maps objects and morphisms in $\mathcal{C}$ to objects and morphisms within the same category.
- $\eta: 1_{\mathcal{C}} \Rightarrow T$ is a natural transformation, called the **unit** of the monad, which assigns to each object $a \in \mathcal{C}$ an arrow $\eta_a: a \to T(a)$. This maps each object to a "wrapped" version of itself within the structure provided by T.
- $\mu: T^2 \Rightarrow T$ is a natural transformation, called the **multiplication** of the monad. It "flattens" two applications of T into one. More formally, for each object $a \in \mathcal{C}$, $\mu_a: T(T(a)) \to T(a)$ combines two layers of the structure into one.

These structures must satisfy two key axioms, which ensure coherence and consistency:

- Left identity law: For any object $a \in \mathcal{C}$, we have:

$$\mu_a \circ \eta_{T(a)} = 1_{T(a)} \tag{12}$$

This condition ensures that applying the unit η and then "flattening" with μ gives back the original structure.

- Right identity law: For any object $a \in \mathcal{C}$,

$$\mu_a \circ T(\eta_a) = 1_{T(a)} \tag{13}$$

This ensures that applying T to the unit and then "flattening" with μ also returns the original structure.

- Associativity law: For any object $a \in \mathcal{C}$, the following diagram must commute:

$$\mu_a \circ \mu_{T(a)} = \mu_a \circ T(\mu_a) \tag{14}$$

3.4.2 Examples

1. **Powerset Monad** Consider the power set functor $P: \mathbf{Set} \to \mathbf{Set}$, which assigns to each set X the set of all its subsets $P(X)$. The monad structure on P consists of:

- Unit: The unit $\eta_X: X \to P(X)$ maps each element $x \in X$ to the singleton subset $\{x\}$:

$$\eta_X(x) = \{x\} \tag{15}$$

This embeds each element of X as a subset of X.

- Multiplication: The multiplication $\mu_X: P(P(X)) \to P(X)$ takes a set of subsets $\{A_i\}$ and returns their union:

$$\mu_X(\{A_i\}) = \bigcup_i A_i \tag{16}$$

This operation "flattens" a collection of subsets into a single set by taking their union.

The unit and multiplication satisfy the monad axioms, ensuring that embedding a set into its power set and then taking unions returns the original set. The associativity condition ensures that repeated applications of the power set and union behave in a consistent way.

2. **The Maybe Monad** The Maybe monad can be defined as a functor $T: \mathbf{Set} \to \mathbf{Set}$ that assigns to each set X the set $T(X) = X \cup \{*\}$, where $*$ is a new element representing a possible failure or absence of information.

The monad structure on T consists of:

- Unit: The unit $\eta_X: X \to T(X)$ maps each element $x \in X$ to itself in $T(X)$:

$$\eta_X(x) = x \tag{17}$$

preserving successful values and adding the possibility of failure.

- Multiplication: The multiplication $\mu_X: T(T(X)) \to T(X)$ "flattens" nested occurrences of the failure or success condition:

$$\mu_X(*) = *, \quad \mu_X(x) = x \text{ if } x \in X \tag{18}$$

This monad is often used in contexts where computations may "fail" or return a result that represents the absence of a value.

3.5 Yoneda Lemma

The Yoneda Lemma is a fundamental result in category theory that provides deep insights into how objects relate to other objects in a category. It allows us to fully understand an object by examining how it interacts with other objects through morphisms (arrows).

Let $\mathcal{C}$ be a locally small category, and let $a \in \mathcal{C}$ be an object. Suppose $X: \mathcal{C} \to \mathbf{Set}$ is a functor that maps objects in $\mathcal{C}$ to sets. The Yoneda Lemma asserts that the set of natural transformations from the hom-functor H^a (which represents morphisms from a to other objects) to X is naturally isomorphic to $X(a)$:

$$[\mathcal{C}, \mathbf{Set}](H^a, X) \cong X(a) \tag{19}$$

This isomorphism is natural in both X and a. Essentially, the Yoneda Lemma tells us that an object is fully determined by its relationships (morphisms) with other objects in the category.

3.5.1 Consequences and Intuition

A key consequence of the Yoneda Lemma is that if two objects a and b have hom-functors that are naturally isomorphic (i.e., $H^a \cong H^b$), then $a \cong b$, meaning the objects themselves are isomorphic. This highlights how the lemma reduces the study of objects to the study of their relationships.

The intuition behind the Yoneda Lemma is that if two objects "see" the rest of the category in the same way, through the same morphisms, then they must be the same object. This shifts the focus from internal properties of objects to how they relate to others, providing a powerful conceptual tool for studying categories.

4. Functional Programming

Functional programming is a declarative paradigm that emphasizes the application of mathematical functions while avoiding state changes and data mutations. Instead of using state machines, functional programming promotes the composition of pure functions black-box like entities that take inputs and produce outputs without side effects. This focus on function purity simplifies unit testing and facilitates effective memory management, which is essential for teaching structured problem-solving in science and mathematics education.

Understanding how data is organized is a crucial first step in any programming language. In Haskell, common structures such as lists, vectors, and trees are treated as functors, providing an ideal platform for exploring connections between functional programming and category theory. The concepts of categories, functors, and natural transformations naturally emerge, helping students grasp the mathematical underpinnings of these structures.

Interpreting Haskell as a category (denoted as Hask) allows students to engage with important questions:

1. What are the objects and arrows (morphisms) in Hask?
2. How are arrows composed, and how do they satisfy the axioms of a category?
3. How can functors and other categorical constructions be represented in Hask?

These questions provide a roadmap for educators to introduce functional programming through a mathematical lens, enhancing interdisciplinary learning. Students can engage with Haskell code through practical online compilers [11], gaining hands-on experience with concepts like functors and natural transformations while consolidating their understanding of mathematical principles.

4.1 Category Hask

4.1.1 Objects of Hask

Objects of Hask are Types, for examples:

1. **Int** is a type representing integer data types. Any integer between $2^{31} - 1 = 2147483647$ and -2147483647 belongs to the Int type class. Int uses 32 bits of memory, with one bit reserved for the sign.
2. **Integer** can be seen as a superset of Int. It has no upper limit, allowing integers of arbitrary length without restrictions.
3. **Float** is a floating-point number type.
4. **Bool** is a boolean type. It can either be True or False. Here's an example code snippet to demonstrate how Bool works in Haskell:

4.1.2 Arrows in Haskell

Arrows in Hask are Haskell functions, a function in Hask is built in the following manner:

Examples 1.1

sp :: (Float, Float, Float) -> (Float, Float) sp (a,b,c)=(a+b+c,a*b*c) main = do print(sp(1,2,3))-- (6.0,6.0)

We observe that the construction of the *sp* function in Haskell follows the same structure as a mathematical function.

$$\begin{array}{rccc} sp: & \mathbb{D}^3 & \longrightarrow & \mathbb{D}^2 \\ & (a,b,c) & \mapsto & (a+b+c, abc) \end{array} \tag{20}$$

```
-- Example: Calculate the area and circumference of a circle

-- Function returning a tuple (area, circumference)
circleProperties :: Float -> (Float, Float)
circleProperties radius = (area, circumference)
  where
    area        = pi * radius * radius
    circumference = 2 * pi * radius

-- Main program
main :: IO ()
main = do
    let result = circleProperties 5.0
    putStrLn ("Area: " ++ show (fst result) ++
              ", Circumference: " ++ show (snd result))
    -- Output: Area: 78.53982, Circumference: 31.41593
```

We can see this Haskell function follows the same structure as a mathematical function:

$$\begin{array}{rccc} f: & \mathbb{R} & \longrightarrow & \mathbb{R} \times \mathbb{R} \\ & r & \mapsto & (\pi r^2, 2\pi r) \end{array} \tag{21}$$

The where clause organizes the calculations similarly to how we might show our work in mathematics

4.1.3 The Composition Law in Hask

```
-- Function type declarations
g :: Int -> Bool
f :: Bool -> String

-- Why is f . g well defined?
-- g takes an Int and returns a Bool.
-- f takes a Bool and returns a String.
-- So (f . g) takes an Int and returns a String.

-- Function definitions
g x = if x `rem` 2 == 0
        then True
        else False

f x = if x == True
        then "This is an even Number"
        else "This is an ODD number"

-- Main program
main = do
    putStrLn "Example of Haskell function composition"
    print ((f . g) 16)  -- "This is an even Number"
```

```
    print ((f . g) 27)  -- "This is an ODD number"

 -- Function to calculate circle area
circleArea :: Float -> Float
circleArea radius = pi * radius * radius

-- Function to format the result with a message
formatResult :: Float -> String
formatResult area = "The area is " ++ show area ++ " square units"

-- Function to add a decorative header to the message
addHeader :: String -> String

-- Composed function that calculates and formats the area
calculateAndFormat :: Float -> String
calculateAndFormat = addHeader . formatResult . circleArea

-- Main program
main :: IO ()
main = do
   putStrLn "Composition with Data Transformation Example:"

   putStrLn (calculateAndFormat 5.0)
   -- Output:
   -- ==== RESULT ====
   -- The area is 78.53982 square units
   -- =================

   putStrLn (calculateAndFormat 2.5)
   -- Output:
   -- ==== RESULT ====
   -- The area is 19.63495 square units
   -- =================
```

This composition follows the mathematical structure:

$$\begin{array}{lccccc} f = & h & \circ & g & \circ & a \\ & \downarrow & & \downarrow & & \downarrow \\ & \text{String} \to \text{String} & & \text{Float} \to \text{String} & & \text{Float} \to \text{Float} \end{array} \quad (22)$$

Where:

- $a(r) = \pi r^2$ (circleArea function)
- $g(x) = \text{"Theareais"} + \text{show}(x) + \text{"squareunits"}$ (formatResult function)
- $h(s) = \text{decorativeheader} + s + \text{footer}$ (addHeader function)
- The composition creates a pipeline: $f(r) = h(g(a(r)))$

4.1.4 The Identity Arrow in Hask

The identity arrow id is predefined in Haskell. In the previous example, we composed two functions f and g. We can verify that composing with id does not change the function's result.

```
 -- Identity Arrow Example

-- Function declaration and definition
g :: Int -> Bool
g x = if x `rem` 2 == 0
      then True
      else False

-- Main program
main = do
```

```
putStrLn "Example of Haskell function composition"

print (g 16)       -- True
print ((id . g) 16) -- True

print (g 27)       -- False
print ((id . g) 27) -- False
```

We note that Haskell retains the same name, id, for the identity arrow as it is named in category theory. This choice has significant pedagogical value.

4.2 Functors in Hask

4.2.1 List [ ]

In Haskell, a List is an endofunctor whose construction is inspired by the functor P, see Examples (3.5.2).

$$\begin{array}{lccc} [\,]: & \text{Hask} & \longrightarrow & \text{Hask} \\ & T & \mapsto & [\,T\,] \\ & f\downarrow & & \downarrow[\,f\,] \\ & S & \mapsto & [\,S\,] \end{array}$$

By analogy to $P(f)(A) = f(A)$, the application of a function to the elements of a list is encoded in Haskell by fmap.

```
main = do print(fmap (\x -> [x, x+1]) [1..5])
```

This code produces the output:

```
[[1,2],[2,3],[3,4],[4,5],[5,6]]
```

```
-- Example demonstrating that `fmap` satisfies the Functor laws

main :: IO ()
main = do
  -- Identity Law: fmap id should return the original list
  print (fmap id [1..5])
  -- Expected output: [1,2,3,4,5]

  -- Composition Law: fmap (f . g) == fmap f . fmap g
  let f = (+1)   -- Function f adds 1
  let g = (*2)   -- Function g multiplies by 2

  -- Verify the composition law
  print (fmap (f . g) [1..5])
  -- Output: [3,5,7,9,11]

  print (fmap f (fmap g [1..5]))
  -- Output: [3,5,7,9,11] (the same)
```

4.3 Monads of Hask

4.3.1 List [ ]

In fact, List in Haskell [ ] is more than just a functor. It is a monad equipped with a natural transformation, coded by return, which is similar to η of P, and a natural transformation coded by >>=, which is similar to μ of P.

```
-- Example demonstrating the list monad with `>>= return`
main :: IO ()
main = do
  print ([1..10] >>= return)
  -- Output: [1,2,3,4,5,6,7,8,9,10]
```

In this example, we verify that the first law of monads is satisfied for the functor. Indeed, the order of composition here is: we apply return to the contents of the list and then apply >>= to the obtained list, which corresponds to $\mu \circ P(\eta) = 1_P$. The question now is, how does >>= work?

main = do print([1..10] >>= (\x -> if odd x then [x*2] else []))

This Haskell code produces the output:

[2,6,10,14,18]

Composing with return neutralizes the action of >>=:

```
-- Example demonstrating list monad with a conditional transformation

main :: IO ()
main = do
  print ([1..10] >>= return . (\x -> if odd x then [x*2] else []))
  -- Output: [[2],[],[6],[],[10],[],[14],[],[18],[]]
```

Thus, it is clear that the function >>= works exactly in the same way as the natural transformation μ of the monad P.

4.3.2 Application

For an integer n, we want to determine all triplets (x, y, z) that satisfy the Pythagorean relation $z^2 = x^2 + y^2$ with $z \leq n$.

Let $\phi: \mathbb{N} \to \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, with

$$\phi(n) = \{(x, y, z): z \leq n and z^2 = x^2 + y^2\} \tag{23}$$

We can see ϕ as

$$\phi(n) = \cup_{1 \leq x \leq n} (\cup_{1 \leq y \leq n} (\cup_{1 \leq z \leq n} \{(x, y, z): z^2 = x^2 + y^2\})) \tag{24}$$

The only Haskell function that allows us to merge this union of unions is the bind function, coded in Haskell by >>=. Thus, we can construct the function ϕ in Haskell as follows:

```
-- Example: Finding Pythagorean triples using the list monad

phi :: Integer -> [(Integer, Integer, Integer)]
phi n = [1 .. n] >>= (\x ->
      [1 .. n] >>= (\y ->
      [1 .. n] >>= (\z ->
         if x^2 + y^2 == z^2
            then [(x, y, z)]
            else [])))

main :: IO ()
main = do
  print (phi 9)
  -- Output: [(3,4,5),(4,3,5)]
```

4.3.3 Maybe Monad

A value y is of type Maybe a if and only if y can be expressed as: $y = \text{Just}(x)$ where x is of type a, or $y = \text{Nothing}$.

For example, the function $f: x \mapsto \frac{1}{x}$ is defined on $\mathbb{R}^*$. If we associate Nothing with 0, then f is defined on $\mathbb{R}$. We say that f is a total function. Consider the following Haskell code:

```
-- Example: Using Maybe to handle partial functions safely

f :: Float -> Maybe Float
f x = if x >= 0
      then Just (1 - sqrt x)
      else Nothing

main :: IO ()
main = do
```

```
   print (f 3)   -- Just (-0.7320508)
   print (f (-3))-- Nothing
   print (f 0)   -- Just 1.0
   print (f (-1))-- Nothing
```

Notice that Just is an instance of a functor. We can then construct with fmap a function $g: \text{MaybeFloat} \rightarrow \text{MaybeFloat}$:

```
-- Example: Using fmap with Maybe to apply a function safely

import Prelude

g :: Maybe Float -> Maybe Float
g x = fmap (\x -> log x) x

main :: IO ()
main = do
   print (g (Just 5))    -- Just 1.609438
   print (g (Just 0))    -- Just (-Infinity)
   print (g Nothing))    -- Nothing
   print (g (Just (-1))) -- Just NaN
```

Let us note that g is composable with the function f since the domain of f is the codomain of g.

```
-- Example: Composing partial functions using Maybe and fmap

f :: Float -> Maybe Float
f x = if x >= 0
      then Just (1 - sqrt x)
      else Nothing

g :: Maybe Float -> Maybe Float
g x = fmap (\x -> log x) x

main :: IO ()
main = do
   print ((g . f) 3)   -- Just NaN
   print ((g . f) (-3))-- Nothing
   print ((g . f) 0)   -- Just 0.0
   print ((g . f) 1)   -- Just (-Infinity)
```

We see that g returns different results for degenerate cases: Just NaN, Nothing, Infinity. The Maybe monad simplifies calculations into Just(x) or Nothing:

```
-- Example: Composing partial functions safely using Maybe and explicit case

f :: Float -> Maybe Float
f x = if x >= 0
      then Just (1 - sqrt x)
      else Nothing

g :: Maybe Float -> Maybe Float
g mx = case mx of
   Nothing  -> Nothing
   Just x   -> if x > 0
               then Just (log x)
               else Nothing

main :: IO ()
main = do
   print ((g . f) 3)    -- Nothing
   print ((g . f) (-3)) -- Nothing
```

```
print ((g . f) 0)    -- Just 0.0
print ((g . f) 1)    -- Nothing
```

4.4 Missing Data Problem

In Haskell, the Maybe monad allows us to address the problem of missing information in a database. Suppose, for example, that our database consists of a collection of lists. We want to query this database for the head of each list. Each empty list can cause an error that disrupts the execution of the program. The Maybe monad allows us to solve this problem once and for all. We can redefine the head function so that it returns Nothing when the list is empty.

```
-- Example: Redefining `head` safely using Maybe

import Data.List

-- Safe version of head
safehead :: [Int] -> Maybe Int
safehead []     = Nothing
safehead (x:_)  = Just x

main :: IO ()
main = do
   print $ safehead []       -- Nothing
   print $ safehead [6,1,2] -- Just 6
```

The function safehead retrieves the head of a list. Sometimes, a list might be empty, in which case the safehead function returns Nothing and automatically continues to search for the head of other lists. In typical programming, this problem is generally handled manually through an operation called database cleaning. This cleaning is unnecessary in the case of monadic programming.

Now suppose that our database consists of a list of phone numbers Table 2:

Table 2. Contact information of individuals

Ali	96552233
Belgacem	98555111
Salha	27211211
Mohsen	
Massaoud	55222333

We want to query this database for the number associated with a given name. If the name does not appear in the database, we risk encountering an error message. The Maybe monad allows us to always return an answer, even if it means returning Nothing when the name does not exist in the list. In Haskell, the function that queries a list for such data is called lookup.

```
-- Example: Using lookup and defining a safe lookup function

-- Sample phonebook
phonebook :: [(String, String)]
phonebook =
   [ ("Ali", "96552233")
   , ("Belgacem", "98555111")
   , ("Salha", "27211211")
   , ("Mohsen", "")
   , ("Massaoud", "55222333")
   ]

main :: IO ()
main = do
   print (lookup "Ali" phonebook)    -- Just "96552233"
   print (lookup "Salem" phonebook)  -- Nothing
   print (lookup "Mohsen" phonebook) -- Just ""

-- Safe lookup function: treats empty strings as missing data
safeLookup :: String -> [(String, String)] -> Maybe String
safeLookup name book = case lookup name book of
   Just "" -> Nothing  -- empty string treated as missing
```

```
    result  -> result   -- otherwise return the result

main2 :: IO ()
main2 = do
    print (safeLookup "Ali" phonebook)    -- Just "96552233"
    print (safeLookup "Mohsen" phonebook) -- Nothing
    print (safeLookup "Salem" phonebook)  -- Nothing
```

In a more advanced framework, the functorial interpretation of databases in Haskell has allowed for optimizing the execution time of programs in big data [10].

Consider this richer example that shows how the Maybe monad cleanly propagates Nothing when some data is missing. This example chains lookups across two tables: an *employee → manager* table and a *person → phone* table. The students will see how a missing manager or a missing phone number stops the computation without explicit error-checking at every step.

```
-- ManagerPhone.hs
import Data.Maybe (maybe)

-- A small helper: treat empty string as "missing" (Nothing)
safeLookup :: String -> [(String, String)] -> Maybe String
safeLookup name book = case lookup name book of
    Just "" -> Nothing   -- empty string denotes "no data"
    result  -> result    -- either Just non-empty string or Nothing

-- Sample data: employee -> manager
employees :: [(String, String)]
employees =
    [ ("Alice", "Bob")
    , ("Bob", "Carol")
    , ("Eve", "Carol")
    , ("Mallory", "")   -- Mallory has no recorded manager
    ]

-- Sample contact table: name -> phone (empty = missing)
contacts :: [(String, String)]
contacts =
    [ ("Bob", "555-1234")
    , ("Carol", "")     -- Carol's phone is missing
    , ("Eve", "555-9999")
    ]

-- Compose two lookups using the Maybe monad (do-notation)
findManagerPhone :: String -> [(String,String)] -> [(String,String)] -> Maybe String
findManagerPhone emp emps book = do
    mgr   <- lookup emp emps      -- lookup returns Maybe String
    phone <- safeLookup mgr book  -- safeLookup returns Maybe String
    return phone

-- Same logic using >>= (bind)
findManagerPhone' :: String -> [(String,String)] -> [(String,String)] -> Maybe String
findManagerPhone' emp emps book =
    lookup emp emps >>= (\mgr -> safeLookup mgr book)

-- Query several employees at once: mapM will return Nothing if any query fails
findManyManagerPhones :: [String] -> [(String,String)] -> [(String,String)] -> Maybe [String]
findManyManagerPhones names emps book =
    mapM (\n -> findManagerPhone n emps book) names

-- A small display helper using Data.Maybe.maybe
displayManagerPhone :: String -> [(String,String)] -> [(String,String)] -> String
displayManagerPhone name emps book =
    maybe ("No phone found for manager of " ++ name)
```

```
        ("Manager's phone: " ++)
        (findManagerPhone name emps book)

-- Example main that prints results (expected results shown as comments)
main :: IO ()
main = do
  print $ findManagerPhone "Alice" employees contacts
  -- Just "555-1234" (Alice -> Bob -> "555-1234")

  print $ findManagerPhone "Bob" employees contacts
  -- Nothing (Bob -> Carol, but Carol has empty phone -> Nothing)

  print $ findManagerPhone "Mallory" employees contacts
  -- Nothing (Mallory -> "" (no manager) -> lookup "" fails -> Nothing)

  print $ findManyManagerPhones ["Alice","Eve"] employees contacts
  -- Just ["555-1234","555-9999"]

  print $ findManyManagerPhones ["Alice","Bob"] employees contacts
  -- Nothing (because Bob's manager Carol has no phone)

  putStrLn $ displayManagerPhone "Alice" employees contacts
  -- "Manager's phone: 555-1234"

  putStrLn $ displayManagerPhone "Bob" employees contacts
  -- "No phone found for manager of Bob"
```

This example shows the pedagogically useful property of the Maybe monad: it abstracts and centralizes the missing-data handling, removing repetitive checks and making programs easier to reason about. For classroom exercises, mix do-notation, >>= and type-level modeling (Maybe in the data) so students see the full picture.

Although Haskell is a theoretical programming language not widely used in practice, some languages are influenced by Haskell's monadic structures. This example shows how R can treat a list as a monad using the purrr library. We will apply a function to each element of a list and accumulate the results.

```
 # Example: Treating a list as a monad in R using the purrr library

# Load the purrr library
library(purrr)

# Create a list of numbers
numbers <- list(1, 2, 3, 4, 5)

# Apply a function to each element (square it) and reduce by summing
result <- numbers %>%
  map(~ .x^2) %>%   # Square each element
  reduce(`+`)       # Sum all elements

# Print the result
print(result)
# Outputs: 55
```

The map function applies a function to each element of the list, while reduce combines the list values into a single result by summing them. This example illustrates how R can treat a list functionally, similar to monads in Haskell. While R is not built around monads, libraries like purrr enable a functional programming approach, providing an elegant way to manipulate data.

4.5 Yoneda Lemma in Hask

So far, we have seen how category theory can be applied to functional programming. Conversely, functional programming provides a rich setting where category theory finds practical applications in computer science. It also helps students deepen their understanding of category theory. For instance, students often face challenges when trying to grasp the Yoneda Lemma, as discussed in various online forums [12, 13]. These challenges are shared by educators tasked with teaching category theory to students who have a purely computer science-oriented background. Writing out

the Yoneda Lemma in the context of the Hask category can help make these ideas clearer. Consider the functor $H: \text{Hask} \longrightarrow \mathbf{Set}$. For each type T in Hask, the functor $H(T)$ assigns the set of all Haskell functions from Integer to T. In other words:

$$H(T) = \{f: \text{Integer} \to T | f \text{isaHaskellfunction}\} \tag{25}$$

For example:

$$H(\text{Bool}) \text{isthesetoffunctionsInteger} \to \text{Bool} \tag{26}$$

which could include functions like:

$$f(n) = \begin{cases} \text{True} & \text{if} n > 0 \\ \text{False} & \text{otherwise} \end{cases} \tag{27}$$

or

$$f(n) = \text{Trueforall} n \tag{28}$$

$$H(\text{Int}) \text{isthesetoffunctionsInteger} \to \text{Int} \tag{29}$$

For each Haskell function $g: T \to U$, the functor H assigns a function

$$H(g): H(T) \to H(U) \tag{30}$$

which maps a function $f: \text{Integer} \to T$ to a new function $H(g)(f): \text{Integer} \to U$ defined by:

$$H(g)(f)(n) = g(f(n)) \tag{31}$$

Suppose $T = \text{Bool}$ and $U = \text{Int}$:

$$g(\text{True}) = 1, \quad g(\text{False}) = 0 \tag{32}$$

Now, for any $f: \text{Integer} \to \text{Bool}$, the functor H maps this to a new function

$$H(g)(f): \text{Integer} \to \text{Int} \tag{33}$$

where:

$$H(g)(f)(n) = g(f(n)) \tag{34}$$

For example, if $f(n) = (n > 0)$, then:

$$H(g)(f)(n) = g(f(n)) = g(n > 0) \tag{35}$$

So:

$$H(g)(f)(n) = \begin{cases} 1 & \text{if} n > 0 \\ 0 & \text{if} n \leq 0 \end{cases} \tag{36}$$

Let $X: \text{Hask} \longrightarrow \mathbf{Set}$ be a set-valued functor that associates to a type T in Hask the set of elements of type T:

$$X(T) = \{x | x \text{ is anelement of type} T\} \tag{37}$$

For each Haskell function $f: T \to U$, the functor X maps this function to a function $X(f): X(T) \to X(U)$ defined by:

$$X(f)(x) = f(x) \tag{38}$$

for every $x \in X(T)$.

The Yoneda Lemma states that the set of natural transformations from H^{Integer} to X is in bijection with $X(\text{Integer}) = \mathbb{Z}$. This is a powerful result, as it simplifies the problem of finding natural transformations: instead of considering all types T, it suffices to understand the transformation on just the object Integer. Thus, category theory is a very beautiful story. Told through Hask, this story becomes very entertaining.

Now, let's construct a natural transformation from H^{Integer} to X for each integer $n \in \mathbb{Z}$. For each $n \in \mathbb{Z}$, we construct a natural transformation $\eta_n : H^{\text{Integer}} \to X$. By the Yoneda Lemma, such a natural transformation corresponds to an element of $X(\text{Integer})$, and since $X(\text{Integer}) = \mathbb{Z}$, each $n \in \mathbb{Z}$ gives rise to a natural transformation η_n.

For each type $T \in \text{Hask}$, the component of the natural transformation η_n at T, denoted $\eta_n(T)$, is a function:

$$\eta_n(T) : H^{\text{Integer}}(T) = \text{Hom}(\text{Integer}, T) \longrightarrow X(T) \tag{39}$$

This function maps any Haskell function $f : \text{Integer} \to T$ to the element $f(n) \in X(T)$. In other words:

$$\eta_n(T)(f) = f(n) \tag{40}$$

To verify that η_n is a natural transformation, we must show that for every morphism $g : T \to S$ in Hask, the following diagram commutes:

$$\begin{array}{ccc} H^{\text{Integer}}(T) & \xrightarrow{\eta_n(T)} & X(T) \\ \downarrow H^{\text{Integer}}(g) & & \downarrow X(g) \\ H^{\text{Integer}}(S) & \xrightarrow{\eta_n(S)} & X(S) \end{array} \tag{41}$$

This means that for every $f : \text{Integer} \to T$, we need to show:

$$X(g)(\eta_n(T)(f)) = \eta_n(S)(H^{\text{Integer}}(g)(f)) \tag{42}$$

Substituting the definitions of η_n and $H^{\text{Integer}}(g)$, we get:

$$X(g)(f(n)) = (g \circ f)(n) \tag{43}$$

Since $(g \circ f)(n) = g(f(n))$, the diagram commutes, proving that η_n is indeed a natural transformation.

5. Conclusion

Category theory and functional programming are essential fields in both mathematics and computer science. However, category theory remains under-represented in many academic programs, while functional programming is often taught without a strong theoretical foundation. This paper introduces a pedagogical framework that bridges these two domains, making abstract concepts such as monads and the Yoneda Lemma more accessible to undergraduate students.

By integrating theoretical principles with practical programming, we offer an innovative approach that allows students to engage with complex ideas early in their academic journey. This interdisciplinary method not only strengthens students' understanding of functional programming but also deepens their comprehension of category theory's abstract principles.

To implement this strategy, we successfully introduced a new course titled Category Theory and Functional Programming into the curriculum of the National Engineer's Degree in Computer Science: Artificial Intelligence [14].

The interdisciplinary nature of this framework has the potential to contribute to innovative teaching strategies, collaborative learning, and interdisciplinary projects. Although pedagogical at its core, this study also highlights the practical application of theoretical frameworks in addressing real-world computational challenges, such as big data and database structures.

Moving forward, future research could explore further integration of category theory into computer science education and its application across various disciplines. A significant challenge is using Haskell to understand advanced concepts of category theory, as discussed in [15, 16, 17].

References

[1] S. MacLane, *Category Theory for the Working Mathematician*, 2nd ed. Springer-Verlag, 1998.
[2] T. Leinster, *Basic Category Theory*. Cambridge University Press, 2014.
[3] D. I. Spivak, *Category Theory for the Sciences*. The MIT Press, 2014.
[4] B. Milewski, *Category Theory for Programmers*. Leanpub, 2019.

[5] D. Jiang, "Teaching Partial Order Relations: A Programming Approach," *Int. J. Educ. Manag. Eng. (IJEME)*, vol. 14, no. 1, pp. 25–32, 2024, doi: 10.5815/ijeme.2024.01.03.
[6] F. Kadhi, *Basic Principles in Numerical Analysis*. Andalus Publishing House, Hail, 2005.
[7] F. Kadhi, *The Appropriate Explanation for Computer Mathematics*. Rushd Bookstore, Riyadh, 2005.
[8] R. P. Dobrow, *Introduction to Stochastic Processes with R*. Wiley, 2016.
[9] W. J. Braun and D. J. Murdoch, *A First Course in Statistical Programming with R*, 3rd ed. Cambridge University Press, 2021.
[10] Y. Boughanmi, "Categorical Modeling of Databases," Master's thesis, Data Science track, ENSI, Feb. 04, 2022.
[11] Online Haskell Compiler. https://onecompiler.com/haskell/42utp6rzq (accessed Oct. 2024).
[12] Discussion on Lemme de Yoneda, Les-Mathématiques.net. https://les-mathematiques.net/vanilla/discussion/987027/lemme-de-yoneda (accessed Oct. 2024).
[13] Discussion on Yoneda Lemma, Mathematics Stack Exchange. https://math.stackexchange.com/questions/tagged/yoneda-lemma (accessed Oct. 2024).
[14] National School of Computer Science, Tunisia, Category Theory and Functional Programming. https://ensi.rnu.tn/fra/plaquette-pedagogique/ (accessed Oct. 2024).
[15] M. Ghazel and F. Kadhi, "Reedy Diagrams in V-Model Categories," *Appl. Categ. Struct.*, vol. 27, pp. 549–566, 2019, doi: 10.1007/s10485-019-09566-w.
[16] F. Kadhi, "Lemme de Yoneda pour les foncteurs à valeurs monoidales," *arXiv*, 2024, doi: 10.48550/arXiv.2410.07766.
[17] F. Kadhi, "Optimisation Abstraite," *arXiv*, 2024, doi: 10.48550/arXiv.2410.07734.
[18] F. Kadhi, "Accessible Bridge Between Category Theory and Functional Programming," in *Conference SMT 2023*, *arXiv*, 2024, doi: 10.48550/arXiv.2410.07918.

Authors' Profiles

Dr. Fethi Kadhi is an Assistant Professor at the National School of Computer Science (ENSI), University of Manouba, Tunisia. In addition to his teaching duties, he holds several leadership roles within the university's research groups and laboratories. Notably, he led the Department of Image and Mathematical Modeling at ENSI and coordinated with other departments to establish two Master's degree programs. Dr Kadhi earned his B.S. in Mathematics and his M.S. in Optimization Theory from the Faculty of Sciences of Tunis, where he later completed his Ph.D. in Applied Mathematics. His research interests are interdisciplinary, encompassing Optimization, Dynamical Systems, Computer Science, and Category Theory.

How to cite this paper: Fethi Kadhi, "Bridging Category Theory and Functional Programming for Enhanced Learning", International Journal of Mathematical Sciences and Computing(IJMSC), Vol.11, No.3, pp. 1-18, 2025. DOI: 10.5815/ijmsc.2025.03.01