Turkish Journal of Mathematics

http://journals.tubitak.gov.tr/math/

Research Article

Turk J Math
(2019) 43: 1566 – 1577
© TÜBİTAK
doi:10.3906/mat-1807-189

Relationships between category theory and functional programming with an application

Alper ODABAŞ[*], **Elis SOYLU YILMAZ**
Department of Mathematics and Computer Sciences, Faculty of Arts and Sciences, Eskişehir Osmangazi University, Eskişehir, Turkey

Received: 25.07.2018 • **Accepted/Published Online:** 08.04.2019 • **Final Version:** 29.05.2019

Abstract: The most recent studies in mathematics are concerned with objects, morphisms, and the relationship between morphisms. Prominent examples can be listed as functions, vector spaces with linear transformations, and groups with homomorphisms. Category theory proposes and constitutes new structures by examining objects, morphisms, and compositions. Source and target of a morphism in category theory corresponds to input and output in programming language. Thus, a connection can be obtained between category theory and functional programming languages. From this point, this paper constructs a small category implementation in a functional programming language called Haskell.

Key words: Category theory, functional programming, Haskell

1. Introduction

Eilenberg and MacLane ([7]) are the pioneers who built the structures of the categories, functors, and natural transformations which are revealed first in 1945. A broader literature review reveals an important connection between homology and theoretical homology theory. These findings relieve mathematics from theoretical constraint and enables branches of science to involve the above relationship.

The most significant transition in computer science is between category theory and computation. One of the most important aspects of computation is composing the new functions or modules by using the primitive functions, recursive structures, etc. These requirements refer to category theory inheriting a proper algebraic model. Indeed, categorical logic is now a well-defined field about type theory and logic. For example, functional programming, domain theory and lambda calculus are highly related with category theory in theoretical respect [16].

Since functional programming languages are useful to solve problems and construct new structures, arrow language that describes morphism classes becomes a requirement for analyzing systems. The 'monad' structure taken from category theory is an advanced example for this situation. Therefore, functional programming languages such as Automata, ML, and Haskell are strongly bound with the concept of the category theory [3].

Almost all notions in category theory have corresponding algorithms in programming. Every programming language has to be defined on a consistent foundation. In order to respond to this consistency, types must be in programming language. A transformation between types builds new structure and this process can be carried on as needed. It is clear from the following diagram that the notions are particularly compatible for this

[*]Correspondence: aodabas@ogu.edu.tr
2010 *AMS Mathematics Subject Classification:* 18A05 · 68N18 · 68N15

type of systems.

Category theory	Programming
object	type
morphism	function
functor	polymorphic type
natural transformation	polymorphic function

Considering similarities, category theory could be applied to many programming fields. Functional programming is the most suitable illustration for expressing structures in category theory. The link between functional programming and category theory allows us to take into consideration data types as objects, functions as morphisms and resultant functions as compositions. Following this method, a category structure can be obtained by a functional program. Deriving categorical structure with functional programming approach provides us an interdisciplinary powerful equivalence [11].

The semantics of programming languages are fundamentally the most leading notions in computer science. This concept is based on generalizing set theory in a categorical sense. Too many properties arise in programming languages via mathematics. However, research fields of mathematics concern with abstract notions and its unifying environment provides to establish vigorous algorithms easily. For instance, instead of side effects of object-oriented programming, functional programming can handle this issue through mathematical operation called 'currying'. Furthermore, modular, dynamic, and high-level computing languages can be generated by the properties of lazy evaluation, referential transparency and mature type system [4].

In a computer program, parameters are associated with outputs. A function has a proper way for computation in order to reduce execution errors. The mathematical base forms a great advantage for any language such as Lisp, Miranda, ML, Scheme, Haskell, Gofer, which are well-known languages. While some of these languages are developed purely functional, others are constructed with a theoretical and educational base, meaning that each of these languages are defined in a special way. Contrary to the imperative programming language, the programming languages mentioned above do not affect loops, recursions, and outputs since they operate with a sequence of functions. This property is an outstanding feature for developing high-performance computers and parallel computing architecture [8].

In addition, category theory procures a link with type theory providing great opportunity to define strong data types. Recognizing strong types, a programming language is built in productive fields. Integer, Float, Bool, and Array are the most common types for any programming language. Different from these common types, any specific conceptual type releases a unique language defining new properties. In this perspective, category theory generates an abstract point of view, which fosters monads, domain theory, abstract data types, and λ-type theory [9].

Any algorithm is written in a simple, short, and apparent syntax. For example, in Haskell data declaration is

```
data List a = Nil | Cons a (List a)
```

and a '*sum*' function is defined

```
sum:: [Int] -> [Int]
sum xs=foldr (+) 0 xs
```

as well.

In this paper, categorical notions will be defined and Haskell, a functional programming language, will be used for a simple algorithm to compute whether any list is a category or not. Algorithms which are used in the Haskell implementations of these structures are analyzed in detail in the second author's MSc thesis, [14].

2. Preliminaries

In this section, we will recall some definitions and examples of category theory about functional programming [2].

A category $\mathcal{C}$ for which;

(i) Let Ob($\mathcal{C}$) be a class, elements of this class are called *objects*.

(ii) For $A, B \in$Ob($\mathcal{C}$); Mor(A,B) or $\mathcal{C}$(A,B) denotes the *morphism* set from A to B.

(iii) For all $A \in$Ob($\mathcal{C}$),

$$1_A : A \longrightarrow A$$

denotes the by unit morphism.

(iv) For $f : A \longrightarrow B$ and $g : B \longrightarrow C$ morphisms, the only morphism of *composition* of f and g is

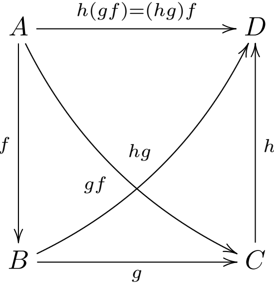
$$g \circ f : A \longrightarrow C$$

must satisfy the following conditions:

K1) For $f : A \longrightarrow B, g : B \longrightarrow C$ and $h : C \longrightarrow D$,

$$h(gf) = (hg)f$$

in other words,



this diagram must have associativity.

K2) Identity law: For any morphism $f : A \longrightarrow B$,

$$f.1_A = 1_B.f = f$$

equation is satisfied. With these four structures, any category $\mathcal{C}$ is denoted by

$$\mathcal{C} \sim (Ob(\mathcal{C}), Mor(\mathcal{C}), Mor(\mathcal{C}) \times Mor(\mathcal{C}) \xrightarrow{\circ} Mor(\mathcal{C});\ \text{Conditions})$$

2.1. Examples of category

1) $\mathcal{C} = Sets; Set$ Category

Ob($\mathcal{C}$): Class of sets

Mor($\mathcal{C}$): Sets of functions on sets

Composition: Composition of functions.

2) $\mathcal{C} = Grp; Group$ Category

Ob($\mathcal{C}$): Class of groups

Mor($\mathcal{C}$): Sets of homomorphisms of groups

Composition: Composition of homomorphisms(functions).

3) $\mathcal{C} = Top;$Category of topological space

Ob($\mathcal{C}$): Class of topological spaces

Mor($\mathcal{C}$): Sets of continuous functions between topological spaces

Composition: Composition of continuous functions.

4) $\mathcal{C} = hTop;$Category of homotopy

Ob($\mathcal{C}$): Class of topological spaces

Mor($\mathcal{C}$): Homotopy classes of continuous functions

Composition: Composition of homotopy classes.

2.2. Small categories

Definition: A small category is a category whose classes of objects are sets. Small category's objects and morphisms are denoted by O and A, respectively. Thus, for $f \in A$

$$s(f) \in O \text{ and } t(f) \in O$$

and

$$A \underset{t}{\overset{s}{\rightrightarrows}} O$$

the diagram are obtained. In this diagram, s and t functions are called source and target functions. For

$$O \xrightarrow{c} A$$

function, for every object $x \in A$,

$$c(x) = 1_x : x \longrightarrow x$$

is the identity morphism. That is, for $x = s(c_x) = t(c_x)$ is denoted by the following diagram.

$$\overset{c(x)=c_x}{\circlearrowright}\ \bullet_x$$

With these functions, the composition

$$\begin{aligned} \circ : A \times A &\longrightarrow A \\ (f,g) &\longmapsto \circ(f,g) = g \circ f = gf \end{aligned}$$

is defined with

$$t(f) = s(g)$$

condition. Because

$$s(f) \xrightarrow{f} t(f) = s(g) \xrightarrow{g} t(g)$$

the diagram gives equation. Thus, the composition is defined by

$$A \times_o A = \{(f,g)|t(f) = s(g)\} \subseteq A \times A$$

fiber product.

2.3. Special objects and morphisms

Let $\mathcal{C}$ be a category. For every object X, if

$$B \xrightarrow{\exists!} X$$

is the only morphism, so B is $\mathcal{C}$'s initial object and denoted by 0.

Dual notion of an initial object is the terminal object. Thus, for every object X, if

$$X \xrightarrow{\exists!} V$$

arrow is the only morphism, so V is $\mathcal{C}$'s terminal object and denoted by 1.

2.4. Constants

In the set category, for any x in A is denoted by

$$\begin{aligned} 1 = \{*\} &\xrightarrow{f} A \\ * &\longmapsto f(*) = x \end{aligned}$$

function. Thus, any element of A

$$x \in A \longleftrightarrow \{1 \xrightarrow{x} A\}$$

gives this element representation. In category theory,

$$x : 1 \longrightarrow A$$

morphism corresponds with constant for any element of A.

2.5. Functors and natural transformations

A functor is a structure-preserving map between categories in the same way that a homomorphism is a structure-preserving map between graphs.

Let $\mathcal{C}$ and $\mathcal{D}$ be the categories.

F1) For an object A of $\mathcal{C}$,

$$F(id_A) = id_{F(A)}$$

F2) For a composition $g \circ f$ of $\mathcal{C}$

$$F(g \circ f) = F(g) \circ F(f)$$

are satisfied, the function F is a functor from category $\mathcal{C}$ to $\mathcal{D}$. This functor is denoted by

$$F : \mathcal{C} \longrightarrow \mathcal{D}.$$

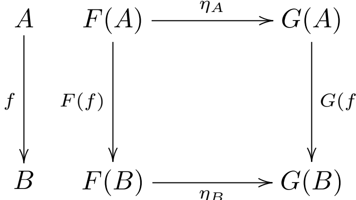
Meanwhile, F and G are functors from category $\mathcal{C}$ to $\mathcal{D}$.

NT1) For any A object of $\mathcal{C}$

$$\eta_A : F(A) \longrightarrow G(A)$$

is a morphism of $\mathcal{D}$.

NT2) Let f be a morphism from A to B in category $\mathcal{C}$. The diagram,



is commutative. If the above conditions are satisfied, the morphism

$$\eta : F \Rightarrow G$$

is a natural transformation from the functor F to the functor G.

3. Category theory and functional programming

Mathematical conception of computational algorithms is identified as a key factor evaluated as time-saving and stable process for programming. Various algorithms are obtained by different proceedings such as asymmetric encryption in parallel programming. These crucial features of programming languages must be compatible with the mathematical thinking to construct strong algorithms. In this manner, the most suitable language corresponds to functional programming language. Number of coding lines, processing time of the algorithms, and errors derived from compiler can be minimized to an acceptable amount.

Imperative computing including object-oriented programming languages precisely focuses on the result displayed by the compiler. Except from the acquired result, each output is called as a side effect. As a matter of fact, these side effects can be preventive factors to reach the result. On the contrary, functional programming languages have a tendency to put emphasis on calculations. Concluding, functional programming languages could have strong properties compared to the other programming languages [11].

Functional programming languages have four main elements:

FPL-1) Primitive data types

FPL-2) Constants of every data type

FPL-3) Functions between data types

FPL-4) Constructors

In the second element, the constant is a nonparametric value constructor. Also, the constructor is a function whose variables are functions. With these features, any functional programming language generates a category. For composing this category, we need some little changes which are:

A-1) This language must have a do-nothing function (id_A).

A-2) For calling a constant with a function, language needs to have a type called 1. This type looks like a terminal object in language's category.

A-3) Language's composition has input and output types. Thus, a composition looks like a derived program.

With these changes, for a functional programming language, $\mathcal{L}$ has a category structure $C(\mathcal{L})$. $C(\mathcal{L})$ has the following properties:

FPC-1) Objects of $C(\mathcal{L})$ are the types of $\mathcal{L}$.

FPC-2) Arrows of $C(\mathcal{L})$ are operations of $\mathcal{L}$.

FPC-3) The sources and targets of arrows are input and output types of operation of $\mathcal{L}$.

FPC-4) Compositions of $C(\mathcal{L})$ are the constructors of $\mathcal{L}$.

FPC-5) The identity arrows of $C(\mathcal{L})$ are do-nothing operations of $\mathcal{L}$ [4].

Similar to this structure, a functional programming language's category, for example, Haskell, has functors and natural transformations. A simple example of a functor is $\mathbf{List} : \mathbf{Set} \to \mathbf{Set}$ that corresponds to the list type constructor in Haskell. The object part of the functor maps a set A to the set of lists over A, i.e. sequences of the form $[x_1, ..., x_n]$ where each x_i is an element of A. The morphism part of the functor maps a function $f : A \to B$ to the function normally written as map f in Haskell which sends a list $[x_1, ..., x_n]$ to $[f(x_1), ..., f(x_n)]$. In the categorical notation, the function map f will be written as $\mathbf{List}\ f : \mathbf{List}\ A \to \mathbf{List}\ B$ (see [12]).

Natural transformations of functional programming language Haskell are functions which satisfy the natural transformations conditions of the category. For example, 'reverse' is a natural transformation of Haskell's category (Hask). This function reverses the lists. For

$$List : C(\mathcal{L}) \longrightarrow C(\mathcal{L})$$

functor,

$$\eta = reverse : Ob(C(\mathcal{L})) \longrightarrow Mor(C(\mathcal{L}))$$

is a natural transformation.

Natural transformations represent polymorphic functions. Polymorphic functions are maps between type constructors. A few examples of polymorphic functions:

$$\begin{array}{lcl} append[A] & : & \mathbf{List}\ A \times \mathbf{List}\ A \to \mathbf{List}\ A \\ map[A,B] & : & [A \to B] \to [\mathbf{List}\ A \to \mathbf{List}\ B] \\ foldr[A,B] & : & [A \times\ B \to B] \times\ B \to [\mathbf{List}\ A \to B] \end{array}$$

More details about this may be found in [5, 12, 13].

As a result, one of the functional programmings, Haskell gives rise to a category, functor, and natural transformations with language's structures [15].

Let $\mathcal{L}$ be a functional programming language with 3 data types as follows:

$$NAT: \text{ Natural numbers}$$

$$BOOLEAN = \{true, false\}$$

$$CHAR: \text{ ASCII characters}$$

Then we can construct the $C(\mathcal{L})$ category. Objects of $C(\mathcal{L})$ are

$$\mathrm{Ob}(C(\mathcal{L})) = \{NAT, BOOLEAN, CHAR, 1\},$$

where 1 is a singleton. The arrows of $C(\mathcal{L})$ together with

$$Mor(C(L)) \underset{t}{\overset{s}{\rightrightarrows}} Ob(C(L))$$

source and target functions are as follows:

$$\begin{array}{lll}
s(id_{NAT}) = t(id_{NAT}) = NAT & & \\
s(id_{CHAR}) = t(id_{CHAR}) = CHAR & & \\
s(id_{BOOLEAN}) = t(id_{BOOLEAN}) = BOOLEAN & & \\
s(id_1) = t(id_1) = 1 & & \\
s(0) = 1 & \text{and} & t(0) = NAT \\
s(c) = 1 & \text{and} & t(c) = CHAR \\
s(false) = 1 & \text{and} & t(false) = BOOLEAN \\
s(true) = 1 & \text{and} & t(true) = BOOLEAN \\
s(n) = BOOLEAN & \text{and} & t(n) = BOOLEAN \\
s(succ) = NAT & \text{and} & t(succ) = NAT \\
s(ord) = CHAR & \text{and} & t(ord) = NAT \\
s(chr) = NAT & \text{and} & t(chr) = CHAR \\
s(id_1) = 1 & \text{and} & t(id_1) = 1 \\
s(x) = NAT & \text{and} & t(x) = 1 \\
s(y) = CHAR & \text{and} & t(y) = 1 \\
s(z) = BOOLEAN & \text{and} & t(z) = 1
\end{array}$$

$$Mor(\mathcal{L}) = \{id_1, id_{NAT}, id_{CHAR}, id_{BOOLEAN}, 0, c, false, true, n, succ, ord, chr, x, y, z\}.$$

The constant 0 in NAT and the function $succ$ are defined by

$$\begin{array}{rccc} 0: & 1 & \longrightarrow & NAT \\ & x & \longmapsto & 0 \end{array} \qquad \begin{array}{rccc} succ: & NAT & \longrightarrow & NAT \\ & x & \longmapsto & x+1 \end{array}$$

and all natural numbers can be generated with the composition of 0 and $succ$.

$$1 \xrightarrow{\ 0\ } NAT \circlearrowleft^{succ}$$

Short for American Standard Code for Information Interexchange, ASCII is a standard that assigns letters, numbers, and other characters in the 256 slots available in the 8-bit code. chr and ord functions which are 8-bit character code are used to convert characters into their ASCII value and vice versa.

$$chr : NAT \longrightarrow CHAR$$

takes an ASCII value and returns the equivalent character, and

$$ord : CHAR \longrightarrow NAT$$

performs the reverse operation by converting a character to its numeric value.

$$c : 1 \longrightarrow CHR$$

is constant in CHR and all elements of the CHR object can be generated with below composition.

$$1 \xrightarrow{\ c\ } CHR \xrightarrow{\ ord\ } NAT \circlearrowleft^{succ} \xrightarrow{\ chr\ } CHR$$

There should be two constants $true$ and $false$

$$\begin{array}{rrcl} true: & 1 & \longrightarrow & BOOLEAN \\ & x & \longmapsto & true \end{array} \qquad \begin{array}{rrcl} false: & 1 & \longrightarrow & BOOLEAN \\ & x & \longmapsto & false \end{array}$$

and a function n which is defined by

$$\begin{array}{rrcl} n: & BOOLEAN & \longrightarrow & BOOLEAN \\ & true & \longmapsto & false \\ & false & \longmapsto & true \end{array}$$

From here, all elements of the $BOOLEAN$ object can be generated with

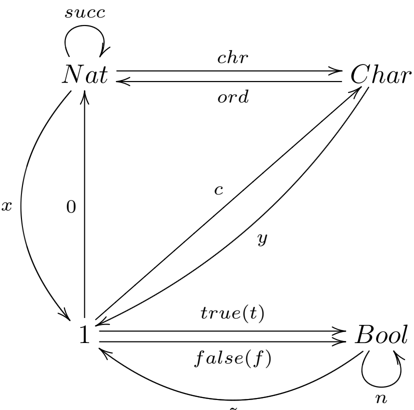
$$1 \xrightarrow{true} BOOLEAN \circlearrowleft^{n}$$

composition.

Then $C(\mathcal{L})$ is a category with the following equations being satisfied (see for details [4, 14]) .

$$\begin{array}{rcl} n \circ true & = & false \\ n \circ false & = & true \\ n \circ n & = & id_{BOOLEAN} \\ chr \circ ord & = & id_{CHAR} \end{array}$$

The diagram of the category $C(\mathcal{L})$ looks like this:



4. Haskell implementation

As we mentioned above, a small category is a category whose classes of objects are sets. We established the small category of a functional programming language and different structures. Reversely, a small category can be controlled by the package of functional programming language. That is to say, we can inspect the conditions of a category step by step under the control and implementation of Haskell programming[1].

[1] Odabas A, Soylu Yılmaz E. Haskell implementations of small categories, http://fef.ogu.edu.tr/aodabas/files/small_category.hs

In this implementation, a small category is denoted by a graph. The details of these implementations can be found in [14]. A graph can be called with nodes and edges which provides a different view to the common definition of the graph. That is, nodes are the points and the edges define the relations between these points. A graph with directed edges is called directed graph. Many functions in Haskell are implemented on this directed graph. Mainly, functions are established as defining the compositions, source and target objects and controlling the conditions of the category in the package. By the steps mentioned above, we can conclude with the following results. Firstly, the package that consists of functions must be uploaded in Haskell compiler. Right after the uploading procedure is completed, compiler outcomes can be achieved.

```
Prelude> :load "small_category.hs"
[1 of 1] Compiling Main  ( small_category.hs, interpreted )
Ok, one module loaded.
```

In this section, we will give an algorithm for checking the axioms of a category.

Algorithm 1: iscategory

Input: dG, directed graph
Output: true iff dG is a category
begin
- *obj* ← the objects of dG
- *mor* ← the morphisms of dG
- *comp* ← the compositions of morphisms of dG
- *id* ← the identity morphisms of *obj*
- **while** *different results arise for the composition in comp* **do**
  - *clist* ← add the selection for composition in *comp*
- **end**
- **if** *not For each morphism* $f : A \to B$ *in mor,* $f \circ id_A = f$ *and* $id_B \circ f = f$ **then**
  - **return** *false*
- **else if** *not associativity for right type (obj ,mor ,comp,id,clist)* **then**
  - **return** *false*
- **else**
  - **return** *true*
- **end**

end

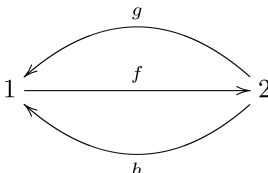
We can assume a small category as a directed graph. The functions `ob`, `mor`, `comp` are used to determine objects, morphisms and compositions of given directed graph.

```
ob :: Graf -> [Int]
Main> ob [((1,1),'f')]
[1]
Main> ob [((1,2),'s'),((2,3),'d')]
[1,2,3]
mor :: Graf -> [(Int,Int)]
Main> mor [((1,2),'s'),((2,3),'d')]
[(1,2),(2,3)]
comp :: Graf -> [(Int, Int)]
Main> comp [((1,2),'s'),((2,3),'d')]
[(1,3)]
```

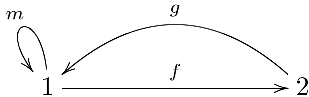
On the other hand, `source` and `target` functions are used to determine source and target objects of given morphisms.

```
source :: [(Int,Int)] -> [Int]
Main> source [(1,2),(2,3)]
[1,2]
target :: [(Int,Int)] -> [Int]
Main> target [(1,3),(1,1)]
[3,1]
```

With these functions, some controlling structures are given for determining if the directed graph is a category or not. The function `iscategory` is used to verify that the axioms are satisfied. In the following Haskell session, we will see whether some different directed graphs are categories.




```
Main> iscategory
Please give a list like [((Int,Int),Char)]
[((1,2),'f'),((2,1),'g'),((2,1),'h')]
false
```




```
Main> iscategory
Please give a list like [((Int,Int),Char)]
[((1,2),'f'),((2,1),'g'),((1,1),'m')]
choose a number from [1,2] respectively for different compositions :
[('f','g',"1"),('f','g',"m")]
2
choose a number from [1,2] respectively for different compositions :
[('m','m',"1"),('m','m',"m")]
2
true
Main> iscategory
Please give a list like [((Int,Int),Char)]
[((1,2),'f'),((2,3),'g'),((1,3),'h')]
true
```

Acknowledgements

We wish to thank the referee for helpful comments.

References

[1] Arbib M, Manes E. Arrows, Structures, and Functors : The Categorical Imperative. London, UK: Academic Press, 1975.

[2] Arvasi Z. Categories and Computer Science, Lecture Notes in Osmangazi University. Eskişehir, Turkey: Osmangazi Universit,y 2013.

[3] Asperti A, Longo G. Categories, Types, and Structures, Foundations of Computing Series. London, UK: MIT Press, 1991.

[4] Barr M, Wells C. Category Theory Lecture Notes for ESSLLI, 1999. (http://www.let.uu.nl/esslli/Courses/barr/barrwells.ps)

[5] Barr M, Wells C. Category Theory for Computing Science. New York, NY, USA: Prentice Hall, 1990.

[6] Dillon L. Functional vs Imperative Programs, Lecture Notes, 2008.

[7] Eilenberg S, MacLane S. Categories for the Working Mathematician. New York, NY, USA: Springer, 1998.

[8] Fokker J. Functional Programming, Utrecht University Department of Computer Science Lecture Notes, 1995.

[9] Hagino T. A Categorical Programming Language. PhD, Kyoto University, Kyoto, Japan, 1987.

[10] Hutton G. Programming in Haskell. London, UK: Cambridge University Press, 2007.

[11] Pitt D, Abramsky S, Poigne A, Rydeheard D. Category Theory and Computer Programming, U.K. Tutorial and Workshop. Guildford, UK: Springer, 1985.

[12] Reddy US. Categories and Functors, Lecture Notes for Midlands Graduate School in The University of Birmingham. Birmingham, UK: University of Birmingham, 2013.

[13] Rydeheard DE, Burstall RM. Computational Category Theory, Prentice-hall International Series in Computer Science. New Jersey, USA: Prentice Hall 1988.

[14] Soylu E. Category Theory with Haskell. MSc, Osmangazi University, Eskişehir, Turkey, 2013.

[15] Vimal SP. Functional Programming, BITS Plain, 2012.

[16] Walters RFC. Categories and Computer Science, Cambridge Texts in Computer Science. Cambridge, UK: Cambridge University Press, 1991.