JAM CM Journal of Applied Mathematics and Computational Mechanics 2017, 16(2), 145-157
www.amcm.pcz.pl p-ISSN 2299-9965
DOI: 10.17512/jamcm.2017.2.12 e-ISSN 2353-0588

COALGEBRAS FOR MODELLING OBSERVABLE BEHAVIOUR OF PROGRAMS

William Steingartner, Valerie Novitzká

Faculty of Electrical Engineering and Informatics, Technical University of Košice
Slovakia
e-mail: william.steingartner@tuke.sk, valerie.novitzka@tuke.sk

Received: 31 March 2017; accepted: 15 May 2017

Abstract. A useful tool for modelling behaviour in theoretical computer science is the concept of coalgebras. Coalgebras enable one to model execution of programs step by step using categorical structures and polynomial endofunctors. In our paper, we start with a short introduction of basic notions and we use this method for modelling structural operational semantics of a simple imperative language.

MSC 2010: *16T15, 18A22, 18C50*
Keywords: *category, coalgebra, observable behaviour, polynomial endofunctor, semantics*

1. Introduction

The development of computers has contributed to the investigation of dynamical features in formal structures. The dynamics involve a state of affairs which can be possibly observed and modified [1]. We can consider a computer state as a combined content of all memory cells. A user can observe only a part of this state, e.g. on display and he can modify this state by typing and executing commands. As a reaction, the computer displays certain behaviour [2]. The aim of programming is to force the computer to execute some actions and to generate desired behaviour. This behaviour can be positive, e.g. expected behaviour; or negative, e.g. side effects that must be excluded from the system. To describe the behaviour of a computer system is a non-trivial matter. But some formal descriptions of such complex systems are needed when we wish to reason formally about their behaviour. This reasoning is needed to achieve the correctness or security of these systems [3].

The basic idea of behavioural theory is to determine a relation between internal states and their observable properties. The internal states are often hidden. Computer scientists have introduced many formal structures to capture the state-based

dynamics, e.g. automata, transition systems, Petri nets, etc. In [4] the notion of behaviour in algebraic specifications was firstly introduced.

Coalgebras are a very important part of theoretical computer science. Their main *rôle* is in modelling the generated behaviour of running programs [5]. This is the behaviour that can be observed on the outside of a machine, for instance on the screen. Coalgebra is a study of states and their operations and properties. The set of states is usually seen as a black box, to which we have limited access [6]. A relation between what is actually inside and what can be observed externally is the foundation of the theory of coalgebras [7].

In this paper we introduce basic concepts and constructions of coalgebras. We start with the notion of signature, category of states and construction of a polynomial endofunctor, which is determined by a corresponding signature. Then we introduce the notion of coalgebra and illustrate it on a simple example of a bank account. The main part of our paper consists of the coalgebraic definition of structural operational semantics of a simple imperative language together with an example how the semantics of a program can be modelled by coalgebra.

2. Basic notions

Program execution can be considered as a mapping from input values to output ones. Giving some input values, the execution causes changes of internal state, i.e. computer memory, and we can observe the behaviour of a program only by its output values. There are also programs changing the internal state of a computer, that do not produce outputs, e.g. the programs running infinitely, sleeping processes in operating systems which wake up and work only in the case when some event occurs. An observer cannot see any changes of internal states because they are hidden [8]. Therefore, it is important to model the behaviour of programs before their implementation, and one of the appropriate methods is to use special categorical structures called coalgebras.

The starting notion in the coalgebraic approach is a *signature* used in the theory of algebraic specifications [9]. A signature Σ consists of *types*, e.g. $\sigma, \tau, \ldots$ and *operation symbols* of the form $f: \sigma_1, \ldots, \sigma_n \to \tau$. In a signature we distinguish:

- *constructor operation symbols* defined inductively. They tell us how to generate (algebraic) data elements;
- *destructor operation symbols*, also called observers or transition functions defined coinductively. They tell us what we can observe about our data elements;
- *derived operations* that can be defined inductively or coinductively.

If we define a derived operation f inductively, we can define the value of f on all constructors. In a coinductive definition of derived operation f we can define the values of all destructors on each outcome $f(x)$ [10, 11].

The next step is to define a basic category as a state space. A *category* $\boldsymbol{C} = (\boldsymbol{C}_{obj}, \boldsymbol{C}_{morp})$ is a mathematical structure consisting of a class of *objects*,

e.g. $A, B, \ldots$ and a class of *morphisms* of the form $f: A \to B$ between them [12, 13]. Every object has its identity morphism $id_A: A \to A$ and morphisms are composable. Because the objects of a category can be arbitrary structures, categories are useful in computer science, where we often use more complex data structures not expressible by sets. In the coalgebraic approach, the objects are states and morphisms are the operations that change states, obviously destructor operations from the corresponding signature.

The basic category of states for a coalgebra has to satisfy several properties. First, this category has to have terminal object 1 to indicate abnormal ending of execution. Second, it needs to have finite products, coproducts and exponentials [14] for constructing polynomial endofunctors introduced below. Third, to model the infinite execution of a program, it needs to have a colimit for any cocone consisting of an infinite sequence of states, i.e. for infinite composition of morphisms [15].

Execution of a program is modelled by a polynomial endofunctor. In the following text we introduce this concept. A functor $F: \boldsymbol{C} \to \boldsymbol{D}$ from a category $\boldsymbol{C}$ to a category $\boldsymbol{D}$ is a morphism defined as a pair of functions:

$$F_0: \boldsymbol{C}_{obj} \to \boldsymbol{D}_{obj} \quad \text{and} \quad F_1: \boldsymbol{C}_{morp} \to \boldsymbol{D}_{morp}$$

which are functorial, i.e. they preserve identities and composition. For a morphism $f: A \to B$ in $\boldsymbol{C}$ its image is a morphism in $\boldsymbol{D}$:

$$F_1(f): F_0(A) \to F_0(B). \tag{1}$$

A special functor is endofunctor $F: \boldsymbol{C} \to \boldsymbol{C}$ over a category $\boldsymbol{C}$. In the coalgebraic approach, polynomial endofunctors are widely used. They are constructed by using constants, identities, products, coproducts and exponentials over a state space [16]. A polynomial endofunctor is determined by the corresponding signature, especially by its destructor operations [17]. The syntax of a polynomial endofunctor F can be described by the following inference rule:

$$F(X) ::= X | X \times Y | X + Y | X^Y, \tag{2}$$

where X stands for a state space and Y is a set of constants. The product $X \times Y$ expresses a change of state from X together with some observable input or output value from Y. The coproduct $X + Y$ expresses either a change of state or some observable value, possibly an undefined end of execution [18]. For instance, when an execution of a program can crash, we define this possibility by polynomial endofunctor $F(X) = 1 + X$. Exponential objects express functions from Y to X. In summary, a polynomial endofunctor models the steps of execution and it captures the kind of behaviour that can be observed.

We use the category $\boldsymbol{Set}$ of sets as a basic category for coalgebras. This category has sets as objects and functions as morphisms. Therefore, we introduce the

concept of polynomial endofunctors for this category. Polynomial endofunctors must be defined for objects and also for morphisms.

Consider the category $\boldsymbol{Set}$ and its objects A and B. The product $A \times B$ is defined by

$$A \times B = \{(a, b) | a \in A, b \in B\}. \tag{3}$$

The product operation which assigns its cartesian product to a pair of objects can be also applied for morphisms. Let $f: A \to A'$ and $g: B \to B'$ be morphisms, then we can define a function $f \times g: A \times B \to A' \times B'$ by

$$(f \times g)(a, b) = (f(a), g(b)). \tag{4}$$

The coproduct operation $A + B$ is defined by

$$A + B = \{(0, a) | a \in A\} \cup \{(1, b) | b \in B\}. \tag{5}$$

The first members of pairs, 0 and 1 serve to force the union to be disjoint. The coproduct can also be applied for morphisms, e.g. for $f: A \to A'$ and $g: B \to B'$ we can define the coproduct of morphisms $f + g: A + B \to A' + B'$ such that for $z \in A + B$

$$(f + g)(z) = \begin{cases} \big(0, f(a)\big), & \text{if } z = (0, a); \\ \big(1, g(b)\big), & \text{if } z = (1, b). \end{cases} \tag{6}$$

Now we have defined polynomial endofunctors for objects and morphisms in the category $\boldsymbol{Set}$ and we can introduce the concept of coalgebras.

Let F be a polynomial endofunctor determined by its corresponding signature. An F-coalgebra, also called coalgebra of type F is a pair

$$(X, c) \tag{7}$$

where X stands for a state space and c is a structure map

$$c: X \to F(X) \tag{8}$$

where c is a finite tuple of destructors

$$c = (destructor_1, \dots, destructor_n). \tag{9}$$

The structure map acts as a destructor, it decomposes elements into their constituent parts. Depending of the definition of F, it provides the next state possibly with observable values. In other words, a coalgebra investigates states, operations on them. It uses destructor operations returning elements of data structures. The essence of the coalgebraic behavioural theory is the tension between what is actually inside and what can be observed externally [19].

At the end of this section, we show a simple example of how bank account can be modelled by coalgebra.

Example 1. Assume a program of bank account with the following fragment of signature:

$$\Sigma_{BankAccount=}$$

$$\begin{array}{ll} \underline{types}: & State, Value \\ \underline{opns}: & \ldots \\ & balance: State \to Value \\ & deposit: State, Value \to State \end{array}$$

We present only destructor operation symbols of a bank account that are important for constructing coalgebra. We assign the following representations to types and operations:

- the type $Value$ we represent as the set of real numbers $\mathbb{R}$;
- the type $State$ we represent as a set X of lists of values in the form

$$\langle x, y, \ldots, z \rangle, \tag{10}$$

where $x, y, z \in \mathbb{R}$. Double colon denotes the operation append which extends a list by appending element at the end, e.g. a state $\langle x, y, z \rangle$ is obtained by an operation $\langle x, y \rangle :: z$.

To the operation $balance$ we assign the function

$$bal: X \to \mathbb{R} \tag{11}$$

that for a state $x \in X$ returns the actual balance attribute of this state, i.e. a sum of all values on an account; and to the operation $deposit$ we assign the following function

$$dep: X \times \mathbb{R} \to X, \tag{12}$$

that for a state *x* and input value *r* returns a new state – it appends the new amount at the end of list.

The basic category $\boldsymbol{Account}$ consists of states $x \in X$ as objects and destructors bal and dep as morphisms. Each state is a finite list of real numbers representing the history of an account.

Now we construct polynomial endofunctor $F: \boldsymbol{Account} \to \boldsymbol{Account}$ using currying on dep as

$$F(X) = X^{\mathbb{R}} \times \mathbb{R}. \tag{13}$$

The structure map of coalgebra is a tuple of destructors $c = (dep, bal)$:

$$c: X \to X^{\mathbb{R}} \times \mathbb{R}. \tag{14}$$

A bank account with these functions is an *F-coalgebra*

$$(X, c).$$

It holds

$$\begin{aligned}(\pi_2 \circ c)(x) &= bal(x)\\ (\pi_1 \circ c)(x)(r) &= dep(x, r),\end{aligned} \tag{15}$$

where the maps π_1 and π_2 stand for obvious projections defined to a product.

This coalgebra models a correct bank account which records in the state the history of all deposited sums and returns the total balance on the account when the function bal is applied.

Let the state space X be the set of finite strings over reals and let the symbol $Sum(x)$ denote the sum of all reals in x. Then the functions in coalgebra are defined as follows:

$$\begin{aligned}bal(x) &= Sum(x),\\ dep(x, r) &= x :: r,\end{aligned} \tag{16}$$

where a double colon denotes the operation append which appends an object r at the end of the list x.

■

3. Structural operational semantics as coalgebra

Structural operational semantics is a very popular semantic method and it is also called small-steps semantics. It describes a meaning of a program in elementary steps which can be considered as transitions between memory states [20]. A state is considered as an abstraction of memory. and each change of some value stored in memory is considered as a change of state. This change of state is described by a transition relation [21, 22].

It is useful to use the concepts of the category theory for modelling structural operational semantics. Categories provide a powerful tool for expressing and modelling the program execution. If we consider memory states as category objects, then execution steps (states' changes) are morphisms between states.

3.1. Simple imperative language *Jane*

For our aims, we define a simple language *Jane* which consists of all traditional van Dijkstra's constructs (D-Charts) for imperative languages and we formulate principles of structural operational semantics in a category. The category serves here as an abstract model of computer memory with possible states, i.e. it can be considered as a state space. Our approach can also be extended for blocks, declara-

tions and user input but here we concerned only with basic imperative features of language.

We introduce for *Jane* the following well-known syntactic domains:

$n \in \mathbf{Num}$	digit strings
$x \in \mathbf{Var}$	variables' names
$e \in \mathbf{Aexpr}$	arithmetic expressions
$b \in \mathbf{Bexpr}$	Boolean expressions
$S \in \mathbf{Statm}$	statements

The *Jane* language contains the following statements: assignment, empty statement (considered also as an empty sequence of statements), sequences of statements, conditional statement and prefix logical cycle statement:

$$S ::= x := e \mid \texttt{skip} \mid S; S \mid \texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ S \mid \texttt{while}\ b\ \texttt{do}\ S$$

Semantics of arithmetic and Boolean expressions are formulated in [23].

Each variable in a program is stored in computer memory. A variable is considered as a container for some value. In our language we consider only implicit types of variables - integer numbers. Here the transient data are Boolean values, but they are never stored as stable values in memory.

We assume that each variable occurring in a program is implicitly allocated and we do not consider the variables' declarations. A value of allocated variable can be assigned and modified inducing a change of state.

3.2. Signature for states and their representation

States we define as an abstract data type. Its signature consists of types and operation symbols on the type $State$:

$$\begin{aligned} \Sigma_{State=} \quad & \underline{types}: && State, Var, Value, Statm \\ & \underline{opns}: && init: \to State \\ & && get: Var, State \to Value \\ & && next: Statm, State \to State \end{aligned}$$

The operation symbols have the following intuitive meaning: $init$ creates the initial state of a program and get returns a variable value in a given state.

Now we assign the representation to the signature Σ_{State}. We assign to the type $Value$ the set of integers together with the undefined value $\perp$:

$$\mathbf{Value} = \mathbb{Z} \cup \{\perp\}. \tag{17}$$

The type Var is represented by a countable set of variables' names **Var**. The type $State$ is represented by a set of states **State**. Every particular state $s \in$ **State** is defined as a function

$$s\colon \mathbf{Var} \rightarrow \mathbf{Value}. \tag{18}$$

Each state expresses one moment of program execution, roughly speaking a snapshot. Then any state s is expressed as a sequence

$$s = \langle (x_1, v_1), \ldots, (x_n, v_n) \rangle \tag{19}$$

of ordered tuples (x_i, v_i) where x_i is the name of variable with its actual value v_i. The first two operations in signature we define as follows. The operation $[\![init]\!]$ creates the initial state s_0 of a program with no variable defined by

$$s_0 = [\![init]\!] = \langle (\varepsilon, \varepsilon) \rangle, \tag{20}$$

where ε represents an empty position and an empty value. The operation $[\![get]\!]$ returns the value of a selected variable and is defined as follows:

$$[\![get]\!](x_i, s) = \begin{cases} v_i, & \text{if } (x_i, v_i) \in s; \\ \bot, & \text{otherwise.} \end{cases} \tag{21}$$

Such defined states we consider as category objects in our model. We also consider a special state expressing the undefined state

$$s_\bot = \langle (\bot, \bot) \rangle. \tag{22}$$

3.3. Operational semantics of *Jane* and category of states

In this section we sketch how to construct a categorical model based on structural operational semantics. The most important feature of this semantic method is the detailed description of a program execution in small steps.

We construct a model of structural operational semantics as a transition system, which models program behaviour on a state space [24]. The execution of a statement is defined by inference rules which are listed in [24, 25].

A step of execution with a possible change of state is in structural operational semantics expressed as a transition $\langle S, s \rangle \Rightarrow s'$. This transition is a relation between an input state s and an output state s'. Any change of state is done as one-step action [23]. On the other hand, if a statement S is not executed in one step, then the transition is expressed as follows:

$$\langle S, s \rangle \Rightarrow \langle S', s' \rangle. \tag{23}$$

where S' is a sub-statement of S that still should be executed. In both cases the transition rules express one action during the program execution.

A categorical model, the category $\boldsymbol{States}$ we construct as follows. We consider:

- category objects as states from **State**, and
- category morphisms as transitions.

The change of state we define by the following function $next$ from signature:

$$next\colon \mathbf{Statm} \to (\mathbf{State} \to \mathbf{State}). \tag{24}$$

For a statement S the function $next[\![S]\!]\colon \mathbf{State} \to \mathbf{State}$ is defined as follows:

$$next[\![S]\!](s) =$$

$$= \begin{cases} s' = s[x \mapsto [\![e]\!]s] & \text{if } S = x := e; \\ s & \text{if } S = \texttt{skip} \\ & \text{or } S = \texttt{while } b \texttt{ do } S \text{ and } [\![b]\!]s = \mathbf{false}; \\ next[\![S_1'; S_2]\!](s') & \text{if } S = S_1; S_2 \text{ and } \langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle; \\ next[\![S_2]\!](s') & \text{if } S = S_1; S_2 \text{ and } \langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle; \\ next[\![S_1]\!](s) & \text{if } S = \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \text{ and } [\![b]\!]s = \mathbf{true}; \\ next[\![S_2]\!](s) & \text{if } S = \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \text{ and } [\![b]\!]s = \mathbf{false}; \\ next[\![S; \texttt{while } b \texttt{ do } S]\!](s) & \text{if } S = \texttt{while } b \texttt{ do } S \text{ and } [\![b]\!]s = \mathbf{true}; \\ abort(s) & \text{otherwise.} \end{cases}$$

So any morphism in the category of states can be considered as a function $next[\![S]\!]$. A morphism $abort$ is a unique morphism which sends any state to the undefined state $s_\perp$:

$$abort\colon s \dashrightarrow s_\perp \tag{25}$$

and it represents the situation when an error occurs during the program execution and the program cannot continue its execution. Because from any object in a category, there exists only one morphism into the undefined state, it is an object which has a property of terminal object 1 in the category.

3.4. Coalgebra for *Jane* as a transition system

An F-coalgebra, also called coalgebra of type F or F-system, is a pair (X, c) where X is a state space of the coalgebra and c is the structure map of the coalgebra on X (8):

$$c\colon X \to F(X).$$

This structure map acts as a destructor operation. It takes an element of the F-coalgebra and decomposes the elements into their constituent parts. This is a common feature of coalgebras, and this point of view is dual to the point of view that algebras are objects together with combinatory principles [26-28].

An endofunctor on the category of states

$$F: \boldsymbol{States} \to \boldsymbol{States} \tag{26}$$

for coalgebra is of the type $F(X) = 1 + X$ where 1 stands for a singleton of an undefined value and X is a placeholder for a state space.

In our case, a state space is the set of category objects $\boldsymbol{States}_{Obj} = \mathbf{State}$ and $\{s_\perp\} = 1$ is a singleton set containing only an undefined state. The endofunctor F on the category of states is defined as follows:

$$F(\mathbf{State}) = 1 + \mathbf{State}. \tag{27}$$

This endofunctor sends objects $s \in \mathbf{State}$ to objects:

$$F(s) = s_\perp + next[\![S]\!]s, \tag{28}$$

and morphisms to morphisms:

$$F(next[\![S]\!]) = abort + next[\![S]\!]. \tag{29}$$

Applying the functor F on statements in program, we get steps of its execution.

Example 2. We show a simple example of defining structural operational semantics of a program written in *Jane*.

We consider the simple program with input values x and y that would be sorted according the descending order. Namely if $x \leq y$, x and y should switch their values.

We show the solution for both possible cases, when the Boolean expression in the conditional statement evaluates to **true** and then to **false**.

Let the program contains only one statement S:

$$S = \texttt{if}\ (x \leq y)\ \texttt{then}\ (z := x; x := y; y := z)\ \texttt{else skip}$$

and let the input states be:

a) $s_0 = [x \mapsto \mathbf{5}, y \mapsto \mathbf{10}, z \mapsto \mathbf{0}]$; and
b) $s_0' = [x \mapsto \mathbf{10}, y \mapsto \mathbf{7}, z \mapsto \mathbf{0}]$.

In the case a) when the condition evaluates to **true**,

$$[\![x \leq y]\!](s_0) = \mathbf{true}, \tag{30}$$

we have the following sequence of states:

$$\begin{aligned} F(s_0) = 1 + next[\![S]\!](s_0) \quad &= next[\![z := x; x := y; y := z]\!](s_0) = \\ &= next[\![x := y; y := z]\!](s_1) = \\ &= next[\![y := z]\!](s_2) = \\ &= s_3 \end{aligned} \tag{31}$$

The sequence of states during the program execution is depicted in Figure 1.

s_0	
x	5
y	10
z	0

s_1	
x	5
y	10
z	5

s_2	
x	10
y	10
z	5

s_3	
x	10
y	5
z	5

Fig. 1. States during the program execution - case a

In the case b) the condition evaluates to **false**,

$$[\![x \leq y]\!](s_0') = \mathbf{false}, \tag{32}$$

the execution of program is evaluated as follows:

$$next[\![S]\!](s_0') = next[\![\texttt{skip}\,]\!](s_0') = s_0', \tag{33}$$

so the initial state is unchanged (Fig. 2).

s_0'	
x	10
y	7
z	0

Fig. 2. An unchanged state - case b

In our program, the variables x and y store the observable values. The variable z is an auxiliary variable and it is not observable for a program user.

■

4. Conclusion

In this short contribution, we presented the main ideas of the coalgebraic approach of the behavioural theory and the process of defining coalgebra for observing the behaviour of programs and program systems. The main part of our paper is the coalgebra for the *Jane* simple imperative language, constructed as transition system in the sense of structural operational semantics. Such modelled semantics are easy to comprehend and it gives illustrative information about execution steps for programs.

We would like to extend this approach for language with declarations, input/output statements and procedures.

Our next goal is to investigate bisimilarity, the relation between states that look to be the same and to formulate coinductive proofs of behaviour.

Acknowledgment

This work has been supported by Grant No. 002TUKE-4/2017: Innovative didactic methods of education process at university and their importance in increasing education mastership of teachers and development of students competences.

References

[1] Novitzká V., Mihályi D., Steingartner W., Coalgebraic behaviour of algebraic programs, Analele Universitatii din Oradea, Proc. 8th International Conference on Engineering of Modern Electric Systems,University of Oradea, Romania 2007, 9, 60-64.

[2] Rutten J., Universal Coalgebra: A Theory of Systems, Technical Report CS-R9652, CWI, Amsterdam 1996.

[3] Jacobs B., Introduction to Coalgebra, Towards Mathematics of States and Observations, Version 2.0, 2012.

[4] Reichel H., Behavioural equivalence - a unifying concept for initial and final specifications, 3rd Hungarian Computer Science Conference, Akadémia kiadó, 3, 1981.

[5] Gumm P., Elements of the General Theory of Coalgebras, Notes of Lecture given at LUATCS'99: Logic, Universal Algebra, Theoretical Computer Science, Johanesburg 1999.

[6] Jacobs B., Objects and Classes, Co-Algebraically, [In:] Object Orientation with Prallelism and Persistence, Volume 370 of the series The Kluwer International Series in Engneering and Computer Science, Springer US, 1996, 83-103.

[7] Jacobs B., Rutten J., A tutorial on (co)algebras and (co)induction, Bulletin of the European Association for Theoretical Computer Science 1997, 62, 222-259.

[8] Slodičák V., Macko P., Some New Approaches in Functional Programming Using Algebras and Coalgebras, [In:] Electronic Notes in Theoretical Computer Science 2011, 279, 3, 41-62.

[9] Ehrig H., Mahr B., Fundamentals of Algebraic Specification 1: Equations and Initial Semantics, EATCS Monographs on Theoretical Computer Science, 1985.

[10] Jacobs B., Rutten J., An introduction to (co)algebras and (co)induction, [In:] D. Sangiorgi, J. Rutten (eds), Advanced Topics in bisimulation and coinduction, 2011, 38-99.

[11] Chin W., A brief introduction to coalgebra representation theory, [In:] J. Bergen, S. Catoiu, W. Chin (eds.), Hopf Algebras, Marcel Dekker Inc., USA, 2004, 109-133.

[12] Brandenburg M., Einführung in die Kategorientheorie, Springer Spektrum 2016.

[13] Walters R.F.C., Categories and Computer Science, Cambridge University Press, New York 1992.

[14] Awodey S., Category Theory, Carnegie Mellon University, 2005.

[15] Escardó M.H., Streicher T., Induction and recursion on the partial real line with applications to real PCF, Theoretical Computer Science 1999, 210, 1, 121-157.

[16] Goldblatt R., A calculus of terms for coalgebras of polynomial functors, Electr. Notes of Theroretical Computer Science 2001, 14, 1.

[17] Kock J., Notes on polynomial endofunctors, Universitat Autonoma de Barcelona, 2007.

[18] Deák A., Mihályi D., Jakab F., Exception modeling in the category, Proc. ICETA, IEEE, New York 2016, 49-53.

[19] Crole R.L., Lectures on (co)induction and (co)algebras, Dept. Mathematics and Computer Science, University Leicester 2006.

[20] Fernández M., Programming Languages and Operational Semantics: A Concise Overview, Springer, 2014.

[21] Nygaard M., Transition Systems, University of Aarhus, 2004.

[22] Ivaniga T., Ovseník Ľ., Turán J., Experimental Model of Passive Optical Network Technical University of Košice. ICCC 2015: 16th International Carpathian Control Conference, May 27-30, 2015, Szilvásvárad, Hungary, 186-189.

[23] Steingartner W., Novitzká V., Categorical model of structural operational semantics of imperative language, J. Informational and Organizational Sciences 2016, 40, 2.

[24] Plotkin G.D., The origins of structural operational semantics, J. of Logic and Algebraic Programming 2004, 60-61, 3-15.

[25] Sculthorpe N., Torrini P., Mosses P.D., A modular structural operational semantics for delimited continuations, Proc. Workshop in Continuations, London 2015, 63-80.

[26] Hughes J., A Study of Categories of Algebras and Coalgebras, Ph.D. Thesis, Carnegie Mellon University, Pittsburgh PA 2001.

[27] Adámek J., Milius S., Moss L.S., Initial algebras and terminal coalgebras, 2010 (unpublished).

[28] Herceg Đ., Radaković D., Extensibility of an Interpreted Language Using Plugin Libraries, Numerical Analysis and Applied Mathematics ICNAAM 2011, AIP Conf. Proc. 2011, 1389, 837-840.