Some aspects about coalgebras as foundation for expressing the semantics of imperative languages

William Steingartner[1], Valerie Novitzká[1],
Mohamed Ali M. Eldojali[1], Davorka Radaković[2]
william.steingartner@tuke.sk, valerie.novitzka@tuke.sk,
eldojalimohamed@gmail.com, davorkar@dmi.uns.ac.rs

Technical University of Košice[1]
Faculty of Electrical Engineering and Informatics
Košice, Slovakia

University of Novi Sad[2]
Faculty of Sciences
Novi Sad, Serbia

Abstract

The semantics of programs written in some languages is concerned with the interpretation in various types of models. Denotational semantics expresses the meaning of programs by functions from syntactical domains to semantic domains which can be non-trivial mathematical structures. On the other hand, structural operational semantics describes the program behaviour in the form of states' changes caused by the execution of elementary steps. We present in this paper categorical representation of denotational semantics in category of states as objects and semantic functions as morphisms. We present also categorical representation of structural operational semantics in category as a transition system where states are objects and morphisms are elementary transitions. We also sketch how to define endofunctor on the category of states and how to construct a Q-coalgebra of the functor.

1 Introduction

The semantics of programming languages belongs to the indisseverable parts of the definition of the languages. Its role is to assign a meaning to a program written in a given programming language [Ded16]. There exist several semantic methods varying in the way how to assign a meaning to a program text, e.g. denotational semantics, natural semantics, operational semantics, axiomatic semantics, game semantics, action semantics and others. These methods are equivalent mutually and we can choose such of them that is suitable for our purposes.

In this paper we deal with two semantic methods, denotational and operational semantics. Denotational semantics expresses the meaning of a program in terms of domains and functions between them. Semantic domains are mathematical structures, mainly lattices, and the meaning of a program is expressed as the elements of them [Jas11]. The semantic functions map syntactic elements to the semantic domains and they are mainly higher order functions. Denotational semantics was formulated by Dana Scott and Christopher Strachey [Sto77] and later by David Schmidt [Sch97]. This method requires rather deep knowledge of mathematics; therefore it is harder to perceive its principles for people that are not experienced in formal methods.

In: E. Vatai (ed.): Proceedings of the 11th Joint Conference on Mathematics and Computer Science, Eger, Hungary, 20th – 22nd of May, 2016, published at http://ceur-ws.org

Operational semantics expresses each step of program execution in detail using transition relations. It provides not only a meaning of a program but also its observable behaviour. This method requires medial knowledge of mathematics and it is more understandable for practical programmers. Structural operational semantics was formulated by Gordon Plotkin in [Plo81] and the main ideas and motivations are explained in [Plo04].

In the last decades the categorical structures have become useful for modeling the meanings of programs. Categories are mathematical structures that consist of objects and morphisms between them. They enable to work with more complex structures as the sets are and their properties can be represented also graphically. There are several publications concerning with categorical denotational semantics, e.g. in [Ste15b]; they are based on a category where the category objects, the semantic domains, represent the types of data structures and category morphisms are operations. Such models are suitable primarily for functional programming languages.

When we are interested in behaviour of programs, operational semantics seems to be the most adequate method. The useful categorical structures for operational semantics are coalgebras. They enable to generate observable behaviour of a running program step by step [Plo04, Tur97]. A coalgebra can be considered as a study of states and operations on them. The states form a state space that is hidden from an observer. A relation between what is actually inside and what can be observed externally is the foundation of coalgebras [Jac97]. Coalgebras are constructed using polynomial endofunctors over a category of states. In [Abo14] is elaborated coalgebraic semantics for imperative languages with non-determinism and other effects. The author constructs a monad over state space for the effects of a language and the behaviour of a program is obtained by the composition of a polynomial endofunctor and a monad.

By contrast to the works in the area of denotational semantics and operational semantics mentioned above, we attempt to define them so that they are both intelligible and demonstrative for students of computer science and also for practical software engineers. The aim of our paper is to present principles of denotational and operational semantics in terms of categories as simple as it is possible without loss of exactness. We define a simple imperative language $\mathscr{J}ane$ with basic statements. In the case of denotational semantics we construct a category of states, where the objects are states and morphisms are statements. For operational semantics we construct also a category of states where the objects are states but morphisms are transition functions. We model behaviour of programs as an appropriate endofunctor over this category as a coalgebra. In the both cases we use the categories which objects are sets and we discuss the properties of them to be legal categories. The advantage of our approach is that it provides precise graphically illustrated denotational and operational semantics without need of deeper knowledge of mathematics and therefore it is suitable for teaching.

The effectiveness of categories is in their expressive power. Categorical models are highly illustrative and their graphical representations are very good readable and understandable. The effectiveness of coalgebras we showed in [Ste16] where we presented construction of coalgebra for concrete programming problem taught on SLGeometry tool. Then the next step could be a construction of an appropriate transition system as category of states. Moreover, categorical semantic methods are suitable also for e-Learning education process. They can be easily implemented and integrated into standard e-Learning tools such as LMS Moodle.

2 Language $\mathscr{J}ane$

In our paper we use a simple language $\mathscr{J}ane$. This language consists of traditional syntactic constructions of imperative languages, namely arithmetic and Boolean expressions and statements. For this language the well-known syntactic domains are introduced:

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

Five Dijkstra's elementary statements that are elements of the syntactic domain **Statm**, $S \in \mathbf{Statm}$, are considered: assignment, empty statement, sequence of statements, conditional statement and cycle statement:

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

The semantics of arithmetic and Boolean expressions was formulated in [Ste15a]. For preservation of simplicity, here we do not consider blocks, input statement and declarations.

3 The states as memory abstraction

A program in $\mathscr{J}ane$ is a sequence of the statements. Execution of a program causes a change of some memory cells. Every snapshot of a memory during program execution can be abstracted as a state where program variables have assigned some values. Execution of a statement can modify some values of program variables, i.e. a state is changed. A meaning of a program is then the change of an initial state before program execution to a final state after program execution. Therefore *state* is a basic concept in the definition of formal semantics of imperative languages.

We define the semantic domain *State* as an abstract data type with the following signature:

$$\begin{array}{lll}\Sigma_{State} = & & \\ & \underline{types:} & State, Var, Value \\ & \underline{opns:} & init : \to State \\ & & get : Var, State \to Value \end{array}$$

Var and $Value$ are type names for program variables and values, respectively. The operation specifications have the following intuitive meaning:

- *init* merely creates the initial state of a program;
- *get* returns a variable value in a given state.

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

$$\mathbf{Value} = \mathbf{Z} \cup \{\perp\}\,. \tag{1}$$

We assign to the type Var a countable set **Var** of variable names. Our representation of an element of type *State* has to express a variable name and its value.

We assign to the type *State* the set **State** of states. Every state $s \in \mathbf{State}$ is represented as a function

$$s : \mathbf{Var} \to \mathbf{Value}. \tag{2}$$

Each state s expresses one moment of program execution. We express a state s as a sequence:

$$s = \langle (x, v_1)\,, \dots, (z, v_n) \rangle$$

of ordered tuples

$$(x, v)\,,$$

where x is the name of a variable with its actual value v.

We define representations of operation specifications from the signature Σ_{State} as follows. The operation $[\![init]\!]$ defined by

$$[\![init]\!] = s_0, \tag{3}$$

creates the initial state s_0 of a program possibly with input values of variables. Its role is to create a new sequence of state at the beginning of program execution.
The operation $[\![get]\!]$ returns the value of a variable and is defined by

$$[\![get]\!](x, \langle \dots, (x, v_i), \dots, (z, v_k), \dots \rangle) = v_i, \tag{4}$$

from the definition of state.

States defined above will be category objects in our model. We also consider a special state

$$s_\perp = \langle (\perp, \perp) \rangle \tag{5}$$

expressing the undefined state.

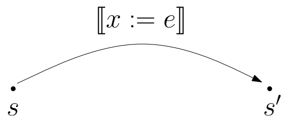

Figure 1: Execution of assignment statement

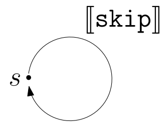

Figure 2: Execution of empty statement

4 Categorical denotational semantics of $\mathscr{J}ane$

Now we construct a model of language $\mathscr{J}ane$ as a category of states, $\mathscr{C}_{State}$. In this category we consider:

- states as category objects; and
- semantic functions as category morphisms.

Statements execute program actions, i.e. they get values from the actual state and possibly provide new values. A state is changed if a value of a program variable is modified. We model the change of states as the semantic function $[\![S]\!]$ between states, where S is a statement:

$$[\![S]\!] : s \to s', \tag{6}$$

where s and s' are states. This function is partially defined, because the resulting state s' can be undefined. To be a semantic function a morphism in $\mathscr{C}_{State}$, we extend $[\![S]\!]$ to total function using a special object of undefined state $s_\perp$:

$$[\![S]\!] : s \to s_\perp \tag{7}$$

if a statement S has undefined meaning in its input state s.

Category $\mathscr{C}_{State}$ has to satisfy the following axioms from the definition of categories [Bar90]:

- each object has identity morphism;
- for two composable morphisms there exists a morphism that is their composition.

A morphism is $\mathscr{C}_{State}$ is an application of the semantic function $[\![S]\!]$. As we see bellow, the role of identity morphism plays the semantic function applied on the empty statement. The second axiom is satisfied in the case of sequence of statements. Therefore we can state that $\mathscr{C}_{State}$ is a category.

We follow with the definition of the semantic function $[\![S]\!]$ for $\mathscr{J}ane$ statements. Assignment statement $x := e$ assigns a value of an arithmetic expression e in a state s to the variable x.
The semantic function for assignment statement is defined as follows:

$$[\![x := e]\!]s = \begin{cases} s\left[(x,v) \mapsto (x, [\![e]\!]s)\right], & \text{for } (x,v) \in s; \\ s_\perp, & \text{otherwise} \end{cases} \tag{8}$$

and it is illustrated in Figure 1.

The empty statement `skip` does not do anything, i.e. it does not change the state. Clearly, the semantic function applied on the empty statement is the identity morphism on a state s (Figure 2) and it is defined by:

$$[\![\texttt{skip}]\!]s = s. \tag{9}$$

A sequence of statements $S_1; S_2$ is executed one by one and can be modeled as a composition of morphisms (Figure 3)

$$[\![S_1; S_2]\!] = [\![S_2]\!] \circ [\![S_1]\!] \tag{10}$$

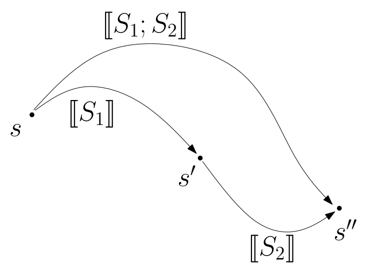

Figure 3: Composition of functions for statement sequence

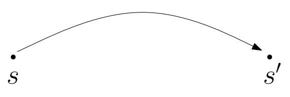

Figure 4: Execution of conditional statement

defined for a state s by

$$[\![S_1;S_2]\!]s = [\![S_2]\!]\,([\![S_1]\!])\,s. \tag{11}$$

If a state s' is undefined, i.e. $s' = s_\perp$, then the execution of the whole sequence of statements provides undefined state:

$$[\![S]\!]s_\perp = s_\perp. \tag{12}$$

From this definition follows that achieving undefined state $s_\perp$ is similar as falling into "black hole". It means that execution of a program is immediately stopped without resulting state. Because a program in $\mathscr{J}ane$ is a sequence of statements, we can state that a meaning of a program is either a path (morphism composition) from the initial state s_0 to a final state s or it is undefined when the path ends in the undefined state $s_\perp$.

Conditional statement

$$\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2$$

causes branching of execution depending on the value of a Boolean expression. The semantic function for conditional statement is defined as:

$$[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]s = \begin{cases} [\![S_1]\!]s, & \text{if } [\![b]\!]s = \textbf{true}; \\ [\![S_2]\!]s, & \text{if } [\![b]\!]s = \textbf{false}; \\ s_\perp, & \text{otherwise.} \end{cases} \tag{13}$$

It is interesting that the path of a program can follow either to the state $[\![S_1]\!]s$ or to the state $[\![S_2]\!]s$, i.e. deterministically and the branching is unseen in $\mathscr{C}_{State}$ as it is illustrated in Figure 4. The result state s' can be either $[\![S_1]\!]s$ or $[\![S_2]\!]s$, but only one of them.

Now we consider cycle statement

$$\texttt{while } b \texttt{ do } S$$

Its execution also depends on the value of Boolean expression b. If $[\![b]\!]s$ evaluates to **true** in an actual state s, the body S of a cycle is executed, then again b is evaluated in a possibly modified state. If $[\![b]\!]s$ evaluates to **false**, execution of cycle statement is finished and we obtain the result state.

The traditional denotational semantics of a cycle statement is defined by using fixpoint operator. This approach is obvious if the categorical model is a category of types. Existence of the least fixed point ensures that while statement finishes, in the other case the execution of while statement is infinite, i.e. the denotational semantics of this statement is not defined. This approach is discussed in detail in [Nie07, Sch97]. In general, the computational categorical models have continuous lattices as objects and continuous functions as morphisms [Sto77]. Such models require some structure on endomorphisms [Esc07].

In our approach we use categorical model with states as objects and functions as morphisms. The domains (states) are sets, not lattices; therefore we use another concept for handling infinite cycles. The execution of a while statement is a path of morphisms, i.e. a composition of morphisms. This path can be either finite or infinite and we need some construction in our category to solve both cases. The useful construct is the colimit of a diagram. Consider a diagram $\mathcal{D}$ consisting of the composition of morphisms

$$\mathcal{D} : s_0 \to s_1 \to s_2 \to \ldots s_i \to s_{i+1} \to \ldots \tag{14}$$

This composition of morphisms is a composition of semantic functions applied on while statement with actual states. This infinite composition is a morphism

$$s_0 \to s_\infty$$

for which there are morphisms $f_i^\infty : s_i \to s_\infty$ for $i \geq 0$ such that the cocone in Figure 5 is a colimit of the diagram $\mathcal{D}$.

$$\begin{tikzcd}
 & & & s_\infty & & & \\
s_0 \arrow[r] \arrow[urrr, "f_0^\infty"] & s_1 \arrow[r] \arrow[urr, "f_1^\infty"] & s_2 \arrow[ur, "f_2^\infty"] & \cdots & s_i \arrow[r] \arrow[ul, "f_i^\infty"] & s_{i+1} \arrow[ull, "f_{i+1}^\infty"'] & \cdots
\end{tikzcd}$$

Figure 5: Colimit of infinite composition of states

Then we can define the semantic function for while statement as:

$$[\![\texttt{while } b \texttt{ do } S]\!]s = \begin{cases} s', & \text{if } \mathcal{D} \text{ is finite} \\ \operatorname{colim}(\mathcal{D}), & \text{if } \mathcal{D} \text{ is infinite} \end{cases} \tag{15}$$

Because the objects in $\mathscr{C}_{State}$ are sets, such colimits always exist.

5 Example of categorical denotational semantics

We present in this section a short example. We show how to find a semantics of program, that calculates integer division and modulo. Let the program S_0 be the following:

$$\begin{array}{l} z := 0; \\ \texttt{while } (y \leq x) \texttt{ do } (z := z + 1; x := x - y); \end{array}$$

and let the input state be $s_0 = [x \mapsto \mathbf{17}, y \mapsto \mathbf{5}]$. We assume in variable x a divident, and in variable y a divisor. We introduce the following substitutions in our program:

$$\begin{array}{l} S_1 = z := 0 \\ S_2 = \texttt{while } (y \leq x) \texttt{ do } (z := z + 1; x := x - y) \end{array}$$

A meaning of the program S in an input state s_0 is given as follows:

$$\begin{aligned}
[\![S_1; S_2]\!]s_0 &= [\![S_2]\!]\left([\![S_1]\!]s_0\right) = [\![S_2]\!]s_1 = [\![S_2]\!]\left([\![x := x - y]\!]\left([\![z := z + 1]\!]s_1\right)\right) \\
&= [\![S_2]\!]\left([\![x := x - y]\!]s_2\right) = [\![S_2]\!]s_3 = [\![S_2]\!]\left([\![x := x - y]\!]\left([\![z := z + 1]\!]s_3\right)\right) \\
&= [\![S_2]\!]\left([\![x := x - y]\!]s_4\right) = [\![S_2]\!]s_5 = [\![S_2]\!]\left([\![x := x - y]\!]\left([\![z := z + 1]\!]s_5\right)\right) \\
&= [\![S_2]\!]\left([\![x := x - y]\!]s_6\right) = [\![S_2]\!]s_7 = id(s_7) = s_7
\end{aligned}$$

Particular states during the program execution are listed in Figure 6.
Expected results are: a quotient is after the execution stored in the variable z and remainder in variable x.

We can observe that program is expressed as a compound function consisting of more morphisms which form together a path in category. The compound function is constructed as follows:

$$[\![S]\!] = [\![\texttt{skip}]\!] \circ [\![x := x - y]\!] \circ [\![z := z + 1]\!] \circ [\![x := x - y]\!] \circ [\![z := z + 1]\!] \circ [\![x := x - y]\!] \circ [\![z := z + 1]\!] \circ [\![z := 0]\!]. \tag{16}$$

	s_0	s_1	s_2	s_3
x	**17**	**17**	**17**	**12**
y	**5**	**5**	**5**	**5**
z	ε	**0**	**1**	**1**

	s_4	s_5	s_6	s_7
x	**12**	**7**	**7**	**2**
y	**5**	**5**	**5**	**5**
z	**2**	**2**	**3**	**3**

Figure 6: States during the program S execution

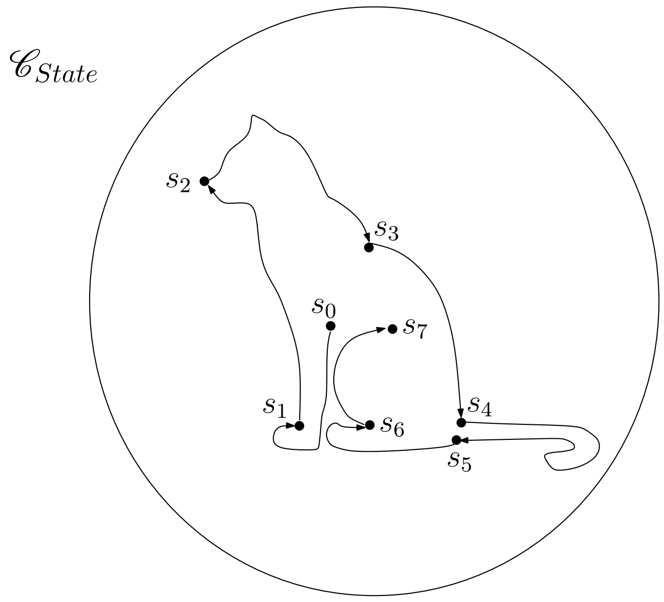

Figure 7: Path in category of states

An illustration of a path in category $\mathscr{C}_{State}$ representing a compound morphism for a semantics of the whole program S is depicted in Figure 7.
A categorical operational semantics of this program is in Section 7.

□

6 Categorical operational semantics as transition system

We defined categorical semantics of language $\mathscr{J}ane$ in the previous section based on denotational approach. In this section we shortly describe how to construct a categorical model based on operational semantics. Its main features are detailed description of program execution in small steps and following observable behaviour of a program. The execution of a program can be expressed also graphically which is highly illustrative and it accentuates the dynamics of structural operational semantics.

A model of structural operational semantics is a transition system which models program behaviour on a state space [Jon03, Plo04]. The change of state is defined for particular statements by inference rules.

A transition

$$\langle S, s\rangle \Rightarrow s'$$

is a relation between input state s and output state s'. A change of state is done as one-step action [Rad13]. If a statement is not being executed in one step, then a transition can be written as:

$$\langle S, s\rangle \Rightarrow \langle S', s'\rangle\,.$$

In both cases a transition rule expresses one action during the program execution. The inference rules for $\mathcal{J}ane$ in structural operational semantics are the following [Nie07]:

$$\langle x := e, s\rangle \Rightarrow s[x \mapsto [\![e]\!]s] \qquad (1_{os})$$

$$\langle \mathtt{skip}, s\rangle \Rightarrow s \qquad (2_{os})$$

$$\frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle}\ (3_{os}^1) \qquad \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}\ (3_{os}^2)$$

$$\frac{[\![b]\!]s = \mathbf{true}}{\langle \mathtt{if}\ \ b\ \ \mathtt{then}\ \ S_1\ \ \mathtt{else}\ \ S_2, s\rangle \Rightarrow \langle S_1, s\rangle}\ (4_{os}^{\mathbf{true}}) \qquad \frac{[\![b]\!]s = \mathbf{false}}{\langle \mathtt{if}\ \ b\ \ \mathtt{then}\ \ S_1\ \ \mathtt{else}\ \ S_2, s\rangle \Rightarrow \langle S_2, s\rangle}\ (4_{os}^{\mathbf{false}})$$

The inference rule for prefix cycle uses semantic equivalence:

$$\langle \mathtt{while}\ \ b\ \ \mathtt{do}\ \ S, s\rangle \Rightarrow \langle \mathtt{if}\ \ b\ \ \mathtt{then}\ \ (S; \mathtt{while}\ \ b\ \ \mathtt{do}\ \ S)\ \ \mathtt{else}\ \ \mathtt{skip}, s\rangle \qquad (5_{os})$$

In these inference rules, $[\![e]\!]$ and $[\![b]\!]$ are semantic functions, which assign to any syntactic well-formed expressions their meanings - an integer value, a Boolean value, resp. The result of the functions $[\![e]\!]$ and $[\![b]\!]$, where e stands for an arithmetic expression and b stands for a Boolean expression, depend on an actual state

$$[\![e]\!] : \mathbf{State} \to \mathbf{Value},$$
$$[\![b]\!] : \mathbf{State} \to \{\mathbf{false}, \mathbf{true}\}.$$

Function produces transient data, that are consumed during the program execution and their values are never stored into memory - they are used only for affecting the control flow in a program.

A categorical structure that models observable behaviour of programs is a coalgebra. It is constructed over a base category of states using a polynomial endofunctor. A base category $\mathscr{D}_{State}$ consists of

- objects that are states from **State**; and
- morphisms that are transitions.

The objects of $\mathscr{D}_{State}$ are the same as for the category $\mathscr{C}_{State}$ constructed for categorical denotational semantics but these two categories differ in morphisms. Because we need transitions as morphisms, we define the transition function *next*:

$$next : \mathbf{Statm} \to (\mathbf{State} \to \mathbf{State}),$$

that return for a statement S

$$next[\![S]\!] : \mathbf{State} \to \mathbf{State}$$

the next state obtained from the execution of the first step of a statement $[\![S]\!]$. We define this function for statements in $\mathcal{J}ane$ by:

$$next[\![S]\!](s) = \begin{cases} s' = s\,[x \mapsto [\![e]\!]] & \text{if } S = x := e; \\ s & \text{if } S = \mathtt{skip} \\ & \text{or } S = \mathtt{while}\ b\ \mathtt{do}\ S \text{ and } [\![b]\!]s = \mathbf{false}; \\ next[\![S_1'; S_2]\!](s') & \text{if } S = S_1; S_2 \text{ and } \langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle\,; \\ next[\![S_2]\!](s') & \text{if } S = S_1; S_2 \text{ and } \langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle\,; \\ next[\![S_1]\!](s) & \text{if } S = \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2 \text{ and } [\![b]\!]s = \mathbf{true}; \\ next[\![S_2]\!](s) & \text{if } S = \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2 \text{ and } [\![b]\!]s = \mathbf{false}; \\ next[\![S; \mathtt{while}\ b\ \mathtt{do}\ S]\!](s) & \text{if } S = \mathtt{while}\ b\ \mathtt{do}\ S \text{ and } [\![b]\!]s = \mathbf{true}; \\ abort(s) & \text{otherwise.} \end{cases} \qquad (17)$$

From this definition follows that any morphism in the category of states $\mathscr{D}_{State}$ can be considered as an application of function $next[\![S]\!]$ for a given statement.

Now we construct the polynomial endofunctor

$$Q : \mathscr{D}_{State} \to \mathscr{D}_{State}$$

that to any statement S and an object s assigns the next state or undefined state. The name polynomial [Koc12] indicates the polynomial form of a functor constructed using products and coproducts. For our purposes we define a functor

$$Q(\mathbf{State}) = 1 + \mathbf{State}$$

where **State** is a state space. We define this functor for objects and morphisms in $\mathscr{D}_{State}$ as follows:

$$\begin{array}{ll} Q(s) = & s_\perp + next[\![S]\!]s \\ Q(next[\![S]\!]) = & abort + next[\![S]\!] \end{array}$$

where *abort* is a unique morphism which sends any state to the undefined state $s_\perp$:

$$abort : s \dashrightarrow s_\perp \tag{18}$$

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.

A Q-coalgebra, also called coalgebra of type Q or Q-system, is a pair $(\mathbf{State}, next[\![S]\!])$, where **State** is a state space of the coalgebra and $next[\![S]\!]$ is the structure map of the coalgebra on **State**:

$$next[\![S]\!] : \mathbf{State} \to Q(\mathbf{State}).$$

This structure map acts as a destructor. It takes an element of the Q-coalgebra and decomposes the element into its 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 [Hug01].

7 Example of categorical operational semantics

We present in this section a short example of finding a meaning of a program in categorical operational semantics. We continue with the same program as in Section 5 - we show how to find a semantics of program, that calculates integer division and modulo. The code of program S_0 is:

$$\begin{array}{l} z := 0; \\ \texttt{while}\ (y \le x)\ \texttt{do}\ (z := z + 1; x := x - y); \end{array}$$

and let the input state be again $s_0 = [x \mapsto \mathbf{17}, y \mapsto \mathbf{5}]$. We assume in variable x a divident, and in variable y a divisor. We follow the substitutions in our program from Section 5:

$$\begin{array}{l} S_1 = z := 0 \\ S_2 = \texttt{while}\ (y \le x)\ \texttt{do}\ (z := z + 1; x := x - y) \end{array}$$

Here the coalgebra state space is the set of category objects $Ob(\mathscr{C}_{State}) = \mathbf{State}$ and $1 = \{s_\perp\}$ is a singleton set containing only an undefined state. The endofunctor on category of states is defined as follows:

$$Q(\mathbf{State}) = 1 + \mathbf{State},$$

and it sends objects $s \in \mathbf{State}$ to objects:

$$Q(s) = s_\perp + next[\![S]\!]s,$$

and morphisms to morphisms:

$$Q(next[\![S]\!]) = abort + next[\![S]\!]$$

Now we construct the sequence of states:

$$\begin{aligned} Q(s_0) = \quad 1 + next[\![S_0]\!]s_0 \quad &= next[\![S_1; S_2]\!]s_0 \\ &= next[\![S_2]\!]s_1 \\ &= next[\![z := z+1; x := x-y; S_2]\!]s_1 \\ &= next[\![x := x-y; S_2]\!]s_2 \\ &= next[\![S_2]\!]s_3 \\ &= next[\![z := z+1; x := x-y; S_2]\!]s_3 \\ &= next[\![x := x-y; S_2]\!]s_4 \\ &= next[\![S_2]\!]s_5 \\ &= next[\![z := z+1; x := x-y; S_2]\!]s_5 \\ &= next[\![x := x-y; S_2]\!]s_6 \\ &= next[\![S_2]\!]s_7 \\ &= s_7 \end{aligned}$$

Particular states and evaluation of Boolean condition during the program execution are as follows:

$$\begin{array}{ll} s_0 = \langle (x, \mathbf{17}), (y, \mathbf{5}) \rangle & \\ s_1 = \langle (x, \mathbf{17}), (y, \mathbf{5}), (z, \mathbf{0}) \rangle & [\![y \le x]\!]s_1 = \mathbf{true} \\ s_2 = \langle (x, \mathbf{17}), (y, \mathbf{5}), (z, \mathbf{1}) \rangle & \\ s_3 = \langle (x, \mathbf{12}), (y, \mathbf{5}), (z, \mathbf{1}) \rangle & [\![y \le x]\!]s_3 = \mathbf{true} \\ s_4 = \langle (x, \mathbf{12}), (y, \mathbf{5}), (z, \mathbf{2}) \rangle & \\ s_5 = \langle (x, \mathbf{7}), (y, \mathbf{5}), (z, \mathbf{2}) \rangle & [\![y \le x]\!]s_5 = \mathbf{true} \\ s_6 = \langle (x, \mathbf{7}), (y, \mathbf{5}), (z, \mathbf{3}) \rangle & \\ s_7 = \langle (x, \mathbf{2}), (y, \mathbf{5}), (z, \mathbf{3}) \rangle & [\![y \le x]\!]s_7 = \mathbf{false} \end{array}$$

Expected results are: a quotient is after the execution stored in the variable z and remainder in the variable x. □

8 Conclusion

We presented in our paper a new approach how to define denotational semantics and operational one in terms of categories. In both cases a state is a basic concept. A state represents a snapshot of a memory that can be changed during the program execution. In denotational approach category objects are states and morphisms are functions on states. We discussed also how to solve the situation of infinite sequence of states using colimits of diagrams. In the operational semantics the category objects are also states but the morphisms are transitions, i.e. elementary actions representing detailed steps of execution. The behaviour of a program is obtained by polynomial endofunctor and structure map of coalgebra.

Very high illustrative power of categories is very fruitful for innovative didactic methods used in education process of young software engineers and IT experts.

We want to extend our approach by introducing and modeling also user inputs and outputs, blocks, variable declarations and procedures and then to construct denotational and operational semantics for real programming language in categorical terms by following our ideas in [Ste17].

Acknowledgements

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

[Abo14] Abou-Saleh, F.: A coalgebraic semantics for imperative programming languages. Imperial College London, UK 2014.

[Ada06] Adámek, J., Herrlich, H., Strecker, G.: Abstract and concrete categories: The joy of cats. Reprints in Theory and Applications of Categories, No. 17, 2006.

[Bar90] Barr, M., Wells, C.: Category theory for computing science, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[Ded16] Dedera L., Benčík M.: Language representations based on C-BML and their processing (2016). Advances in Military Technology, Vol. 11, No. 2, pp. 159-170.

[Esc07] Escardó, M.H.: Recursion and Induction on the Real Line, In: Proceedings for the Second Imperial College Department of Computing Workshop on Theory and Formal Methods, Møller Centre, Cambridge, 11-14 September 1994.

[Hug01] Hughes J.: A Study of Categories of Algebras and Coalgebras, Ph.D. thesis, Carnegie Mellon University, Pittsburgh PA, USA, 2001.

[Jac97] Jacobs, B., Rutten, J.: A Tutorial on (Co)Algebras and (Co)Induction, EATCS Bulletin Vol. 62, 1997, pp. 222-259.

[Jac05] Jacobs, B.: Introduction to coalgebra. *Towards Mathematics of States and Observations (draft)* (2005).

[Jas11] Jaskelioff, M., Ghani, N., Hutton, G.: Modularity and Implementation of Mathematical Operational Semantics, *Electronic Notes in Theoretical Computer Science*, Vol. 229, No. 5, 2011, pp. 75–95, Proceedings of the Second Workshop on Mathematically Structured Functional Programming, MSFP 2008.

[Jay91] Jay, B.: Fixpoint And Loop Constructions As Colimits, *Lecture Notes In Mathematics*, Vol. 1488, pp. 187-192, 1991.

[Jon03] Jones, C. B.: Operational Semantics: Concepts and Their Expression, *Information Processing Letters*, Vol. 88, No. 1-2, 2003, pp. 27–32.

[Koc12] Kock, J.: Data types with symmetries and polynomial functors over groupoids. In *Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics*, Bath 2012, Electronic Notes in Theoretical Computer Science, Vol. 286, 2012, pp. 351–365.

[Kur01] Kurz, A.: Coalgebras and Modal Logic. Course Notes for ESSLLI 2001, Version of October 2001. Appeared on the CD-Rom ESSLLI'01, Department of Philosophy, University of Helsinki, Finland.

[Nie07] Riis Nielson, H., Nielson, F.: *Semantics With Applications: An Appetizer*, Springer-Verlag London, 2007.

[Nov08] Novitzká, V., Mihályi, D., and Verbová, A.: Coalgebras as models of systems behaviour. In *International Conference on Applied Electrical Engineering and Informatics, Greece, Athens* (2008), pp. 31–36.

[Per15] Perháč, J., Mihályi, D.: Intrusion Detection System Behavior as Resource-Oriented Formula, Acta Electrotechnica et Informatica, Vol. 15, No. 3, Technical University of Košice, Slovakia, 2015.

[Plo81] Plotkin, G. D.: A Structural Approach to Operational Semantics, Technical Report DAIMI FN-19, University of Aarhus, 1981.

[Plo04] Plotkin, G. D.: The Origins of Structural Operational Semantics, *J. Log. Algebr. Program.*, Vol. 60-61, 2004, pp. 3–15.

[Rad13] Radaković, D., Herceg, D.: A Platform for Development of Mathematical Games on Silverlight, *Acta Didactica Napocensia*, Vol. 6, No. 1, Babeş-Bolyai University, pp. 77-90, 2013.

[Sch97] Schmidt, D.: Denotational Semantics. A methodology for language development, 1997.

[Slo11] Slodičák, V., Macko, P.: Some New Approaches in Functional Programming Using Algebras and Coalgebras, *Electronic Notes on Theoretical Computer Science*, Vol. 279, No. 3, 2011, pp. 41–62.

[Ste15a] Steingartner, W., Novitzká, V.: A new approach to operational semantics by categories, *Proceedings of the 26th Central European Conference on Information and Intelligent Systems, CECIIS 2015*, Varaždin, University of Zagreb, 2015.

[Ste15b] Steingartner, W., Novitzká, V.: A new approach to semantics of procedures in categorical terms, *Proceedings of 2015 IEEE* 13^{th} *International Conference Informatics*, Poprad, Slovakia, IEEE Danvers, 2015, pp. 252–257.

[Ste16] Steingartner, W., Radaković, D., Valkošák, F., Macko, P.: Some properties of coalgebras and their rôle in computer science, *Journal of Applied Mathematics and Computational Mechanics*, Vol. 15, No. 4, 2016, pp. 145-156.

[Ste17] Steingartner, W., Radaković, D., Novitzká, V., Eldojali, M. A.: An Analysis of Some Aspects of Component-Based Programming for Selecting Appropriate Categorical Structures as their Models, Acta Electrotechnica et Informatica, Vol. 17, No. 2, Technical University of Košice, Slovakia, 2017.

[Sto77] Stoy, J. E.: *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA, USA, 1977.

[Tay93] Taylor, P.: An Exact Interpretation of While. *In: Theory and Formal Methods 1993: Proceedings of the First Imperial College Department of Computing Workshop on Theory and Formal Methods*, Isle of Thorns Conference Centre, Chelwood Gate, Sussex, UK, 29–31 March 1993, Springer London, pp. 302–313, 1993.

[Tur97] Turi, D., Plotkin, G.: Towards a Mathematical Operational Semantics, *In Proc. 12 th LICS Conf.*, IEEE, Computer Society Press, 1997.