Categorical Approach to Denotational and Operational Semantics

Novitzká, Valerie, Perháč, Ján and Steingartner, William

Abstract: ***The aim of this paper is to define a categorical model for operational and denotational semantics of a simple procedural language and to construct the functors between these two semantical methods. This article develops and describes the relationship between the two most used semantical methods that describe the meaning of a program. Construction of these functors determines that their composition is an identity functor.***

Index Terms: ***category, coalgebra, denotational semantics, operational semantics***

1. Introduction

Categories [1, 3] are useful mathematical structures, which in the theory of programming languages stand for modeling processes of the computer programs. In this paper we use the categories as the basic structures for describing the meaning and the behavior of the programs. The category theory was applied in the mathematical algebraic study of structures and topology. Categories find their applications widely also in the world of computer science. They help to describe construction and also behavior of the programs. Categories are often used in functional programming, e.g. using monads that are special endofunctors. This article focuses on defining the semantics and on the construction of the functors between these models. The objects of these categories represent the space, in which the programs are implemented and each category describes the execution of program commands using its respective method. One of the methods we use is denotational semantics [10], where the objects serve as semantic areas and the functions (morphisms) assign the value of a program. We are not interested in the implementation of the programs, but only in their results. The second method we use is structural operational semantics [9]. This method focuses on the description of the programs execution step by step.

We construct the functors from one category to the other and we show that their composition is identity functor. That means the both methods are equivalent. We have to show for each object (a state of the program) and for each morphism (a command) that the composition of these functors returns the origin object and morphism.

2. Simple Programming Language

We use a simple programming language that consists of five Dijkstra's statements that are present in the most imperative programming languages. A statement S can have the following structure:

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

The first statement is assignments statement, where e stands for arithmetical expression. The second one is the empty statement. The syntax follows with the sequence of statements, conditional statement and loop statement. The symbol b stands for Boolean expression. In our approach we do not use declaration of the variables, we assign input values before the start of the execution of the program.

Manuscript received March, 2019.

This work was supported by the Slovak Research and Development Agency under the contract No. SK-AT-2017-0012: Semantics technologies for computer science education.

The authors' affiliations: Technical University of Košice, Slovakia, email: valerie.novitzka@tuke.sk, jan.perhac@tuke.sk, william.steingartner@tuke.sk.

3. Categorical Denotational Semantics

In this section we define denotational semantics (DS) of the simple programming language in categorical terms. We construct the categorical model as a category of states. A state $s \in \mathbf{State}$ is the basic notion in the semantical methods and it is an abstraction of a computer memory. The semantic domain $\mathbf{State}$ is called state space.

We represent a state as a sequence of pairs (x, v):

$$s = \langle (x_1, v_1), \ldots, (x_n, v_n) \rangle,$$

where x_i is a variable and v_i is its value in this state, for $i = 1, \ldots, n$. The values are in our simplified approach integer numbers

$$v \in \mathbf{Value} = \mathbf{Z}.$$

We denote a change of a state s as an actualization:

$$s' = s[(x, v) \mapsto (x, v')]$$

that means: a state s is changed for a variable x, which obtains a new value v'.

Execution of a statement S obviously changes the content of a memory, i.e. it changes a state. The objects of our category are states and the morphisms are functions that model change of states, i.e. the execution of the statements. Such defined category defines the semantics of a program as a composition of morphisms that is called a path in the category of states. We can denote a morphism as a function:

$$[\![S]\!] : s \to s'$$

where $[\![S]\!]$ denotes the semantics of a statement S. It is clear that a state is changed in the case a variable in a statement achieves a new value.

We can summarize that our category $\mathcal{C}_{State}$ of states has:

- states of the program as category objects,
- execution of program statements as morphisms,
- for two composable morphisms there exists a morphism that is their composition,
- each object has an identity morphism.

Morphisms model execution of statements. Some statements cannot provide the changed state, e.g. in the case of infinite loop. Therefore the functions $[\![S]\!]$ are partial ones, i.e. it can have undefined value as the results. Because a category deals with total functions as morphisms, we extend the semantical domain $\mathbf{State}$ with udefined state $s_\perp$:

$$s \in \mathbf{State} \cup \{s_\perp\}$$

It is clear that a program that finishes its execution in an undefined state has no semantics. This special state $s_\perp$ is a terminal object in the category $\mathcal{C}_{State}$, i.e. it exists a unique morphism from each object to $s_\perp$.

Now we can define the morphisms in the category $\mathcal{C}_{State}$, i.e. the denotational semantics of statements. The program can execute only these five statements:

- assignment statement,
- empty statement, a program does nothing,
- sequence of statements,
- conditional statement,
- loop statement.

Assignment statement $x := e$ computes a value of arithmetic expression *e* and assigns it to the memory cell associated to variable *x*.

$$[\![x := e]\!]s = s[(x, v) \mapsto (x, [\![e]\!]s)]$$

A value $[\![e]\!]s$ is computed as a value of arithmetic expression. The detailed semantics of expressions is defined in [7].

Empty statement `skip` does nothing and it is represented as an identity morphism on a given state.

$$[\![\texttt{skip}]\!]s = s$$

In the case a statement $S = S_1; S_2$ is a sequence of statements, the semantics is a composition of morphisms for both statements that are executed one after the other:

$$[\![S]\!]s = [\![S_1; S_2]\!]s = [\![S_2]\!]([\![S_1]\!]s)$$

The function $[\![S_1]\!]s$ returns a new state s' and then is computed the function $[\![S_2]\!]s'$ that returns a state s''. The semantics of a sequence of statements is a morphism (path) $s \to s''$.

Conditional statement `if` b `then` S_1 `else` S_2 firstly computes a Boolean expression b in the state s that specifies which branch is executed. Semantics of conditional statement is defined as:

$$[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]s = \begin{cases} [\![S_1]\!]s, & \text{if (a)} \\ [\![S_2]\!]s, & \text{if (b).} \end{cases}$$

where

- (a) is a condition $[\![b]\!]s = \mathbf{true}$
- (b) is a condition $[\![b]\!]s = \mathbf{false}$

Loop statement `while` b `do` S also depends on the value of a Boolean expression b. As long as $[\![b]\!]s = \mathbf{true}$ in an actual state, then the body S of loop statement is executed. When $[\![b]\!]s = \mathbf{false}$, the execution of the cycle ends. We have to consider the situation when the condition is always evaluated as **true**, which means that the body of loop statement is executed infinitely:

$$\mathbf{D} : s_0 \to s_1 \to \ldots s_i \to s_{i+1} \to \ldots$$

To deal with an infinite composition of morphisms in our category, we use the concept of colimits [8, 11]. Then the semantics of infinite loop is a colimit of the infinite composition $\mathbf{D}$. A colimit in this case is the terminal object, the undefined state $s_\perp$. The denotational semantics of a program is a path, a composition of morphisms defined above.

4. Coalgebraical Structural Operational Semantics

Structural operational semantics (SOS) describes the execution of programs in detail and models a program behavior [5]. Categorical structure suitable for defining operational semantics is a coalgebra [4]. A coalgebra is a pair (X, c), where X is a state space, $c : X \to T(X)$ is a coalgebraic mapping, and T is a polynomial endofunctor [6] over a category of states.

Assume the types $Variable$ for variable names, $Value$ for possible values, $Statm$ for statements of a language and $State$ for states. All these types are sets [2]. We define a signature of state space

$$\begin{aligned} &\Sigma_{State} = \\ &\quad types : State, Variable, Value, Statm \\ &\quad opns : \ init :\to State \\ &\qquad\quad next : Statm, State \to State \\ &\qquad\quad abort : Statm, State \to State \end{aligned}$$

which define two important operations, the destructors. The operation $next$ gets a statement in an actual state and returns a new state. The operation $abort$ serves for udefined new state, e.g. if a loop is infinite, or a program has not defined an initial value for a variable. The operation $init$ is auxiliary, it provides an initial state before an execution of a program starts.

Now we assign to the types and operations from Σ_{State} their representations. We associate to the type $Value$ the set of integer numbers together with an undefined value:

$$\mathbf{Value} = \mathbf{Z} \cup \{\perp\}.$$

The type $Variable$ is represented by a finite set of program variables **Var**. The type $State$ is represented as a set **State** and it is our state space. Each state $s \in$ **State** is defined by a function:

$$s : \mathbf{Var} \to \mathbf{Value}$$

Assume a statement S and a state s. The representations of the operations $next$ and $abort$ are as follows:

$$[\![next]\!][\![S]\!]s = \begin{cases} s' = s[x \mapsto [\![e]\!]s] & \text{if (a)} \\ s & \text{if (b)} \\ [\![next]\!][\![S_1'; S_2]\!](s') & \text{if (c)} \\ [\![next]\!][\![S_2]\!]s' & \text{if (d)} \\ [\![next]\!][\![S_1]\!](s) & \text{if (e)} \\ [\![next]\!][\![S_2]\!](s) & \text{if (f)} \\ [\![next]\!][\![S]\!]; \texttt{while}\ b\ \texttt{do}\ S] & \text{if (g)} \\ [\![abort]\!]s & \text{otherwise} \end{cases}$$

where

- (a) stands for $S = x := e$;
- (b) stands for $S = \texttt{skip}$ or $S = \texttt{while}\ b\ \texttt{do}\ S$ and $[\![b]\!]s = \mathbf{false}$;
- (c) stands for $S = S_1; S_2$ and $\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle$;

- (d) stands for $S = S_1; S_2$ and $\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle$;
- (e) stands for $S = \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2$ and $[\![b]\!]s = \mathbf{true}$;
- (f) stands for $S = \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2$ and $[\![b]\!]s = \mathbf{false}$;
- (g) stands for $S = \texttt{while } b \texttt{ do } S$ and $[\![b]\!]s = \mathbf{true}$.

The function $[\![abort]\!]$ we represent by

$$[\![abort]\!]s = s_\perp.$$

A category of states $\mathcal{D}_{State}$ has as objects states s as defined above and the morphisms are functions $[\![next]\!]$ and $[\![abort]\!]$. The polynomial endofunctor $T : \mathcal{D}_{State} \to \mathcal{D}_{State}$ for our programming language is defined as:

$$T(\mathbf{State}) = s_\perp + \mathbf{State}$$

Then the T-coalgebra is a pair

$$(\mathcal{D}_{State}, ([\![next]\!], [\![abort]\!]))$$

Each application of the functor T models one step of program execution in the terms of structural operational semantics.

5. Relationship between Semantic Methods

In this section we show that both categorical semantical methods introduced above are equivalent. In the previous sections we constructed the categories $\mathcal{C}_{State}$ for denotational semantics and $\mathcal{D}_{State}$ for structural operational semantics. Here we construct two functors

$$Q : \mathcal{D}_{State} \to \mathcal{C}_{State}$$
$$P : \mathcal{C}_{State} \to \mathcal{D}_{State}$$

and we show that their composition is an identity functor. Both categories $\mathcal{D}_{State}$ and $\mathcal{C}_{State}$ have as objects all possible states. The category $\mathcal{D}_{State}$ uses in defining structural semantics more objects than $\mathcal{C}_{State}$, because SOS defines semantics in detailed steps. The morphisms in these categories differ, because the semantics is defined by different functions.

5.1 Construction of Functor Q

First we define the functor Q from operational semantics to denotational semantics. In general, this functor assigns to a state from $\mathcal{D}_{State}$ a state from $\mathcal{C}_{State}$:

$$Q(s_{\mathcal{D}_{State}}) = s_{\mathcal{C}_{State}}$$

and to a morphism in $\mathcal{D}_{State}$ a morphism in $\mathcal{C}_{State}$:

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

We define the functor Q for morphisms for each statement in a language. To a morphism from the $\mathcal{D}_{State}$ category has to be assigned a morphism from the $\mathcal{C}_{State}$ category.

For the assignment statement, that is executed in one step, we investigate the domains and codomains of the morphisms defining particular semantics of assignment statement:

in $\mathcal{D}_{State}$:

$$dom([\![next]\!][\![x := e]\!]) = s,$$
$$cod([\![next]\!][\![x := e]\!]) = s',$$

in $\mathcal{C}_{State}$:

$$dom([\![x := e]\!]) = s$$
$$cod([\![x := e]\!]) = s'.$$

It is clear that for assignment statement the morphism $[\![next]\!]$ is mapped to the morphism $[\![S]\!]$

$$Q([\![next]\!][\![x : e]\!]) = [\![x := e]\!]$$

Empty statement is also executed in one step, and the domain and codomain morphisms in both categories are the same, i.e. they can be mapped directly:

in $\mathcal{D}_{State}$:

$$dom([\![next]\!][\![\texttt{skip}]\!]) = cod([\![next]\!][\![\texttt{skip}]\!]) = s,$$

in $\mathcal{C}_{State}$:

$$dom([\![\texttt{skip}]\!]) = cod([\![\texttt{skip}]\!]) = s.$$

That means, the identity morphism in $\mathcal{D}_{State}$ is mapped to the identity morphism in $\mathcal{C}_{State}$

$$Q([\![next]\!][\![\texttt{skip}]\!]) = id_{s_{\mathcal{C}_{State}}}$$

For sequence of statements $S_1; S_2$, its operational semantics consists of several morphisms $[\![next]\!]$. Let n be a number of steps to execute the statement S_1 and m the number of steps to execute the statement S_2. Then the execution of the statement sequence is the composition:

$$\bigcirc_{i=1}^{n+m} [\![next]\!][\![S_1; S_2]\!]$$

The denotational semantics defines the meaning of statement sequence as a morphism $[\![S_1; S_2]\!]$. Assume that this sequence of statements starts its execution in a state s and finishes in a state s'. We define the functor Q for statement sequence as follows. In $\mathcal{D}_{State}$, we map the $n + m - 1$ morphisms $[\![next]\!]$ to an identity on $s_{\mathcal{C}_{State}}$ and the $n + m$-th morphism $[\![next]\!]$ to the composition $[\![S_1; S_2]\!]$.

The morphisms for conditional statement and loop statement are mapped by the functor Q in similar manner.

We should also consider the undefined state, which is caused by infinite loop. From each state there is morphism to the object representing the undefined state. Undefined state $s_\perp$ from the category $\mathcal{D}_{State}$ is displayed as an undefined state of the $\mathcal{C}_{State}$ category using functor Q. The resulting s_n state is displayed in object s' in $\mathcal{C}_{State}$ category.

We have shown how all morphisms of $\mathcal{D}_{State}$ for operational semantics are mapped to the morphisms in $\mathcal{C}_{State}$ for denotational semantics by the functor Q. This functor is illustrated in Fig.1.

5.2 Construction of Functor P

Now, we construct a functor $P : \mathcal{C}_{State} \to \mathcal{D}_{State}$ from categorical denotational semantics into a coalgebraic structural operational semantics:

$$P : \mathcal{C}_{State} \to \mathcal{D}_{State}$$

The functor assigns to a state $s_{\mathcal{C}_{State}}$ in the category $\mathcal{C}_{State}$ a state $s_{\mathcal{D}_{State}}$ in the category $\mathcal{D}_{State}$:

$$P(s_{\mathcal{C}_{State}}) = s_{\mathcal{D}_{State}}$$

The functor P has to assign also to the morphisms from $\mathcal{C}_{State}$ a morphism in $\mathcal{D}_{State}$ category:

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

Let the $\mathcal{C}_{State}$ and $\mathcal{D}_{State}$ categories have an initial state s. We have already stated that a morphism for assignment statement in the $\mathcal{C}_{State}$ category saves the resulting value of the expression e in a variable x in state s, afterwards the program acquires a new state s'. With this semantics the only possible result is the corresponding new state in both categories.

$$\text{in } \mathcal{C}_{State}:$$

$$dom([\![x := e]\!]) = s$$
$$cod([\![x := e]\!]) = s'$$

$$\text{in } \mathcal{D}_{State}:$$

$$dom([\![next]\!][\![x := e]\!]) = s$$
$$cod([\![next]\!][\![x := e]\!]) = s'.$$

That means, the functor P:

$$P([\![x := e]\!]) = [\![next]\!][\![x := e]\!]$$

For the $\mathcal{C}_{State}$ category and an arbitrary object s the result of the morphism for empty statement is $[\![\texttt{skip}]\!]s = s$. This morphism assigns the state s of the $\mathcal{C}_{State}$ into $\mathcal{D}_{State}$ category using the functor P. We can define

$$P([\![\texttt{skip}]\!]) = [\![next]\!][\![\texttt{skip}]\!]$$

For the sequence of statements $[\![S_1;\ S_2]\!]s$ the result is the state s' or $s_\perp$. In denotational semantics, the sequence of statements is perceived as the composition of morphisms, which intermediate states are unknown, in SOS the result is some state s_n. We assign the resulting state s' from category $\mathcal{C}_{State}$ to state s_n in $\mathcal{D}_{State}$ category.

If the sequence of statements is composed from language commands that are executed in one step S_1,S_2,...,S_{n-1},S_n, then functor P assigns morphism:

$$P([\![S_1;S_2]\!]) = \bigcirc_{i=1}^{n} ([\![next]\!][\![S_i]\!])$$

The remaining statements (conditional statement and loop statement) are mapped by the functor P in similar manner.

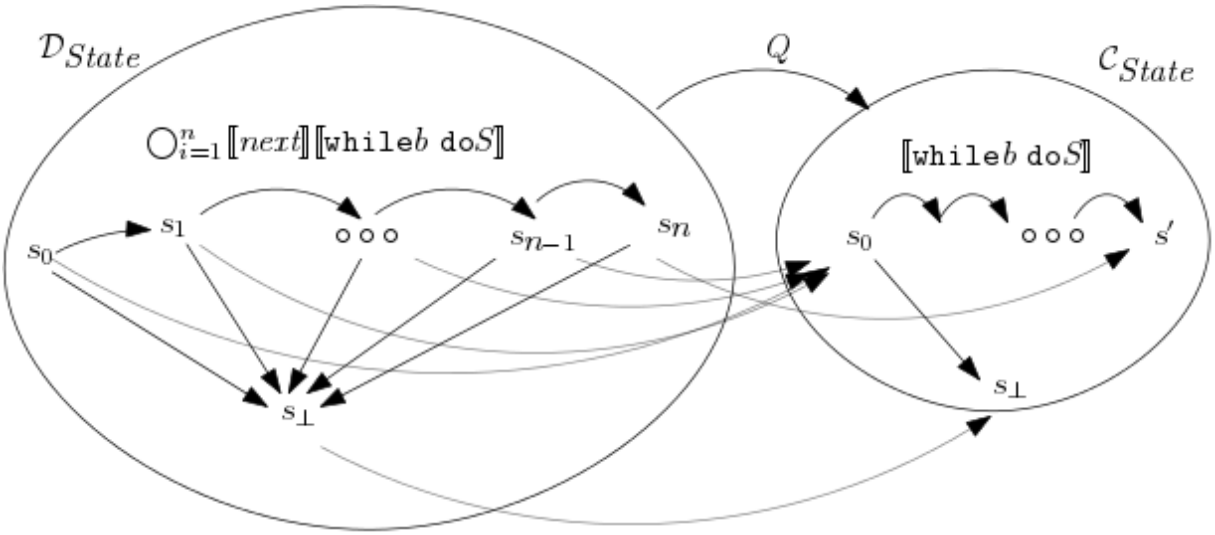

Figure 1: Example of how functor Q displays its morphisms and states

5.3 Equivalence of Categorical Models

We have constructed the functor P from categorical denotational semantics into coalgebraic structural operational semantics and the functor Q from structural operational semantics to denotational semantics. We construct the composition of these functors and we have to discuss that such composition is an identity functor on the category $\mathcal{D}_{State}$:

$$P \circ Q = Id_{\mathcal{D}_{State}}$$

The composition $P \circ Q$ on objects returns the same object

$$(P \circ Q)s_{\mathcal{D}_{State}} = s_{\mathcal{D}_{State}}.$$

from construction of P and Q.

We investigate the morphisms and their mapping by functors. For assignment statement and empty statement it is clear that

$$(P \circ Q)([\![next]\!][\![x := e]\!]) = P([\![x := e]\!]) = \\ = [\![next]\!][\![x := e]\!]$$

$$(P \circ Q)([\![next]\!][\![\texttt{skip}]\!]) = P([\![\texttt{skip}]\!]) = \\ = [\![next]\!][\![\texttt{skip}]\!]$$

Sequence of statements makes a composition, so from category $\mathcal{D}_{State}$ morphisms representing the first statement S_1 or intermediate (sub)statements will be assigned to identical morphism in $\mathcal{C}$. The domain and codomain is assigned to the initial state of the sequence. This identical morphism is assigned back, domain to the initial state of the respective morphism, codomain to the final state of the morphism. The last morphism of $\mathcal{D}_{State}$ is displayed to the morphism of which the domain is the initial state for whole sequence and codomain to the final state. $\mathcal{C}_{State}$ category assigns its morphism to the composition of all intermediate morphisms in $\mathcal{D}_{State}$.

In the case of remaining statements we can reason in the similar manner. The second part, we need to show is that the composition

$$Q \circ P = Id_{\mathcal{C}_{State}}$$

is an identity functor on the category $\mathcal{S}_{State}$. The process of reasoning is analogous and then we can state that the composition of the functor P and Q is identity functor, i.e. the both semantical methods, categorical denotational semantics and coalgebraic operational semantics are equivalent.

6. Conclusion

This paper presents two categorical models for two known semantical methods, denotational semantics and structural operational semantics. We introduced a simple programming language, that is imperative. Then we defined categorical model for denotational semantics in the frame of the category $\mathcal{C}_{State}$ of states, where execution

of statements is modeled by morphisms. The following task was the definition of a coalgebraical model for operational semantics. In the category $\mathcal{D}_{State}$ we needed to define the execution of a statement in small (elementary) steps, therefore we introduced the operation $[\![next]\!]$. The composition of this operation depending on the nature of a given statement defines the behavior of execution.

The main contribution of our paper is the construction of two functors, P and Q, between these models. We proved that the composition of these functors is an identity functor on a corresponding categorical model, what ensures that these methods are equivalent. This paper contains only principles how these results were achieved.

References

[1] J. Adámek. *Matematické struktury a kategorie*. SNTL-Nakladatelství technické literatury, Praha, 1982.

[2] P. Balcar, B. Štěpánek. *Teorie množin*. Academia, Praha, 2000.

[3] C. Barr, M. Wells. *Category Theory for Computing Science*. Prentice Hall International, 1990.

[4] B. Jacobs. *Introduction to Coalgebra. Towards Mathematics of States and Observations*. Cambridge University Press, 2016.

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

[6] J. Kock. Data Types with Symmetries and Polynomial Functors over Grupoids. *Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics, Electronic Notes in Theoretical Computer Science*, 286:351–365, 2012.

[7] H. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.

[8] V. Novitzká, V. Slodičák. *Kategorické štruktúry a ich aplikácie v informatike*. Equilibria, Košice, 2010.

[9] G.D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, 1981.

[10] D.A. Schmidt. *Denotational Semantics. Methodology for Language Development*. Allyn and Bacon, 1986.

[11] W. Steingartner, V. Novitzká, M. Bačíková, and Š. Korečko. A New Approach to Categorical Semantics for Procedural Languages. *Computing and Informatics*, 36:1385–1414, 2017.