A new approach to operational semantics by categories

William Steingartner, Valerie Novitzká

Faculty of Electrical Engineering and Informatics

Technical University of Košice

Letná 9, 04200 Košice, Slovakia

`{william.steingartner, valerie.novitzka}@tuke.sk`

Abstract. *Structural operational semantics is one of the most popular semantic methods in the community of software engineers. It describes program behavior in the form of change of states caused by execution of elementary steps. This feature predestinates Structural operational semantics for implementation of programming languages and also for verification purposes. In our paper we present a new approach to Structural operational semantics: behavior of programs, i.e. state changes we model in category of states. Category morphisms express elementary execution steps and program execution is an oriented path in category, i.e. composition of morphisms. Our approach is able to accentuate dynamics of Structural operational semantics, it is intuitively typed. That's why such model is suitable not only as a model for Structural operational semantics but also for educating young software engineers.*

Keywords. category theory, morphism, semantic function, state, structural operational semantics

1 Introduction

Structural operational semantics is a simple and direct method for describing meaning of programs written in some programming language. It requires minimal knowledge of mathematics and it is easily understandable by practical programmers [14].

There are several semantic methods used simultaneously with structural operational semantics. Denotational semantics formulated by Scott Strachey in [20] and later by David Schmidt [15] requires quite deep knowledge of mathematics. The meaning of programs is expressed by functions from syntactical domains to semantic domains which can be non-trivial mathematical structures, e.g. lattices. Therefore we cannot be surprised that structural operational semantics gained much more attention in community of programmers than denotational semantics [8].

Natural semantics formulated by Gilles Kahn [9] is often called semantics of big steps. The author followed two aims:

- to simplify semantic description for software en-

gineers instead of difficult mathematical notations of currying and continuation functions in denotational semantics; and

- to abstract from elementary steps of execution in structural operational semantics.

Natural semantics describes a change of states caused by execution whole statements [16]. Natural semantics can be useful for specification languages or in program verification [3].

There are known several other semantic methods less or more used in various areas of programming. Axiomatic semantics [6] is based on satisfying postconditions after executing of statements from truth preconditions before this action. Algebraic semantics [5, 23] specifies abstract data types and it models them by heterogeneous algebras. Game semantics [1, 4] describes meaning of programs in the form of game trees and game arenas.

The author of structural operational semantics is Gordon Plotkin. In his work [12] he formulated this semantic method as a formal tool for describing detailed execution of programs by transition relations between configurations before and after performing an elementary step of every operation. The main ideas about his approach and his motivation is explained in [13].

Structural operational semantics generates labeled transition system consisting of transition rules describing modification of states [22]. A state is a basic notion of structural operational semantics and it can be considered as some abstraction of computer memory. Every transition rule has its premise or premises and a conclusion. Premises and a conclusion are transitions. The rules can be decorated by additional conditions in the form of predicates. A rules has a form

$$\frac{premise_1, \ldots, premise_n, condition}{conclusion}$$

If all premises and (if exist) all conditions are satisfied, then a conclusion is valid [2].

By [22], structural operational semantics is essentially a description of program behavior. Because it provides a detailed description of program performing, its main application area is in implementation of programming [11]. By the years, this semantic method

became very popular among software engineers and it has many extensions for various purposes.

One of the advantages of structural operational semantics is the notion of environment expressing context dependencies. Context dependencies are the relationships required between the declarations and usage of variables in nested blocks with respect to scope rules.

In the last decades many new results were published about structural operational semantics. Turi [21] in his PhD. thesis formulated coalgebraic categorical model of this method and he showed its duality with denotational approach. New approaches to operational semantics were published in [17, 18]. Among new research results in the area of this semantic methods belongs also formulation of modular structural operational semantics published in [7, 10, 19].

2 The language $\mathscr{J}ane$

In our approach to define categorical operational semantics we use a sample imperative language $\mathscr{J}ane$. It consists of traditional syntactic constructions of imperative languages, namely arithmetic and Boolean expressions, variable declarations and statements. For defining formal syntax of $\mathscr{J}ane$ we introduce the following syntactic domains:

$n \in \mathbf{Num}$ - digit strings;
$x \in \mathbf{Var}$ - variable names;
$e \in \mathbf{Expr}$ - arithmetic expressions;
$b \in \mathbf{Bexpr}$ - Boolean expressions;
$S \in \mathbf{Statm}$ - statements;
$D \in \mathbf{Decl}$ - sequences of variable declarations.

The elements $n \in \mathbf{Num}$ have no internal structure from semantic point of view. Similarly, $x \in \mathbf{Var}$ are only variable names without internal structure significant for defining semantics.

The syntactic domain $\mathbf{Expr}$ consists of all well-formed arithmetic expressions created by the following production rule:

$$e ::= n \mid x \mid e + e \mid e - e \mid e * e.$$

A Boolean expression from $\mathbf{Bexpr}$ can be of the following structure:

$$b ::= \mathtt{false} \mid \mathtt{true} \mid e = e \mid e \leq e \mid \neg b \mid b \wedge b.$$

The variables used in programs have to be declared. We consider $D \in \mathbf{Decl}$ as a sequence of declarations:

$$D ::= \mathtt{var}\ x; D \mid \varepsilon$$

where ε is the empty sequence. We assume that variables are implicitly of type integer. This restriction enables us to focus on main ideas of our approach.

We consider five Dijkstra's statements as statemets in language $S \in \mathbf{Statm}$: assignment, empty statement, sequence of statements, conditional statement

and cycle statement together with block statement and input statement:

$$S ::=$$
$$x := e \mid \mathtt{skip} \mid S; S \mid \mathtt{if}\ b\ \mathtt{then}\ S\ \mathtt{else}\ S \mid$$
$$\mathtt{while}\ b\ \mathtt{do}\ S \mid \mathtt{begin}\ D; S\ \mathtt{end} \mid \mathtt{input}\ x.$$

3 Specification of states

A state is a basic concept of structural operational semantics. It can be considered as some abstraction of computer memory. Every variable occurring in a program has to be allocated, i.e. some memory cell is reserved and named within its declaration. We can assign and modify a value of allocated variable inducing change of state. Because of block structure of $\mathscr{J}ane$, we have to consider also a level of block nesting.

According to previous ideas we formulate the signature Σ_{State} for states. We define abstract data type $State$ using types Var and $Value$ of variables and values. A signature Σ_{State} consists of types and operation specifications on the type $State$:

$$\Sigma_{State} =$$

$\underline{types:}$	$State, Var, Value$
$\underline{opns:}$	$init :\rightarrow State$
	$alloc : var, State \rightarrow State$
	$get : Var, State \rightarrow Value$
	$del : State \rightarrow State$

The operation specifications have the following intuitive meaning:

- $init$ creates a new state, the initial state of a program;

- $alloc$ reserves a new memory cell for a variable in a given state (and nesting level);

- get returns a variable value in an actual state;

- del deallocates (releases) all variables together with their values on a given nesting level.

The signature Σ_{State} serves as a basis for constructing our model of $\mathscr{J}ane$ as the category of states.

4 Operational semantics

We construct operational model of $\mathscr{J}ane$ as the category $\mathscr{C}_{State}$ of states. First, we assign to states their representation. The representation of the elements of $Value$ we consider integer numbers together with the undefined value $\perp$:

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

Our representation of type $State$ has to express variable, its value with respect to actual nesting level. Let

$Level$ be a finite set of nesting levels denoted by natural numbers l:

$$l \in Level, \quad Level = \mathbb{N}.$$

Now, we can represent every state $s \in State$ as a function

$$s : Var \times Level \rightharpoonup Value.$$

This function is partially defined, because a declaration does not assign a value to declared variable. Every state s expresses one moment of program execution. Our definition of states can be considered as a table with possibly unfilled cells denoted by $\perp$.

Every state s can be expressed as a sequence:

$$s = \langle ((x, 1), v_1), \ldots, ((z, l), v_n) \rangle$$

of ordered triples

$$((x, l), v),$$

where (x, l) is declared variable x on nesting level l with actual (possibly undefined) value v. Another representation of state is table which contains names of variables, the level of their declaration and actual value stored in the variable:

variable	level	value
x	1	v_1
$\vdots$		
z	l	v_n

The last representation is as a graph of function:

$$graph(s) = \{((x, 1), v_1), \ldots, ((z, l), v_n)\}$$

Now, we can define the representation of operations from Σ_{State} as follows. The operation $[\![init]\!]$ defined by

$$[\![init]\!] = s_0 = \langle ((\perp, 1), \perp) \rangle$$

creates the initial state of a program, with no declared variable. Its role is to set nesting level to value 1:

variable	level	value
$\perp$	1	$\perp$

The operation $[\![alloc]\!]$ is defined by

$$[\![alloc]\!](x, s) = s \circ ((x, l), \perp),$$

where '$\circ$' is concatenation of sequences. This operation sets actual nesting level to declared variable. Because of undefined value of declared variable, the operation $[\![alloc]\!]$ does not change the state:

variable	level	value
$\vdots$	$\vdots$	$\vdots$
x	l	$\perp$

The operation $[\![get]\!]$ returns a value of a variable declared on the highest nesting level and can be defined by

$$[\![get]\!](x, \langle \ldots, ((x, l_i), v_j), \ldots, ((x, l_k), v_{k'}), \ldots \rangle) = v_l,$$

where $l_i < l_k$.

The operation $[\![del]\!]$ deallocates (forgets) all variables declared on the highest nesting level l_i:

$$[\![del]\!](s \circ \langle ((x_i, l_j), v_k), \ldots, ((x_n, l_j), v_m) \rangle) = s.$$

variable	level	value
$\vdots$	$\vdots$	$\vdots$
x	l_i	v
x_i	l_j	v_k
$\vdots$	$\vdots$	$\vdots$
x_n	l_j	v_m

We construct the category $\mathscr{C}_{State}$ as a category of states defined above. Category objects are states s with special object $s_\perp = ((\perp, \perp), \perp)$ expressing an undefined state.

Category morphisms express change of states caused by execution of statements and they will be defined later.

5 Arithmetic and Boolean expressions

Arithmetic and Boolean expressions serve for computing values of two implicit types of the language $\mathscr{J}ane$. In defining semantics of both types of expressions, an actual state is used but not changed in the process of evaluation. The following tables (Table 1 and Table 2) define semantic functions together with corresponding semantic operations for arithmetic and Boolean expressions.

$$[\![e]\!] : \textbf{State} \rightarrow \textbf{Value}.$$

$$[\![n]\!]s = \mathbf{n}$$

$$[\![x]\!]s = [\![get]\!](x, s)$$

$$[\![e_1 + e_2]\!] = [\![e_1]\!]s \oplus [\![e_2]\!]s$$

$$[\![e_1 - e_2]\!] = [\![e_1]\!]s \ominus [\![e_2]\!]s$$

$$[\![e_1 * e_2]\!] = [\![e_1]\!]s \otimes [\![e_2]\!]s$$

Table 1: Semantics of arithmetic expressions

$$[\![b]\!] : \textbf{State} \rightarrow \textbf{Bool}.$$

$$[\![\texttt{true}]\!]s = \textbf{true}$$

$$[\![\texttt{false}]\!]s = \textbf{false}$$

$$[\![e_1 = e_2]\!]s = \begin{cases} \textbf{true} & \text{if } [\![e_1]\!]s = [\![e_2]\!]s \\ \textbf{false} & \text{otherwise} \end{cases}$$

$$[\![e_1 \le e_2]\!]s = \begin{cases} \textbf{true} & \text{if } [\![e_1]\!]s \le [\![e_2]\!]s \\ \textbf{false} & \text{otherwise} \end{cases}$$

$$[\![\neg b]\!]s = \begin{cases} \textbf{true} & \text{if } [\![\neg b]\!]s = \textbf{false} \\ \textbf{false} & \text{otherwise} \end{cases}$$

$$[\![b_1 \wedge b_2]\!]s = \begin{cases} \textbf{true} & \text{if } [\![b_1]\!]s \le [\![b_2]\!]s = \textbf{true} \\ \textbf{false} & \text{otherwise} \end{cases}$$

Table 2: Semantics of Boolean expressions

Evaluation of expressions has no effect on states, i.e. objects of our category $\mathscr{C}_{State}$. **Value** and **Bool** are semantic domains for integers and Booleans, resp.

$$\textbf{Value} = \mathbb{Z}, \quad \textbf{Bool} = \mathbb{B},$$

where $\mathbb{B}$ is the set containing Boolean values $\{\textbf{true}, \textbf{false}\}$.

6 Declarations

Every variable occurring in a $\mathscr{J}ane$ program has to be declared. Declarations are elaborated, i.e. a memory cell is allocated and named by a declared variable. Therefore elaboration of a declaration

$$\texttt{var } x$$

is represented as an endomorphism:

$$[\![\ ]\!]_D : s \to s$$

for a given state s and defined by

$$[\![\texttt{var } x]\!]s = alloc(x, s).$$

A sequence of declarations is represented as a composition of corresponding endomorphisms:

$$[\![\texttt{var } x; D]\!]s = [\![D]\!] \circ alloc(x, s).$$

If we consider a state as a table, a declaration create new entry (raw) for declared variable with the actual level of nesting and undefined value

$$((x, l), \bot).$$

7 Statements

Statements are the most important constructions of procedural/imperative languages. They execute program

actions, i.e. they get values from the actual state and provide new values. A state is changed if a value of allocated variable is modified. This change of state we model in category $\mathscr{C}_{State}$ by morphisms between objects:

$$[\![S]\!] : \textbf{State} \rightharpoonup \textbf{State}. \tag{1}$$

The morphism (1) is a partial because execution of some statements does not need to provide defined state, e.g. infinite cycle.

Statements are executed in sequence, as they are written in program text. In this contribution we do not consider the statements breaking sequential execution, e.g. `goto` statement or exceptions.

Assignment statement $x := e$ stores a value of arithmetic expression e in a state s in a memory cell allocated for variable x on maximal (highest) level of nesting. This condition ensures that local variable visible in given scope is used.
The semantics is as follows

$$[\![x := e]\!]s = \begin{cases} s\,[\![[\![e]\!]s/x] & \text{for } ((x, max\,l), v) \in s; \\ \bot & \text{otherwise.} \end{cases}$$
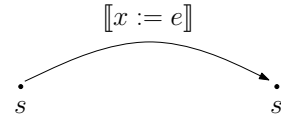
and it is expressed by a morphism in the Fig. 1.

Figure 1: Morphism for assignment

The notation

$$s\,[\![[\![e]\!]s/x]$$

describes that initial state s is modified for variable x by a value $[\![e]\!]s$ of arithmetic expression in the state s.

The empty statement `skip` does nothing, i.e. it does not change state. Clearly, it is identity on state s (Fig. 2).

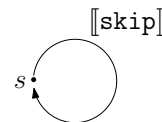$$[\![\texttt{skip}]\!] = id_s$$
$$[\![\texttt{skip}]\!]s = s$$

Figure 2: Morphism for empty statement

A sequence of statements is executed one by one and can be modeled as composition of morphisms (Fig 3)

$$[\![S_1; S_2]\!] = [\![S_2]\!] \circ [\![S_1]\!]$$
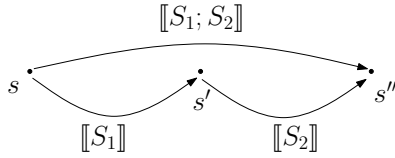
Figure 3: Composition of morphisms

and defined for a state s by

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

Because the body of a program is a sequence of statements, program semantics is a path in category $\mathscr{C}_{State}$.

Conditional statement

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

causes branching of execution depending on a value of Boolean expression (Fig. 2):

$$[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]s = \begin{cases} [\![S_1]\!]s & \text{if } [\![b]\!]s = \mathbf{true}; \\ [\![S_2]\!]s & \text{otherwise.} \end{cases}$$
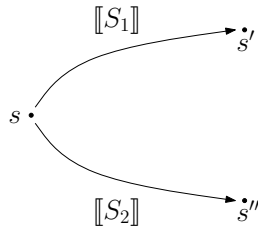
Figure 4: Conditional branching

A cycle

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

also depends on a value of Boolean expression b. If b is true in initial state, the body S of a cycle is executed, then again b is evaluated in modified state. If a value b is not valid, execution of cycle statement is finished. Cycle statement is semantically equivalent with the following conditional statement:

$$[\![\texttt{while } b \texttt{ do } S]\!]s = \\ [\![\texttt{if } b \texttt{ then } (S; \texttt{while } b \texttt{ do } S) \texttt{ else skip}]\!]s$$

Input statement $\texttt{input } x$ serves for reading input value that is stored in given variable x. Because the value of variable is changed, execution of input statement causes modification of state. If a variable x is not declared, the final state is undefined:

$$[\![\texttt{input } x]\!]s = \begin{cases} s\,[v/x] & \text{for } ((x, max\ l)\,, v') \in s; \\ \bot & \text{otherwise.} \end{cases}$$

The notation

$$s\,[v/x]$$

describes a new state where the value v' of local variable x is modified by an input value v.

Programs in $\mathscr{J}\!ane$ can have nested blocks together with declarations of local variables. Execution of block statement

$$\texttt{begin } D; S \texttt{ end}$$

follows in steps:

- nesting level l is incremented. We represent this step by fictive entry in state table

$$((\texttt{begin}, l+1)\,, \bot)$$

  i.e. endomorphism $s \rightarrow s$;

- local declarations are elaborated on new nesting level $l + 1$;

- the body S of block is executed;

- locally declared variables are forgotten at the end of block. We model this situation using operation $[\![del]\!]$.

The semantics of block statement is the following composition of morphisms:

$$[\![\texttt{begin } D; S \texttt{ end}]\!]s = \\ ([\![del]\!] \circ ([\![S]\!] \circ [\![D]\!])) (s \circ ((\texttt{begin}, l+1)\,, \bot)))$$

It follows from the construction of category of states fulfilling of its base properties:

- each object has identity morphism defined;

- for any two composable morphisms there exists such a morphism which is their composition.

8 An example

We show our approach on a simple example. We consider here trivial a program written in $\mathscr{J}\!ane$ with one nested block with local variables which are modified inside the block:

$$
\begin{aligned}
&\texttt{var } x; \texttt{var } y; \\
&x := 1; \\
&y := 10; \\
&\texttt{begin} \\
&\quad \texttt{var } x; \\
&\quad x := 5; \\
&\quad y := y + x; \\
&\texttt{end}; \\
&y := y - x;
\end{aligned}
$$

$$s_0$$

$\perp$	1	$\perp$

Figure 5: Initial state
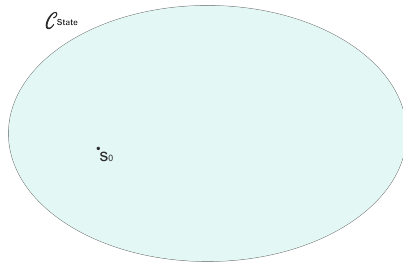
$\mathcal{C}_{\text{State}}$

$\cdot s_0$

Figure 6: Initial state in category

The initial state has only starting information in the state table (Fig. 5) and we start to construct a path in category of states with an initial state in category (Fig. 6).

When entering the local block, we must to note this information into state table with fictive entry. Then actual nesting level is incremented to the new value which means that all variables declared in this local block have nesting level actualized. For example, the state table after local variable declaration is depicted in the Fig. 7.

$$s_2$$

x	1	1
y	1	10
begin	2	$\perp$
x	2	$\perp$

Figure 7: State table with enetring local block

The category contains a path which is depicted in the Fig. 8.
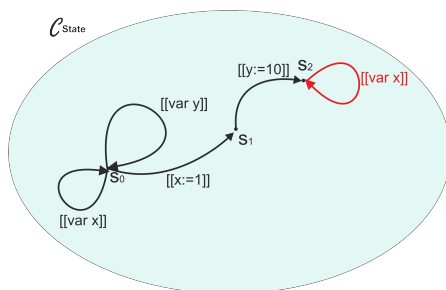
$\mathcal{C}_{\text{State}}$

Figure 8: Path with entering the local block

When we are at the end of the block, we simply forget all local variables with the actual level of nesting (Fig. 9) which leads to resuming values of variables

from previous level.

$$s_5$$

x	1	1
y	1	15
begin	2	$\perp$
x	2	5

Figure 9: End of the local block execution

Actual path in category is depicted in the Fig. 10.

$\mathcal{C}_{\text{State}}$
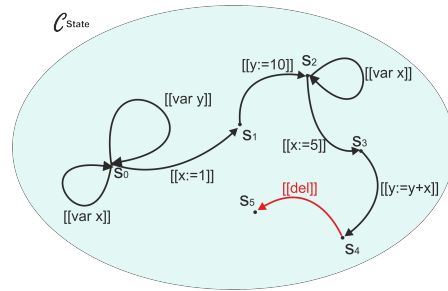
Figure 10: End of the local block

Finally, the last morphism completes the path in category (Fig. 11) and state table (Fig. 12).

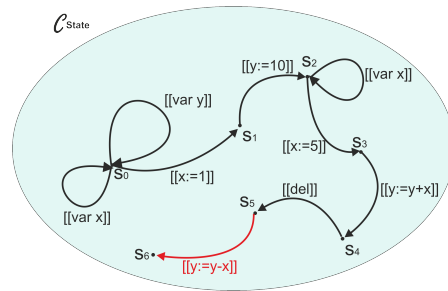$\mathcal{C}_{\text{State}}$

Figure 11: Complete path representing the whole program

$$s_6$$

x	1	1
y	1	14

Figure 12: State table after the program finishes

The semantics of the program is expressed as a path in category of states from initial state into final (the last) state s_6.

9 Conclusion

We presented a new approach to operational semantics by categories. We constructed the category of states

$\mathscr{C}_{State}$ where states of memory are objects and state changes (computations) are morphisms. The semantics of program is defined as composition of morphisms from initial state into final state and is represented in category as a path of all morphisms that represent each program step. Categories have beautiful illustrative power when expressing some relations graphically, our approach is very good understandable for students and also for software engineers. We would like to focus on types of data structures, exceptions and jumps and procedures. We assume that computation by procedure shall be defined in separate category. Each category that represent computation of procedure will be defined as object of total category for procedure environments.

Acknowledgments

References

[1] ABRAMSKY, S. Semantics of interaction: an introduction to game semantics. In *Proceedings of the 1996 CLiCS Summer School, Isaac Newton Institute* (1997), Cambridge University Press, pp. pp. 1–31.

[2] ACETO, L., J., F. W., AND VERHOEF, C. Structural operational semantics. In *Handbook of Process Algebra* (1999), Elsevier, pp. pp. 197–292.

[3] BAGNARA, R., HILL, P. M., PESCETTI, A., AND ZAFFANELLA, E. Verification of C programs via natural semantics and abstract interpretation. In *Proceedings of the C/C++ Verification Workshop* (Oxford, UK, 2007), p. 75–80.

[4] CURIEN, P. Notes on game semantics. http://www.pps.jussieu.fr/ curien/Game-semantics.pdfElectronic Edition, 2006.

[5] EHRIG, H., AND MAHR, B. *Fundamentals of Algebraic Specification 1, 2*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, New York, Berlin, Heidelberg, New York, Tokio, 1985, 1990.

[6] HOARE, C. A. R. An axiomatic basis for computer programming. *Commun. ACM Vol. 12*, No. 10 (1969), pp. 576–580.

[7] JASKELIOFF, M., GHANI, N., AND 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).

[8] JONES, C. B. Operational semantics: concepts and their expression. *Information Processing Letters 88*, 1-2 (2003), 27–32.

[9] KAHN, G. Natural semantics. In *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings (LNCS 247)* (1987), Springer-Verlag, pp. 22–39.

[10] MOSSES, P. Modular structural operational semantics. *Journal of Logic and Algebraic Programming Vol. 60-61* (2004).

[11] NIELSON, F., AND NIELSON, H. R. *Semantics with applications. A formal introduction*. Wiley and Sons, 1992.

[12] PLOTKIN, G. D. A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, University of Aarhus, 1981.

[13] PLOTKIN, G. D. The origins of structural operational semantics. *J. Log. Algebr. Program. Vol. 60-61* (2004), pp. 3–15.

[14] RISTIĆ, S., ALEKSIĆ, S., ČELIKOVIC, M., DIMITRIESKI, V., AND LUKOVIĆ, I. Database reverse engineering based on meta-models. *Central Europ. J. Computer Science Vol. 4*, No. 3 (2014), pp. 150–159.

[15] SCHMIDT, D. A. *Denotational semantics. Methodology for language development*. Allyn and Bacon, 1986.

[16] SCHMIDT, D. A. Natural-semantics-based abstract interpretation (preliminary version). In *SAS* (1995), vol. 983 of *Lecture Notes in Computer Science*, Springer, pp. 1–18.

[17] SCHMIDT, D. A. Abstract interpretation of small-step semantics. In *Proceedings of the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages. LNCS 1192* (1996), Springer-Verlag, pp. pp. 76–99.

[18] SCHMIDT, D. A. Trace-based abstract interpretation of operational semantics. *LISP and Symbolic Computation Vol. 10*, No. 3 (1998), pp. 237–271.

[19] STATON, S. General structural operational semantics through categorical logic. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science (LICS 2008)* (2008), IEEE Computer Society Press, pp. pp. 166–177.

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

[21] TURI, D. *Functorial Operational Semantics and its denotational dual (Ph.D. thesis)*. PhD thesis, University of Amsterdam, 1996.

[22] TURI, D., AND PLOTKIN, G. Towards a mathematical operational semantics. In *In Proc. 12 th LICS Conf* (1997), IEEE, Computer Society Press, pp. pp. 280–291.

[23] WIRSING, M. Handbook of theoretical computer science (vol. b). MIT Press, Cambridge, MA, USA, 1990, ch. Algebraic Specification, pp. pp. 675–788.