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 engineers 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 post-conditions after executing of statements from truth pre-conditions 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

\[
\text{premise}_1, \ldots, \text{premise}_n, \text{condition} \quad \Rightarrow \quad \text{conclusion}
\]

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 \textit{J}ane

In our approach to define categorical operational semantics we use a sample imperative language \textit{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 \textit{J}ane we introduce the following syntactic domains:

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

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

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

\[
e := n \mid x \mid e + e \mid e - e \mid e \ast e.
\]

A Boolean expression from \textit{Bexp} can be of the following structure:

\[
b := \text{false} \mid \text{true} \mid e = e \mid e \leq e \mid \neg b \mid b \land b.
\]

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

\[
D := \text{var } x; D \mid \varepsilon
\]

where \(\varepsilon\) 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 statements in language \(S \in \text{Statm}\): assignment, empty statement, sequence of statements, conditional statement and cycle statement together with block statement and input statement:

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

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 \textit{J}ane, we have to consider also a level of block nesting.

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

\[
\Sigma_{\text{State}} = \begin{array}{c}
\text{types}: \text{State, Var, Value} \\
\text{ops}: \text{init} : \rightarrow \text{State} \\
\text{alloc} : \var, \text{State} \rightarrow \text{State} \\
\text{get} : \text{Var, State} \rightarrow \text{Value} \\
\text{del} : \text{State} \rightarrow \text{State}
\end{array}
\]

The operation specifications have the following intuitive meaning:

- \text{init} creates a new state, the initial state of a program;
- \text{alloc} reserves a new memory cell for a variable in a given state (and nesting level);
- \text{get} returns a variable value in an actual state;
- \text{del} deallocates (releases) all variables together with their values on a given nesting level.

The signature \(\Sigma_{\text{State}}\) serves as a basis for constructing our model of \textit{J}ane as the category of states.

4 Operational semantics

We construct operational model of \textit{J}ane as the category \(\mathcal{E}_{\text{State}}\) of states. First, we assign to states their representation. The representation of the elements of \textit{Value} we consider integer numbers together with the undefined value \(\bot\):

\[
\text{Value} = \mathbb{Z} \cup \{\bot\}.
\]

Our representation of type \textit{State} has to express variable, its value with respect to actual nesting level. Let
Variable level value

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

\([get][x, ((x, l_1), v_1), \ldots, ((x, l_k), v_k)] = v_1,
where \(l_i < l_k\).

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

\([del]((x, l_1), v_1), \ldots, ((x, l_j), v_j) = s, \ldots, ((x, l_m), v_m) = s,\]

We construct the category \(\mathcal{C}_{State}\) as a category of states defined above. Category objects are states \(s\) with special object \(s_\bot = (((\bot), \bot), \bot)\) 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 \(\texttt{Jane}\).

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] : \text{State} \to \text{Value}.\]

\([n]s = n\]

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

\([e_1 + e_2] = [e_1]s \oplus [e_2]s\]

\([e_1 - e_2] = [e_1]s \otimes [e_2]s\]

\([e_1 \ast e_2] = [e_1]s \odot [e_2]s\]

Table 1: Semantics of arithmetic expressions

\([b] : \text{State} \to \text{Bool}.\]
A sequence of declarations is represented as a composition of corresponding endomorphisms:

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

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 \(\mathcal{C}_{\text{State}}\) by morphisms between objects:

\([S] : \text{State} \rightarrow \text{State}\).

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} \[e\]s/x \text{ for } ((x, \text{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](image)

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 \(\text{skip}\) does nothing, i.e. it does not change state. Clearly, it is identity on state \(s\) (Fig. 2).

![Figure 2: Morphism for empty statement](image)

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

\([S_1; S_2] = [S_2] \circ [S_1]\)
The notation 
\[ \llbracket S_1; S_2 \rrbracket(s) \]
and defined for a state \( s \) by
\[
\llbracket S_1; S_2 \rrbracket s = \llbracket S_2 \rrbracket (\llbracket S_1 \rrbracket s).
\]

Because the body of a program is a sequence of statements, program semantics is a path in category \( \mathcal{C}_{\text{State}} \).

Conditional statement
\[
\text{if } b \text{ then } S_1 \text{ else } S_2
\]
causes branching of execution depending on a value of Boolean expression (Fig. 2):
\[
\llbracket \text{if } b \text{ then } S_1 \text{ else } S_2 \rrbracket(s) = \begin{cases} 
\llbracket S_1 \rrbracket s & \text{if } \llbracket b \rrbracket s = \text{true}; \\
\llbracket S_2 \rrbracket s & \text{otherwise}.
\end{cases}
\]

A cycle
\[
\text{while } b \text{ do } S
\]
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:
\[
\llbracket \text{while } b \text{ do } S \rrbracket(s) = 
\llbracket \text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip} \rrbracket(s).
\]

Input statement \( \text{input } x \) 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:
\[
\llbracket \text{input } x \rrbracket s = \begin{cases} 
{s[v/x]} & \text{for } ((x, \max l), v') \in s; \\
\bot & \text{otherwise}.
\end{cases}
\]

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 \( \mathcal{F}ane \) can have nested blocks together with declarations of local variables. Execution of block statement
\[
\begin{array}{c}
\text{begin } D; S \text{ end}
\end{array}
\]
follows in steps:

- nesting level \( l \) is incremented. We represent this step by fictive entry in state table
\[
((\text{begin}, l + 1), \bot)
\]
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 \( \llbracket \text{del} \rrbracket \).

The semantics of block statement is the following composition of morphisms:
\[
\llbracket \text{begin } D; S \text{ end} \rrbracket s = 
(\llbracket \text{del} \rrbracket \circ (\llbracket S \rrbracket \circ [D]) \circ (\text{begin}, l + 1), \bot))
\]

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 \( \mathcal{F}ane \) with one nested block with local variables which are modified inside the block:

\[
\begin{array}{l}
\text{var } x; \text{var } y; \\
x := 1; \\
y := 10; \\
\text{begin} \\
\text{var } x; \\
x := 5; \\
y := y + x; \\
\text{end}; \\
y := y - x;
\end{array}
\]
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.

![Figure 7: State table with entering local block](image)

The category contains a path which is depicted in the Fig. 8.

![Figure 8: Path with entering the local block](image)

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.

![Figure 9: End of the local block execution](image)

Figure 9: End of the local block execution

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

![Figure 10: End of the local block](image)

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

![Figure 11: Complete path representing the whole program](image)

![Figure 12: State table after the program finishes](image)

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 Central European Conference on Information and Intelligent Systems

Faculty of Organization and Informatics

Varaždin, Croatia

September 23-25, 2015
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

This work has been supported by KEGA grant project No. 050TUKE-4/2012: "Application of Virtual Reality Technologies in Teaching Formal Methods", and by the Slovak Research and Development Agency under the contract No. APVV-0008-10: "Modelling, simulation and implementation of GPGPU-enabled architectures of high-throughput network security tools."

References


