Computing and Informatics, Vol. 36, 2017, 1385–1414, doi: 10.4149/cai_2017_6_1385

NEW APPROACH TO CATEGORICAL SEMANTICS FOR PROCEDURAL LANGUAGES

William Steingartner, Valerie Novitzká
Michaela Bačíková, Štefan Korečko

Faculty of Electrical Engineering and Informatics
Technical University of Košice
Letná 9
042 00 Košice, Slovakia
e-mail: {william.steingartner, valerie.novitzka,
michaela.bacikova, stefan.korecko}@tuke.sk

Abstract. The semantics of programs written in some languages is concerned with the interpretation in various types of models. The purpose of structural operational semantics is to describe how a computation is performed. This method is one of the most popular semantic methods in the community of software engineers. It describes program behavior in the form of state changes caused by the execution of elementary steps. This feature predestinates the usage of the structural operational semantics for implementation of programming languages and also for verification purposes. Another semantic method, denotational semantics, defines changes of states by functions. In this paper a new approach to semantics is presented: behavior of programs, i.e., changes of states are modeled in the category of states. The morphisms category expresses elementary execution steps and the program execution is an oriented path in the category, i.e. composition of morphisms. Our categorical model is constructed for a simple procedural language that contains all basic van Dijkstra's constructs. We enriched our approach also with procedures forming a collection of categories interconnected by functors. This method enables the repeated call of procedures, nesting of procedure calls and recursive calls. Moreover, it allows to illustrate and accentuate dynamics of the program execution. The simplicity of this method does not exclude its mathematical exactness.

Keywords: Category theory, structural operational semantics, state, programming languages

Mathematics Subject Classification 2010: 18B05, 18C50, 68Q55

1 INTRODUCTION

Formal description of programming languages belongs to the important methods serving for assigning exact meanings to programs and helping to implement compilers. Semantics of programming languages is concerned with the interpretation of programs written in programming languages. The main role of semantics is to predict the outcome of a program execution. The semantics can be viewed as a function which maps syntactic elements to the semantic domains. There are several known semantic methods used for various purposes. Denotational semantics formulated by Scott and Strachey in [32] and later by Schmidt [24] 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. Up to the present time, categorical models have been used for denotational semantics of programs. They are based mainly on the category of domains representing types [8, 15, 17, 33] and are suitable particularly for functional languages. Among disadvantages of the denotational approach we can mention the impossibility to describe execution steps and to observe behavior of executed programs. Therefore this method is mainly used in the design of programming languages [28].

Programmers are often interested not only in the meaning of programs but also in observing their behavior. Therefore, other methods, e.g. operational semantics, received much more attention in the community of programmers than denotational semantics [12].

The first form of operational semantics is natural semantics formulated by Kahn [13] and is often called semantics of big steps. The author pursued 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 of whole statements [14, 25]. Natural semantics can be useful for specification languages or in program verification [3].

Structural operational semantics (semantics of small steps) is a simple and direct method for describing the behavior of programs written in a programming language. It requires minimal knowledge of mathematics and is easily understandable by practical programmers [23]. The author of structural operational semantics is Gordon Plotkin. In his work [21], 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 of his approach and his motivation are explained in [22].

Structural operational semantics generates labeled transition systems consisting of transition rules describing modification of states [35]. A transition rule has a form

$(S, s) \to s'$, where S stands for statement and s, s' for states. A state is a basic notion of structural operational semantics and it can be considered as an abstraction of computer memory [2].

According to [35], structural operational semantics is essentially a description of program behavior. Because it provides a detailed description of program execution, its main application area is in implementation of programming languages [19]. Over the years, this semantic method has become very popular with 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 relationships required between the declarations and usage of variables in nested blocks with respect to scope rules.

In the last decades many new results on structural operational semantics were published. Turi in his Ph.D. thesis [34] formulated coalgebraic categorical model of this method and showed its duality with the denotational approach. New approaches to operational semantics were published in [26, 27]. Other research results in the area of this semantic method include also the formulation of modular structural operational semantics published in [10, 18, 30].

Several other semantic methods more or less used in various areas of programming exist. Axiomatic semantics [9] is based on satisfying postconditions after executing statements from truth preconditions before this action. Algebraic semantics [7, 36] specifies abstract data types and models them by heterogeneous algebras. Game semantics [1, 6] describes the meaning of programs in the form of game trees and game arenas and it is suitable for expressing non-determinism.

We presented the basic and preliminary ideas of our approach in [31]. Here we extend this approach with other constructs appearing in real programming languages and with detailed discussion. We provide a new approach to defining semantics of a programming language in terms of categories. We construct an integrated categorical model using denotational and operational features. The basic notion of our approach is state as an abstraction of computer memory. We construct a model as a category of states where environment expressing context dependencies in structural semantics is built in category objects, states, using the nesting level. The dynamics of program execution is modeled by category morphisms that express change of states. Morphisms are defined as functions $[\![S]\!] : s \to s'$, where S stands for statement and s, s' for states, as in denotational approach. The advantage of our model is that the execution of program can be expressed also graphically which is highly illustrative. This idea comes from our categorical model of the intrusion detection system [16].

We construct our integrated categorical model for a simple procedural language *Jane*. At first we specify the signature of states and their representations as partially defined functions. We define semantics of statements as morphisms and then we construct category of states as a model of a procedural programming language. Our model enables to define semantics of local declarations within blocks, therefore it is suitable also for languages with block structure. We extend our approach by in-

troducing procedures to a language and we model them as a collection of categories of states together with suitable functors for procedure call and return to calling item.

2 THE LANGUAGE *JANE*

In our approach to define categorical semantics a simple imperative language *Jane* is used. It consists of traditional syntactic constructions of imperative languages, namely arithmetic and Boolean expressions, variable declarations and statements. For defining formal syntax of *Jane*, the following 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,
- $D \in \mathbf{Decl}$ – sequences of variable declarations.

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

The syntactic domain **Aexpr** consists of all well-formed arithmetic expressions created by the following syntax:

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

A Boolean expression from **Bexpr** can be of the following structure:

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

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

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

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

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

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

3 SIGNATURE OF STATES AND THEIR REPRESENTATION

A state is a basic concept in semantics of imperative languages. It can be considered as an abstraction of computer memory. Every variable occurring in a program has to be allocated, i.e., a memory cell is reserved and named within elaboration of declaration. A value of the allocated variable can be assigned and modified inducing a change of state. Because of the block structure of $\mathscr{J}ane$, the level of block nesting has also to be considered.

3.1 Signature of States

According to the previous ideas about the concept of state we formulate the signature Σ_{State} for abstract data type *State*. A signature is a well-known notion used in algebraic specification of abstract data types [7]. This signature uses types *Var* and *Value* for variables and values, respectively.

The signature Σ_{State} consists of types and operation specifications on the type *State*:

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

The operation specifications have the following intuitive meaning:

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

3.2 Representation of States

In this subsection 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 $\perp$:

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

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 with respect to actual nesting level. Let **Level** be a finite set of nesting levels denoted by natural numbers l:

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

The declaration level allows us to create variable environment, the notion known from structural operational semantics, and enables to distinguish local declarations from global ones.

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

$$s : \mathbf{Var} \times \mathbf{Level} \rightharpoonup \mathbf{Value}. \tag{2}$$

This function is partially defined, because a declaration does not assign a value to the declared variable. We denote partial function with the symbol $\rightharpoonup$. Each state s expresses one moment of program execution. We express a state s as a finite sequence:

$$s = \langle ((x_1, 1), v_1), \ldots, ((x_n, l), v_n) \rangle$$

of ordered triples

$$((x_i, l_j), v_i)$$

where (x_i, l_j) is the declared variable x_i on the nesting level l_j with actual (possibly undefined) value v_i for $i = 1, \ldots, n$ and $j = 1, \ldots, l$. This sequence can also be infinite, e.g. in the case of infinite loop. Sequence can be illustrated by a table with possibly unfilled cells denoted by $\perp$ expressing an undefined value which increases readability (Figure 1).

variable	level	value
x_1	1	v_1
⋮		
x_n	l	v_n

Figure 1. Representation of a state by table

In this paper, we use the sequence representation of states in definitions and we illustrate states by tables in examples.

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

$$[\![init]\!] = s_0 = \langle ((\perp, 1), \perp) \rangle \tag{3}$$

creates the initial state s_0 of a program, with no declared variable. Its role is to set the nesting level to a value 1 at the beginning of program execution (Figure 2).

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

Figure 2. Initial state of the program

The operation $[\![alloc]\!]$ appends a new item to the sequence (creates a new entry in the table of) s and is defined by

$$[\![alloc]\!](x, s) = s \diamond \langle ((x, l), \perp) \rangle \tag{4}$$

where "$\diamond$" is concatenation operation on sequences (representation of states). This operation sets the actual nesting level to the declared variable. Because of the undefined value of the declared variable, the operation $[\![alloc]\!]$ does not change the state (Figure 3).

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

Figure 3. Allocation of the new state

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

$$[\![get]\!](x, \langle \ldots, ((x, l_i), v_i), \ldots, ((x, l_k), v_k), \ldots \rangle) = v_k \tag{5}$$

where $l_i < l_k$, $i < k$ for all i, from the definition of state.

The operation $[\![del]\!]$ deallocates (forgets) all the variables declared on the highest nesting level l_j (Figure 4):

$$[\![del]\!](s \diamond \langle ((x_i, l_j), v_k), \ldots, ((x_n, l_j), v_m) \rangle) = s. \tag{6}$$

variable	level	value
⋮	⋮	⋮
x	l_{j-1}	v
x_i	l_j	v_k
⋮	⋮	⋮
x_n	l_j	v_m

Figure 4. Deallocation of all variables declared on the level l_j

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

$$s_\perp = \langle ((\perp, \perp), \perp) \rangle \tag{7}$$

expressing the undefined state. We close this section with definition of semantics for arithmetic and Boolean expressions. The meaning of these expressions depends on

an actual state s which stays unchanged. The evaluation of arithmetic and Boolean expressions produces transient values that are used within execution of statements.

3.3 Semantics of Arithmetic and Boolean Expressions

Arithmetic and Boolean expressions serve for computing values of two implicit types of the language *Jane*, integer and Boolean, respectively. In defining semantics of both types of expressions, an actual state is used but not changed in the process of evaluation. We introduce a new semantic domain **Bool** for Boolean values, that is defined as

$$\mathbf{Bool} = \mathbb{B} \tag{8}$$

where $\mathbb{B}$ is the set containing values $\{\mathbf{true}, \mathbf{false}\}$. The following Table 1 presents the semantic functions together with the corresponding semantic definitions for arithmetic and Boolean expressions.

$[\![e]\!] : \mathbf{State} \to \mathbf{Value}$	$[\![b]\!] : \mathbf{State} \to \mathbf{Bool}$
$[\![n]\!]s = \mathbf{n}$	$[\![\texttt{true}]\!]s = \mathbf{true}$
$[\![x]\!]s = [\![get]\!](x, s)$	$[\![\texttt{false}]\!]s = \mathbf{false}$
$[\![e_1 + e_2]\!]s = [\![e_1]\!]s \oplus [\![e_2]\!]s$	$[\![e_1 = e_2]\!]s = \begin{cases} \mathbf{true}, & \text{if } [\![e_1]\!]s{=}[\![e_2]\!]s \\ \mathbf{false}, & \text{otherwise} \end{cases}$
$[\![e_1 - e_2]\!]s = [\![e_1]\!]s \ominus [\![e_2]\!]s$	$[\![e_1 \leq e_2]\!]s = \begin{cases} \mathbf{true}, & \text{if } [\![e_1]\!]s{\leq}[\![e_2]\!]s \\ \mathbf{false}, & \text{otherwise} \end{cases}$
$[\![e_1 * e_2]\!]s = [\![e_1]\!]s \otimes [\![e_2]\!]s$	$[\![\neg b]\!]s = \begin{cases} \mathbf{true}, & \text{if } [\![\neg b]\!]s{=}\mathbf{false} \\ \mathbf{false}, & \text{otherwise} \end{cases}$
	$[\![b_1 \wedge b_2]\!]s = \begin{cases} \mathbf{true}, & \text{if } [\![b_1]\!]s{=}[\![b_2]\!]s{=}\mathbf{true} \\ \mathbf{false}, & \text{otherwise} \end{cases}$

Table 1. Semantics of arithmetic and Boolean expressions

4 CATEGORY OF STATES AS MODEL OF *JANE*

In the previous section we defined states as sequences of tuples. Now we construct a model of language *Jane* as a category of states, $\mathscr{C}_{State}$. In this category we consider:

- states as category objects; and
- functions on states, possibly partially defined, as category morphisms.

Functions on states represent elaborations of declarations and executions of statements. In the following section we define the semantics of variable declarations. Then we define execution of statements as functions from state to state and in the last section we verify that so constructed structure $\mathscr{C}_{State}$ is a category.

4.1 Elaboration of Declarations

Each variable occurring in a *Jane* 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 a function on state s:

$$[\![\texttt{var}\ x]\!] : s \to s \tag{9}$$

for a given state s and defined by

$$[\![\texttt{var}\ x]\!]s = [\![alloc]\!](x, s). \tag{10}$$

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

$$[\![\texttt{var}\ x; D]\!]s = [\![D]\!] \circ [\![alloc]\!](x, s). \tag{11}$$

Each declaration of variable actualizes an environment of variables in terms of operational semantics.

Graphically, a variable declaration is depicted in the table by creating a new entry for the declared variable with the actual level of the nesting and undefined value

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

4.2 Execution of Statements

Statements are the most important constructions of procedural 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 the allocated variable is modified. We model the change of state by functions between states.

Let S be a statement. Its semantics is a function:

$$[\![S]\!] : s \to s' \tag{12}$$

where s and s' are states. This function can be partially defined in the case the resulting state s' is undefined, $s' = s_\bot$ as it was introduced in Section 3.2.

Statements are executed in the order, as they are written in the program text. In this paper, the statements breaking sequential execution, e.g. `goto` statement or exceptions are not considered.

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

The semantics is as follows:

$$[\![x := e]\!]s = \begin{cases} s\left[\left((x,l),v\right) \mapsto \left((x,l),[\![e]\!]s\right)\right], & \text{for } \left((x,l),v\right) \in s, \\ s_\perp, & \text{otherwise,} \end{cases} \tag{13}$$

and it is illustrated in Figure 5.

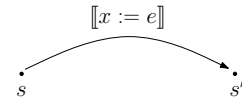

Figure 5. Execution for assignment statement

The notation

$$s' = s\left[\left((x,l),v\right) \mapsto \left((x,l),[\![e]\!]s\right)\right] \tag{14}$$

describes a new state s' that is an actualization of the state s in its tuple for the declared variable x whose value is changed to $[\![e]\!]s$. If x is not declared then s' is undefined,

$$s' = s_\perp.$$

The empty statement `skip` does do nothing, i.e., it does not change the state. Clearly, it is an identity function on a state s (Figure 6) and is defined by:

$$[\![\texttt{skip}]\!]s = s. \tag{15}$$

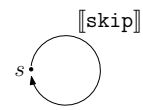

Figure 6. Execution of empty statement

Typical usage of `skip` statement can be rewriting the statement `if` b `then` S by the semantically equivalent statement

$$\texttt{if } b \texttt{ then } S \texttt{ else skip}.$$

That means, if $[\![b]\!]s = \textbf{false}$, the execution of the conditional statement is modeled by identity on s, i.e., the state s is not changed.

A sequence of statements is executed one by one and can be modeled as a composition of functions (Figure 7)

$$[\![S_1; S_2]\!] = [\![S_2]\!] \circ [\![S_1]\!] \tag{16}$$

defined for a state s by

$$[\![S_1; S_2]\!]s = [\![S_2]\!]\,([\![S_1]\!])\,s. \tag{17}$$

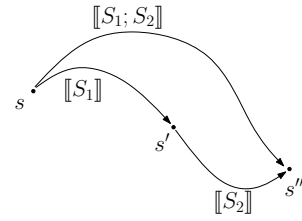

Figure 7. Composition of functions for statement sequence

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

$$[\![S]\!]s_\perp = s_\perp. \tag{18}$$

From this definition follows that achieving undefined state $s_\perp$ is similar as falling into "black hole". It means that execution of program is immediately stopped without resulting state.

Conditional statement

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

causes branching of execution depending on the value of the Boolean expression b. The semantics of conditional statement is defined by:

$$[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]s = \begin{cases} [\![S_1]\!]s, & \text{if } [\![b]\!]s = \textbf{true}, \\ [\![S_2]\!]s, & \text{otherwise.} \end{cases} \tag{19}$$

The function defined in (19) returns the final state

$$s_i = [\![S_i]\!]s$$

where $i = 1$ if $[\![b]\!]s = \textbf{true}$ and $i = 2$ if $[\![b]\!]s = \textbf{false}$. It means that execution of conditional statement is modeled by one morphism depending on the value of Boolean expression b as it is illustrated in Figure 8.

Execution of a cycle statement

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

also depends on the value of Boolean expression b. If b evaluates to **true** in the actual state, the body S of a cycle is executed, then again b is evaluated in a possibly modified state. If b evaluates to **false**, execution of the cycle statement is

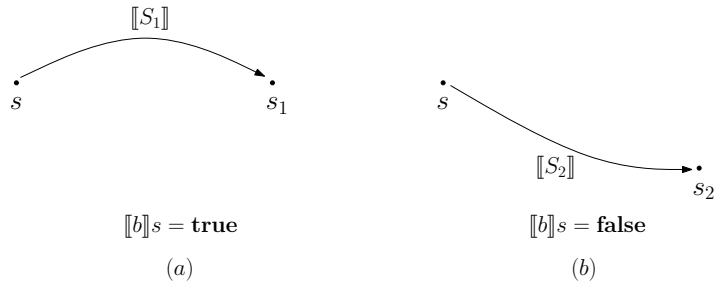

Figure 8. Execution of conditional statement

finished. The cycle statement is semantically equivalent to the following conditional statement:

$$[\![\texttt{while}\ b\ \texttt{do}\ S]\!]s = [\![\texttt{if}\ b\ \texttt{then}\ (S; \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip}]\!]s. \tag{20}$$

The proof can be found in [19]. Therefore, the semantics of the cycle statement is defined as a (possibly infinite) composition of functions.

When modeling execution of this statement, two situations can appear. The first situation depicted in Figure 9 comes when the execution of cycle statement finishes in some state s_n.

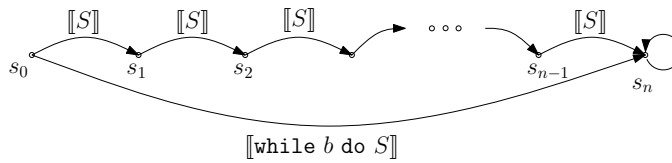

Figure 9. Terminated execution of cycle statement

However, if the condition b is always evaluated to **true**, the cycle statement is executed as infinite composition of functions. In this case, no resulting state is provided by the cycle statement. In Figure 10 we illustrate execution of two possible kinds of infinite cycles. Mostly, the execution in Figure 10 a) is realized when some sequence of states is repeated infinitely. The trivial example of the execution in Figure 10 b) can be the statement `while true do` $x := x + 1$, where each state in the sequence is different. Both these situations can be detected in our approach by observations thanks to graphical representation.

Input statement `input` x serves for reading the input value v' and storing it in the tuple for given variable x. Because the value of the variable is changed, execution of input statement causes modification of the state. If the variable x is not declared, i.e., the tuple for x does not exist in s, the final state of `input` statement is undefined.

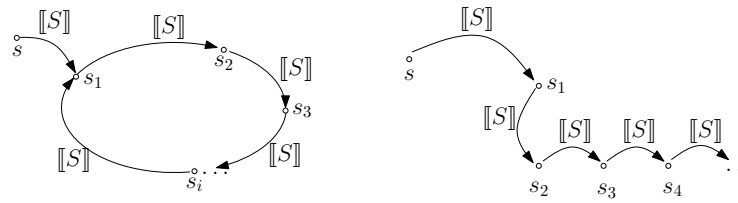

Figure 10. Infinite cycles

$$[\![\texttt{input } x]\!]s = \begin{cases} s\left[((x,l),v) \mapsto ((x,l),v')\right], & \text{for } ((x,l),v) \in s, \\ s_\perp & \text{otherwise.} \end{cases} \tag{21}$$

Programs in *Jane* can have nested blocks together with declarations of local variables. Execution of a block statement

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

follows in several steps:

- nesting level l is incremented. This step is represented by a fictive entry in the state table
$$((\texttt{begin}, l+1), \perp);$$
- local environment of variables in terms of operational semantics is created;
- local declarations are elaborated on the new nesting level $l+1$;
- the body S of block is executed;
- locally declared variables are forgotten at the end of the block. This situation is modeled by using the operation $[\![del]\!]$ in the Equation (6).

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

$$[\![\texttt{begin } D; S \texttt{ end}]\!]s = [\![del]\!] \circ [\![S]\!] \circ [\![D]\!](s \diamond \langle((\texttt{begin}, l+1), \perp)\rangle). \tag{22}$$

4.3 Constructing the Category $\mathscr{C}_{State}$

Now we can define the category $\mathscr{C}_{State}$ of states as follows:

- category objects are states defined in Section 3.2 as sequences of tuples for variables together with special state $s_\perp$;
- category morphisms are functions $[\![S]\!] : s \to s'$ defined in the previous section.

The semantics of a program in $\mathscr{J}ane$ is modeled in $\mathscr{C}_{State}$ as a path, possibly infinite, i.e. a composition of morphisms.

Next, we have to verify that the structure $\mathscr{C}_{State}$ constructed above is a category. Each object in a category has to have the identity morphism. In $\mathscr{C}_{State}$ each state has the identity morphism because after any sequence of statements the `skip` statement represented by identity can follow.

For any two composable morphisms in a category a morphism which is their composition has to exist. This property of category is satisfied in $\mathscr{C}_{State}$ because the semantics of a statement sequence (program) is defined by composition of morphisms. Composition of category morphisms has to be associative. Our category $\mathscr{C}_{State}$ satisfies this property because morphisms are functions and composition of functions is associative.

Our category $\mathscr{C}_{State}$ has the following properties:

- the special object $s_\perp = \langle((\perp, \perp), \perp)\rangle$, undefined state, is a terminal object of our category; from any object there is a unique morphism to this state;
- the initial state $s_0 = \langle((\perp, 1), \perp)\rangle$ is the initial object of our category;
- the category $\mathscr{C}_{State}$ has no products, because a program written in $\mathscr{J}ane$ cannot be simultaneously in more than one state.

We can state that $\mathscr{C}_{State}$ is a category without products and with initial and terminal objects.

5 EXAMPLES OF MODELING PROGRAM EXECUTION IN $\mathscr{C}_{STATE}$

In this section we present the modeling of programs execution in category $\mathscr{C}_{State}$ on selected examples. For better illustration we use only graphical representation of states by tables.

Example 1. We consider a simple program P_1 written in $\mathscr{J}ane$ with one nested block with local variables which are modified inside the block (Listing 1).

```
var x;
var y;
x := 1;
input y;
begin
        var x;
        x := 5;
        y := y + x;
end;
y := y − x
```

Listing 1. Program P_1 with nested block

s_0		
$\perp$	1	$\perp$

Figure 11. Initial state

Let the input value for y be **10**. The initial state s_0 has only starting information in the state table (Figure 11).

Declarations of variables x and y are modeled as endomorphisms on s_0. They create global environment of variables. Assignment statement changes the state s_0 to s_1 according to Equation (13). Input statement stores the value **10** to the variable y and the state is changed to s_2. Entering the local block causes incrementation of the nesting level and the local declaration of the variable x creates a new entry, actual local environment of variables and the state s_2 becomes unchanged (Figure 12).

s_2		
x	1	**1**
y	1	**10**
`begin`	2	$\perp$
x	2	$\perp$

Figure 12. State s_2 with new environment of variables

Execution of two block statements leads to the state s_4 according to Figure 13 a). The ending of local block deletes all entries with the maximum nesting level and creates the state s_5 in Figure 13 b).

s_4		
x	1	**1**
y	1	**15**
`begin`	2	$\perp$
x	2	**5**

(*a*)

s_5		
x	1	**1**
y	1	**15**
`begin`	2	$\perp$
x	2	**5**

(*b*)

Figure 13. Creating and deleting the local environment of variables

Finally, the last assignment statement creates the final state s_6 in Figure 14. The semantics of the program is expressed as follows:

$$\begin{aligned}&(\llbracket y := y - x \rrbracket \circ \llbracket del \rrbracket \circ \llbracket y := y + x \rrbracket \ \circ \llbracket x := 5 \rrbracket \circ \llbracket \texttt{var}\ x \rrbracket \circ \\ &\circ \llbracket \texttt{input}\ y \rrbracket \circ \llbracket x := 1 \rrbracket \circ \llbracket \texttt{var}\ y \rrbracket \circ \llbracket \texttt{var}\ x \rrbracket)\,(s_0) = s_6\end{aligned} \tag{23}$$

s_6		
x	1	**1**
y	1	**14**

Figure 14. State table after the program finishes

and is illustrated as the finite path in the category of states from the initial state s_0 into the final state s_6 (Figure 15).

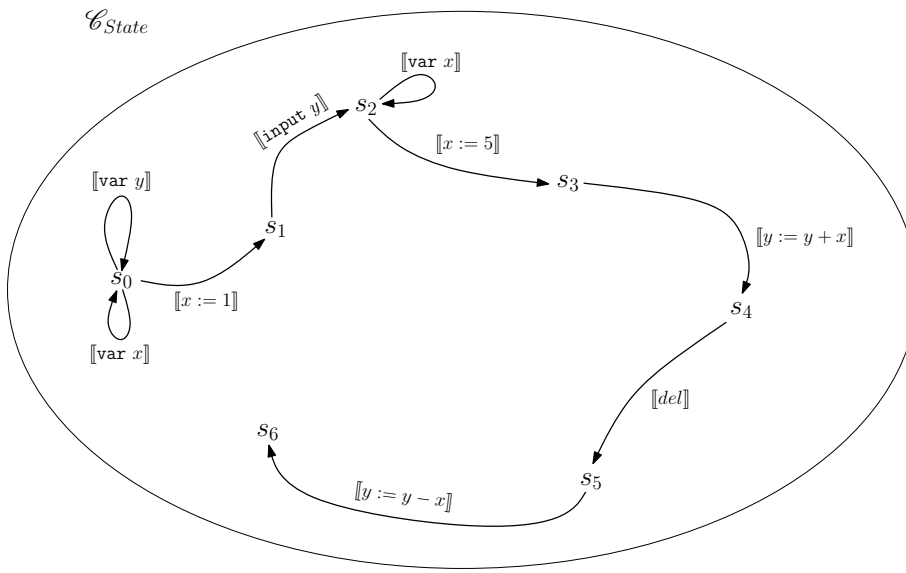

Figure 15. The path of execution P_1

Example 2. Consider the program P_2 in Listing 2 computing the factorial of its input value. Following the definitions in Sections 4.1 and 4.2, states changed during execution of this program are in Figure 16, and Figure 17 shows the path (composition of morphisms) how P_2 is executed step-by-step from the initial state s_0 to the final state s_6. In this figure, `while` statement is executed in five steps that are illustrated from state s_2 to state s_6, i.e., this cycle is composition of all these steps.

```
var x;
var y;
input x;
y := 1;
while ¬(x = 1) do (y := y * x; x := x − 1)
```

Listing 2. Program P_2 for factorial computation

The semantics of the program is expressed as follows:

$$(\llbracket \texttt{skip} \rrbracket \circ \llbracket x := x - 1 \rrbracket \circ \llbracket y := y * x \rrbracket \circ \llbracket x := x - 1 \rrbracket \circ \llbracket y := y * x \rrbracket \circ \\ \circ \llbracket y := 1 \rrbracket \circ \llbracket \texttt{input } x \rrbracket \circ \llbracket \texttt{var } y \rrbracket \circ \llbracket \texttt{var } x \rrbracket)(s_0) = s_6. \tag{24}$$

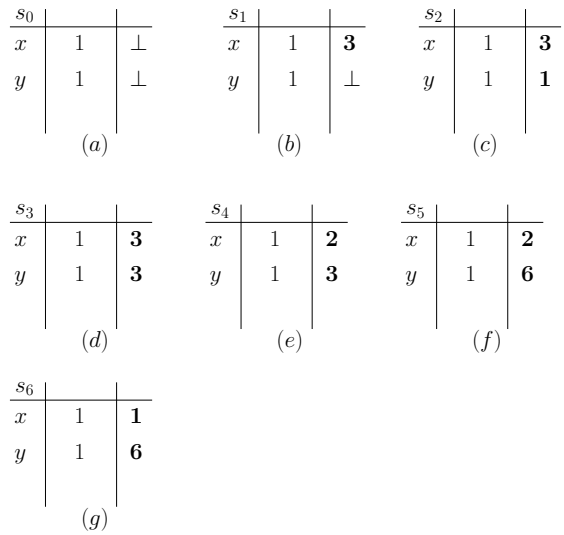

Figure 16. States changes during execution of P_2

6 CATEGORICAL SEMANTICS FOR PROCEDURES

In the previous sections, the categorical model of programming language *Jane* was defined. Our model is based on category $\mathscr{C}_{State}$ of states. In this section, our approach for the language with procedure declarations and procedure calls is extended. A procedure is a named block that can be called possibly more times by its name from the main program or from another procedure. A procedure has to be declared within block declarations. A procedure declaration consists of its name (possibly with parameters), the local declarations and the body of a procedure. When a procedure is called, its parameters are replaced by arguments and the body of a procedure is assigned to its name. In this paper, for simplification only one

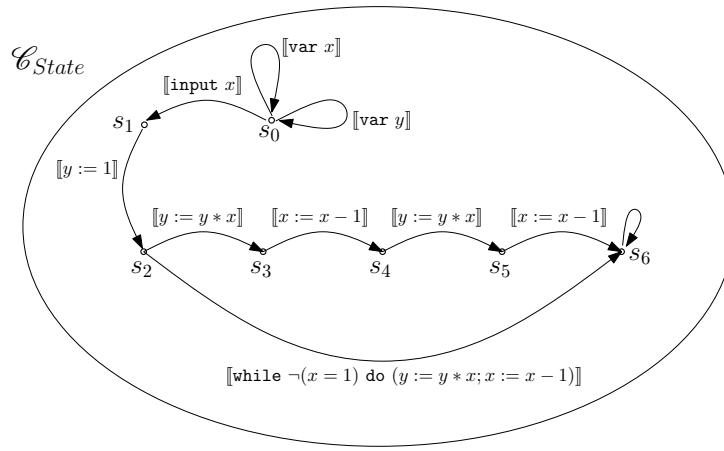

Figure 17. The path of execution P_2

parameter is considered but it is easy to extend this approach to a finite number of parameters.

Firstly, formal description of extended language *Jane* is formulated. We introduce a new syntactic domain $Dp \in \mathbf{ProcDecl}$ for sequences of procedure declarations with the syntax:

$$Dp ::= \texttt{proc}\ p(t);\ S\ \texttt{return}; Dp \mid \varepsilon$$

where Dp is a sequence of procedure declarations and ε denotes the empty sequence of declarations. A procedure declaration contains procedure name p, its parameter t and its body, a block statement S.

Because of the new sort of declarations, we have to change also the syntax of the block statement and add the syntax for the procedure call with the argument that can be an arithmetical expression e:

$$S ::= ... \mid \texttt{begin}\ D; Dp; S\ \texttt{end} \mid \texttt{call}\ p(e).$$

The semantics of a program is modeled as a collection of categories of states. The category $\mathscr{C}_{State}$ constructed above serves for a main program. A declaration of a procedure p causes the construction of the category $\mathscr{C}_p$ similarly as the category $\mathscr{C}_{State}$. Constructing a new category of states for every declared procedure enables multiple and nested calling of procedures. Every procedure call can start with a different initial state depending on the passed value of its argument and the values of global variables in this state.

Let p be a declared procedure with the parameter t. Its categorical model $\mathscr{C}_p$ has the initial state $s_0^p = \langle((\bot, 1), \bot)\rangle$ and a special object for undefined state $s_\bot^p = \langle((\bot, \bot), \bot)\rangle$, from definition. By construction of such categories for every

declared procedure we build step by step a procedure environment of a program. In other words, every procedure declaration allows to create the corresponding category of states and to update a procedure environment.

The connection between constructed categories of states can be carried out by functors. Two functors are constructed:

$$C : \mathbf{Statm} \to \mathscr{C}_{State} \to \mathscr{C}_p,$$
$$R : \mathbf{Statm} \to \mathscr{C}_p \to \mathscr{C}_{State}.$$

The functor C serves for calling a procedure p. Its definition comes from the following considerations. Let p be a procedure declared by

$$\texttt{proc}\ p(t); S_p\ \texttt{return}$$

where S_p is the body of procedure p. If the statement S in $\mathscr{C}_{State}$ is a call of the procedure p with argument e, $\texttt{call}\ p(e)$, in a state s, then the functor C has to:

- update the initial state s_0^p in $\mathscr{C}_p$ by the state s in $\mathscr{C}_{State}$;
- append a new entry in s_0^p for parameter t;
- increment the nesting level;
- pass the value $[\![e]\!]s$ of the argument to the new entry for parameter t.

If the statement S is other than a call of a procedure, then the image of a state s is the undefined state $s_\perp^p = \langle((\perp, \perp), \perp)\rangle$, the terminal object in $\mathscr{C}_p$. Formally, the functor C works on objects as follows:

$$C(S)s = \begin{cases} s_0^p[\langle((\perp, 1), \perp)\rangle \mapsto s \diamond \langle((t, l+1), [\![e]\!]s)\rangle], & \text{if } S = \texttt{call}\ p(e), \\ s_\perp^p, & \text{otherwise.} \end{cases} \tag{25}$$

For any morphism $s \to s'$ in $\mathscr{C}_{State}$ its image by C is defined as follows to satisfy functoriality of C:

$$C(S)(s \to s') = \begin{cases} s_0^p \to s_\perp^p, & \text{if } S = \texttt{call}\ p(e), \\ id_{s_\perp^p}^p, & \text{otherwise.} \end{cases}$$

Notation (25) denotes replacing the original state s_0^p by a new sequence of entries from the calling program. That can be considered as an extension of state actualization (14) introduced in Section 7.

Executing a procedure p can be modeled in the corresponding category $\mathscr{C}_p$ of states as a finite path of states. The final state is denoted by s_f^p and it is indicated by `return`. The role of functor R is to:

- forget entries in s_f^p of locally declared variables; and
- pass the possibly changed values of global variables to the category $\mathscr{C}_{State}$;

because finishing the procedure body will cause forgetting the values of locally declared variables and decrementation of the nesting level. Therefore, the formal definition of functor R is simpler:

$$\begin{aligned} R(S)s^p &= \begin{cases} [\![del]\!](s^p), & \text{if } s^p = s^p_f, \\ s_\perp, & \text{otherwise,} \end{cases} \\ R(S)(s^p \to s'^p) &= \begin{cases} s_\perp \to s', & \text{if } S = \texttt{return}, \\ id_{s_\perp}, & \text{otherwise.} \end{cases} \end{aligned} \tag{26}$$

The semantics of the statement `call` $p(e)$ is then defined by the commutative diagram (Figure 18) as a composition

$$[\![\texttt{call } p(e)]\!] = R \circ ([\![S_p]\!] \circ C(\texttt{call } p(e))).$$

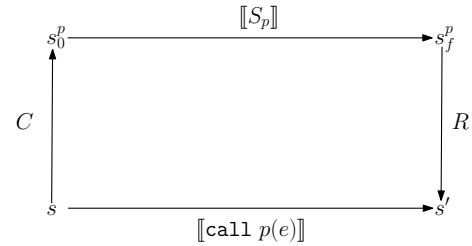

Figure 18. The semantics of procedure calling

As states can be considered as sets of functions, the morphisms $s_\perp \to s'$ always exist [4]. Similarly, the functors C and R can be defined in a similar way between any two categories of states depending on the declared and called procedures. In this paper, recursive procedures are not yet discussed; they are the subject of further research.

Example 3. Consider a fragment of program P in the language *Jane* with procedures.

P :

```
...
proc q(t2); Sq  return
proc p(t1); ...
   proc r(t3); Sr  return
   ...
   call r(e3);
   ...
   call q(e2);
return
...
call p(e1);
...
```

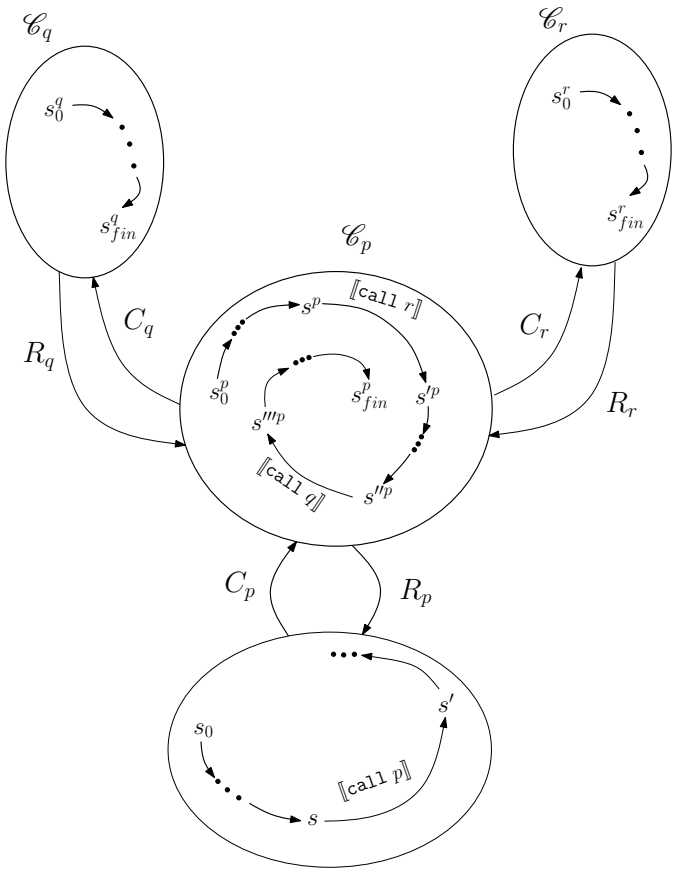

Figure 19. The collection of categories for program P

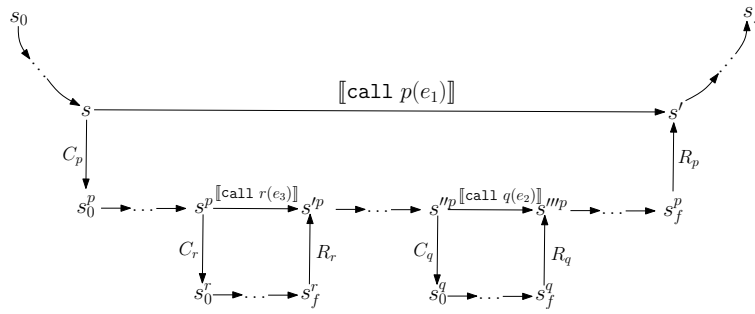

Figure 20. Diagram for program P

In the main program P, procedures p and q are declared. The procedure p is called from the main program. In the procedure p, the local procedure r is declared and called and the global procedure q is also called. Semantics of this program is defined according to the commutative diagram in Figure 20 and illustrated in Figure 19.

Program P is executed as follows: declaration of procedure q in main program causes the construction of category $\mathscr{C}_q$ and declaration of procedure p causes the construction of category $\mathscr{C}_p$.

Procedure p is called in the main program. Functor C_p initiates the state s_0^p from s and the body of p is executed. In p a local procedure r is declared, i.e., category $\mathscr{C}_r$ is constructed. Calling of R in some state s^p initiates the state s_0^r by functor C_r and the body of S_r is executed to some final state s_f^r. Then control is passed back to the body of p by functor R_r and execution follows from some state s'^p. Similarly, execution of the statement `call` $q(e_2)$ starts in some state s''^p and ends in a state s'''^p. Execution of p ends in some final state s_f^p and control is passed back to main program by functor R_p to some state s'.

7 DISCUSSION ABOUT ACHIEVED RESULTS AND OPEN PROBLEMS

In this section we discuss our results and their applicability in real programming. We also present advantages and disadvantages of this approach and several open problems together with ideas how to solve them in the future research. We analyzed these topics on three real imperative programming languages: Modula 3, Python and ABAP. Each of them is used in programming practice but represents different application areas. Modula 3 [5] is a strongly typed extension of Niklaus Wirth's Modula 2 language enabling modular structure of programs and because of its precise definition and understandability is now used mostly for educational purposes. Python [29] is widely used programming language designed for general purposes for writing applications of wide spectrum. ABAP [20] is a language used in SAP for developing business application support and development.

After analysis of these programming languages we recognized that they have many common constructions differing in concrete syntax but with the same abstract syntax which is important for semantic definitions. Therefore we introduced sample programming language *Jane* containing most of the constructions known from real procedural languages and we defined categorical semantics for it. This definition is simply applicable for corresponding constructions in real languages.

The basic model of categorical semantics is the construction of category of states. A state is the fundamental concept and the role of categorical semantics is to define a change of states during the execution of a program. It can be considered as some tracing but based on exact mathematical definitions. In this regard our approach can be useful for several purposes.

First, it enables to model execution of programs or their problematic parts before their actual execution on computers and helps to avoid undesirable runtime errors. This can significantly contribute to the higher reliability of final programs. Second, the execution of programs can be illustrated graphically in contrast with the other semantic methods. Graphical demonstration of program execution is more responsive for humans and leads to better understanding what happens during the execution on computer. Third, our approach shows the dynamics of program execution. Last but not least, we hope that our approach supported by suitable application with graphical output can be well applied also in education of programmers to show program execution step by step.

Our approach presented in this paper can be divided into two parts. In the first part we define categorical semantics for conventional statements of imperative languages that are often called Dijkstra's statements: assignment, empty statement, composition of statements, conditional statement and cycle statement. Each of analyzed languages has these basic statements with inconspicuous differences in concrete syntax. The conditional statement `if` b `then` S is semantically equivalent with the statement

$$\texttt{if } b \texttt{ then } S \texttt{ else skip}$$

in our language $\mathscr{J}ane$. Similarly, the cycle statement `for` $x = e_1$ *to* e_2 `do` S is sematically equivalent with the following statements

$$x := e_1; \texttt{while } x \leq e_2 \texttt{ do } (S;\ x := x + 1).$$

All analyzed languages contain unnamed block statement with local declarations. The semantics of this statement is defined in our approach.

In this paper we limit our language on two basic types: integer and Boolean. It is by the reason to concentrate on our idea without many technical details. Extending $\mathscr{J}ane$ with other types requires to extend the concept of state with type, i.e.

$$s : \mathbf{Var} \times \mathbf{Type} \times \mathbf{Level} \rightharpoonup \mathbf{Value}$$

together with corresponding representation of the semantic domain **Value**. For instance, let T_1, T_2 and T_3 be types. Table representation of a variable $a : T$ of record type T

$$\begin{array}{ll} \texttt{type } T = \texttt{ record} & \\ & \mathit{first} : T_1 \\ & \mathit{scnd} : T_2 \\ & \mathit{third} : T_3 \\ \texttt{end record} & \end{array}$$

can be as follows:

var	type	level	value
...	...	...	...
a	T	l	v_1
		l	v_2
		l	v_3
...	...	...	...

where $v_1 : T_1, v_2 : T_2$ and $v_3 : T_3$ are particular values of record fields. In similar way further types can be introduced into language. Type information needs to be reflected also in declarations extending them with type information. We note that our concept of state works as abstraction of static memory, therefore it is not appropriate for dynamic structures as pointers.

Each of analyzed languages has input statement with the semantics defined in the categorical semantics. The output statement does not affect semantics; therefore we do not concern it.

Another problem arises when we consider unconditional jump statement `go to` and exception. In such cases indirect semantics [19] has to be used. Its principle is to construct continuation functions

$$cont_i : S \times State \rightharpoonup State$$

for $i = 0, \ldots, n$, where c_i defines a change of state arising from execution of the rest of program. For instance, if a program consists of the sequence of statements

$$S_1; S_2; \ldots; S_{n-1}; S_n$$

then c_0 is the identity on final state s_{fin}, $c_0 = id_{s_{fin}}$, c_1 returns the change of state caused by execution of the statement S_n, c_2 returns state change caused by execution of the statements $S_{n-1}; S_n$ and so on. The semantics of the whole program is defined by a function c_n returning the state after executing the sequence $S_1; S_2; \ldots; S_{n-1}; S_n$.

Exceptions also break program control and causes execution of corresponding handler. Therefore indirect semantics shall be defined also for them. We note that if we consider programming language with jumps and executions we will have to construct a new category of states with the same objects, states; but the morphisms will be continuation functions.

The second part of our paper concerns procedures. We construct model of a program with declared procedures as a collection of state categories. The construction of category of states for each declared procedure allows us to call it repeatedly, moreover with nested procedures calls. This collection of categories is also graphically better arranged as modeling of the whole program execution merely in one category of states.

A procedure declaration causes construction of a new state category and the statement `call` is modeled by a functor C copying arguments and actual state to the initial state of called procedure. We use call-by-value approach. A functor R returns control to calling program copying values of global variables. This idea

enables repeated calls of a procedure with different values of arguments and also nested calls of procedures. Further, recursive procedures can be also appropriately modeled by this approach by constructing a new category of states for each unfolding of recursion (Figure 21).

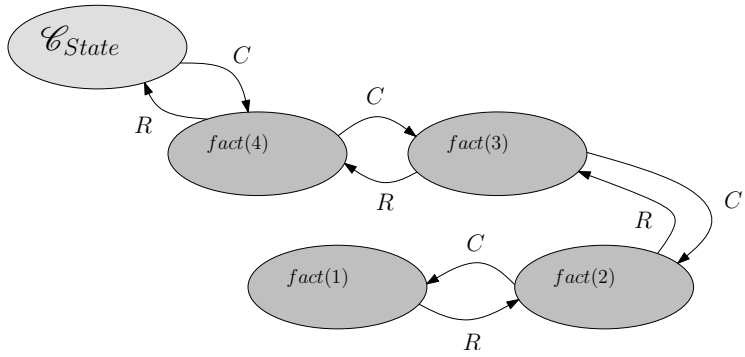

Figure 21. Unfolding recursion

We can point out that we can define the categorical semantics of programs written in real procedural languages using the language constructions discussed above. This research is now in process and there are several open problems.

Modern programming languages enable to program parallel processes. Calling a process causes starting a new task without stopping the current calling program. Independent parallel processes can work without any problems and can be modeled by our categorical semantics. In the case when communication between processes appears, i.e. sharing and modifying data are necessary, the synchronization is needed.

The language Python contains special module `Thread` with synchronization functions `acquire` and `release` serving for synchronization. Modula 3 has special interface `Thread` containing procedures supplying synchronization services. ABAP solves creation of new task using mechanism of Remote Function Call (RFC) and synchronization has to be implemented additionally.

The idea how to model synchronization between processes by categories can be solved as follows. Creating of new process can be solved by a functor *Thread* similarly as calling of procedure. The communication between parallel processes is realized through shared data. To ensure mutual exclusion we need to lock shared data during modifying them and unlock them after the transmission. That can be modeled as a special functor Mut between state categories of communicating processes which writes special token *locked* into the state where shared data are being modified. This idea requires to extend the definition of a state with a new column indicating locking or unlocking the data.

We note that our approach is not available for non-deterministic processes. For modeling them another semantic method is more appropriate – game semantics [11]. Other open problems for our further research are how to model modules (as in

Modula 3) and objects as object-oriented paradigm. It could be interesting to define categorical semantics for component-based program systems.

8 CONCLUSION

A new approach semantics is presented based on categories of states. We constructed the category of states $\mathscr{C}_{State}$ where states are objects and state changes (statements executions) are morphisms. Starting from the analysis of real procedural programming languages we have defined a sample programming language $\mathscr{J}ane$ to illustrate our approach. The semantics of a program is defined as a composition of morphisms from the initial state into a final state and it is represented in our category as a path of morphisms that represent each program step.

In the second part of our paper we extended the language $\mathscr{J}ane$ with procedures and defined the model as a collection of categories connected by functors for procedure calls and returns. Construction a collection of categories of states enables to model repeated calls of procedures, nesting of them and recursive calling.

Our categorical model can express the dynamics of program execution and has a great illustrative power when expressing execution of programs graphically, i.e. step by step.

We have discussed several open problems, for instance how to extend our model by adding types and how to model parallel processes. We also stated which constructs of programming languages are not appropriate for this approach.

A collection of categories inspire us to use our approach as a basis for modeling component composition into component based systems. Functors C and R are foundations for modeling interactions between components which are necessary to investigate in details. Our approach, in which an actual state within calling subroutine is copied into the initial state of the called program, seems to be a suitable way for the instantiation of a particular component state when it is invoked.

Acknowledgment

This work has been supported by Grant No. FEI-2015-18: Coalgebraic models of component systems.

REFERENCES

[1] Abramsky, S.: Semantics of Interaction: An Introduction to Game Semantics. Proceedings of the 1996 CLiCS Summer School, Isaac Newton Institute, Cambridge University Press, 1997, doi: 10.1017/CBO9780511526619.002.

[2] Aceto, L.—Fokking, J. W.—Verhoef, C.: Structural Operational Semantics. Handbook of Process Algebra. Elsevier, 1999, doi: 10.7146/brics.v6i30.20099.

[3] Bagnara, R.—Hill, P. M.—Pescetti, A.—Zaffanella, E.: Verification of C Programs via Natural Semantics and Abstract Interpretation. Proceedings of the C/C++ Verification Workshop, Oxford, UK, 2007.

[4] Barr, M.—Wells, C.: Category Theory for Computing Science. Prentice Hall, 1998.

[5] Cardelli, L.—Donahue, J.—Glassman, L.—Jordan, M.—Kalsow, B.—Nelson, G.: Modula-3 Report (Revised). DEC Systems Research Center (SRC), Research Report 52, 1989.

[6] Curien, P.: Notes on Game Semantics. 2006, `http://www.pps.jussieu.fr/~curien/Game-semantics.pdf` (Electronic Edition).

[7] Ehrig, H.—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, doi: 10.1007/978-3-642-61284-8.

[8] Gunter, C.: Semantics of Programming Languages: Structures and Techniques. MIT Press, Cambridge, MA, USA, 1992.

[9] Hoare, C. A. R.: An Axiomatic Basis for Computer Programming. Communications of the ACM, Vol. 12, 1969, No. 10, pp. 576–580.

[10] Jaskelioff, M.—Ghani, N.—Hutton, G.: Modularity and Implementation of Mathematical Operational Semantics. Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008). Electronic Notes in Theoretical Computer Science, Vol. 229, 2011, No. 5, pp. 75–95, doi: 10.1016/j.entcs.2011.02.017.

[11] Japaridze, G.: In the Beginning Was Game Semantics. In: Majer, O., Pietarinen, A.-V., Tulenheimo, T. (Eds.): Games: Unifying Logic, Language and Philosophy. Springer, 2009, doi: 10.1007/978-1-4020-9374-6_11.

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

[13] Kahn, G.: Natural Semantics. Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science (STACS '87). Springer-Verlag, Lecture Notes in Computer Science, Vol. 247, 1987, pp. 22–39, doi: 10.1007/BFb0039592.

[14] Kuśmierek, J. D. M.—Bono, V.: Big-Step Operational Semantics Revisited. Fundamenta Informaticae, Vol. 103, 2010, No. 1-4, pp. 137–172.

[15] Mihályi, D.—Lukáč, M.—Novitzká, V.: Categorical Semantics of Reference Data Type. Acta Electrotechnica et Informatica, Vol. 13, 2013, No. 4, pp. 64–69, doi: 10.15546/aeei-2013-0051.

[16] Mihályi, D.—Novitzká, V.: Towards the Knowledge in Coalgebraic Model of IDS. Computing and Informatics, Vol. 33, 2014, No. 1, pp. 61–78.

[17] Moggi, E.: Notions of Computations and Monads. Information and Computation, Vol. 93, 1991, No. 1, pp. 55–92.

[18] Mosses, P.: Modular Structural Operational Semantics. The Journal of Logic and Algebraic Programming, Vol. 60-61, 2004, pp. 195–228, doi: 10.1016/j.jlap.2004.03.008.

[19] Nielson, H. R.—Nielson, F.: Semantics with Applications. A Formal Introduction. Wiley and Sons, 1992.

[20] O'Neill, B.: Getting Started with ABAP: Beginner's Guide to SAP ABAP – Introduction to SAP ABAP. 1st edition. SAP PRESS, 2015.

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

[22] Plotkin, G. D.: The Origins of Structural Operational Semantics. The Journal of Logic and Algebraic Programming, Vol. 60-61, 2004, pp. 3–15, doi: 10.1016/j.jlap.2004.03.009.

[23] Ristić, S.—Aleksić, S.—Čeliković, M.—Dimitrieski, V.—Luković, I.: Database Reverse Engineering Based on Meta-Models. Central European Journal of Computer Science, Vol. 4, 2014, No. 3, pp. 150–159, doi: 10.2478/s13537-014-0218-1.

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

[25] Schmidt, D. A.: Natural-Semantics-Based Abstract Interpretation. Static Analysis (SAS '95). Springer, Lecture Notes in Computer Science, Vol. 983, 1995, pp. 1–18, doi: 10.1007/3-540-60360-3_28.

[26] Schmidt, D. A.: Abstract Interpretation of Small-Step Semantics. Proceedings of the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages. Springer-Verlag, Lecture Notes in Computer Science, Vol. 1192, 1996, pp. 76–99.

[27] Schmidt, D. A.: Trace-Based Abstract Interpretation of Operational Semantics. LISP and Symbolic Computation, Vol. 10, 1998, No. 3, pp. 237–271.

[28] Slodičák, V.—Macko, P.: Some New Approaches in Functional Programming Using Algebras and Coalgebras. Electronic Notes in Theoretical Computer Science, Vol. 279, 2011, No. 3, pp. 41–62, doi: 10.1016/j.entcs.2011.11.037.

[29] Summerfield, M.: Programming in Python 3: A Complete Introduction to the Python Language. 2nd edition, Addison-Wesley, 2010.

[30] Staton, S.: General Structural Operational Semantics Through Categorical Logic. Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science (LICS 2008), IEEE Computer Society Press, 2008, doi: 10.1109/LICS.2008.43.

[31] 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, pp. 247–254.

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

[33] Szabó, C.—Slodičák, V.: Software Engineering Tasks Instrumentation by Category Theory. 2011 IEEE 9th International Symposium on Applied Machine Intelligence and Informatics (SAMI), 2011, pp. 195–199.

[34] Turi, D.: Functorial Operational Semantics and Its Denotational Dual. Ph.D. thesis, University of Amsterdam, 1996.

[35] Turi, D.—Plotkin, G.: Towards a Mathematical Operational Semantics. Proceedings of the Twelfth Annual IEEE Symposium on Logic in Computer

Science (LICS '97), IEEE, Computer Society Press, 1997, pp. 280–291, doi: 10.1109/LICS.1997.614955.

[36] Wirsing, M.: Algebraic Specification, In: van Leeuwen, J. (Ed.): Handbook of Theoretical Computer Science (Vol. B). MIT Press, Cambridge, MA, USA, pp. 675–788, 1990, doi: 10.1016/B978-0-444-88074-1.50018-4.

William Steingartner works as Assistant Professor of informatics at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia. He defended his Ph.D. thesis "The *Rôle* of Toposes in Informatics" in 2008. His main fields of research are semantics of programming languages, category theory, compilers, data structures and recursion theory. He also works with software engineering.

Valerie Novitzká is Full Professor of informatics at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia. Her fields of research include semantics of programming languages, non-classical logical systems and their applications in computing science. She also works with type theory and behavioural modeling of large program systems based on categories.

Michaela Bačíková received her Ph.D. in 2014 at the Department of Computers and Informatics of the Technical University of Košice. She is currently Research Assistant at the same department since 2014 and since 2016 she is also the Head of the Information Systems Laboratory. Her research interests include HCI, usability, graphical user interfaces, domain usability, domain-specific languages, software languages and innovations in the teaching process.

Štefan Korečko defended his Ph.D. thesis at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics at Technical University (DCI FEEI TU) in Košice. The title of his thesis was "Integration of Petri Nets and B-Method for the mFDT Environment". Since 2004 he has worked as Assistant Professor at the DCI FEEI TU in Košice. His scientific research is primarily focused on formal methods, Petri nets and B-Method in particular, their integration and utilization in modelling and simulation. He also deals with virtual and augmented reality technologies.