Towards the semantics of recursive procedures in categorical terms

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. *Semantics is concerned with the interpretation of programs written in some programming languages. The purpose of categorical semantics is to model how the computation is performed in categorical terms. Categories are mathematical structures with great illustrative power enabling to model also processes or computations. In this paper we extend our categorical semantics of simple imperative language $\mathscr{J}ane$ to named blocks - procedures. Semantics of procedures we define as a collection of categories interconnected by functors. Our approach enables nested declarations, repeated calls with different arguments and recursive calls without construction of the fixed point known from denotational approach. Such model allows to illustrate and accentuate the dynamics of categorical semantics.*
Keywords. Category theory, imperative language, procedure, recursion, semantics

1 Introduction

Definition of formal semantics of programs belongs to important methods in informatics providing exact and unambiguous meaning of programs written in programming languages of various paradigms. There are several well-known methods, e.g. denotational semantics, operational semantics, action semantics, game semantics, etc. suitable for different purposes and widespreadly used in the community of programmers. Categorical semantics belongs among newer methods that use mathematical structures called categories as models. There are many publications dealing with categorical semantics for functional programming languages, e.g. (Eades, 2012; Jeltsch, 2014) based on type theory modeled by categories of types. But only a few papers concern with imperative paradigm, e.g. (Todoran, 2014).

Therefore our research is concerned on how to define categorical semantics for imperative programming languages. We presented categorical semantics of simple imperative language $\mathscr{J}ane$ which contains five traditional Dijkstra's constructs, user input, unnamed block and variable declarations in (Steingartner & Novitzká, 2015a). In this approach, the semantics of a program was modeled as a path in category of states $\mathscr{C}_{State}$. This path is a composition of morphisms in category representing particular steps of computation.

Procedural abstraction allows the programmer to be concerned mainly with the interface between the function (procedure) and what it computes, ignoring the details of how the computation is accomplished. A procedure is an abstraction of a sequence of commands. The procedure call is a reference to the abstraction. The semantics of the procedure call is determined by the semantics of the procedure body (Plotkin, 2004). For many languages with non-recursive procedures, the semantics may be viewed as simple textual substitution (Novitzká & Slodičák, 2007; Solus, Ovseník, & Turán, 2015).

The first approach to categorical semantics of procedures in language $\mathscr{J}ane$ we presented in (Steingartner & Novitzká, 2015b). Here we defined semantics of procedure execution in separate category which is interconnected with the main category $\mathscr{C}_{State}$ by two functors - the first of them models procedure invocation, the second one ensures return from procedure. This approach allows nested procedure declarations and possibly repeated calls with different parameters without recursion. The semantics of program with procedure invocations is then modeled as the collection of interconnected categories.

We extend our approach in this paper with the semantics of recursive procedures in categorical way. We do not use fixed point approach known from denotational semantics (Schmidt, 1997) for recursive procedures. We model the semantics of recursive procedures in very simple way - as a composition of appropriate morphisms across the collection of categories. Each unfolding of recursion is modeled in a separately constructed category for procedure, with possibly different internal states. This approach is similar to the technique which is used by structural operational semantics and it expresses the dynamics during the processing of recursive procedures.

2 Basic notions

Category theory investigates the internal architecture of mathematics (Brandenburg, 2016). More precisely, category theory is the algebra of functions (Walters, 1992). A category is an abstract structure: a collection of objects, together with a collection of arrows between them. Precisely, a category $\mathscr{C}$ is mathematical structure that consists of a set of objects, e.g. $A, B, \ldots$ and a set of morphisms or arrows of the form $f : A \to B$ between them. Given any object A there is designated identity morphism $id_A : A \to A$. Given morphisms $f : A \to B$, $g : B \to C$, there is designated composite morphism $g \circ f : A \to C$. Because the objects of category can be arbitrary structures, categories are useful in computer science (Barr & Wells, 2002, 1990; Slodičák, 2011), where we often use more complex structures not expressible by sets. Morphisms between categories are called functors, e.g. a functor $F : \mathscr{C} \to \mathscr{D}$ from a category $\mathscr{C}$ into a category $\mathscr{D}$ is considered as a structure-preserving mapping between categories.

3 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. We defined signature on the type of states and its operations in (Steingartner & Novitzká, 2015a). Here we briefly recall basic definitions.

A state is defined as abstract data type. Its signature Σ_{State} contains four operation specification on states:

$$\begin{aligned} &init :\to State \\ &alloc : Var, State \to State \\ &get : Var, State \to Value \\ &del : State \to State \end{aligned}$$

where $Value$ and Var stand for types of values and variables, respectively. The operation $init$ creates a new state which is an initial state of a program. The operation $alloc$ reserves a new memory cell for a variable in a given state and the nesting level. The operation get returns a variable value in an actual state and operation del deallocates all variables together with their values on the highest nesting level.

States are assigned to their representation. 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 $\mathbf{Var}$ 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 $\mathbf{Level}$ be a finite set of nesting levels denoted by natural numbers l:

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

Level of declaration allows us to create variable environment known from structural operational semantics (Plotkin, 2004; Staton, 2008) and enables to distinguish local declarations from global ones.

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

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

which is partially defined because a declaration does not assign a value to the declared variable. We express a state s 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 the declared variable x on the nesting level l with actual (possibly undefined) value v. Sequence can be illustrated by a table with possibly unfilled cells denoted by $\perp$ expressing an undefined value which increases readability. We use tables for better illustration of program execution in Section 6.

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

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

merely creates the initial state s_0 of a program, with no declared variable. This operation is applied only at the beginning of the main 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. 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. The operation $[\![get]\!]$ returns the value of a variable declared on the highest nesting level and is defined by

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

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

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

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

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.

4 Semantics of statements and declarations

Here we briefly introduce the language $\mathscr{J}ane$. This language consists of traditional syntactic constructions of imperative languages, namely arithmetic and Boolean expressions, variable declarations and statements. For this language the well-known syntactic domains are introduced:

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

Five Dijkstra's elementary statements that are elements of syntactic domain $\mathbf{Statm}$, $S \in \mathbf{Statm}$, are considered: assignment, empty statement, sequence of statements, conditional statement and cycle statement. Moreover, we add into syntax a statement for user input and block statement:

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

Declarations of variables are defined as follows:

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

where ε stands for an empty sequence of declarations.

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

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

Statements are executed in the order, as they are written in the program text. The execution of statements we define as functions from state to state. Statements 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:

$$[\![S]\!] : s \to s', \tag{8}$$

where s and s' are states. This function can be partially defined in the case the resulting state s' becomes undefined, $s' = s_\perp$.

The categorical semantics of statements in $\mathscr{J}ane$ is defined as follows:

$$[\![x := e]\!]s = \begin{cases} s\left[((x,l),v) \mapsto ((x,l),[\![e]\!]s)\right], \\ \quad \text{for } ((x,l),v) \in s; \\ \\ s_\perp \\ \quad \text{otherwise.} \end{cases} \tag{9}$$

$$[\![\texttt{skip}]\!]s = s. \tag{10}$$

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

$$[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]s = \begin{cases} [\![S_1]\!]s, \\ \quad \text{if } [\![b]\!]s = \mathbf{true}; \\ \\ [\![S_2]\!]s \\ \quad \text{otherwise.} \end{cases} \tag{12}$$

$$[\![\texttt{while}\ b\ \texttt{do}\ S]\!]s = [\![\texttt{if}\ b\ \texttt{then}\ (S; \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip}]\!]s \tag{13}$$

$$[\![\texttt{input}\ x]\!]s = \begin{cases} s\left[((x,l),v) \mapsto ((x,l),[\![x]\!]s)\right], \\ \quad \text{for } ((x,l),v) \in s \\ \\ s_\perp, \\ \quad \text{otherwise.} \end{cases} \tag{14}$$

$$[\![\texttt{begin}\ D; S\ \texttt{end}]\!]s = [\![del]\!] \circ [\![S]\!] \circ [\![D]\!](s \diamond \langle((\texttt{begin}, l+1), \perp)\rangle). \tag{15}$$

A proof of semantic equivalence of the prefix logic cycle can be found in (Nielson & Nielson, 1992). Therefore, the semantics of the cycle statement is defined as a (possibly infinite) composition of functions. Generally, whenever during the program execution a state s' becomes undefined, i.e. $s' = s_\perp$, then the execution of the whole sequence of statements provides undefined state:

$$[\![S]\!]s_\perp = s_\perp. \tag{16}$$

Each 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 a function on a state s:

$$[\![\texttt{var}\ x]\!] : s \to s \tag{17}$$

for a given state s and defined by

$$[\![\texttt{var}\ x]\!]s = [\![alloc]\!](x, s). \tag{18}$$

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

$$[\![\texttt{var}\ x; D]\!]s = [\![D]\!] \circ [\![alloc]\!](x, s). \tag{19}$$

5 Semantics of procedures

In this section we extend our approach for the language with procedures declarations and procedures calls. A procedure is named block which can be called (possibly) more times by its name from the main program or from another procedure. Special case is recursive procedure which calls itself with possibly changed (usually decremented) input argument until the closing condition is false.

A procedure declaration consists of its name (possibly with parameters), the local declarations and the sequence of statements. Within calling a procedure, its parameters are replaced by arguments and body of a procedure is assigned to its name. In this paper we consider for simplification only one parameter.

A new syntactic domain **ProcDecl** for sequences of procedure declarations is introduced. A syntax of procedures is as follows:

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

Procedure declarations contain procedure name p, its parameter t and a sequence of statements S_p. Next we extend the syntax of $\mathscr{J}ane$ as follows:

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

The semantics of program containing procedure(s) is modeled as a collection of categories of states. One category denoted $\mathscr{C}_{State}$ constructed above serves for a main program. A declaration of procedure p causes the construction of the new category C_p similarly as the main category. Constructing a new category of states for each declared procedure then enables multiple and nested calling of procedures.

Consider a category $\mathscr{C}_p$ constructed for the procedure p. This category has its initial state denoted $s_0^p = \langle((\bot, \bot), \bot)\rangle$. A level of declaration in an initial state of the procedure is undefined and is replaced by actual value when the procedure is called and an appropriate category is constructed.

The connection between constructed categories of states can be carried out by functors. We construct two functors:

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

The functor C serves for calling a procedure p.

If the statement S in $\mathscr{C}_{State}$ is a call of the procedure p with argument e, `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 undefined state $s_\bot^p = \langle((\bot, \bot), \bot)\rangle$, the terminal object in $\mathscr{C}_p$. Formally, the functor C works on objects as follows:

$$C(S)s = \begin{cases} s_0^p[\langle((\bot, l+1), \bot)\rangle \mapsto s \diamond \langle((t, l+1), [\![e]\!]s)\rangle], \\ \quad \text{if } S = \texttt{call}\ p(e) \\ \\ s_\bot^p, \\ \quad \text{otherwise.} \end{cases} \tag{20}$$

Notation (20) denotes replacing the original state s_0^p by a new sequence of entries from the calling program. For any morphism $s \to s'$ in $\mathscr{C}_{State}$ its image by C is defined as follows to satisfy functoriality of C:

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

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_{fin}^p and it is indicated by `return`.

The *rôle* of functor R is:

- to forget entries in s_{fin}^p of locally declared variables; and
- to 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:

$$R(S)s^p = \begin{cases} [\![del]\!](s^p), & \text{if } s^p = s_{fin}^p \\ \\ s_\bot, & \text{otherwise} \end{cases}$$

$$R(S)(s^p \to s'^p) = \begin{cases} s_\bot \to s', & \text{if } S = \texttt{return} \\ \\ id_{s_\bot}, & \text{otherwise} \end{cases} \tag{21}$$

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

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

Now we extend this way of defining the categorical semantics of procedures introduced in (Steingartner & Novitzká, 2015b) and we define the semantics of recursive procedures. Roughly speaking, a recursive procedure is a procedure that makes a call to itself. To prevent infinite recursion, we usually need a conditional statement of some sort where one branch makes

a recursive call, and the other branch does not. The branch without a recursive call is usually the base case. Base cases do not make recursive calls to the function. When a recursive call is made, technically the procedure clones itself, making new copies of the code, the local variables (with their initial values) and the arguments. Program control jumps to the function where the function's code is being executed. These steps are being repeated until the base case is reached. After the computation inside the base case the result is copied into a return value. When the procedure returns back, that clone goes away, but the previous ones are still there, and know what to execute next because their current position in the code was saved. The stack is rewound to its previous position and control jumps back to where the function was called until the last procedure call is not finished.

We will follow this idea and we define categorical semantics of recursive procedures. We use the following notation:

- $\mathscr{C}_{p|j}$ is the j^{th} category for procedure p, constructed when the procedure is called j^{th} time;
- $s^p_{i|j}$ is the state with index i during the program execution enclosed in category of states for procedure p with an index j;
- $s^p_{fin|j}$ is the final state in the category for procedure p with an index j.

The semantics of recursive procedure with one parameter which calls only itself is as follows:

$$\begin{aligned}[\![\texttt{call}\ p(e)]\!] = \quad & \left[R^i(\texttt{return}) \circ [\![S'']\!]\right]_{i=n-1}^{1} \\ & \circ \left(R^n\left(\texttt{return}\right) \circ [\![S_p]\!]\right) \circ \\ & \circ \left[C^i(\texttt{call}\ p(e_i)) \circ [\![S']\!]\right]_{i=2}^{n} \\ & \circ C^1\left(\texttt{call}\ p(e_1)\right)\end{aligned} \tag{22}$$

In the formula (22), upper index at functors C and R means the order of application. At functor C the index of order is increasing, at functor R is decreasing. Statement S_p represents the whole body of procedure, statement S' stands for some subsequence of statements that is executed when the procedure starts until the new call is invoked, and the statement S'' represents the subsequence that is executed after returning the value of previous call to the end of procedure.

This approach does not use the philosophy of constructing the fixed point of recursive function. It shows the dynamics of computation in the similar way as the structural operational semantics (Mosses, 2004).

6 Example

This approach we show on a simple example. We consider program for factorial using recursion. The code for program is as follows:

```
1  var y;
2  y:=1;
3  call fact(4);
4
5  procedure fact(t)
6    begin
7      var z;
8      z:=t;
9      if not(t=1) then
10         call fact(t-1);
11       else skip;
12     y:=y*z;
13     return;
14   end;
```

Listing 1: Example of the factorial computation

The code in Listing 1 demonstrates one of the traditional ways of factorial calculation using recursion. The procedure passes the parameter with call-by-value method. Inside the procedure one local variable is declared.

We start with the description of the program. On the line 1, a variable y is declared. This variable is considered as global. During the program execution, the particular results are stored in it. After the outermost procedure call is finished, the variable y contains the resulting value of factorial calculation. On the next line, the variable y is initialized to a starting value $\mathbf{1}$. On the line 3, the procedure $fact$ is called with the fixed value of argument, the value $\mathbf{4}$.

Calling of a procedure is expressed as follows. The state s_1 in $\mathscr{C}_{State}$ is copied to the state $s^{fact}_{0|1}$ in the new category $\mathscr{C}_{fact|1}$. That means that an initial state $s^{fact}_{0|1}$ is created (Fig. 1) in category $\mathscr{C}_{fact|1}$. This new state contains a fictive entry in the table which means that all local declarations are elaborated on the new level of declarations, here $l = 2$.

$s^{fact}_{0|1}$

var	level	value
y	1	$\mathbf{1}$
begin	2	$\perp$
t	2	$\mathbf{4}$
z	2	$\perp$

Figure 1: An initial state in the first call of $fact$

Inside the procedure, on the line 7 the new local variable z is declared and declaration is elaborated on the level $l = 2$. Assignment statement $z := t$ on the line 8 is executed with the actual value of parameter t, here $t = \mathbf{4}$ and the new state becomes (Fig. 2).

The next statement is conditional statement on the lines 9 and 10. Evaluation of the Boolean expression results into value **true** and program continues with the next procedure call: the command to be executed is

$s^{fact}_{0|1}$

var	level	value
y	1	**1**
begin	2	$\perp$
t	2	**4**
z	2	$\perp$

$s^{fact}_{1|1}$

var	level	value
y	1	**1**
begin	2	$\perp$
t	2	**4**
z	2	**4**

Figure 2: States in procedure after declaration and assignment of the parameter value

$s^{fact}_{0|4}$

var	level	value
y	1	**1**
begin	2	$\perp$
t	2	**4**
z	2	**4**
begin	3	$\perp$
t	3	**3**
z	3	**3**
begin	4	$\perp$
t	4	**2**
z	4	**2**
begin	5	$\perp$
t	5	**1**
z	5	$\perp$

$s^{fact}_{fin|4}$

var	level	value
y	1	**1**
begin	2	$\perp$
t	2	**4**
z	2	**4**
begin	3	$\perp$
t	3	**3**
z	3	**3**
begin	4	$\perp$
t	4	**2**
z	4	**2**
begin	5	$\perp$
t	5	**1**
z	5	**1**

Figure 3: States in the innermost procedure call: initial and final

`call` $fact(t-1)$. So the new category $\mathscr{C}_{fact|2}$ for the next procedure call is created. Inside this new category its initial state with the new level of declaration ($l = 3$) is created.

In our case, the steps on lines 7-9 repeat two more times with two new categories for particular procedure call: $\mathscr{C}_{fact|3}$ and $\mathscr{C}_{fact|4}$. In the last category, $\mathscr{C}_{fact|4}$, the Boolean expression in conditional statement on the line 9 has resulting value **false** and the statement `skip` is executed. In this point no new procedure call is executed and program reached the "bottom" of recursion. The next statement after the conditional one is an assignment $y := y * z$ on the line 11. The state $s^{fact}_{0|4}$ after declaration of variable z is in Fig. 3 on the left hand side. The state $s^{fact}_{fin|4}$ is the final state of the innermost procedure call and is depicted on the right hand side. It is the state before executing the keyword `return`, i.e. before applying the functor R which sends the final state of procedure into the category from which the procedure was called.

After mapping the state $s^{fact}_{fin|4}$ into the previous category $\mathscr{C}_{fact|3}$ all declarations elaborated in $\mathscr{C}_{fact|4}$ are forgotten (deleted) and global variable y is actualized. Similarly, those steps repeat in categories $\mathscr{C}_{fact|3}$, $\mathscr{C}_{fact|2}$ and $\mathscr{C}_{fact|1}$ as follows:

- in $\mathscr{C}_{fact|3}$, the statement $y := y * z$ by taking an actual value **2** stored in z declared on level $l = 4$ sends (maps) the state $s^{fact}_{2|3}$ into $s^{fact}_{fin|3}$ and variable y is actualized to value **2** (Fig. 4, the left hand side);

- in $\mathscr{C}_{fact|2}$, also the statement $y := y * z$ taking an actual value **3** stored in z declared on level $l = 3$ sends the state $s^{fact}_{2|2}$ into $s^{fact}_{fin|2}$ and variable y is actualized to new value **6** (Fig 4, right hand side);

$s^{fact}_{fin|3}$

var	level	value
y	1	**2**
begin	2	$\perp$
t	2	**4**
z	2	**4**
begin	3	$\perp$
t	3	**3**
z	3	**3**
begin	4	$\perp$
t	4	**2**
z	4	**2**

$s^{fact}_{fin|2}$

var	level	value
y	1	**6**
begin	2	$\perp$
t	2	**4**
z	2	**4**
begin	3	$\perp$
t	3	**3**
z	3	**3**

Figure 4: Final states in the third and the second procedure calls before `return`

- in $\mathscr{C}_{fact|1}$ the statement $y := y * z$ is executed for the last time in program, it takes an actual value **4** in variable z declared on level $l = 2$ and variable y is actualized to new value **24** (Fig 5, left hand side);

- finally, the resulting state $s^{fact}_{fin|1}$ is sent to its image s_{fin} in the main category $\mathscr{C}_{state}$ by functor R (Fig 5, right hand side).

$s^{fact}_{fin|1}$

var	level	value
y	1	**24**
begin	2	$\perp$
t	2	**4**
z	2	**4**

s_{fin}

var	level	value
y	1	**24**

Figure 5: Final states of the first procedure call and of the main program

The final result is stored in the resulting state s_{fin}. The value **24** stored in variable y is a result of **4** factorial.

The composition of all morphisms that are parts of procedure call is expressed as follows:

$$[\![\texttt{call } fact(e_1)]\!] =$$

$$= \left[R^i(\texttt{return}) \circ [\![y := y * z]\!]\right]_{i=3}^{1}$$

$$\circ \left(R^4\,(\texttt{return}) \circ [\![y := y * z]\!] \circ [\![\texttt{skip}]\!] \circ\right.$$

$$\left.\circ [\![z := 1]\!] \circ [\![\texttt{var } z]\!]\right)$$

$$\circ \left[C^i(\texttt{call } fact(e_i)) \circ [\![z := 1]\!] \circ [\![\texttt{var } z]\!]\right]_{i=2}^{4}$$

$$\circ\, C^1\,(\texttt{call } fact(e_1))$$

where values of particular expressions e_i are:

$$e_1 = \mathbf{4}, e_2 = \mathbf{3}, e_3 = \mathbf{2}, e_4 = \mathbf{1}.$$

The procedure $fact$ is called four times, and index values run from $i = 1$ to $i = 4$. Here, as said before, indices denote the order of applying the functors C and R. The semantics of the program from Listing 1 is expressed as a path in Fig. 6 (see the last page).

7 Benefits and open problems

In this paper we present a new approach of defining categorical semantics of procedural languages. We follow our results introduced in (Steingartner & Novitzká, 2015a) and we extend the language Jane with procedures possibly with parameters. Usage of categories as models has several advantages. Categories are mathematical structures with exactly defined and proved properties that are useful for semantic definitions. Category of states used in our approach consists of sets as category objects and functions between them as category morphisms. Such category has useful properties important for us, e.g. initial and terminal objects, exponentials, commutative diagrams, global variables etc. In the case of procedures we construct a collection of categories of states for the main program and each declared procedure. The connection between categories is ensured by special morphisms, functors, for procedure call and return. Our approach permits repeated call, nesting of procedures and handling recursive procedures. After all, categories afford opportunity for graphical representation of program semantics that is more illustrative and better understandable as pure mathematical inscription.

Our approach can be useful for several groups of users. First, categories are very popular in modeling processes in computer science and we provide a new area of their usefulness. Moreover, categorical model with its mathematical properties either ensures reliability of a drafted program or shows possible errors before its execution on a computer. Last but not least, graphical character of categories enables to use our approach for practical programmers and also for educational purposes because of its simplicity and visualization.

Admittedly, there are open problems not yet solved in this paper that are the subjects of our further research. Now, we can define categorical semantics for traditional procedural languages. We assume that our model is not proper for languages with nondeterminism, where game semantics is more suitable. To model parallelism a new environment for processes should be introduced using functors for synchronization purposes. Possible extension can be to define categorical semantics of objects and classes used in object-oriented paradigm. We plan to proceed our research with defining categorical semantics for component-based program systems. These mentioned extensions will coerce new functors for defining cooperation between objects and for instantiations of classes. In the case of component-based program systems we need to define functors for calling components (with necessary agreements and dependencies) but each component as a closed system works until its final state.

8 Conclusion

We presented categorical semantics of procedures in this paper. We extended our language Jane with named blocks, procedures with parameters. A declaration of a procedure causes construction of a new category of states. Calling of procedures is realized by a pair of functors: C for calling and R for return. A program with procedures is represented as a collection of categories of states. Our approach enables also multiple calling of a procedure. In the case of recursive procedures, we decided to construct a new category of states for each call to illustrate unfolding of recursion. It seems to be more illustrative than using of fixed point operator in denotational method. Creating a chain of categories of states for recursive calling is near to the operational semantics. Hence the semantics of the whole program is expressed as one compound path. Moreover the recursive procedure is expressed also as a sequence of morphisms until the base case is reached without constructing the fixed point known from denotational semantics. We could say that our approach is highly illustrative and good understandable, suitable also for education and teaching some formal methods. Our next objective is to define category as transition system and to construct coalgebra with an appropriate endofunctor over the category of states for our language which will enable to model observable behavior of executed programs.

Acknowledgments

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

References

Barr, M., & Wells, C. (1990). *Category theory for computing science*. Prentice Hall International.

Barr, M., & Wells, C. (2002). *Toposes, triples and theories*. Springer-Verlag.

Brandenburg, M. (2016). *Einführung in die Kategorientheorie*. Springer Spektrum.

Eades, H. (2012). *The semantic analysis of advanced programming languages*. Unpublished doctoral dissertation, University of Iowa.

Jeltsch, W. (2014). Categorical Semantics for Functional Reactive Programming with Temporal Recursion and Corecursion. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014* (pp. 127–142).

Mosses, P. D. (2004). *Modular structural operational semantics.*

Nielson, H. R., & Nielson, F. (1992). *Semantics with applications: A formal introduction*. New York, NY, USA: John Wiley & Sons, Inc.

Novitzká, V., & Slodičák, V. (2007). On applying stochastic problems in higher-order theories. *Acta Electrotechnica et Informatica*, *7*(3), pp. 58–62.

Plotkin, G. D. (2004). The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming*, *60-61*, pp. 3–15.

Schmidt, D. (1997). *Denotational semantics. Methodology for language development*. USA: Allyn and Bacon.

Slodičák, V. (2011). Some useful structures for categorical approach for program behavior. *Journal of Information and Organizational Sciences*, *35*(1), pp. 93–103.

Solus, D., Ovseník, L., & Turán, J. (2015). Inventory system of vertical traffic signs. In *Radioelektronika (RADIOELEKTRONIKA), 2015 25th International Conference* (pp. 121–124). IEEE.

Staton, S. (2008). General structural operational semantics through categorical logic. In *Logic in Computer Science, 2008. LICS'08. 23rd Annual IEEE Symposium on* (pp. 166–177). IEEE Computer Society Press.

Steingartner, W., & Novitzká, V. (2015a). A new approach to operational semantics by categories. In *Proceedings of the 26th Central European Conference on Information and Intelligent Systems, CECIIS 2015* (pp. 247–254). Varaždin, University of Zagreb.

Steingartner, W., & Novitzká, V. (2015b). A new approach to semantics of procedures in categorical terms. In *2015 IEEE 13th International Scientific Conference on Informatics, INFORMATICS 2015* (pp. 252–257). Poprad, Slovakia.

Todoran, E. N. (2014). Continuation semantics for maximal parallelism and imperative programming. *Automation, Computers, Applied Mathematics*, *23*(1), pp. 29–35.

Walters, R. (1992). *Categories and computer science*. New York, USA: Cambridge University Press.

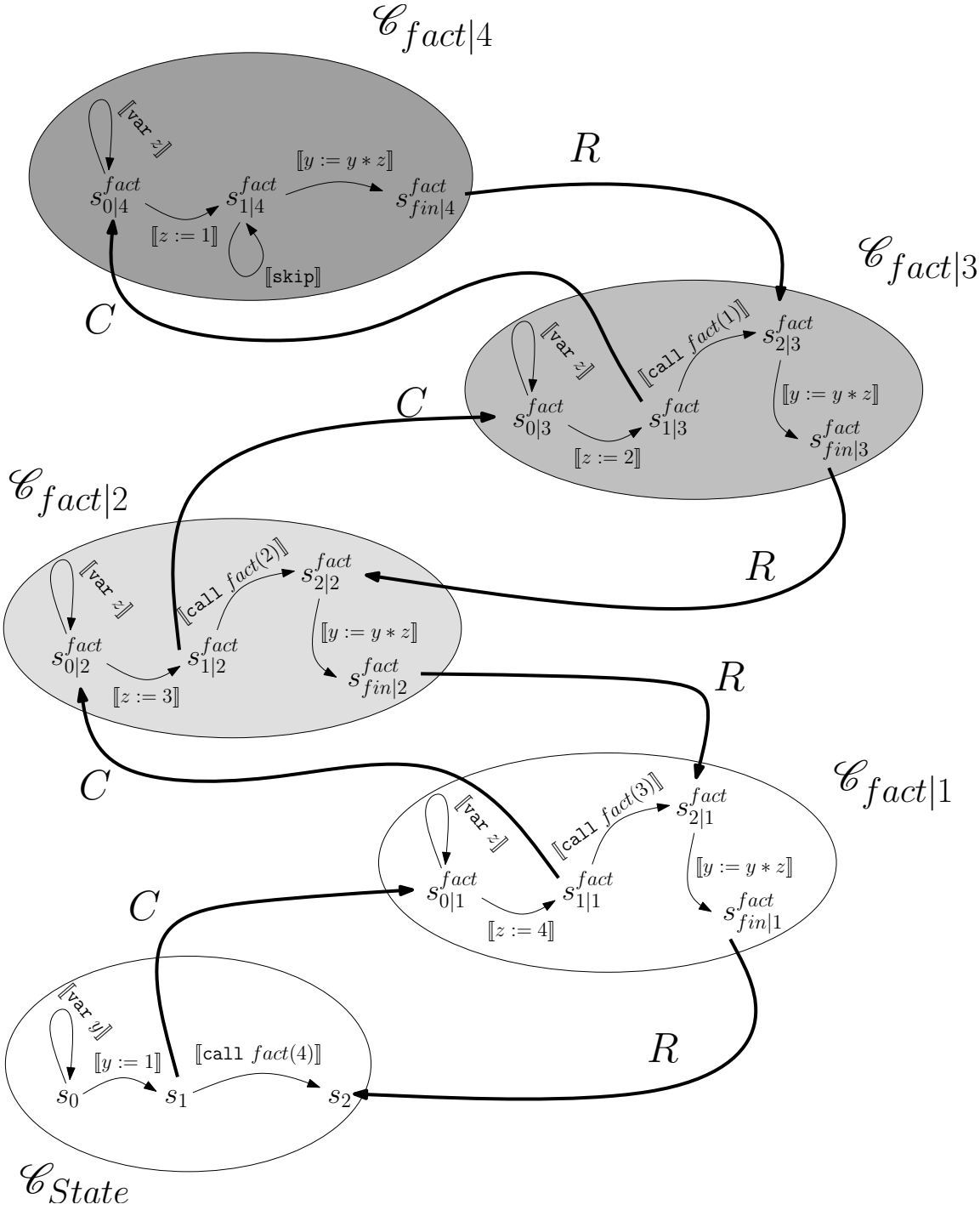

Figure 6: Collection of categories for factorial computation