Selected Topics in Contemporary Mathematical Modeling 2017, 167-192 (Ed. A.Z. Grzybowski)
The Publishing Office of Czestochowa University of Technology
DOI: 10.17512/STiCMM2017.11 ISBN 978-83-7193-668-5

CATEGORICAL SEMANTICS OF PROGRAMMING LANGUAGES

William Steingartner, Valerie Novitzká

Faculty of Electrical Engineering and Informatics, Technical University of Košice
Košice, Slovakia
william.steingartner@tuke.sk, valerie.novitzka@tuke.sk

Abstract. Computer science uses still more formal models to aid the understanding of complex software systems and to reason about their behavior, in particular to verify the correctness of a system or at least some desired aspects of its behavior with respect to a formal specification. All these techniques are grounded in formal models of system execution which are themselves rooted in the formal semantics of the underlying programming languages. Denotational semantics expresses the meaning of programs by functions from syntactical domains to semantic domains which can be non-trivial mathematical structures. We present in this document representation of denotational semanitcs based on category theory. We consider memory states as objects of category and semantic functions as morphisms. Category is then a model and semantics of a program written in our language Jane is modeled in this category as a path, i.e. a composition of morphisms.

Keywords: *categorical model, colimit, denotational semantics, imperative programming language, memory state*

1. Introduction

Formal description of programming languages belongs to the important methods that serve for assigning exact meanings to programs. They also help during the implementation of compilers. Formal semantics of programming languages provides the interpretation of programs written in a given programming language. Its main role is to predict the outcome of program execution.

Formal semantics can be viewed as a mapping from syntactic domains to the semantics domains. There are several semantic methods that are used for various purposes. Denotational semantics defines a meaning of a program in terms of domains and functions. It was formulated by Dana Scott and Christopher Strachey [1] and later by David Schmidt [2]. Operational semantics defines not only a meaning of a program but also a detailed process of its execution. It was formulated by Gordon Plotkin [3, 4] as a transition system that can be modeled by coalgebras [5, 6]. Among other known semantic methods we mention already action semantics [7], axiomatic semantic [8] and game semantics [9].

In this chapter we devote to denotational semantics. We introduce a simple procedural (imperative) programming language *Jane* and we define its denotational semantics. The second part of this chapter contains the construction of categorical denotational semantics of our language.

2. The *Jane* programming language

In this chapter we use a simple programming language called *Jane.* This language consists of traditional syntactic constructions used in imperative/procedural languages including arithmetic and Boolean expressions, statements and variable declarations. Later, we extend this definition with other constructs.

A formal definition of a programming language consists of its abstract syntax and semantics. Abstract syntax defines a structure of the language constructions and it consists of syntactic domains and production rules. For each syntactic domain, there is one production rule.

In our *Jane* language, we recognize the following syntactic domains:

$$n \in \mathbf{Num},$$
$$x \in \mathbf{Var},$$
$$e \in \mathbf{Aexpr},$$
$$b \in \mathbf{Bexpr},$$
$$S \in \mathbf{Statm}.$$

For simplicity, we use the sets as syntactic domains. The syntactic domain **Num** contains finite strings of digits and the syntactic domain **Var** is a countable set of variables names. Both these domains have no internal structure from the semantic point of view, which is significant for defining the semantics of the *Jane* language. The syntactic domain **Aexpr** contains well-structured arithmetic expressions defined by the following production rule:

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

This production rule expresses that an arithmetic expression can be a finite string of digits, a variable, a sum, subtraction or product of arithmetic expressions, or arithmetic expression enclosed in parenthesis, respectively. The syntactic domain **Bexpr** contains Boolean expressions with the structure defined by the following production rule:

$$b ::= true \mid false \mid e = e \mid e \leq e \mid \neg b \mid b \wedge b \mid (b).$$

Boolean expressions can be the truth constants $true$ and $false$, the equality of two arithmetic expressions, the relation less or equal, negation of a Boolean expression, conjunction between two Boolean expressions or Boolean expression enclosed in parentheses. The next syntactic domain **Statm** contains statements. We use only the known Dijkstra's statements as an assignment, empty statement, sequence of statements, conditional statement and pre-test loop statement (while) as it defines the following production rule:

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

3. Denotational semantics for the *Jane* programming language

To define the semantics of the *Jane* language, we need the semantic domains, i.e. the sets containing the meaning of the given syntactic domains. We assume that our expressions are implicitly typed, i.e. arithmetic expressions are evaluated to integer numbers from the set **Z** and Boolean expression are evaluated to truth values from the set

$$\mathbf{B} = \{\mathbf{tt}, \mathbf{ff}\},$$

where **tt** expresses the true value and **ff** the false value. The expressions can contain variables. During the execution of a program, the values of variables can be changed, therefore we need some semantic domain reflecting actual values of variables. We define the semantic domain of states

$$s \in \mathbf{State} = \mathbf{Var} \to \mathbf{Z},$$

as a set of functions

$$s\colon \mathbf{Var} \to \mathbf{Z}$$

assigning its actual value to a variable. The semantic domain **State** can be considered as a snapshot of a computer's memory. Arithmetic and Boolean expressions use states to achieve the actual values of variables, but they do not change the states. The execution of a statement can change the state. The semantics of the whole program defines how the initial state before execution of a program changes to a final state after ending a given program.

Now, we can define the semantic functions for arithmetic and Boolean expressions. The semantic function for arithmetic expressions is

$$\mathcal{E}\colon \mathbf{Aexpr} \to \mathbf{State} \to \mathbf{Z}.$$

This function has two parameters: an arithmetic expression and a state in which a given expression is evaluated. This semantic function expresses that a semantic of an arithmetic expression is its value in an actual state. The function E is defined for each structure from the production rule by semantic equations:

$$\begin{aligned}
&\mathcal{E}[\![n]\!]s = \mathcal{N}[\![n]\!],\\
&\mathcal{E}[\![x]\!]s = s\ x,\\
&\mathcal{E}[\![e_1 + e_2]\!]s = \mathcal{E}[\![e_1]\!]s + \mathcal{E}[\![e_2]\!]s,\\
&\mathcal{E}[\![e_1 - e_2]\!]s = \mathcal{E}[\![e_1]\!]s - \mathcal{E}[\![e_2]\!]s,\\
&\mathcal{E}[\![e_1 * e_2]\!]s = \mathcal{E}[\![e_1]\!]s * \mathcal{E}[\![e_2]\!]s,\\
&\mathcal{E}[\![(e)]\!]s = (\mathcal{E}[\![e]\!]s).
\end{aligned}$$

The function $\mathcal{E}$ is applied on two arguments: an expression $e \in \mathbf{Aexpr}$ and a state $s \in \mathbf{State}$. An expression is a syntactic concept and we enclose it between semantic brackets $[\![\ ,\ ]\!]$.

For Boolean expressions, we define the function

$$\mathcal{B}: \mathbf{Bexpr} \to \mathbf{State} \to \mathbf{B}$$

by the semantic equations:

$$\mathcal{B}[\![false]\!]s = \mathbf{ff}$$
$$\mathcal{B}[\![true]\!]s = \mathbf{tt}$$
$$\mathcal{B}[\![e_1 = e_2]\!]s = \begin{cases} \mathbf{tt}, & \text{if } \mathcal{E}[\![e_1]\!]s = \mathcal{E}[\![e_2]\!]s, \\ \mathbf{ff}, & \text{otherwise,} \end{cases}$$
$$\mathcal{B}[\![e_1 \leq e_2]\!]s = \begin{cases} \mathbf{tt}, & \text{if } \mathcal{E}[\![e_1]\!]s \leq \mathcal{E}[\![e_2]\!]s, \\ \mathbf{ff}, & \text{otherwise,} \end{cases}$$
$$\mathcal{B}[\![\neg b]\!]s = \begin{cases} \mathbf{tt}, & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{ff}, \\ \mathbf{ff}, & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt}, \end{cases}$$
$$\mathcal{B}[\![b_1 \wedge b_2]\!]s = \begin{cases} \mathbf{tt}, & \text{if } \mathcal{B}[\![b_1]\!]s = \mathbf{tt} \text{ and } \mathcal{B}[\![b_2]\!]s = \mathbf{tt}, \\ \mathbf{ff}, & \text{otherwise,} \end{cases}$$
$$\mathcal{B}[\![(b)]\!]s = (\mathcal{B}[\![(b)]\!]s).$$

Before introducing the semantics of statements, we define a new notation. When a state s is changed only for a variable x, and a value $\boldsymbol{a} \in \mathbf{Z}$ is assigned to x, a new state s' is created. We use the following notation for this actualization of a state:

$$s' = s[x \mapsto \boldsymbol{a}].$$

Now, we can introduce the semantics of the statements. We define the semantic function

$$\mathcal{S}_{ds}: \mathbf{Statm} \to (\mathbf{State} \rightharpoonup \mathbf{State}),$$

that assigns to the statement $S \in \mathbf{Statm}$ a partially defined function $\mathbf{State} \rightharpoonup \mathbf{State}$ assigning for an initial state s before executing S a final state s' created after execution of a given statement. This semantic function is defined for the first four statements in *Jane* by the following semantic equations:

$$(1_{ds}) \qquad \mathcal{S}_{ds}[\![x := e]\!]s = s[x \mapsto \mathcal{E}[\![e]\!]s]$$

$$(2_{ds}) \quad \mathcal{S}_{ds}[\![\text{skip}]\!]s = id_{\boldsymbol{State}}$$

where $id_{\boldsymbol{State}}: \mathbf{State} \to \mathbf{State}$ for all $s \in \mathbf{State}$ $id_{\boldsymbol{State}}(\text{s}) = \text{s}$

$$(3_{ds}) \qquad \mathcal{S}_{ds}[\![S_1; S_2]\!] = \mathcal{S}_{ds}[\![S_2]\!] \circ \mathcal{S}_{ds}[\![S_1]\!]$$

$$\mathcal{S}_{ds}[\![S_1; S_2]\!]s = (\mathcal{S}[\![S_2]\!] \circ \mathcal{S}[\![S_1]\!])s =$$
$$= \begin{cases} s', & \text{if there exists } s' \text{ such that } \mathcal{S}_{ds}[\![S_1]\!]s = s' \text{and } \mathcal{S}_{ds}[\![S_2]\!]s' = s''; \\ \perp, & \text{if } \mathcal{S}_{ds}[\![S_1]\!]s = \perp \text{ or if there exists } s' \text{such that } \mathcal{S}_{ds}[\![S_1]\!]s = s' \text{but } \mathcal{S}_{ds}[\![S_2]\!]s' = \perp. \end{cases}$$

The semantics of a conditional statement is defined with the auxiliary function $cond$, which has the following functionality:

$$cond\colon (\mathbf{State} \to \mathbf{B}) \times (\mathbf{State} \rightharpoonup \mathbf{State}) \times (\mathbf{State} \rightharpoonup \mathbf{State}) \rightharpoonup (\mathbf{State} \rightharpoonup \mathbf{State}),$$

and it is defined for $\varphi\colon \mathbf{State} \to \mathbf{B}$ and $f_1, f_2\colon \mathbf{State} \rightharpoonup \mathbf{State}$ as follows:

$$cond(\varphi, f_1, f_2)s = \begin{cases} f_1 s, & \text{if } \varphi(s) = \mathbf{tt}; \\ f_2 s, & \text{if } \varphi(s) = \mathbf{ff}. \end{cases}$$

The first parameter to $cond$ is a function that, when supplied with an argument, will select either the second or the third parameter of $cond$ and then supply that parameter with the same argument. Thus for conditional statement, the clause is

$$(4_{ds}) \quad \mathcal{S}_{ds}[\![\text{if } b \text{ then } S_1 \text{ else } S_2]\!]s = cond(\mathcal{B}[\![b]\!], \mathcal{S}_{ds}[\![S_1]\!], \mathcal{S}_{ds}[\![S_2]\!])s =$$

$$= \begin{cases} s', & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt} \text{ and } \mathcal{S}_{ds}[\![S_1]\!]s = s' \text{or if } \mathcal{B}[\![b]\!]s = \mathbf{ff} \text{ and } \mathcal{S}_{ds}[\![S_2]\!]s = s'; \\ \bot, & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt} \text{ and } \mathcal{S}_{ds}[\![S_1]\!]s = \bot \text{ or if } \mathcal{B}[\![b]\!]s = \mathbf{ff} \text{ and } \mathcal{S}_{ds}[\![S_2]\!]s = \bot. \end{cases}$$

The definition of semantics for a loop statement is not so simple. Consider a loop statement

$$\text{while } b \text{ do } S.$$

It is semantically equivalent to the following conditional statement

$$\text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip.}$$

The proof of this semantic equivalence can be found in [10]. If we use the semantic equation for conditional statement, we get

$$\mathcal{S}_{ds}[\![\text{while } b \text{ do } S]\!] = cond(\mathcal{B}[\![b]\!], \mathcal{S}_{ds}[\![\text{while } b \text{ do } S]\!] \circ \mathcal{S}_{ds}[\![S]\!], id)$$

The problem is that denotational semantics needs to be composable. That means, the semantics of complex structures (statements) can be defined by the semantics of their components (substatements). This definition is not composable because it is recursive. Denotational semantics of a loop statement is defined by the fixed point, a concept known from mathematics [11]. The equation above expresses that $\mathcal{S}_{ds}[\![\text{while } b \text{ do } S]\!]$ must be a fixed point of the functional F

$$F\colon (\mathbf{State} \rightharpoonup \mathbf{State}) \to (\mathbf{State} \rightharpoonup \mathbf{State})$$

defined by

$$F\ (\mathcal{S}_{ds}[\![\text{while } b \text{ do } S]\!]) = cond(\mathcal{B}[\![b]\!], \mathcal{S}_{ds}[\![\text{while } b \text{ do } S]\!] \circ \mathcal{S}_{ds}[\![S]\!], id).$$

That means

$$\mathcal{S}_{ds}[\![\text{while } b \text{ do } S]\!] = F\ (\mathcal{S}_{ds}[\![\text{while } b \text{ do } S]\!] \circ \mathcal{S}_{ds}[\![S]\!]).$$

We define the semantics of loop statement by

$$\mathcal{S}_{ds}[\![\text{while } b \text{ do } S]\!] = fix\ F$$

where

$$fix\colon \big((\mathbf{State} \rightharpoonup \mathbf{State}) \rightharpoonup (\mathbf{State} \rightharpoonup \mathbf{State})\big) \to (\mathbf{State} \rightharpoonup \mathbf{State}).$$

In this way we get a compositional definition of the semantic function $\mathcal{S}_{ds}$, because when we define the functional F, we only apply $\mathcal{S}_{ds}$ to the immediate constituents of the loop statement

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

and not to the construct itself. However, another problem arises. A recursive function can have no, one or more fixed points.

In the case of no fixed point, the semantics of a cycle statement does not exist, i.e. the cycle is infinite. In the case when more fixed points exist, the least fixed point of a functional is the denotational semantics of a cycle statement. The following two theorems define the requirements for existing of the least fixed point.

Theorem 1: A continuous function defined on a chain complete partially ordered sets (ccpos) has the least fixed point.

Theorem 2: Let $(\mathbf{State} \rightharpoonup \mathbf{State}, \sqsubseteq)$ be a partially ordered set with the least element $\perp$ that is an undefined function

$$\perp : \mathbf{State} \rightharpoonup \mathbf{State}.$$

This function assigns to each state $s \in \mathbf{State}$ the undefined state $\perp$

$$\perp (s) = \perp.$$

For a continuous function (functional)

$$F\colon (\mathbf{State} \rightharpoonup \mathbf{State}) \to (\mathbf{State} \rightharpoonup \mathbf{State})$$

the least fixed point is

$$fix\ F = \sqcup \{F^n\ \perp | n \geq 0\},$$

where $\sqcup$ is the least upper bound of the ccpos. The non-recursive functions F^i, for $i = 0, \ldots, n$ are defined by

$$F^0 = id,$$
$$F^{n+1} = F \circ F^n \text{ for } n \geq 0.$$

The detailed description of the least fixed-point together with proofs of the theorems are in [10].
We show in the following example how the least fixed point can be constructed.

Example 1: Assume the statement

$$S = y \coloneqq 1;\ \text{while}\ \neg(x = 1)\text{do}\ (y \coloneqq y * x; x \coloneqq x - 1)$$

and the initial state

$$s_0 = [x \mapsto \mathbf{3}].$$

It is clear that S computes the factorial of x and this variable has the initial value $\mathbf{3}$.
From (1_{ds}) and (5_{ds}) for the initial state s_0 holds:

$$\mathcal{S}_{ds}[\![y \coloneqq 1; \text{while}\ \neg(x = 1)\ \text{do}\ (y \coloneqq y * x; x \coloneqq x - 1)]\!]s_0 = (fix\ F)s_0[y \mapsto \mathbf{1}]$$

where

$$(Fg)s = \begin{cases} g(\mathcal{S}_{ds}[\![y \coloneqq y * x; x \coloneqq x - 1]\!]s), & \text{if}\ \mathcal{B}[\![\neg(x = 1)]\!]s = \mathbf{tt}, \\ s, & \text{if}\ \mathcal{B}[\![\neg(x = 1)]\!]s = \mathbf{ff}, \end{cases}$$

or, equivalently,

$$(Fg)s = \begin{cases} g(s[y \mapsto sx * sy][x \mapsto sx - \mathbf{1}]), & \text{if}\ sx \neq \mathbf{1}, \\ s, & \text{if}\ sx = \mathbf{1}. \end{cases}$$

We compute the functions F^n from the Theorem 2 as follows:

$$(F^0\ \bot)s = \bot,$$

$$(F^1\ \bot)s = \begin{cases} s, & \text{if}\ sx = \mathbf{1}, \\ \bot, & \text{if}\ sx \neq \mathbf{1}, \end{cases}$$

$$(F^2\ \bot)s = \begin{cases} s, & \text{if}\ sx = \mathbf{1}, \\ s[y \mapsto sy * \mathbf{2}][x \mapsto \mathbf{1}], & \text{if}\ sx = \mathbf{2}, \\ \bot, & \text{otherwise}, \end{cases}$$

$$(F^3\ \bot)s = \begin{cases} s, & \text{if}\ sx = \mathbf{1}, \\ s[y \mapsto sy * \mathbf{2}][x \mapsto \mathbf{1}], & \text{if}\ sx = \mathbf{2}, \\ s[y \mapsto sy * \mathbf{3} * \mathbf{2}][x \mapsto \mathbf{1}] & \text{if}\ sx = \mathbf{3}, \\ \bot, & \text{otherwise}, \end{cases}$$

Now, we can generalize:

$$(F^n\ \bot)s = \begin{cases} s[y \mapsto sy * k * \ldots * \mathbf{2} * \mathbf{1}][x \mapsto \mathbf{1}], & \text{if}\ sx = k \text{ and } 1 \leq k \text{ and } k \leq n, \\ \bot, & \text{if}\ sx < \mathbf{1} \text{ or } sx > n. \end{cases}$$

Then the least fixed point is:

$$(fix\ F)s = \begin{cases} s[y \mapsto sy * n * \ldots * \mathbf{2} * \mathbf{1}][x \mapsto \mathbf{1}], & \text{if } sx = n \text{ and } n \geq 1, \\ \bot, & \text{if } sx < \mathbf{1}. \end{cases}$$

For the initial state s_0, we receive that the value computed by the factorial statement S is

$$(fix\ F)(s_0[y \mapsto \mathbf{1}]) = s_0[\boldsymbol{y} \mapsto \mathbf{1} * \mathbf{3} * \mathbf{2} * \mathbf{1}][x \mapsto \mathbf{1}] = [y \mapsto \mathbf{6}, x \mapsto \mathbf{1}]$$

as we expected.

In the following example, we show how the least fixed point is computed for the simple loop statement.

Example 2: Assume the loop statement

$$\text{while } \neg(x{=}0) \text{ do skip}$$

and the initial state

$$s \in \mathbf{State}.$$

From (5_{ds}) holds

$$\mathcal{S}_{ds}[\![\text{while } \neg(x = 0) \text{ do skip}]\!]s = (fix\ F')s,$$

where the functional F' is defined as

$$(F'g)s = \begin{cases} s, & \text{if } sx = \mathbf{0}, \\ gs, & \text{if } sx \neq \mathbf{0}, \end{cases}$$

because from (2_{ds})

$$\mathcal{S}_{ds}[\![\text{skip}]\!]s = s.$$

That means that each partially defined function g: $\mathbf{State} \rightharpoonup \mathbf{State}$

$$gs = s$$

is a fixed point of a functional F'. Now, we need to find the least fixed point $fix\ F'$. We construct the following functions similarly as in Example 1:

$$(F'^0\ \bot)s = \bot$$

$$(F'^1\ \bot)s = \begin{cases} s, & \text{if } sx = \mathbf{0}, \\ \bot, & \text{if } sx \neq \mathbf{0}, \end{cases}$$

$$(F'^2\ \bot)s = \begin{cases} s, & \text{if } sx = \mathbf{0}, \\ \bot, & \text{if } sx \neq \mathbf{0}, \end{cases}$$

...

$$(F'^n\ \bot)s = \begin{cases} s, & \text{if } sx = \mathbf{0}, \\ \bot, & \text{if } sx \neq \mathbf{0}, \end{cases}$$

We can write that the least fixed point of F' is

$$g_0 s = \begin{cases} s, & \text{if } sx = \mathbf{0}, \\ \bot, & \text{if } sx \neq \mathbf{0}, \end{cases}$$

and the denotation of the given cycle statement is

$$\mathcal{S}_{ds}[\![\text{while } \neg(x = 0) \text{ do skip}]\!] = fix\ F' = g_0.$$

4. Basic concepts from category theory

We define in the following sections categorical denotational semantics. Therefore, in this section, we introduce the basic notions from the category theory. The category theory is defined in detail in many books, we recommend only a few, e.g. [12-14]. The basic notion in the category theory is a new mathematical structure, category.

Definition 1: *A category* $\mathbf{C} = (\mathbf{C_{obj}}, \mathbf{C_{morp}})$ is a mathematical structure consisting of

- a set $\mathbf{C_{obj}}$ of objects; and
- a set $\mathbf{C_{morp}}$ of morphisms between them;

together with two functions $\boldsymbol{dom}$ and $\boldsymbol{cod}$ that are depicted in Figure 1.

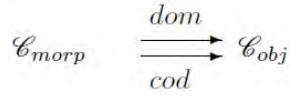

Fig. 1. Functions *dom* and *cod*

In any category, the following conditions must be satisfied:

1. If $\boldsymbol{f}: \boldsymbol{A} \to \boldsymbol{B}$ is a morphism in a category $\mathbf{C}$, then

$$\boldsymbol{dom}(\boldsymbol{f}) = \boldsymbol{A} \textbf{ and } \boldsymbol{cod}(\boldsymbol{f}) = \boldsymbol{B},$$

   that means, $\boldsymbol{dom}(\boldsymbol{f})$ is a domain of $\boldsymbol{f}$ and $\boldsymbol{cod}(\boldsymbol{f})$ is a codomain of $\boldsymbol{f}$.
2. For each object $\boldsymbol{A}$ in $\mathbf{C}$ there exists an identity morphism $\boldsymbol{id_A}: \boldsymbol{A} \to \boldsymbol{A}$.
3. If $\boldsymbol{f}: \boldsymbol{A} \to \boldsymbol{B}$ and $\boldsymbol{g}: \boldsymbol{B} \to \boldsymbol{C}$ are morphisms in $\mathbf{C}$, then there exists a morphism

$$\boldsymbol{g} \circ \boldsymbol{f}: \boldsymbol{A} \to \boldsymbol{C}$$

that is a composition of the morphisms $\boldsymbol{f}$and $\boldsymbol{g}$.

4. A composition of the morphisms is associative, i.e. for the following morphisms $\boldsymbol{f}: \boldsymbol{A} \to \boldsymbol{B}, \boldsymbol{g}: \boldsymbol{B} \to \boldsymbol{C}$ and $\boldsymbol{h}: \boldsymbol{C} \to \boldsymbol{D}$ holds:

$$\boldsymbol{h} \circ (\boldsymbol{g} \circ \boldsymbol{f}) = (\boldsymbol{h} \circ \boldsymbol{g}) \circ \boldsymbol{f}.$$

The special objects in categories are terminal and initial objects. A category can have no, one or more terminal and/or initial objects.

Definition 2: An object $\mathbf{0}$ in a category $\mathbf{C}$ is *initial*, if for any object $\boldsymbol{A}$ in $\mathbf{C}$ there exists just one morphism $\mathbf{0} \to \boldsymbol{A}$.
An object $\mathbf{1}$ in a category $\mathbf{C}$ is *terminal*, if for any object $\boldsymbol{A}$ in $\mathbf{C}$ there exists just one morphism $\boldsymbol{A} \to \mathbf{1}$.

In our approach to categorical denotational semantics we use another important concept from the category theory, a functor.

Definition 3: Let $\mathbf{C}$, $\mathbf{D}$ be categories. A *functor* $\boldsymbol{F}: \mathbf{C} \to \mathbf{D}$ is a pair of functions:

$$\boldsymbol{F_0}: \mathbf{C_{obj}} \to \mathbf{D_{obj}}$$
$$\boldsymbol{F_1}: \mathbf{C_{morp}} \to \mathbf{D_{morp}}$$

satisfying the following conditions:

- if $\boldsymbol{f}: \boldsymbol{A} \to \boldsymbol{B}$ is a morphism in $\mathbf{C}$, then $\boldsymbol{F_1}(\boldsymbol{f}): \boldsymbol{F_0}(\boldsymbol{A}) \to \boldsymbol{F_0}(\boldsymbol{B})$ is a morphism in $\mathbf{D}$
- for any object $\boldsymbol{A}$ in $\mathbf{C}$ it holds $\boldsymbol{F_1}(\boldsymbol{id_A}) = \boldsymbol{id_{F_0(A)}}$
- if a composition $\boldsymbol{g} \circ \boldsymbol{f}$ is a morphism in $\mathbf{C}$, then the composition $\boldsymbol{F_1}(\boldsymbol{g}) \circ \boldsymbol{F_1}(\boldsymbol{f})$ is defined in $\mathbf{D}$ and it holds:

$$\boldsymbol{F_1}(\boldsymbol{g} \circ \boldsymbol{f}) = \boldsymbol{F_1}(\boldsymbol{g}) \circ \boldsymbol{F_1}(\boldsymbol{f}).$$

An important functor is an endofunctor $\boldsymbol{F}: \mathbf{C} \to \mathbf{C}$ and its special case is an identity functor

$$\boldsymbol{Id}: \mathbf{C} \to \mathbf{C}$$

defined for any object $\boldsymbol{A}$ and any morphism $\boldsymbol{f}$ in $\mathbf{C}$ by

$$\boldsymbol{Id}(\boldsymbol{A}) = \boldsymbol{A} \text{ and } \boldsymbol{Id}(\boldsymbol{f}) = \boldsymbol{f}.$$

Now, we have defined all notions needed for describing categorical denotational semantics of the language *Jane.* That is the content of the following sections.

5. Extended language *Jane*

In the previous section we introduced the traditional denotational semantics of the *Jane* language. In the categorical approach, which we present in the following text, we extend this basic language with new, frequently used constructions as declarations, blocks and input statements. The syntax of arithmetic and Boolean expressions remains without changes and their semantics is defined as in Section 3. In the categorical approach, this semantics plays only an auxiliary role, therefore we will concentrate only onto statements and declarations.

All variables occurring in a program written in *Jane* need to be declared. The declarations form a sequence with the following syntax:

$$\boldsymbol{D} ::= \mathbf{var}\ \boldsymbol{x}; \boldsymbol{D} \mid \varepsilon,$$

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

We consider five Dijkstra's statements as above and we extend our language with block statement and input statement:

$$\boldsymbol{S} ::= \cdots \mid \mathbf{begin}\ \boldsymbol{D}; \boldsymbol{S}\ \mathbf{end} \mid \mathbf{input}\ \boldsymbol{x}.$$

From this production rule, it is clear that a block can have declarations of local variables that are invisible in outer blocks. The input statement gets a value and assigns it to a variable that needs to be previously declared.

In the following section we construct a categorical model of the *Jane* language. We start with the most important notion for the semantics - the state.

6. Signature and representation of states

According to the previous ideas about the concept of state, we formulate the signature $\boldsymbol{\Sigma_{State}}$ for the abstract data type $\boldsymbol{State}$. A signature is a well-known notion used in algebraic specification of abstract data types [15]. This signature specifies the type $\boldsymbol{State}$ and uses the trivial types $\boldsymbol{Var}$ for variable names and $\boldsymbol{Value}$ for integer values.

The signature $\boldsymbol{\Sigma_{State}}$ consists of types and operation specifications:

$$\begin{aligned}\boldsymbol{\Sigma_{State}} = \\ &\boldsymbol{types}: \boldsymbol{State}, \boldsymbol{Var}, \boldsymbol{Value} \\ &\boldsymbol{opns}: \boldsymbol{init}: \rightarrow \boldsymbol{State} \\ &\qquad \boldsymbol{alloc}: \boldsymbol{Var}, \boldsymbol{State} \rightarrow \boldsymbol{State} \\ &\qquad \boldsymbol{get}: \boldsymbol{Var}, \boldsymbol{State} \rightarrow \boldsymbol{Value} \\ &\qquad \boldsymbol{del}: \boldsymbol{State} \rightarrow \boldsymbol{State}.\end{aligned}$$

The operation specifications have the following intuitive meaning:

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

Now, we assign the representation to the signature of states. To the type $\boldsymbol{Value}$ we assign the set of integer numbers together with the undefined value $\perp$:

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

To the type $\boldsymbol{Var}$ we assign a countable set $\mathbf{Var}$ of variable names. Our representation of an element of the type $\boldsymbol{State}$ has to express a variable name together with its value with respect to an actual nesting level. Let $\mathbf{Level}$ be a finite set of nesting levels denoted by natural numbers $\boldsymbol{l}$:

$$\boldsymbol{l} \in \mathbf{Level}, \qquad \mathbf{Level} = \mathbf{N}.$$

The declaration (nesting) level allows us to create a variable environment that enables us to distinguish local declarations from global ones.

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

$$\boldsymbol{s}\colon \mathbf{Var} \times \mathbf{Level} \rightharpoonup \mathbf{Value}.$$

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

$$\boldsymbol{s} = \langle\big((\boldsymbol{x_1}, \mathbf{1}), \boldsymbol{v_1}\big), \ldots, \big((\boldsymbol{x_n}, \boldsymbol{l}), \boldsymbol{v_{n1}}\big)\rangle$$

of ordered triples

$$\big((\boldsymbol{x_i}, \boldsymbol{l_j}), \boldsymbol{v_i}\big),$$

where $(\boldsymbol{x_i}, \boldsymbol{l_j})$ is the declared variable $\boldsymbol{x_i}$ on the nesting level $\boldsymbol{l_j}$ with actual (possibly undefined) value $\boldsymbol{v_i}$, for $\boldsymbol{i} = \mathbf{1}, \ldots, \boldsymbol{l}$. This sequence can also be infinite, e.g. in the case of an infinite loop. This sequence can be illustrated equivalently by a table with possibly unfilled cells denoted by the symbol $\perp$ expressing an undefined value which increases readability (Fig. 2).

variable	level	value
x	1	v_1
⋮		
z	l	v_n

Fig. 2. Representation of a state by table

In the following text, we use the sequence representation of states in definitions and we illustrate states by tables in examples. We follow by the representation of operation specifications from the signature $\boldsymbol{\Sigma_{State}}$. The operation $[\![\boldsymbol{init}]\!]$ defined by

$$[\![\boldsymbol{init}]\!] = \boldsymbol{s_0} = \langle((\bot, \boldsymbol{1}), \bot)\rangle$$

creates the initial state $\boldsymbol{s_0}$ of a program with no declared variable. Its role is to set the nesting level to a value $\boldsymbol{1}$ at the beginning of program execution (Fig. 3).

variable	level	value
$\bot$	1	$\bot$

Fig. 3. Initial state of a program

The operation $[\![\boldsymbol{alloc}]\!]$ appends a new item to the sequence, i.e. it creates a new entry in the table of $\boldsymbol{s}$ and is defined by

$$[\![\boldsymbol{alloc}]\!](\boldsymbol{x}, \boldsymbol{s}) = \boldsymbol{s} \lozenge \langle((\boldsymbol{x}, \boldsymbol{l}), \bot)\rangle,$$

where the symbol $\lozenge$ denotes the concatenation operation on sequences. This operation sets the actual nesting level to the declared variable. Because of the undefined value of the declared variable, the operation $[\![\boldsymbol{alloc}]\!]$ does not change the state (Fig. 4).

variable	level	value
⋮	⋮	⋮
x	l	$\bot$

Fig. 4. Allocation of the new state

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

$$[\![\boldsymbol{get}]\!]\big(\boldsymbol{x}, \langle \dots, \big((\boldsymbol{x}, \boldsymbol{l_i}), \boldsymbol{v_i}\big), \dots, \big((\boldsymbol{x}, \boldsymbol{l_k}), \boldsymbol{v_k}\big), \dots\rangle\big) = \boldsymbol{v_k},$$

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

The operation $[\![\boldsymbol{del}]\!]$ deallocates (forgets) all the variables declared on the highest nesting level $\boldsymbol{l_j}$ (Fig. 5) and is defined by

$$[\![\boldsymbol{del}]\!]\left(\boldsymbol{s} \Diamond \langle\left(\left(\boldsymbol{x_i}, \boldsymbol{l_j}\right), \boldsymbol{v_k}\right), \ldots, \left(\left(\boldsymbol{x_n}, \boldsymbol{l_j}\right), \boldsymbol{v_m}\right)\rangle\right) = \boldsymbol{s}.$$

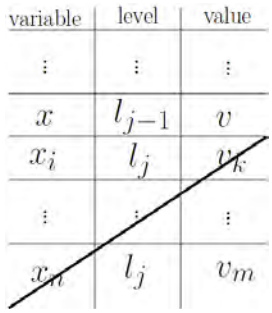

Fig. 5. Deallocation of all variables declared on the level $\boldsymbol{l_j}$

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

$$\boldsymbol{s}_{\perp} = \langle((\perp, \perp), \perp)\rangle$$

expressing the undefined state.

7. Model of *Jane* as a category of states

When we have defined the representation of state, we can construct a model of the *Jane* language as a category of states, $\mathbf{C}_{\boldsymbol{State}}$. In this category we consider:

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

Functions on states represent elaborations of declarations and executions of statements. In the next text we define the semantics of declarations, then the semantics of statements and we verify that so constructed category $\mathbf{C}_{\boldsymbol{State}}$ is a category, i.e. it satisfies the conditions in Definition 1.

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, an elaboration of a declaration of the form

$$\mathbf{var}\ \boldsymbol{x}$$

is represented as a function on a state $\boldsymbol{s}$:

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

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

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

Each declaration of a variable actualizes an environment of variables. Graphically, a variable declaration is depicted in the table by creating a new entry for the declared variable with its actual nesting level and undefined value:

$$\big((x, l), \perp\big).$$

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 states by functions between states.

Let in the following S be a statement. Its semantics is a function:

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

where s and s' are states. This function can be partially defined in the case the resulting state s' does not exist, i.e. it is undefined:

$$s' = s_{\perp}.$$

We extend the states with an undefined state to be the total semantic function. Dealing with partially defined functions in a category is not trivial. Statements are executed in the order as they are written in a program text. In this chapter we do not consider the statements causing a break of sequential execution, e.g. **goto** statements nor exceptions.

Assignment statement $x := e$ stores a value of arithmetic expression e in a state s in a memory cell allocated for the 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 defined as:

$$[\![x := e]\!]s = \begin{cases} s\big[\big((x, l), v\big) \mapsto \big((x, l), [\![e]\!]s\big)\big], & \mathbf{for}\ \big((x, l), v\big) \in s \\ s_{\perp}, & \mathbf{otherwise} \end{cases}$$

and it is illustrated in Figure 6.

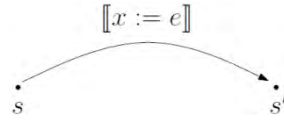

Fig. 6. Execution for assignment statement

The notation

$$\boldsymbol{s}\left[\left((\boldsymbol{x},\boldsymbol{l}),\boldsymbol{v}\right)\mapsto\left((\boldsymbol{x},\boldsymbol{l}),[\![\boldsymbol{e}]\!]\boldsymbol{s}\right)\right]$$

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

The empty statement **skip** does not do anything, i.e. it does not change the state. Clearly, its semantics is an identity function on a state $\boldsymbol{s}$ (Fig. 7) and it is defined as:

$$[\![\mathbf{skip}]\!]\boldsymbol{s}=\boldsymbol{s}.$$

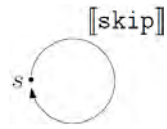

Fig. 7. Execution of empty statement

Typical usage of the empty statement **skip** can be when we rewrite the statement **if** $\boldsymbol{b}$ **then** $\boldsymbol{S}$ by the semantically equivalent statement

$$\textbf{if } \boldsymbol{b} \textbf{ then } \boldsymbol{S} \textbf{ else skip}.$$

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

A sequence of statements is executed one by one and can be modeled as a composition of functions (Fig. 8):

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

defined for a state $\boldsymbol{s}$ by

$$[\![\boldsymbol{S_1};\boldsymbol{S_2}]\!]\boldsymbol{s}=[\![\boldsymbol{S_2}]\!]([\![\boldsymbol{S_1}]\!]\mathbf{s}).$$

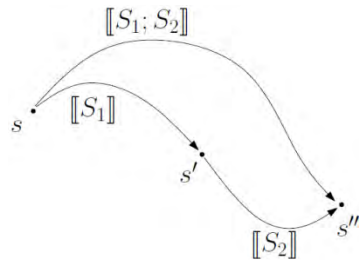

Fig. 8. Composition of functions for statement sequence

If a state $\boldsymbol{s'}$ is undefined, then the execution of the whole sequence of statements provides the undefined state. It follows from this definition that achieving undefined state $\boldsymbol{s_\perp}$ is similar as falling into "black hole", meaning that execution of a program is immediately stopped without resulting state, i.e. without meaning.
A conditional statement causes branching of execution that depends on the value of a Boolean expression $\boldsymbol{b}$. The semantics of conditional statement is defined by:

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

The function defined above returns the final state $\boldsymbol{s_i} = [\![\boldsymbol{S_i}]\!]\boldsymbol{s}$, where $\boldsymbol{i} = \boldsymbol{1}$ for $[\![\boldsymbol{b}]\!]\boldsymbol{s} = \textbf{true}$, and $\boldsymbol{i} = \boldsymbol{2}$ for $[\![\boldsymbol{b}]\!]\boldsymbol{s} = \textbf{false}$, meaning that execution of a conditional statement is modeled by one morphism depending on the value of Boolean expression $\boldsymbol{b}$ as it is illustrated in Figure 9.

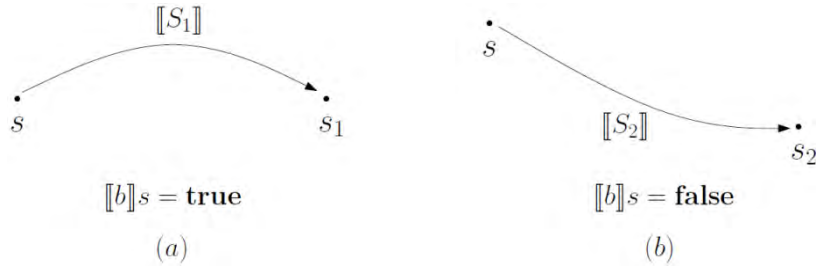

Fig. 9. Execution of conditional statement

Execution of the loop statement $\textbf{while } \boldsymbol{b} \textbf{ do } \boldsymbol{S}$ also depends on the value of Boolean expression $\boldsymbol{b}$. If $\boldsymbol{b}$ evaluates to $\textbf{true}$ in the actual state, the body $\boldsymbol{S}$ of a loop is executed, then again $\boldsymbol{b}$ is evaluated in a possibly modified state. If $\boldsymbol{b}$ evaluates to $\textbf{false}$, execution of loop statement is finished. Loop statement is semantically equivalent to the following conditional statement:

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

Semantic equivalence of two statements means that their execution in the same state provides the same final state. The proof of this semantic equivalence of the conditional statement and the loop statement can be found in [10]. Therefore, the semantics of the loop statement is defined as a (possibly infinite) composition of functions.

When we model execution of the loop statement, two situations can appear. The first of them is in Figure 10, where the execution finishes in some state $\boldsymbol{s_n}$.

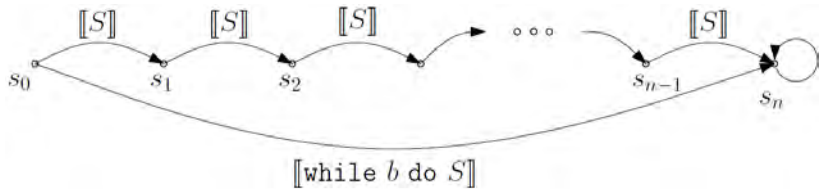

Fig. 10. Terminated execution of loop statement

However, if the condition $\boldsymbol{b}$ is always evaluated to **true**, the loop statement is executed as an infinite composition of functions. In this case, no resulting state is provided by the loop statement. In Figure 11, we illustrate execution of two possible kinds of infinite loops.

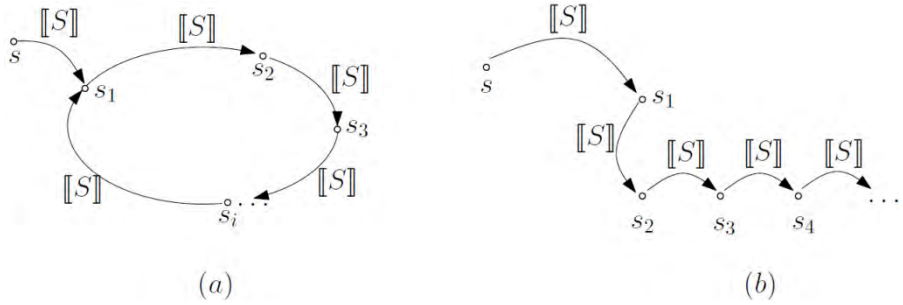

Fig. 11. Infinite loops

The trivial example of the execution in Figure 11, case (b) is the statement

$$\textbf{while true do } x := x + 1,$$

where each state in the sequence of states is different. Both these situations can be detected by observations thanks to graphical representation.

The traditional denotational semantics of a cycle statement is defined by using the fixed-point operator. This approach is obvious if the categorical model is a category of types. Existence of the least fixed point ensures that a loop statement finishes. In the other case the execution of a loop statement is infinite, i.e. the denotational semantics is not defined. This approach is discussed in detail also in [16]. In general, the computational categorical models have continuous lattices as objects and continuous functions as morphisms [1]. Such models require some structure on endomorphisms [17]. In our approach we use the categorical model with states as objects and functions as morphisms. The domains (states) are sets, not lattices; therefore we use another concept for handling infinite cycles. The execution of the while statement is a path of morphisms, i.e. a composition of morphisms. This path can be either finite or infinite, and we need some construction in our category to solve both cases. The useful construct is the colimit of a diagram.

Consider a diagram D consisting of the composition of morphisms:

$$D\colon\ s_0 \to s_1 \to \cdots s_i \to s_{i+1} \to \cdots$$

This composition of morphisms is a composition of semantic functions applied on a loop statement with actual states. This infinite composition is a morphism

$$s_0 \to s_\infty,$$

for which there are morphisms $f_i^\infty\colon s_i \to s_\infty$, for $i \geq 0$, such that the cocone in Figure 12 is a colimit of the diagram D.

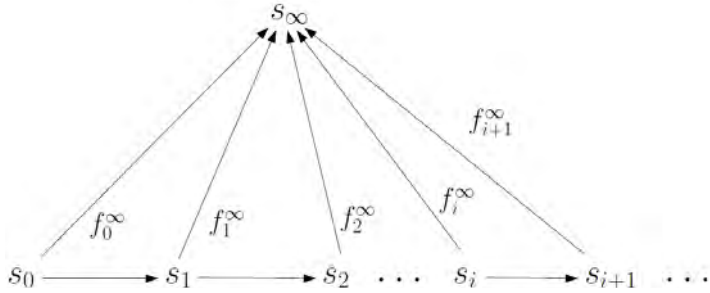

Fig. 12. Colimit of in infinite composition of states

By the requirement on our category of states to have colimits for diagrams, we ensure that the semantics of an infinite cycle is $\boldsymbol{s}_\infty = \boldsymbol{s}_\perp$. That means, in the category of states, we have defined a composition of infinite sequences as a morphism and we can also define the categorical semantics of cycle statement as follows:

$$[\![\textbf{while } \boldsymbol{b} \textbf{ do } \boldsymbol{S}]\!]\boldsymbol{s} = \begin{cases} \boldsymbol{s}', & \textbf{if } \boldsymbol{D} \textbf{ is finite}, \\ \boldsymbol{colim}(\boldsymbol{D}), & \textbf{if } \boldsymbol{D} \textbf{ is infinite}. \end{cases}$$

Because the objects in the category $\mathbf{C}_{\mathbf{State}}$ are sets, such colimits always exist [17].

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

$$[\![\textbf{input } \boldsymbol{x}]\!]\boldsymbol{s} = \begin{cases} \boldsymbol{s}[\big((\boldsymbol{x},\boldsymbol{l}),\boldsymbol{v}\big) \mapsto \big((\boldsymbol{x},\boldsymbol{l}),\boldsymbol{v}'\big), & \textbf{for } \big((\boldsymbol{x},\boldsymbol{l}),\boldsymbol{v}\big) \in \boldsymbol{s}; \\ \boldsymbol{s}_\perp & \textbf{otherwise}. \end{cases}$$

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

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

follows in several steps:

- nesting level l is incremented. This step is represented by a fictive entry in a given object

$$\big((\text{begin}, l+1), \perp\big);$$

- local declarations are elaborated on the new nesting level $l+1$;
- the body S of a block is executed;
- locally declared variables are forgotten at the end of the block. This situation is modeled by using the operation $[\![del]\!]$ defined above.

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

$$[\![\text{begin } D; S \text{ end}]\!]s = [\![del]\!] \circ [\![S]\!] \circ [\![D]\!]\big(s \lozenge \langle\big((\text{begin}, l+1), \perp\big)\rangle\big).$$

Now we can define the whole category $\mathbf{C}_{\text{State}}$ of states:

- category objects are states defined in Section 6 as sequences of tuples for declared variables together with the special state $s_\perp$;
- category morphisms are functions $[\![S]\!]: s \to s'$defined above for all statements of *Jane*.

The semantics of a program written in *Jane* is modeled in this category as a path, i.e. a composition of morphisms.

Example 1: Consider the following program *P*:

var x;
var y;
input x;
$y := 1$;
while $\neg(x = 1)$do $(y := y * x; x := x - 1)$

This program computes the factorial of its input value. In this example we consider that the input value is **3**.

Following the definitions of semantics for declarations and statements, we can follow the changes of states during execution of this program in Figure 13.

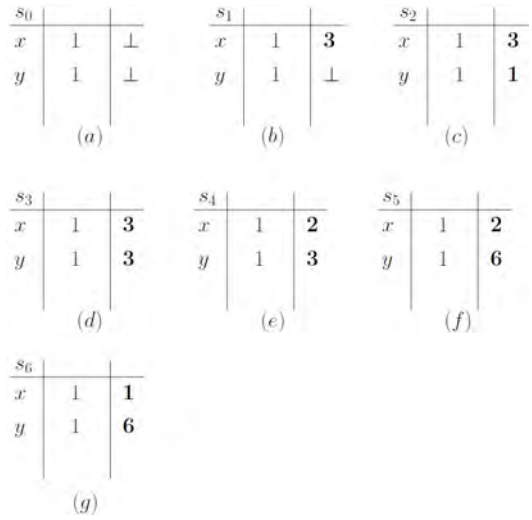

Fig. 13. States changes during execution of the given program

Figure 14 shows the path (composition of morphisms) by which the program P is executed step-by-step from the initial state s_0 to the final state s_6. In this figure, a loop statement is executed in five steps, this part of the program is illustrated from state s_2 to state s_6, i.e. the semantics of the cycle is a composition of all these steps.

The semantics of P is defined as follows:

$$([\![\text{skip}]\!] \circ [\![x := x - 1]\!] \circ [\![y := y * x]\!] \circ [\![x := x - 1]\!] \circ [\![y := y * x]\!] \circ [\![y := 1]\!] \circ [\![\text{input } x]\!] \circ [\![\text{var } y]\!] \circ [\![\text{var}]\!])$$

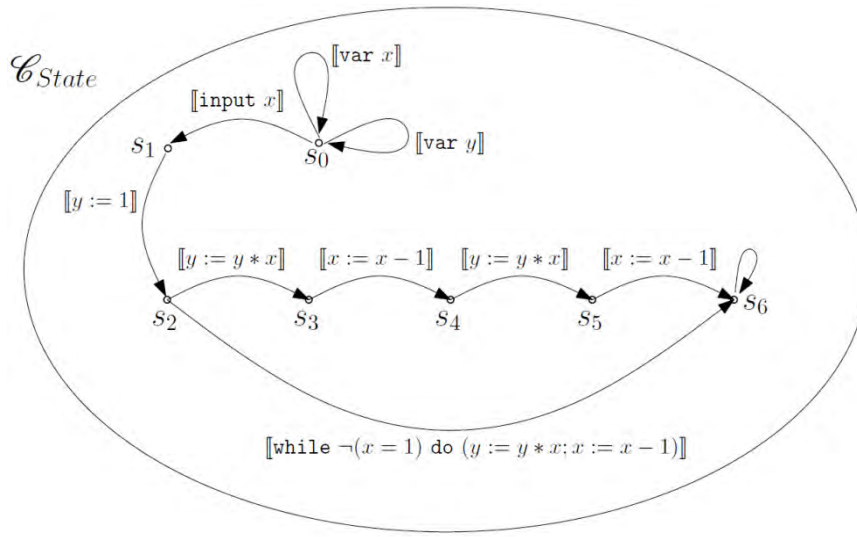

Fig. 14. The path of execution of the given program

8. Categorical semantics for procedures

In the previous sections, we defined the categorical model of the *Jane* programming language. This model is based on the category $\mathbf{C}_{\mathrm{State}}$ of states. In this section, we extend our approach for the language with procedure declarations and procedure calls.

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

Firstly, we formulate the syntax of the extended *Jane* language. We introduce a new syntactic domain $D_p \in \mathbf{ProcDecl}$ for the finite sequences of procedure declarations with the syntax:

$$D_p ::= \mathrm{proc}\, p(t); S\ \mathrm{return}; D_p \mid \varepsilon$$

where D_p is a sequence of procedure declarations and ε denotes the empty sequence. 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 additionally change the syntax of the block statement and to add the syntax for the procedure call with the argument that can be an arithmetical expression e:

$$S ::= \cdots \mid \mathrm{begin}\ D; D_p; S\ \mathrm{end} \mid \mathrm{call}\ p(e)$$

The semantics of a program is modeled as a collection of categories of states. The category $\mathbf{C}_{\mathrm{State}}$ constructed above serves for a main program.

A declaration of a procedure p causes the construction of the category $\mathbf{C}_{\mathrm{p}}$ similarly as the category $\mathbf{C}_{\mathrm{State}}$. 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 $\mathbf{C}_{\mathrm{p}}$ 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 the 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 us 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. We construct two functors:

$$C\colon \mathbf{Statm} \to \mathbf{C}_{\mathrm{State}} \to \mathbf{C}_{\mathrm{p}}$$
$$R\colon \mathbf{Statm} \to \mathbf{C}_{\mathrm{p}} \to \mathbf{C}_{\mathrm{State}}$$

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

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

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

- update the initial state s_0^p in $\mathbf{C}_{\mathrm{p}}$ by the state s in $\mathbf{C}_{\mathrm{State}}$;
- 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 $\mathbf{C}_{\mathrm{p}}$. Formally, the functor C works on objects as follows:

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

where the notation in square brackets denotes replacing of the original state s_0^p by a new sequence of entries from the calling program. This notation can be considered as an extension of state actualization introduced above.

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

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

where $s^* = s_\perp^p$.

Executing a procedure p can be modeled in the corresponding category $\mathbf{C}_{\mathrm{p}}$ 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 $\mathbf{C}_{\mathrm{State}}$;

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

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

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

The semantics of the statement call $p(e)$ is then defined by the commutative diagram in Figure 15 as a composition.

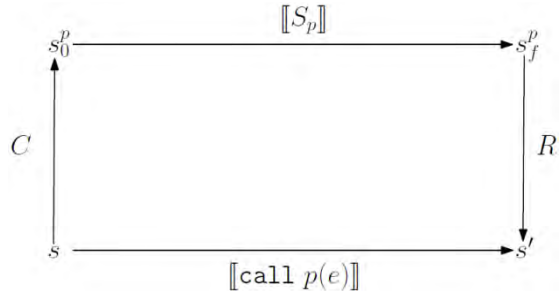

Fig. 15. The semantics of procedure calling

As states can be considered as sets of functions, the morphisms $s_\perp \to s'$ always exist [12]. 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 chapter, recursive procedures are not yet discussed.

Example 2: Consider a fragment of program P in the *Jane* language with procedures:

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

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 16 and illustrated in Figure 17.

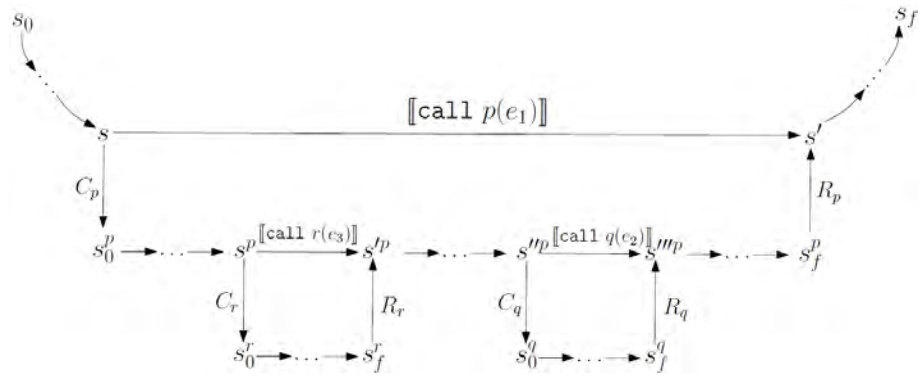

Fig. 16. Diagram for program *P*

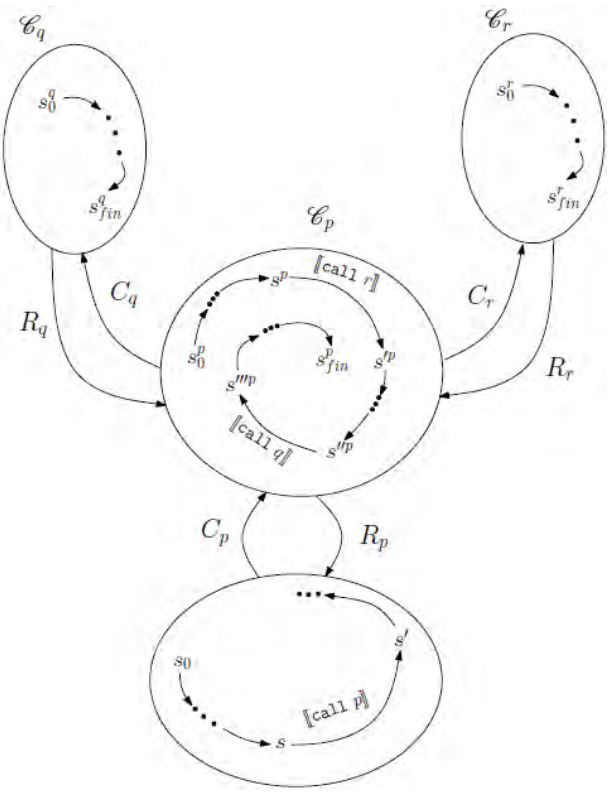

Fig. 17. The collection of categories for program *P*

Program P is executed as follows: declaration of procedure q in the main program causes the construction of category $\mathbf{C}_\mathrm{q}$, and declaration of procedure p causes the construction of category $\mathbf{C}_\mathrm{p}$.

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 $\mathbf{C}_\mathrm{r}$ is constructed. Calling of R in a state s^p initiates the state s_0^r by functor C_r and the body of S_r is executed to a final state s_f^r. Then control is passed back to the body of p by functor R_r and execution follows from a state s'^p.

Similarly, execution of the statement call $q(e_2)$ starts in a state s''^p and ends in a state s'''^p. Execution of p ends in a final state s_f^p and control is passed back to the main program by functor R_p to some state s'.

References

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

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

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

[4] Plotkin G., The origins of structural operational semantics, J. Log. Algebr. Program 2004, 60-61, 3-15.

[5] Steingartner W., Novitzká V., Coalgebras for modelling observable behaviour of programs, J. Appl. Mathem. Comput. Mech. 2017, 16, 2, 145-157.

[6] Jacobs B., Rutten J.: The Tutorial on (Co)algebras and (Co)induction, EATCS Bulletin 1997, 62, 222-259.

[7] Mosses P.D., Action Semantics, Cambridge University Press, 2005.

[8] Hoare C.A.R., Wirth N., An Axiomatic definition of the programming language Pascal, Acta Informat. 1973, 2, 4, 333-355.

[9] Abramsky S., McCusker G., Game Semantics, [In:] U. Berger, H. Schwichtenberg (eds), Computational Logic, NATO ASI Series, Vol. 165, Springer-Verlag, 1999.

[10] Nielson H.R., Nielson F., Semantics with Applications: An Appetizer, Springer-Verlag London Limited, 2007.

[11] Davey B.A., Priestley H.A., Introduction to Lattices and Order, Cambridge University Press, 2002.

[12] Barr M., Wells C., Category Theory for Computer Science, Prentice Hall, 2012.

[13] Adámek J., Herrlich H., Strecker G.E., Abstract and Concrete Categories, Heldermann Verlag, Berlin 2004.

[14] Turi D., Category Theory Lecture Notes, LFCS, University of Edinburgh, 2001.

[15] Ehrig H., Mahr B., Fundamentals of Algebraic Specification 1, 2, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin-Heidelberg, 1985, 1990.

[16] Schmidt D., Denotational Semantics. A methodology for language development, 1997.

[17] Escardó M.H., Recursion and Induction on the Real Line, [In:] Proceedings for the Second Imperial College Department of Computing Workshop on Theory and Formal Methods, Moller Centre, Cambridge, September 1994.