Computing and Informatics, Vol. 38, 2019, 1181–1209, doi: 10.31577/cai_2019_5_1181

COALGEBRAIC OPERATIONAL SEMANTICS FOR AN IMPERATIVE LANGUAGE

William Steingartner, Valerie Novitzka

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

Wolfgang Schreiner

Research Institute for Symbolic Computation
Johannes Kepler University
Altenberger Straße 69, A-4040 Linz, Austria
e-mail: wolfgang.schreiner@risc.jku.at

Abstract. Operational semantics is a known and popular semantic method for describing the execution of programs in detail. The traditional definition of this method defines each step of a program as a transition relation. We present a new approach on how to define operational semantics as a coalgebra over a category of configurations. Our approach enables us to deal with a program that is written in a small but real imperative language containing also the common program constructs as input and output statements, and declarations. A coalgebra enables to define operational semantics in a uniform way and it describes the behavior of the programs. The state space of our coalgebra consists of the configurations modeling the actual states; the morphisms in a base category of the coalgebra are the functions defining particular steps during the program's executions. Polynomial endofunctor determines this type of systems. Another advantage of our approach is its easy implementation and graphical representation, which we illustrate on a simple program.

Keywords: Category theory, coalgebra, operational semantics, programming language

Mathematics Subject Classification 2010: 18C50, 16T99, 97P40

1 INTRODUCTION

Formal semantics belongs inherently to the definition of programming languages. It provides the meaning of the programs in an unambiguous way. There are several well-known semantic methods for defining the formal semantics of programs written in some programming language. The choice of a suitable semantic method depends on the information we desire. In this paper, we are interested in operational semantics of a simple imperative language *E-Jane* defined as a coalgebra over a category of configurations.

Operational semantics is the most popular semantic method. It was defined by Plotkin in [24]; and he discusses its motivation in [25]. This method describes not only the meaning of a program but also the execution details. Operational semantics can be considered as a transition system, where program execution is described in particular steps using transition relations [12, 21, 32, 38]; thus the behavior of programs is defined. And this method is our subject of interest in this paper.

Another approach connected to small-step approach is the K framework. It is an executable rewriting-based semantic definitional framework in which programming languages, calculi, as well as type systems or formal analysis tools can be defined, making use of configurations, computations, and rules. It was introduced by Grigore Roşu in 2003 [26]. The framework consists of two components: general-purpose concurrent rewriting approach (K rewriting) and a definitional technique specialized for concurrent programming languages or systems (K technique) which yields a semantic definitional style [28]. Configurations organize the state in units called cells (K cell structures), which are labeled and can be nested. Computations carry computational meaning as special nested list structures sequentializing computational tasks, such as fragments of program. The rewriting rules of the K framework make it explicit which parts of the term they read-only, write-only, read-write, or do not care about. A complete K definition of *IMP* (*While*) has been presented in [27]. Furthermore, an extended language *IMP++* is presented with the increment construct $++x$ which introduces a side effect for expressions (as in C-like languages), then *print* construct and the *input* constructs are introduced, and halting of program (internal and external ones) is added into the definition of *IMP++*.

A further particular type of small-step semantics is Reduction Semantics with Evaluation Contexts (RSEC), also known as contextual semantics, which models execution as a sequence of atomic rewrites of state, between each of which some small amount of time passes. Derivations are expressed as sequences that progress with time, rather than as trees of inference that conclude instantaneously. Reduction semantics was introduced by Matthias Felleisen and colleagues in 1992 [5]. The evaluation context style improves over small-step structural operational semantics in two ways: it gives a more compact semantics to context-sensitive reduction, by using parsing to find the next redex (a term that can be transformed in a single step); and it provides the possibility to also modify the context in

which a reduction occurs, making it much easier to deal with control-intensive features. Here, an evaluation context is a term with a "hole" (a placeholder) in the place of a subterm. Additionally, one can also include the configuration as a part of the evaluation context, and thus to have full access to semantic information "by need". The contexts allow the designer of a reduction semantics to factor the definition of calculus into one part that specifies the atomic steps of computation and the second part that controls where these steps may occur. Reduction semantics with evaluation contexts does precisely that it allows to formally define evaluation contexts; rules become mostly unconditional and reductions can only happen "in context". Reduction semantics is considered as yet another way to define single-step semantic relation [14], a small-step semantics where the atomic execution step is a rewrite of the program. For modeling programming languages, Felleisen-Hieb-style reduction semantics, and their type systems, a powerful software tool PLT Redex has been designed [6]. PLT Redex is a lightweight, embedded domain-specific programming language and it comes with a suite of tools for working with the semantics. In principle, Redex is hosted in PLT Scheme.

There are some other methods for defining formal semantics, e.g. natural semantics known also as a "big-step" operational semantics [13, 16], axiomatic semantics [9] for verification purposes with various extensions [17], action semantics [20] as a hybrid between denotational and operational semantics and game semantics [10, 11] defining a semantics by game trees and strategies; however, we do not consider these approaches in this paper.

The denotational semantics, also called mathematical semantics that expresses the meaning of a program in terms of mathematical structures and mappings between them, belongs to popular semantic methods. In a simplified case, sets are used as the semantic domains and the execution of statements is described by functions [30]. The general definition of denotational semantics uses lattices and homomorphisms between them [36]; an alternative formulation is based on relations [33]. Denotational semantics provides the results of program execution, but it does not consider the details during the execution process.

In the last decades, categories have become useful mathematical structures for modeling programs and program systems [2, 18, 22]. Categories enable to work not only with sets that are carriers of the most mathematical structures but also with more complex structures that are often used in computer science [23]. The morphisms can express the changes between these structures. Functors, the morphisms between categories, express mappings between categorical structures; they are suitable for describing useful properties of systems. In [34], we have defined a categorical denotational semantics of a procedural language, where we constructed a category of states as a model of a language.

In this paper, we construct a coalgebraic semantics of an imperative language. It is a further contribution to our research project to prepare a package of modules serving to define the semantics of programs by several semantic methods. This paper together with our previous work in [34] presents a theoretical foundation

for this package and is designed to be easily implementable. Therefore we try to use similar notions that can be implemented uniformly. But the coalgebraic approach requires to define behavior of programs step by step and it induces fundamental changes and necessity of new concepts different from the approach published in [34].

Categories of states are the bases for important and useful mathematical structures: algebras and coalgebras. The objects of these categories form a state space and their morphisms express the changes of states. Polynomial endofunctors over these categories characterize different kinds of systems and they model the changes of states. If we assume a state space X and a polynomial endofunctor Q over a state space, then an algebra a is defined as a mapping $a : Q(X) \to X$ and a coalgebra c, as its dual notion, is defined as $c : X \to Q(X)$. While algebras are useful for modeling the construction of programs, coalgebras enable to model their behavior. Our own results on the coalgebraic approach were published in [19, 35]. There are several publications defining coalgebraic semantics. The main ideas about coalgebraic approach come from [29]. The author considers coalgebras as transition systems describing the execution of programs in particular steps. Some additionally used language elements mentioned above are elaborated in a few works, e.g. declarations in [7, 31] and input/output using process algebras in [3, 8]. Also in the categorical approach, many publications define coalgebras for simplified kinds of systems and they ignore some details occurring in real ones.

We present a new approach on how to define the coalgebraic semantics of an imperative language containing the major features of a real imperative programming language with common statements, variable declarations, and input/output statements. We construct our coalgebra gradually from the signatures, their representations, base category, polynomial endofunctor up to coalgebraic representation. This detailed definition is more understandable also for practical programmers. The graphical representation of coalgebra can provide a good background also for educational purposes of young IT experts, and it seems to be easily implementable within our package of semantics.

In the next section, we present a short overview of the traditional definition of an operational semantics for a simple imperative language well-known as language *While* or *IMP*, presented e.g. in [21]. We adopted the structure of this language, and for pedagogical reasons, we refer to this language as *Jane*. In Section 2, we introduce traditional definition of operational semantics. Basic concepts and definitions for coalgebras are in Section 3. Then we extend our language to *E-Jane* (Section 4) with the additional constructs that are common in imperative languages. Our extended language *E-Jane* does not contain only the five basic statements of Dijkstra's language (variable assignment statement, empty statement, composed statement, conditional statement, and loop statement) but also variable declarations, input and output statements, and a block statement. To define the operational semantics in coalgebraic terms, we define the concepts of memory abstraction, statement list, declaration list, and configuration as abstract data types (Section 5). We represent these abstract types so that a rep-

resentation of a configuration contains the information what is to be executed together with an actual snapshot of a memory, and lists of input and output. We consider the set of configurations as our state space. In Section 6, we define how statements of a program are executed; in Section 7, we give the semantics of declarations as the morphisms between configurations. In Section 8, we construct a category of configurations, suitable polynomial endofunctor Q and Q-coalgebra. We finish the paper with a simple example illustrating our approach (Section 9).

2 TRADITIONAL DEFINITION OF OPERATIONAL SEMANTICS

Before we explain our approach of defining a coalgebraic operational semantics, an overview of the traditional definition of operational semantics is given. We introduce a simple imperative language *Jane* consisting of expressions (arithmetic and Boolean ones) and statements; then we give it an operational semantics.

2.1 The Language Jane

The formal syntax of *Jane* has been inspired by the formal syntax of *While* [21]. The language *Jane* is considered as a folklore (toy) language, without an official inventor; it has been used in many textbooks and papers, often with slight syntactic variations. The syntax of the language is described by syntactic domains and production rules. We consider the following syntactic domains:

- $n \in \mathbf{Num}$ – numerals (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.

The elements of **Num** and **Var** have no internal structure significant for the semantics. The syntactic domain **Aexpr** consists of all well-formed arithmetic expressions created by the following syntax:

$$e ::= n \mid x \mid e + e \mid e - e \mid e * e. \tag{1}$$

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

$$b ::= \texttt{false} \mid \texttt{true} \mid e = e \mid e \leq e \mid \neg b \mid b \wedge b. \tag{2}$$

As elements S of the syntactic domain **Statm**, we consider Dijkstra's five elementary statements, namely variable assignment, empty statement, sequential composition of statements, conditional statement, and loop statement:

$$S ::= x := e \mid \texttt{skip} \mid S; S \mid \texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ S \mid \ \texttt{while}\ b\ \texttt{do}\ S. \tag{3}$$

2.2 Operational Semantics of Jane

A structural operational semantics is defined as a transition system that describes each step of a program execution using transition relations. The traditional approach to this method defines transition relations by inference rules that describe the changes in memory during a program execution. As some abstraction of computer memory, the concept of a "state" is used. The set **State** of states is the basic semantic domain and its elements are functions from variables to values. For simplicity, we consider that all variables are implicitly typed as integer values from the set **Z**. Thus a state s is defined as a function

$$s : \mathbf{Var} \to \mathbf{Z}. \tag{4}$$

A change of a state means an actualization of a state which is written using a substitution. If a value of a variable y is changed to some new value $\mathbf{n}$, then a new state s' is defined as

$$s' = s[y \mapsto \mathbf{n}]. \tag{5}$$

This means that a new state s' is the same as s excluding the value of y, which was changed (substituted) to a new value $\mathbf{n} \in \mathbf{Z}$. Formally:

$$s'x = (s[y \mapsto \mathbf{n}])x = \begin{cases} \mathbf{n}, & \text{if } x = y; \\ sx, & \text{if } x \neq y. \end{cases} \tag{6}$$

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

Table 1. Semantics of arithmetic and Boolean expressions

Arithmetic and Boolean expressions serve for computing values of two implicit types of the language *Jane*, the type of integer values and the type of Boolean values, respectively. In defining the semantics of both types of expressions, an actual state is used but not changed in the process of evaluation. So the state plays only a passive rôle in the evaluation of expressions. The semantic domain for arithmetic expressions

is the set **Z** of integer numbers; for Boolean values, we introduce a new semantic domain **Bool** containing two elements – **true** and **false**:

$$\mathbf{Bool} = \{\mathbf{true}, \mathbf{false}\}. \tag{7}$$

Table 1 defines the semantic functions $[\![e]\!]$ and $[\![b]\!]$ mapping arithmetic expressions respectively Boolean expressions to functions from states to integer values respectively Boolean values. These functions produce transient data that are consumed during the program execution and whose values are never directly stored into memory except by assigning them to variables.

The changes of states are defined for particular statements by inference rules. An inference rule consists of a finite number of assumptions and a conclusion:

$$\frac{assumption_1, \ldots, assumption_n}{conclusion}\ (rule\_name)$$

$$\langle x := e, s\rangle \Rightarrow s[x \mapsto [\![e]\!]s]\ (1_{os}) \qquad \langle \texttt{skip}, s\rangle \Rightarrow s\ (2_{os})$$

$$\frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle}\ (3^1_{os}) \qquad \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}\ (3^2_{os})$$

$$\frac{[\![b]\!]s = \mathbf{true}}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \Rightarrow \langle S_1, s\rangle}\ (4^{\mathbf{true}}_{os})$$

$$\frac{[\![b]\!]s = \mathbf{false}}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \Rightarrow \langle S_2, s\rangle}\ (4^{\mathbf{false}}_{os})$$

$$\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \Rightarrow \langle \texttt{if}\ b\ \texttt{then}\ (S; \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip}, s\rangle\ (5_{os})$$

Table 2. Semantics of statements

The assumptions and the conclusion are transition relations between particular configurations. A configuration

$$\alpha = \langle S, s\rangle \tag{8}$$

expresses that a statement S is to be executed in a state s. A transition has form

$$\alpha \Rightarrow \alpha'. \tag{9}$$

Here α' can be either a state, if the statement is executed in one step, or a configuration $\langle S', s'\rangle$, where S' stands for a statement that is to be executed in the following step(s). So a transition describes an one-step action [37].

The inference rules for *Jane* in structural operational semantics [21] are given in Table 2. An operational semantics can also be defined for block statements

with declarations of local variables, e.g. in [24]. Our short overview of operational semantics serves only for illustration of the traditional approach, and we will treat these language constructions fully in our coalgebraic approach.

3 BASIC NOTIONS OF COALGEBRAS

Coalgebras are a useful tool for modeling the behavior of dynamic systems. They are defined over a base category whose objects create a state space and whose category morphisms are transitions.

To define a coalgebra for a system, we start with the signature of an abstract data type of states. A signature contains operation symbols defined on data types that can be:

- constructors – these generate the algebraic data types; they "work into" the data types;
- selectors (destructors) – these describe changes of states; they are also called behavioral operations, because they can provide observable values; they "work out" of a data structure;
- derived operations – these help to work with corresponding data structures.

Selectors play the most important rôle among the operation symbols for constructing coalgebras. They are interpreted as morphisms in a base category of states.

The dynamics, i.e. the execution of particular steps, is supplied by a polynomial endofunctor

$$F : \mathscr{C} \to \mathscr{C} \tag{10}$$

defined over a category $\mathscr{C}$ of states; it is indicated by the corresponding signature. The notion of a polynomial endofunctor [15] comes from its shape that is similar to that of polynomials, because it can be constructed from constants, products, coproducts and exponentials, e.g.:

$$FX = A_0 + A_1 \times X^{B_1} + A_2 \times X^{B_2} + \ldots + A_n \times X^{B_n}. \tag{11}$$

Here, X stands for a state space, the A_i, for $i = 0, \ldots, n$, are sets of observable values, the B_i are some fixed sets and $\times$ and $+$ are the operations on category objects: products and sums, respectively. The concrete shape of a polynomial endofunctor determines (characterizes) a particular kind of systems.

Then a coalgebra can be defined as a mapping from the state space to the result of the endofunctor applied to this state space; this mapping can be represented as a tuple of selectors:

$$\langle [\![sel_1]\!], \ldots, [\![sel_n]\!] \rangle : X \to FX. \tag{12}$$

Based on these general definitions, we show in the following sections how a coalgebra defining the operational semantics of an imperative programming language can be constructed.

4 EXTENDED IMPERATIVE LANGUAGE *E-JANE*

Because our aim is to treat a language with the constructs available in real programming languages, we extend the language introduced in Section 2 to the language *E-Jane* that contains the constructs common in the most imperative programming languages.

Assume a set **Decl** as a syntactic domain of declarations. Each variable used in a program has to be declared. We introduce a declaration $D \in \mathbf{Decl}$ as

$$D ::= \texttt{var}\ x. \tag{13}$$

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

We add three further statements: a block statement enclosed in the brackets `begin` and `end`; an input statement `read`, and an output statement `print`:

$$S ::= \ldots \mid \texttt{begin}\ D_1; \ldots; D_k; S\ \texttt{end} \mid \texttt{read}\ x \mid \texttt{print}\ e. \tag{14}$$

where $D_1; \ldots; D_k$ is a finite list of local declarations and $k \in \mathbf{N}_0$. The local declarations are visible only inside a given block. We have to take into consideration also the program global declarations that are located at the beginning of a program and they are visible within a whole program.

A program in *E-Jane* has then a form for $k \in \mathbf{N}_0, n \in \mathbf{N}$:

$$D_1; \ldots; D_k; S_1; \ldots; S_n, \tag{15}$$

i.e., it is a finite list of global declarations followed by a finite list of statements. So *E-Jane* becomes closer to many real imperative languages.

The following sections describe how we construct an operational semantics by a coalgebra for this language.

5 MEMORY AND ITS REPRESENTATION

Observing the behavior of a program during its execution means to define how a program is being executed step by step and how the snapshots of the memory are being changed in detail. First, we specify a structure (a data type) for the state space. We consider as our state space the data type *Config* of configurations for which we introduce a signature.

Each configuration is a tuple whose first item is a finite list of declarations together with a list of statements to be executed; its second item is the actual memory content. The third and the fourth items are lists of input and output values, respectively. We start with the signature for the lists of declarations and statements; we follow with the signature for memory; then we construct a signature of configurations. We put emphasis on the selector operations that play an impor-

tant rôle in coalgebras. Then we define a suitable representation of the specified types.

5.1 Signature of Configurations

As we have mentioned, a program in the imperative language *E-Jane* is a finite list of declarations of global variables followed by a finite list of statements. Declarations do not affect the computer memory content, but they are important because they reserve cells for declared variables.

We start with specifying a parametrized signature [4] of finite lists, then we define its two instantiations for the declaration list and statement list. Then we specify a signature for abstract memory. Using these three signatures, we specify the data type *Config* of configurations.

$$\begin{array}{l} \Sigma_{List} = List_{fin}\,[Item] \\ types : List, Item \\ opns : \\ \quad init : \rightarrow Item \\ \quad head : List \rightarrow Item \\ \quad tail : List \rightarrow List \end{array} \tag{16}$$

The operation *init* creates the empty list of items. The operation *head* extracts the first item and *tail* returns the rest of a list of items.

Let *Decl* and *Statm* be the type names for declarations and statements, respectively. Then we define the signature of declaration lists by setting $Item = Decl$ as the instantiation

$$\Sigma_{Decl\_List} = \Sigma_{List}\,[Decl]\,.$$

Similarly, setting $Item = Statm$ we get the signature of statement lists

$$\Sigma_{Statm\_List} = \Sigma_{List}\,[Statm]\,.$$

An abstract memory is a basic concept in the semantics of imperative languages. It is often called *state* in many publications, but for our purposes the notion *memory* is more appropriate. Each variable occurring in a program has to be allocated, i.e., a memory cell is reserved and named by the elaboration of a declaration. The value of an allocated variable can be assigned and modified inducing a change of the actual memory.

According to these ideas about the concept of abstract memory, we formulate a signature Σ_{Memory} as an abstract data type which uses types *Var* and *Value* for variables and values, respectively. This signature consists of types and operation

specifications on the type *Memory*:

$$\begin{array}{lll} \Sigma_{Memory} = & & \\ & types: & Memory, Var, Value \\ & opns: & init: \to Memory \\ & & alloc: Var, Memory \to Memory \\ & & get: Var, Memory \to Value \\ & & del: Memory \to Memory \end{array} \tag{17}$$

The operation specifications have the following intuitive meaning (the following subsection will explain the notion of "nesting levels" in more detail):

- *init* merely creates the initial memory of a program;
- *alloc* reserves a new memory cell for a variable at its actual nesting level;
- *get* returns a variable value in a given memory cell at its actual nesting level;
- *del* deallocates (releases) all memory cells for the variables declared on the highest nesting level.

The language *E-Jane* contains the statements for user input and output. Input and output values are of the type *Value* and they form the lists. The corresponding signatures are

$$\Sigma_{Input} = \Sigma_{List}[I\_Value], \tag{18}$$

$$\Sigma_{Output} = \Sigma_{List}[O\_Value] \tag{19}$$

where *I_Value* and *O_Value* are type names for input and output values, respectively.

To specify the process of program execution we introduce a new type *Config* by its signature:

$$\begin{array}{lll} \Sigma_{Config} = & \Sigma_{Decl\_List} + \Sigma_{Statm\_List} + \Sigma_{Memory} + \Sigma_{Input} + \Sigma_{Output} + & \\ & types: Config & \\ & opns: next: Config \to Config & \\ & \quad read: Config, I\_Value \to Config & \\ & \quad print: Config \to O\_Value, Config & \end{array} \tag{20}$$

This data type introduces a new type *Config* with its operations (transition functions), where

- *next* provides the next configuration;
- *read* takes an input value and stores it in a corresponding memory cell;
- *print* computes a value of an argument and produces it as an observable value.

5.2 Representation of Types

Now we assign a representation to the data types specified above. First, we represent the unstructured data type *Value* as a set of integers **Z** together with the undefined value $\perp$:

$$\mathbf{Value} = \mathbf{Z} \cup \{\perp\}\,. \tag{21}$$

We assign to the type *Var* a countable set **Var** of variable names. To deal with nested block statements, we extend this set with special (dummy) variables `begin` and `end` that are not declared. An undefined variable $\perp$ is also in **Var**, but it serves only for the initial memory of a program.

We represent the type *List* in Σ_{Decl_List} as a set of finite lists **Decl_List**, which elements are finite lists of declarations D^*. Analogously, the type *State* in Σ_{Statm_List} is represented also as a set of finite lists **Statm_List** with elements S^* standing for the finite lists of statements. That means

$$D^* = D_1; \ldots; D_m, \tag{22}$$

$$S^* = S_1; \ldots; S_n, \tag{23}$$

where $D_i \in \mathbf{Decl}$, for $i = 1, \ldots, m$, and $S_i \in \mathbf{Statm}$, for $i = 1, \ldots, n$. The representations of the operation symbols *head* and *tail* are defined as it is regular for lists.

Because our language contains also a block statement, possibly with local variables declarations, we will consider the nesting level of a block. This nesting level allows us to create a variable environment, the notion known from operational semantics, and it enables us to distinguish local declarations from global ones. Therefore we introduce the set **Level** of nesting levels denoted by natural numbers l:

$$l \in \mathbf{Level}, \quad \text{where} \quad \mathbf{Level} = \mathbf{N}. \tag{24}$$

We assign to the type *Memory* the set **Memory** of all possible non-empty memory contents:

$$\mathbf{Memory} = \{m : \mathbf{Var} \times \mathbf{Level} \to \mathbf{Value}\}\,.$$

Each memory m expresses one moment of program execution, an actual snapshot of a computer memory. The function m is identified with its graph [30], $graph(m)$, i.e., a set of pairs, where the first member of each pair is an argument of this function and the second member is the value of the function:

$$graph(m) = \{((x, l), v) \mid (x, l) \in dom(m) \wedge m(x, l) = v\}\,. \tag{25}$$

For a visualization of an actual memory, we can write the function m as a table with possibly unfilled cells denoted by $\perp$ expressing an undefined value for a declared variable. This visualization increases the readability; it is illustrated on the left-hand side in Figure 1.

variable	level	value
x_1	1	v_1
⋮		
x_n	l	v_n

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

Figure 1. Visualization: actual memory and initial memory

Now we define the representations of the operation specifications from the signature Σ_{Memory} as follows. The operation specification *init* is represented by a function $[\![init]\!]$ defined by

$$[\![init]\!] = m_0 = \{((\perp, 1), \perp)\} \tag{26}$$

which creates an initial memory m_0 of a program, with no declared variable. Its rôle is only to set the nesting level to a value 1 at the beginning of program execution as it is on the right-hand side in Figure 1.

For defining the representation of the further operations on a memory, we need to specify an actual (maximal) level of a nesting. Let

$$m = \{((x_1, l_1), v_1), \ldots ((x_m, l_n), v_k)\}$$

be an actual memory. We define an auxiliary function

$$maxlevel : \mathbf{Memory} \to \mathbf{Level} \tag{27}$$

defined by

$$maxlevel(m) = L, \tag{28}$$

such that $\exists j = 1, \ldots n. L \geq l_j$.

The operation $[\![alloc]\!]$ adds a new item to the memory m (creates a new entry in the table); it is defined as

$$[\![alloc]\!](x, m) = m \cup \{((x, maxlevel(m)), \perp)\}. \tag{29}$$

This operation sets the actual nesting level l to the declared variable (left table in Figure 2).

The operation $[\![get]\!]$ returns the value of a variable declared on the highest nesting level. We introduce an abbreviation *Highest* which expresses the maximum nesting level where a variable x is declared:

$$Highest(m, x) = \max \{l' \mid l' \in \mathbf{Level} \text{ and } (x, l') \in dom(m)\};$$

For simplification of the formulation, we define the following predicate:

$$Defined(m, x) \equiv_{def} \exists l' \in \mathbf{Level}.(x, l') \in dom(m)$$

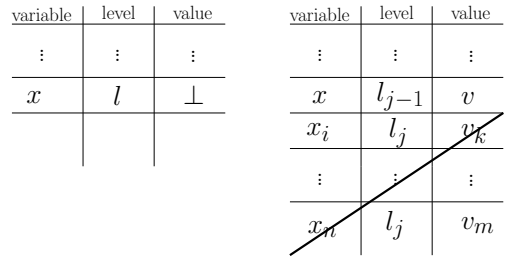

Figure 2. Variable allocation and deallocation

that expresses whether the variable x is declared in an actual memory m.

The operation $[\![get]\!]$ is then defined as

$$[\![get]\!](x,m) = \begin{cases} m(x,l), & \text{where } l = \mathit{Highest}(m,x), \\ & \text{if } \mathit{Defined}(m,x); \\ \bot, & \text{otherwise.} \end{cases} \tag{30}$$

The operation $[\![del]\!]$ deallocates (releases from the table) all the variables declared on the highest nesting level (right table in Figure 2):

$$[\![del]\!]m = m \setminus \{((x, \mathit{maxlevel}(m)), v) \mid x \in \mathbf{Var} \wedge v \in \mathbf{Value}\}. \tag{31}$$

We also consider a special memory content

$$m_\bot = ((\bot, 0), \bot) \tag{32}$$

expressing the undefined memory content when a program aborts.

The representation of the type *List* in Σ_{Input} is a finite list $i^* = i_1; \ldots; i_m$. Similarly, the type *List* in Σ_{Output} is a finite list $o^* = o_1; \ldots; o_n$, where $i_j, o_k \in \mathbf{Value}$. For simplicity, we consider here only finite lists of input and output values. These lists form the semantic domains **Input** of lists of input values and **Output** of lists of output values.

The last type to be represented is *Config*. We represent it by a set **Config** of configurations as a cartesian product

$$\mathbf{Config} = \mathbf{Program} \times \mathbf{Memory} \times \mathbf{Input} \times \mathbf{Output}, \tag{33}$$

where **Program** consists of the lists of global declarations/statements to be yet elaborated and/or executed. A configuration is a quadruple

$$\mathit{config} = ([\![D^*; S^*]\!], m, i^*, o^*) \tag{34}$$

for $m \in \mathbf{Memory}$, $i^* \in \mathbf{Input}$ and $o^* \in \mathbf{Output}$. We consider the representation **Config** as the state space; its elements will be the objects in the base category of

our coalgebra. We note that we consider at the beginning of program execution that o^* is empty, $o^* = \varepsilon$.

The most important operations for defining the operational semantics by a coalgebra are the transition operations $[\![next]\!]$, $[\![read]\!]$ and $[\![print]\!]$ that we will define later. These are the morphisms in our category of configurations.

In the next section, we define how a step of the execution of statements can be defined as a morphism between configurations.

6 EXECUTION OF STATEMENTS

The interpretation of the operations *next*, *read*, and *print* represents the execution of statements. Here we discuss how each of these operations can be defined. We note that in operational semantics we model only one step of a statement execution.

Generally, the interpretation of *next* is a mapping

$$[\![next]\!] : \mathbf{Config} \to \mathbf{Config}. \tag{35}$$

The finite list S^* of statements is changed in each execution step according to the correspondingly executed statement. The first member of this sequence is a statement to be executed in a given memory m. Now we define the function $[\![next]\!]$ for each kind of statement.

The assignment statement $x := e$ is to be executed in a memory m in one step. The operation $[\![next]\!]$ for this statement is defined as

$$[\![next]\!]([\![x := e; S^*]\!], m, i^*, o^*) = ([\![S^*]\!], m', i^*, o^*) \tag{36}$$

where

$$m' = \begin{cases} m\left[((x, \mathit{Highest}(m, x)), v) \quad \mapsto \quad ((x, \mathit{Highest}(m, x)), [\![e]\!]m)\right], \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \text{if } \mathit{Defined}(m, x); \\ m_\perp, \qquad\qquad\qquad\qquad\qquad\qquad\quad \text{otherwise.} \end{cases}$$

The transition mapping returns the tail of the statement list together with a new memory that contains a new value $[\![e]\!]m$ for the variable x.

In like manner, we define the semantics for the empty statement `skip` that executes also in one step but without any change of memory. Its semantics is defined as a morphism without change of a memory m:

$$[\![next]\!]([\![\texttt{skip}; S^*]\!], m, i^*, o^*) = ([\![S^*]\!], m, i^*, o^*). \tag{37}$$

This statement is considered as an identity on memory. On the other hand, this morphism is not an identity on configuration because of shortening of the statement list.

To define the semantics of a sequence of statements, we need to distinguish two situations. Assume that the rest of the program has a form

$$S_1; S_2; S^*. \tag{38}$$

The role of operational semantics is to define only the first execution step. A statement S_1 can be executed either in one step or in more steps. In the first case, e.g., if it is a statement being executed in one step, after this step the statements $S_2; S^*$ remain to be executed. In the second case, after the first step a sequence $S_1'; S_2; S^*$ has to be executed. Thus we define the mapping $[\![next]\!]$ for these situations as

$$[\![next]\!]([\![(S_1; S_2); S^*]\!], m, i^*, o^*) = \begin{cases} ([\![S_2; S^*]\!], m', i'^*, o'^*), \\ \qquad \text{if } \langle S_1, m\rangle \Rightarrow m'; \\ ([\![S_1'; S_2; S^*]\!], m', i'^*, o'^*), \\ \qquad \text{if } \langle S_1, m\rangle \Rightarrow \langle S_1', m'\rangle. \end{cases} \tag{39}$$

The memory m' depends on the actual execution of the statement S_1. The lists i'^* and o'^* represent the situation when the statement S_1 stands for a user input or output.

Consider now that the first statement to be executed is a conditional statement S as the first statement in the list $S; S^*$:

$$S = \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2. \tag{40}$$

The first step depends on the value of the Boolean expression b in the actual memory m. If $[\![b]\!]m = \textbf{true}$, then the execution follows with the statement S_1; and with the statement S_2, otherwise. We note that the execution of the conditional statement is still deterministic and the memory m is not changed during this first step. Therefore, the semantics of conditional is

$$[\![next]\!]([\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2; S^*]\!], m, i^*, o^*) = \begin{cases} ([\![S_1; S^*]\!], m, i^*, o^*), & \text{if } [\![b]\!]m = \textbf{true}; \\ ([\![S_2; S^*]\!], m, i^*, o^*), & \text{if } [\![b]\!]m = \textbf{false}. \end{cases} \tag{41}$$

The first step of the execution of the loop statement `while` b `do` S is the same as the execution of the following conditional statement:

$$\begin{aligned} &[\![next]\!]\,([\![\texttt{while } b \texttt{ do } S; S^*]\!], m, i^*, o^*) \\ &\quad = ([\![\texttt{if } b \texttt{ then } S; \texttt{while } b \texttt{ do } S \texttt{ else skip}; S^*]\!], m, i^*, o^*)\,. \end{aligned} \tag{42}$$

Now we define the semantics of the user input statement `read` x. It is executed in one step and it assigns to a declared variable x an input value v, i.e., it changes

a configuration as described by the transition function

$$[\![read]\!] : \mathbf{Config} \to \mathbf{Config}^{\mathbf{Value}} \tag{43}$$

defined as

$$[\![read]\!]([\![\texttt{read}\ x; S^*]\!], m, i^*, o^*) = \begin{cases} \lambda v'.([\![S^*]\!], m', \mathit{tail}(i^*), o^*), \\ \quad \text{if } \mathit{Defined}(m, x); \\ ([\![S^*]\!], m_\perp, \mathit{tail}(i^*), o^*), \\ \quad \text{otherwise}, \end{cases} \tag{44}$$

where $m' = m[((x, \mathit{Highest}(m, x)), v) \mapsto ((x, \mathit{Highest}(m, x)), v')]$. That means that any value stored in a variable x declared on the highest nesting level is changed to the input value v'.

The output statement `print` e is also executed in one step. It computes a value of an arithmetic expression e in an actual memory m and it provides a result as an observable value. A memory is not changed but the configuration is. The semantics of this statement can be described by the transition function

$$[\![print]\!] : \mathbf{Config} \to \mathbf{Value} \times \mathbf{Config} \tag{45}$$

defined as

$$[\![print]\!]([\![\texttt{print}\ e; S^*]\!], m, i^*, o^*) = ([\![e]\!]m, ([\![S^*]\!], m, i^*, [\![e]\!]m; o^*))\,. \tag{46}$$

The first step of the execution of a block statement $S = \texttt{begin}\ D^*; S'\ \texttt{end}$ is modeled as

$$\begin{gathered} [\![next]\!]([\![\texttt{begin}\ D^*; S'\texttt{end}; S^*]\!], m, i^*, o^*) = \\ ([\![D^*; S'\ \texttt{end}; S^*]\!], [\![begin]\!]m, i^*, o^*) \end{gathered} \tag{47}$$

where the special declaration `begin` (see in Section 7) is elaborated and the execution proceeds with elaborating the local declarations and executing the body of the block.

For indicating that an execution step yields an undefined result, we introduce an auxiliary mapping

$$[\![abort]\!] : \mathbf{Config} \to \mathbf{Config} \tag{48}$$

defined as

$$[\![abort]\!](\mathit{config}) = (\varepsilon, m_\perp, \varepsilon, \varepsilon) \tag{49}$$

where ε denotes the empty list. This definition ensures that an aborted program stops in a stuck configuration that does not contain any statements to be executed.

In the next section, we define the elaboration of declarations in a uniform way.

7 SEMANTICS OF DECLARATIONS

Each variable occurring in an *E-Jane* program has to be declared. Declarations are elaborated, i.e., a new memory cell is allocated on the actual nesting level with the undefined value. Declarations form a finite list $D_1; D_2, \ldots; D_n$, where each $D_i, i = 1, \ldots, n$ has the structure `var` x_i.

We represent each declaration as a function on a memory:

$$[\![\texttt{var}\ x]\!] : \mathbf{Memory} \to \mathbf{Memory}. \tag{50}$$

We define it for a given memory m as follows:

$$[\![\texttt{var}\ x]\!]m = [\![alloc]\!](x, m). \tag{51}$$

This definition expresses that a declaration is elaborated in one step.

Declarations may appear in a program either on the global level with $l = 1$ or in a block statement with a nesting level $l > 1$. In case of global declarations, new entries are allocated with the initial nesting level $l = 1$; for instance, for a variable x on the level $l = 1$ the following entry is being created:

$$((x, 1), \bot). \tag{52}$$

To supply the incrementation of a nesting level in the case of a block statement, we introduce two special variables: `begin`, `end` $\in$ **Var**. A fictive declaration `begin` serves to bound the locally declared variables to an incremented nesting level while `end` indicates the end of a block statement where locally declared variables are released from the table. These declarations are also elaborated in one step by functions $[\![begin]\!]$ and $[\![end]\!]$ on a memory m:

$$[\![begin]\!], [\![end]\!] : \mathbf{Memory} \to \mathbf{Memory}, \tag{53}$$

defined as follows:

$$\begin{aligned} [\![begin]\!]m &= m \cup \{((\texttt{begin}, maxlevel(m) + 1), \bot)\}, \\ [\![end]\!]m &= [\![del]\!]m. \end{aligned} \tag{54}$$

Because the objects of our categorical model are configurations and the elaboration of a declaration affects a given configuration, we define a morphism $[\![next]\!]$ for declarations as

$$[\![next]\!] : \mathbf{Config} \rightharpoonup \mathbf{Config}, \tag{55}$$

where

$$\begin{aligned} [\![next]\!]([\![\texttt{var}\ x; D^*; S^*]\!], m, i^*, o^*) &= ([\![D^*; S^*]\!], [\![\texttt{var}\ x]\!]m, i^*, o^*), \\ [\![next]\!]([\![\texttt{begin}\ D^*; S'\ \texttt{end}; S^*]\!], m, i^*, o^*) &= ([\![D^*; S'\ \texttt{end}; S^*]\!], [\![begin]\!]m, i^*, o^*), \\ [\![next]\!]([\![\texttt{end}; S^*]\!], m, i^*, o^*) &= ([\![S^*]\!], [\![end]\!]m, i^*, o^*). \end{aligned} \tag{56}$$

However, a morphism $[\![next]\!]$ is always defined for declarations, it can be undefined only for statements, as we showed in Section 6. So the semantics of declarations corresponds with the meaning of declarations in traditional operational semantics, i.e., each declaration actualizes an environment of variables.

8 COALGEBRA FOR THE LANGUAGE *E-JANE*

In the previous sections, we defined the notions necessary for the construction of a base category for a coalgebra. Now we construct the category $\mathscr{C}onfig$ consisting of

- configurations $config = ([\![D^*; S^*]\!], m, i^*, o^*)$ as the category objects;
- mappings $[\![next]\!], [\![read]\!], [\![print]\!]$ and $[\![abort]\!]$ as the category morphisms.

The objects in this category form the state space for our coalgebra and the morphisms are transition mappings, each of them modeling one step of program execution.

We check whether so defined structure is a category:

- Each object has to have an identity morphism. However, no morphism defining the operational semantics of *E-Jane* is an identity. To satisfy this category property, we need to define explicitly that each object has an identity morphism id_{config}.
- A composition of two composable morphisms is a morphism in $\mathscr{C}onfig$, e.g., for the execution of a sequence of statements.
- A composition of morphisms is associative, trivially.

Our category $\mathscr{C}onfig$ has a terminal object, the undefined configuration

$$config_\perp = (\varepsilon, m_\perp, \varepsilon, \varepsilon) \tag{57}$$

that indicates aborting of a program. From any object in $\mathscr{C}onfig$ there exists a unique morphism to this object because the running program can abort in any step; so $config_\perp$ is a terminal object in $\mathscr{C}onfig$. The category has no initial object because the starting of execution depends on an actual program that should be executed. Therefore the initial configuration is different for each program.

The execution of a loop statement is modeled as a path of morphisms, i.e. a composition of morphisms modeling the particular steps of an execution. When the loop is executed in a finite number of steps, we get some final configuration and the execution of a program can follow. When this path is infinite, we need to ensure that our model is a category, i.e., there exists an object that is a composition of an infinite path. Thus our category $\mathscr{C}onfig$ needs to have colimits [1] for all diagrams consisting of an infinite composition of configurations. The definition of a colimit in the category $\mathscr{C}onfig$ is explained in [34].

Now we have the base category that is a model of *E-Jane*; thus we can proceed to construct a coalgebra modeling the behavior of programs written in *E-Jane*. The

objects of our category form the state space of a coalgebra and the morphisms are the transition mappings. Now, we construct the polynomial endofunctor for this kind of systems. Generally, the following polynomial endofunctor seems to be appropriate for our purposes:

$$Q(\mathbf{Config}) = 1 + \mathbf{Config} + O \times \mathbf{Config} + \mathbf{Config}^I. \tag{58}$$

Here $I \subseteq \mathbf{Value}$ denotes the domain of input values and $O \subseteq \mathbf{Value}$ denotes the domain of output values of the program execution.

The operation + in the definition of the functor Q expresses distinct, mutual exclusive results of the functor. We discuss the possible results:

- $Q(\mathbf{Config}) = 1$ when a program aborts, i.e. it abnormally finishes and does not return a result. This situation arises when the morphism $[\![abort]\!]$ in $\mathscr{C}onfig$ is performed:
$$Q(config) = [\![abort]\!](config);$$
- if $Q(\mathbf{Config}) = \mathbf{Config}$, a new configuration is achieved by an elaboration of a declaration or an execution of a statement with no input and output. This situation occurs in the category by performing the morphism $[\![next]\!]$:
$$Q(config) = [\![next]\!](config);$$
- if $Q(\mathbf{Config}) = O \times \mathbf{Config}$, a change of configuration happens together with producing some observable output value. The morphism $[\![print]\!]$ is performed:
$$Q(config) = [\![print]\!](config);$$
- if $Q(\mathbf{Config}) = \mathbf{Config}^I$, an input value $i \in I$ is read by the execution of the statement `read`
$$Q(config) = [\![read]\!](config).$$

Now we can define the Q-coalgebra for the programming language *E-Jane* as a mapping:

$$\langle [\![abort]\!], [\![next]\!], [\![print]\!], [\![read]\!] \rangle : \mathbf{Config} \to Q(\mathbf{Config}). \tag{59}$$

This coalgebra models the execution of a program in particular steps, i.e., it provides the operational semantics of any program written in *E-Jane*. We note that our coalgebra models the behavior of programs written in any programming languages containing corresponding constructs.

9 EXAMPLE

We illustrate our approach on a simple program, which uses the most of the constructs of *E-Jane*. Assume a program P:

```
var x; var y;
input x; input y;
if x <= y then begin var z;
                  z:=x; x:=y; y:=z
               end
          else skip;
print x
```

We introduce here some abbreviations:

$$\begin{aligned}
&D_1 = \texttt{var}\ x;\ D_2 = \texttt{var}\ y;\\
&S_1 = \texttt{read}\ x;\ S_2 = \texttt{read}\ y;\\
&S_3 = \texttt{if}\ x <= y\ \texttt{then begin var}\ z;\\
&\qquad z := x; x := y; y := z\ \texttt{end else skip};\\
&S_4 = \texttt{print}\ x
\end{aligned}$$

Let the input values for x and y be **3** and **5**, respectively. An input list is then $i^* = (\mathbf{3}, \mathbf{5})$ and an output list is empty, $o^* = \varepsilon$.

An initial configuration is

$$\mathit{config}_0 = (\llbracket D_1; D_2; S_1; S_2; S_3; S_4 \rrbracket, m_0, i^*, o^*)$$

and an initial memory m_0 contains only information about starting value of declaration nesting, $m_0 = ((\bot, 1), \bot)$ (Figure 3).

m_0		
$\bot$	1	$\bot$

Figure 3. Initial memory

First, the declarations are elaborated in separate steps:

$$\begin{aligned}
Q(\mathit{config}_0) &= \llbracket \mathit{next} \rrbracket(\mathit{config}_0) = \mathit{config}_1\\
&= (\llbracket D_2; S_1; S_2; S_3; S_4 \rrbracket, \llbracket \texttt{var}\ x \rrbracket m_0, (\mathbf{3}, \mathbf{5}), \varepsilon),\\
Q(\mathit{config}_1) &= \llbracket \mathit{next} \rrbracket(\mathit{config}_1) = \mathit{config}_2\\
&= (\llbracket S_1; S_2; S_3; S_4 \rrbracket, \llbracket \texttt{var}\ y \rrbracket m_1, (\mathbf{3}, \mathbf{5}), \varepsilon),
\end{aligned}$$

where $m_1 = \llbracket \texttt{var}\ x \rrbracket m_0$ and $m_2 = \llbracket \texttt{var}\ y \rrbracket m_1$ (Figure 4).

m_1		
x	1	$\bot$

m_2		
x	1	$\bot$
y	1	$\bot$

Figure 4. Memory with declared variables

The execution of statements is realized by applying the functor Q in particular steps. First, two input statements are performed:

$$\begin{aligned} Q(config_2) &= [\![read]\!](config_2) = config_3 \\ &= ([\![S_2; S_3; S_4]\!], m_3, (\mathbf{5}), \varepsilon), \\ Q(config_3) &= [\![read]\!](config_3) = config_4 \\ &= ([\![S_3; S_4]\!], m_4, \varepsilon, \varepsilon), \end{aligned}$$

and memory after performing these two steps is depicted in Figure 5.

m_3		
x	1	**3**
y	1	$\perp$

m_4		
x	1	**3**
y	1	**5**

Figure 5. Memory after user inputs

Next, the conditional statement is executed,

$$\begin{aligned} Q(config_4) &= [\![next]\!](config_4) = config_5 \\ &= ([\![\texttt{begin var } z; z := x; x := y; y := z \texttt{ end}; S_4]\!], m_4, \varepsilon, \varepsilon), \end{aligned}$$

and a Boolean condition is evaluated

$$[\![x \leq y]\!] m_4 = \mathbf{true}.$$

Because the condition is evaluated to true, the next step is an inner block.

$$\begin{aligned} Q(config_5) &= [\![next]\!](config_5) = config_6 \\ &= ([\![\texttt{var } z; z := x; x := y; y := z \texttt{ end}; S_4]\!], m_5, \varepsilon, \varepsilon), \end{aligned}$$

and an actual memory m_5 contains also information about entering the local block (Figure 6).

m_5		
x	1	**3**
y	1	**5**
`begin`	2	$\perp$

Figure 6. Memory after entering the local block

The next step is an elaboration of a declaration inside the block:

$$\begin{aligned} Q(config_6) &= [\![next]\!](config_6) = config_7 \\ &= ([\![z := x; x := y; y := z \texttt{ end}; S_4]\!], m_6, \varepsilon, \varepsilon), \end{aligned}$$

and an actual memory m_6 is in Figure 7.

m_6		
x	1	**3**
y	1	**5**
`begin`	2	$\perp$
z	2	$\perp$

Figure 7. Memory after local declaration inside the block

The next three steps represent performing three variables assignments:

$$\begin{aligned} Q(config_7) &= [\![next]\!](config_7) = config_8 \\ &= ([\![x := y; y := z \texttt{ end}; S_4]\!], m_7, \varepsilon, \varepsilon), \\ Q(config_8) &= [\![next]\!](config_8) = config_9 \\ &= ([\![y := z \texttt{ end}; S_4]\!], m_8, \varepsilon, \varepsilon), \\ Q(config_9) &= [\![next]\!](config_9) = config_{10} \\ &= ([\![\texttt{end}; S_4]\!], m_9, \varepsilon, \varepsilon), \end{aligned}$$

and particular changes of memory are depicted in Figure 8.

m_7		
x	1	**3**
y	1	**5**
`begin`	2	$\perp$
z	2	**3**

m_8		
x	1	**5**
y	1	**5**
`begin`	2	$\perp$
z	2	**3**

m_9		
x	1	**5**
y	1	**3**
`begin`	2	$\perp$
z	2	**3**

Figure 8. Memory after variables assignments

After those statements, the execution of a local block must be finished:

$$\begin{aligned} Q(config_{10}) &= [\![next]\!](config_{10}) = config_{11} \\ &= ([\![S_4]\!], [\![end]\!]m_9, \varepsilon, \varepsilon) \end{aligned}$$

where $[\![\texttt{end}]\!]m_9 = m_{10}$ and actual memory after deleting the record of the block is in Figure 9.

m_{10}		
x	1	**5**
y	1	**3**

Figure 9. Memory after deleting the local declarations

The last step is a performing of an output statement which provides user output of computed value:

$$\begin{aligned} Q(config_{11}) &= [\![print]\!](config_{11}) = config_{12} \\ &= (\mathbf{5}, (\varepsilon, m_{10}, \varepsilon, (\mathbf{5}))). \end{aligned}$$

Our simple program is executed in particular steps by applying the endofunctor Q. These steps form a finite path in the category $\mathscr{C}onfig$ as we can see in Figure 10.

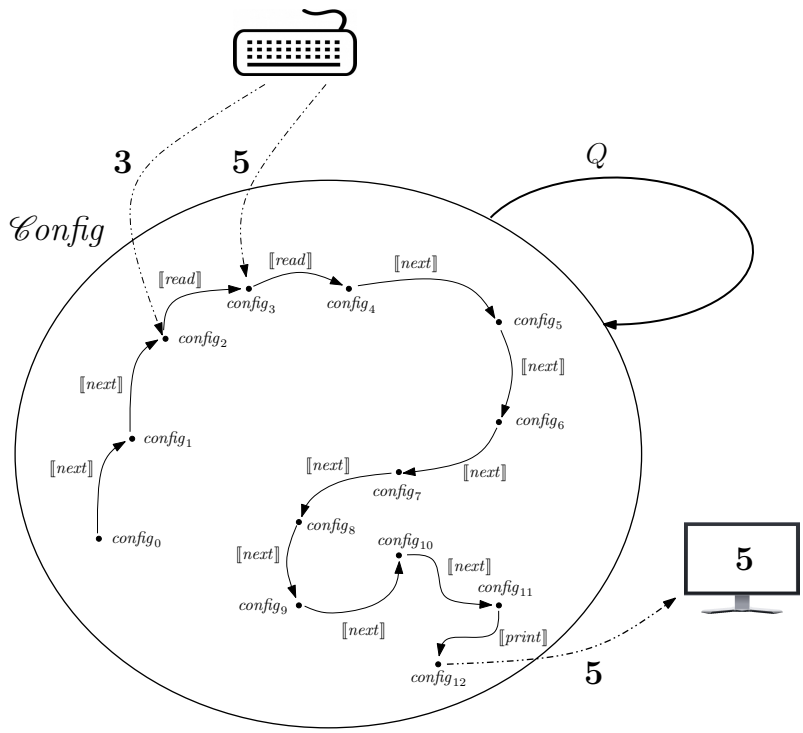

Figure 10. A program execution in coalgebra

10 CONCLUSION

Operational semantics can provide a useful information to programmers on how a program is executed, i.e. on its behavior. This information is important in the process of preparing the program or for implementation purposes. Unfortunately, traditional methods (only minor exceptions) consider some constructions as irrelevant for operational semantics. Therefore, it is hard to provide an operational semantics for a program written in a real imperative language. In this paper, we present a new approach that overcomes the lack of traditional approach. We define a simple imperative language *E-Jane* that contains most of the obvious constructs of imperative programming languages. We construct the operational semantics of this language as a coalgebra over a category of configurations. Our definition of configurations and their choice for the states, instead of memory abstractions, enables us to treat statements and declarations in a uniform way that is a further advantage of our approach. Each step of an execution is described as an application of a polynomial endofunctor Q over the category of configurations that characterizes this kind of systems. Our coalgebra also describes how input and output values go into and go out of a system. Another advantage of our approach is its possibility to get a graphical representation of the particular steps of the execution of a program which is more understandable also for practical programmers.

In this paper, we use the language *E-Jane* that has some simplifications for accentuating the principles of our approach. The constructed coalgebra can serve also as a basis for our further research. We would like to introduce also other types of values into our coalgebra and to define an operational semantics for procedures. This would be a starting point to define a coalgebraic semantics for component-based systems.

Acknowledgment

This work has been supported within the scope of the project "Semantic Technologies for Computer Science Education (SemTech)" by the Slovak Research and Development Agency under the contract No. SK-AT-2017-0012 respectively by the Austrian OeAD under the WTZ contract SK 14/2018, as well as by the Linz Institute of Technology (LIT), Project LOGTECHEDU "Logic Technology for Computer Science Education".

REFERENCES

[1] Adámek, J.—Herrlich, H.—Strecker, G.: Abstract and Concrete Categories. 1st Edition, J. Wiley and Sons, Inc., 1990.

[2] Barr, M.—Wells, C.: Category Theory for Computing Science. 2nd Edition. Prentice Hall, 1995.

[3] Crole, R. L.—Gordon, A. D.: Relating Operational and Denotational Semantics for Input/Output Effects. Mathematical Structures in Computer Science, Vol. 9, 1999, No. 2, pp. 125–158, doi: 10.1017/S0960129598002709.

[4] Ehrig, H.—Mahr, B.: Fundamentals of Algebraic Specification 1: Equations and Initial Semantics. 1st Edition. Springer, 1985, doi: 10.1007/978-3-642-69962-7.

[5] Wright, A. K.—Felleisen, M.: A Syntactic Approach to Type Soundness. Information and Computation, Vol. 115, 1994, No. 1, pp. 38–94, doi: 10.1006/inco.1994.1093.

[6] Felleisen, M.—Findler, R. B.—Flatt, M.: Semantics Engineering with PLT Redex. The MIT Press, Cambridge, MA, 2009.

[7] Fernández, M.: Programming Languages and Operational Semantics: A Concise Overview. Springer, 2014.

[8] Gordon, A. D.: An Operational Semantics for I/O in Lazy Functional Languages. Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93), Copenhagen, Denmark, 1993, pp. 136–145, doi: 10.1145/165180.165199.

[9] Hoare, C. A. R.: An Axiomatic Basis for Computer Programming. Communication of the ACM, Vol. 12, 1969, No. 10, pp. 576–580, doi: 10.1145/363235.363259.

[10] Hyland, J.—Ong, C.-H. L.: On Full Abstraction for PCF I., II., III. Information and Computation, Vol. 163, 2000, No. 2, pp. 285–408, doi: 10.1006/inco.2000.2917.

[11] Japaridze, G.: In the Beginning Was Game Semantics. In: Majer, O., Pietarinen, A. V., Tulenheimo, T. (Eds.): Games: Unifying Logic, Language, and Philosophy. Springer, Dordrecht, Logic, Epistemology, and the Unity of Science, Vol. 15, 2009, pp. 249–350, doi: 10.1007/978-1-4020-9374-6_11.

[12] Jaskelioff, M.—Ghani, N.—Hutton, G.: Modularity and Implementation of Mathematical Operational Semantics. Electronic Notes in Theoretical Computer Science, Vol. 229, 2011, No. 5, pp. 75–95, doi: 10.1016/j.entcs.2011.02.017.

[13] Kahn, G.: Natural Semantics. In: Brandenburg, F. J., Vidal-Naquet, G., Wirsing, M. (Eds.): 4th Annual Symposium on Theoretical Aspects of Computer Sciences (STACS '87). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 247, 1987, pp. 22–39, doi: 10.1007/BFb0039592.

[14] Klein, C.—McCarthy, J.—Jaconette, S.—Findler, R. B.: A Semantics for Context-Sensitive Reduction Semantics. In: Yang, H. (Ed.): Programming Languages and Systems (APLAS 2011). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 7078, 2011, pp. 369-383, doi: 10.1007/978-3-642-25318-8_27.

[15] Kock, J.: Notes on Polynomial Functors. Technical report, Universitat Autonoma de Barcelona, Spain, 2007.

[16] Kuśmierek, J.—Bono, V.: Big-Step Operational Semantics Revisited. Fundamenta Informatica, Vol. 103, 2010, No. 1-4, pp. 137–172, doi: 10.3233/FI-2010-323.

[17] Kryvolap, A.—Nikitchenko, M.—Schreiner, W.: Extending Floyd-Hoare Logic for Partial Pre- and Postconditions. In: Ermolayev, V., Mayr, H. C., Nikitchenko, M., Spivakovsky, A., Zholtkevych, G. (Eds.): Information and Communication Technologies in Education, Research, and Industrial Applications (ICTERI

2013). Springer, Cham, Communications in Computer and Information Science, Vol. 412, 2013, pp. 355–378, doi: 10.1007/978-3-319-03998-5_18.

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

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

[20] Mosses, P.: Action Semantics. Technical report, Cambridge Tracts in Theoretical Computer Science, 1992, doi: 10.1017/CBO9780511569869.

[21] Nielson, H. R.—Nielson, F.: Semantics with Applications: An Appetizer. Undergraduate Topics in Computer Science, Springer, 2007.

[22] Novitzká, V.: Logical Reasoning about Programming of Mathematical Machines. Acta Electrotechnica et Informatica, Vol. 5, 2005, No. 3, pp. 50–55.

[23] Novitzká, V.—Mihályi, D.—Slodičák, V.: Linear Logical Reasoning on Programming. Acta Electrotechnica et Informatica, Vol. 6, 2006, No. 3, pp. 34–39.

[24] Plotkin, G.: A Structural Approach to Operational Semantics. Technical report, Computer Science Department, Aarhus University, 1981.

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

[26] Roşu, G.: CS322 Fall 2003: Programming Language Design. Lecture Notes. Technical Report, UIUCDCS-R-2003-2897, Department of Computer Science, University of Illinois at Urbana-Champaign, Lecture Notes of a Course Taught at UIUC, December 2003.

[27] Roşu, G.—Şerbănuţă, T. F.: An Overview of the K Semantic Framework. The Journal of Logic and Algebraic Programming, Vol. 79, 2010, No. 6, pp. 397–434, doi: 10.1016/j.jlap.2010.03.012.

[28] Roşu, G.: K – A Semantic Framework for Programming Languages and Formal Analysis Tools. In: Pretschner, A., Peled, D., Hutzelmann, T. (Eds.): Dependable Software Systems Engineering. IOS Press, NATO Science for Peace and Security Series D: Information and Communication Security, Vol. 50, 2017, pp. 186–206, doi: 10.3233/978-1-61499-810-5-186.

[29] Rutten, J. J. M. M.: Universal Coalgebra: A Theory of Systems. Theoretical Computer Science, Vol. 249, 2000, No. 1, pp. 3–80, doi: 10.1016/S0304-3975(00)00056-6.

[30] Schmidt, D.: Denotational Semantics: A Methodology for Language Development. William C. Brown Publishers, Dubuque, IA, USA, 1986.

[31] Schmidt, D.: The Structure of Typed Programming Languages, MIT Press, Cambridge, MA, USA, 1994.

[32] Schmidt, D.: Abstract Interpretation of Small-Step Semantics. In: Dam, M. (Ed.): Analysis and Verification of Multiple-Agent Languages (LOMAPS 1996). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 1192, 1996, pp. 76–99, doi: 10.1007/3-540-62503-8_4.

[33] Schreiner, W.: Computer-Assisted Program Reasoning Based on a Relational Semantics of Programs. In: Quaresma, P., Back, R.-J. (Eds.): THedu'11, Wrocław,

Poland, July 31, 2011. Electronic Proceedings in Theoretical Computer Science (EPTCS), Vol. 79, 2012, pp. 124–142, doi: 10.4204/EPTCS.79.8.

[34] Steingartner, W.—Novitzká, V.—Bačíková, M.—Korečko, Š.: New Approach to Categorical Semantics for Procedural Languages. Computing and Informatics, Vol. 36, 2017, No. 6, pp. 1385–1415, doi: 10.4149/cai_2017_6_1385.

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

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

[37] Szymoniak, S.—Siedlecka-Lamch, O.—Kurkowski, M.: SAT-Based Verification of NSPK Protocol Including Delays in the Network. 2017 IEEE 14$^{\text{th}}$ International Scientific Conference on Informatics (Informatics 2017), IEEE, 2017, pp. 388–393, doi: 10.1109/INFORMATICS.2017.8327280.

[38] Turi, D.—Plotkin, G.: Towards a Mathematical Operational Semantics. Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science, 1997, pp. 280–291, doi: 10.1109/LICS.1997.614955.

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

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

Wolfgang Schreiner is Associate Professor at the Research Institute for Symbolic Computation (RISC) of the Johannes Kepler University Linz, Austria. His research fields include formal methods in computer science, formal semantics of programming languages, and parallel and distributed computing. He is the main developer of the formal modeling and verification systems RISCAL, RISC ProgramExplorer, and RISC ProofNavigator.