Semantic methods in our research and teaching

William Steingartner, Valerie Novitzká
Technical University of Košice
Faculty of Electrical Engineering and Informatics
Letná 1/9, 04200 Košice, Slovakia
{william.steingartner, valerie.novitzka}@tuke.sk

Abstract. *The formal semantics of programming languages forms the basis for many other formal methods used in software engineering. Its role is primarily in finding the meanings of programs and language constructs and in verifying the correctness of the design and implementation by means of formal procedures. Formal semantics has an irreplaceable place in the curricula of computer science and related fields of most universities. Its study requires a certain knowledge of mathematics and formal principles, therefore one of the results of our research is the visualization of semantic methods, primarily for educational purposes. In this article, we present an overview of our results in the field of definition and extension of semantic methods and their visualization.*

Keywords. correctness verification, curricula, formal methods, programming languages, semantic methods, software engineering, visualization, university didactic.

1 Introduction

Semantic methods belong to the frequently used formal methods for determining the meaning of programs written in programming languages. Individual methods are formulated to serve some specific purposes.

The first semantic method is denotational semantics, which defines the meaning of programs in terms of sets and functions. This method is not interesting in the process of program execution; it provides only the results with respect to the given input data. Therefore, this method is appropriate in designing programming languages. The second, very popular method is operational semantics. We differentiate two operational semantic methods. Natural semantics, which provides the meaning of the whole statements, and structural operational semantics, which analyzes every detail in program execution, follows the observable behavior of programs. The important part of structural operational semantics is abstract implementation on an abstract machine. Structural operational semantics is useful in the process of program implementation. There exist other semantic methods, for instance, axiomatic semantics, which adds preconditions and postconditions to each statement. These formulas serve to prove partial correctness of programs. Algebraic semantics specifies programs as abstract data types and models them as heterogeneous algebras. All these methods are included in the course Semantics of Programming Languages for graduate students.

In the following text, we present the related research and resources and we sketch the structure of the paper. Categories are very useful mathematical structures for modeling the programs and defining their categorical semantics. Categorical semantics of programs is in the literature oriented mostly on functional programming paradigm, for instance, (Hyland and Ong, 2000), but our approach is for the imperative paradigm modeled by category of states. We defined denotational categorical semantics for imperative languages and we extended it also for procedures (Steingartner et al., 2017). Another categorical structure we defined for structural operational semantics is coalgebra (Steingartner et al., 2020). We followed the ideas published in the book (Jacobs, 2016). The results of both models we explained in Section 2 (denotational semantics) and in Section 3 (operational semantics).

Because graphical illustration of semantics is easy to understand for students, our further work was to prepare visualization software appropriate for the teaching process. We were inspired by the work (Hannan and Miller, 1990) and we used the tool ANTLR 4 (Parr, 2013) as a very strong and useful software for our purposes. Our results (Steingartner et al., 2019; Steingartner, 2020; Steingartner and Sivý, 2023) are shortly characterized in Section 4.

Our results in semantic methods we applied in other languages as concatenative/compositional languages and domain-specific languages. Using (Herzberg and Reichert, 2009) we defined semantics for a simple compositional language KKJ (published in (Mihelič et al., 2021)). During the last decades, a new kind of programming languages, domain-specific languages were developed. We included in our course a simple domain-specific language Robot (Horpácsi and Kőszegi, 2015) and we showed how the semantics of such languages can be defined (Steingartner and Novitzká, 2021; Steingartner et al., 2022). Our results we sum up in Section 5. Section 6 concludes our paper.

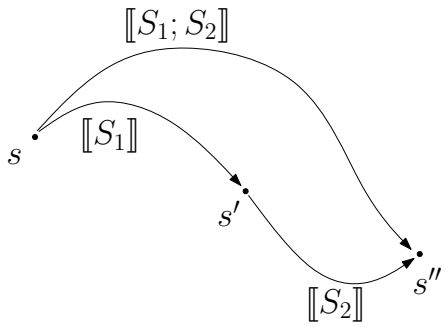

Figure 1: A compound morphism for a sequence of commands

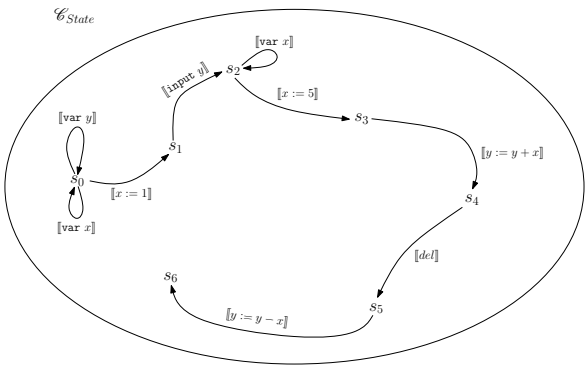

Figure 2: The path of execution P_1

2 Categorical semantics

A very useful mathematical structure for modeling programs and defining their semantics is a category. Therefore, our research focused on the use of categories for defining semantics.

We worked with imperative and procedural languages, where the essential concept is a state. The state is considered as an abstract model of computer memory which is independent from the real architecture and allows to view the changes in computer memory in a formal way. We define a state s as a function

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

where $\mathbf{Var}$ is a set of variables, $\mathbf{Level}$ expresses nesting level and $\mathbf{Value}$ is the set of integers together with undefined value $\perp$. Each state is then a list of triples, where x_i are variables, l is the nesting level and v is a value of a variable of the given level:

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

States are the objects of our model, the category of states. Each variable of a program needs to be declared. A declaration `var` x is modeled as an endomorfism on a given state s and it adds a new triple to a given state:

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

The statements are modeled as category morphisms – for a statement S, we have

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

It expresses a change of state induced by execution of S. In such way, we defined denotational semantics (Steingartner et al., 2017), where we constructed a category of states and defined commands of a simple imperative language using morphisms.

For illustration, in Fig. 1, we show the semantics of compound statement $S_1; S_2$. We also consider block structure of our language. A block can contain declarations and statements. Here, the nesting level is incremented within entry to a block and the locally declared variables are forgotten at the end of a block:

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

where the keywords `begin` and `end` serve as fictive variables and del is a function releasing all declaration on the level $l + 1$, for $l \in \mathbf{N}_0$.

The semantics of a program is then a path in the category of states, as it is illustrated in Fig. 2 (for the Listing 1).

Listing 1: Program P_1 with nested block

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

We extended the simple procedural language with procedures to the procedural language and constructed a collection of state categories that are linked using functors for calling the procedure and returning to the calling subroutine. The advantage of our approach is the possibility of multiple procedure calls and the use of recursive calls (Figure 3).

Categories allow a graphical representation of program execution, which inspired us to introduce graphical visualization of semantics into the teaching process. Categorical structures are quite difficult for students studying technical sciences because they require quite deep knowledge of mathematics, so we tried to visualize the definition of the traditional approach to defining and visualizing semantics.

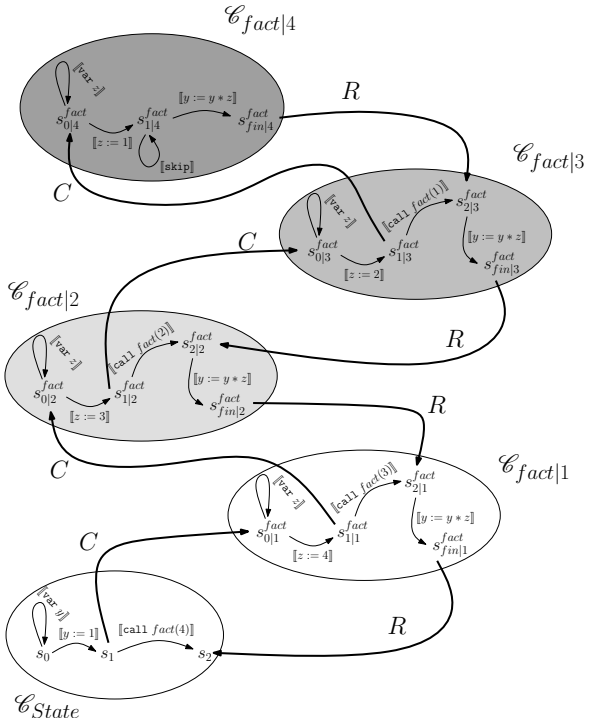

Figure 3: Categorical interpretation of recursive calls

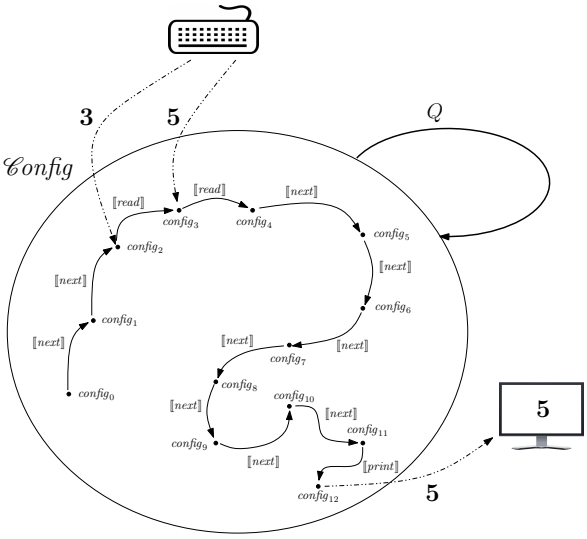

Figure 4: Semantics of program P_2 in coalgebra

3 Categorical operational semantics

The second semantic method, which we modeled with the help of categories, was structural operational semantics, which provides an overview of the detailed steps of the executed program, that is, it describes the behavior of the program. We extended the simple programming language with input and output blocks and commands, which required the construction of another categorical structure, a category of configurations whose objects are lists of commands, states, inputs and outputs (Steingartner et al., 2020). The commands of the language are modeled by morphisms (Figure 4).

Operational semantics provides behavior of programs, that is, it considers every detail in program execution. The best categorical structure, which is appropriate for this purpose, is a coalgebra. Coalgebra is defined as a polynomial endofunctor over state space, here over a base category. Every application of this functor expresses one step of program execution. To construct coalgebra for our purposes, we need to construct base category as a category of configurations. A configuration is a tuple

$$config = (\llbracket D^*; S^* \rrbracket, m, i^*, o^*),$$

where m is actual memory, i^* and o^* are lists of input and output values, respectively. The set **Config** of configuration is our state space. The morphisms of the base category are the transition operations $\llbracket next \rrbracket$ for statements, $\llbracket read \rrbracket$ and $\llbracket print \rrbracket$ for input and output statements, resp. Now we have the base category that is a model of our simple language *Jane*; thus we can proceed to construct a coalgebra modeling the behavior of programs written in *Jane*. The objects of our category form the state space of a coalgebra and the morphisms are the transition mappings. Hence, we construct the polynomial endofunctor for this kind of systems

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

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.

In Fig. 4, we show the semantics of the following program (Listing 2):

Listing 2: Program P_2

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

4 Visualization of traditional semantic methods

The mentioned modules were solved within the KEGA educational projects and are gradually being introduced into the teaching process. The advantage is that they can serve as an illustrative aid for the teacher, either in lectures or exercises. In addition, they allow students to

individually practice the relevant issue and alert them to the mistakes they make when defining semantics. The modules work in interactive mode, that means that the student defines the semantics step by step, or enters a program and the module returns the semantics of the program. The response from students is very positive. This process can be considered as some pre-evaluation of our software. We observe how the students understand the principles of individual semantic methods faster and understand their principles more deeply than is the case with the classic definition of semantics on the blackboard or in a notebook.

4.1 Natural semantics

Natural semantics is one of the methods of operational semantics. This method is used quite often in practice when deriving the properties of languages and programs written in specific languages. It is also a useful method for designing compilers. Its principle is that the meaning of the program is constructed using derivation trees of natural semantics, where in individual nodes there are transition relations expressing changes in memory states (abstract expression of computer memory) as follows:

$$\langle S, s\rangle \rightarrow s'$$

where S represents a statement and s, s' stand for memory states.

The software module for the visualization of natural semantics (Figure 5, presented in (Steingartner et al., 2019)) works on the principle of translating the input program in the *Jane* language into tokens, with the help of which a derivation tree of natural semantics is subsequently generated. In the core of the application is a standard simple compiler working with the grammar of the *Jane* language. The compiler includes standard phases (lexical analysis, syntactic analysis, semantic analysis and target code generation – here in the form of tokens for derivation tree rendering). The module allows to input the program manually or load it from a file. Then the next step is setting the values of variables in the program. After compilation program performs the visualization of semantic evaluation applying the particular rules. The result provided (when the input program was correct) is then a derivation tree for natural semantics and the table with all states during the program execution. User can save the results for later use in graphical and textual form: program provides a cropped image containing the drawn derivation tree and LaTeX source for this tree.

4.2 Structural operational semantics

Structural operational semantics is a very popular semantic method that models each step of program execution based on transitive relations. Transition sessions can take place in two forms (depending on the result of the calculation) – either the result is a simple state or the next step (next configuration) in the computation:

$$\langle S, s\rangle \Rightarrow s', \quad \text{or} \quad \langle S, s\rangle \Rightarrow \langle S', s'\rangle.$$

For defining the grammar and parser construction, the ANTLR tool (Parr, 2013) with a great success was used. After providing the input sequence of statements and the initial state of variables, a parsing of the input code starts. Compilation provides errors if the input code is not correct. Otherwise, a parse tree is created. After the correct input, a new instance of *SOSGrammmarVisitor* class and the *MainSequence* class are created. So the first step is to obtain a sequence of statements and process them using the appropriate compilation (and semantic) rules. After obtaining the necessary objects, a derivation sequence is created based on the rules of structural operational semantics and the visualization is ready to be performed.

When a user launches the emulation (visualization) process, lexical and syntax analysis are performed. The result of a computation is provided and displayed as the whole computational sequence. User then can choose a mode of visualization: either the whole computation at once with final results or the step-wise emulation with showing all changes of states (and variables). At the end, user is allowed to store the work done: the whole output as a complete report (in one document), or only the state table, or the output computational sequence as a source code for LaTeX. Software module is depicted in Figure 6.

4.3 Abstract machine

Obviously, operational semantics is used to specify the meaning of programs and abstract machines to provide an intermediate representation of the language's implementation. Operational semantics can be presented as inference rules or, equivalently, as formulas in a weak meta-logic permitting quantification at first-order and second-order types. Abstract machines can be presented as rewrite rules describing single-step operations on the state of a computation. Such specifications provide an intermediate level of representation for many practical implementations of programming languages (Hannan and Miller, 1990).

For the complete visualization of particular processes of the abstract machine for the operational semantics, an emulator of abstract machine was developed (Steingartner and Sivý, 2023). Our visualization tool implementation fulfills several tasks. In its final form, it forms a tool accessible through a web browser, which is capable of processing various types of input, compilation (translation into tokens), subsequent visualization of program execution and exporting visualization results in various formats (Fig. 7). The implementation also includes a separate compiler executable via the command line, which is capable of output in various formats and is also capable of interpreting input

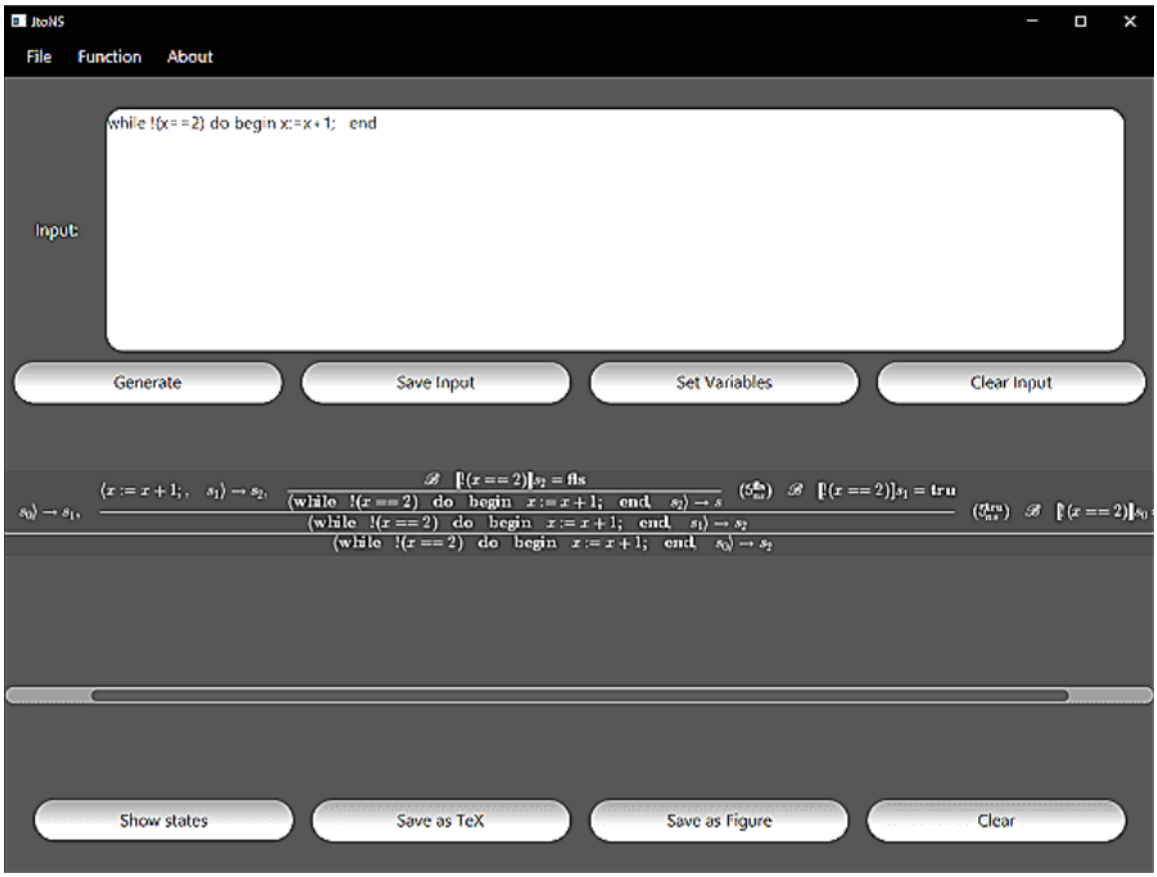

Figure 5: A look at the Natural Semantics Visualizer

programs. The entire tool is conceptually divided into three packages: the editor, which serves to process user input, the compiler, and the visualization tool itself.

4.4 Denotational semantics

The tool for denotational semantics is currently under development. The specificity of the design and development of the tool for denotational semantics results from the nature of the given method. Denotational, also called mathematical semantics, is a semantics that expresses the meaning of programs using mathematical objects (numbers, expressions and functions) and their composition. The main idea is that for a given program we construct a mathematical function that gives the same results as the given program (regardless of the order of individual steps in the program; of course, it is a function on states).

For simple statements and for the conditional statement pattern, the implementation is fairly straightforward. However, the challenge is the implementation of the denotation of the loop statement, for which in practice we construct a functional (a higher-order function) and then search for this functional's least fixed point, which expresses the meaning (denotation) of the loop statement.

It is obvious that for educational purposes automation of selected patterns of solving a fixed point of the functional appears to be a suitable solution, of course in interactive mode.

The development and methodology itself differ only little from the development of previous modules. Also in this case, thanks to the ANTLR tool, a grammar for input processing (*Jane* programs) is constructed. The compiler reads and translates the input program and decomposes it into a sequence of tokens, thanks to which it is possible to visualize calculations in denotational semantics. After completion, we plan to deploy the tool itself in the same way as the previously mentioned programs.

5 Extension of semantic methods for other languages

In addition to the application and extension of semantic methods in the area of the imperative paradigm, we further focused on the use of semantic methods also for languages from other paradigms. We focused on concatenative (or compositional) languages, which have the character of stack-based languages (Sec. 5.1). The very nature of domain-specific languages makes it possible to explore many of their interesting properties, including from the point of view of formal semantics, so we focused our efforts in research further in this direc-

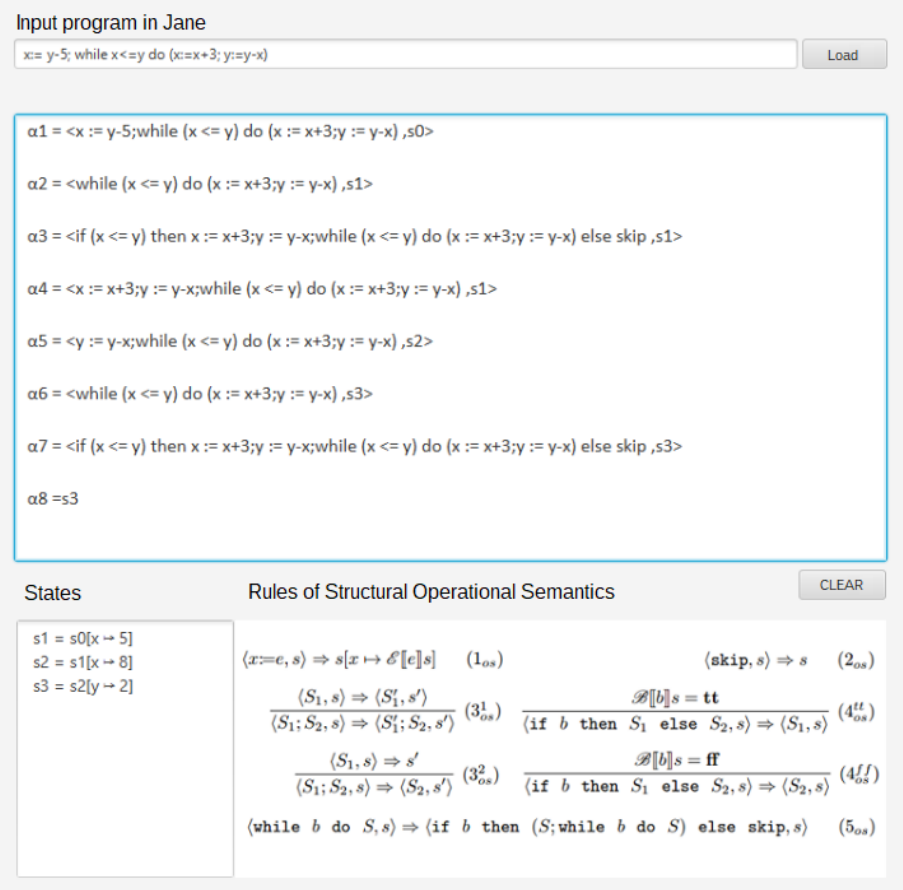

Figure 6: A module for the structural operational semantics

tion (Sec. 5.2).

5.1 Semantics of compositional languages

Another area we focused on was concatenative (stack-based) languages, which demonstrate to students that semantic methods are also suitable for other programming languages.

We have extended our research in the field of formal semantics of programming languages to the field of *concatenative languages*. This group of languages usually belongs to the functional paradigm (Herzberg and Reichert, 2009).

The name of this group of languages is derived from the property that the syntactic chaining (concatenation) of programs corresponds to the semantic composition of functions. The paradigm of concatenative languages is suitable for research into the fundamentals of software engineering and appears to be a suitable foundation for the future of programming.

We present a simple programming language KKJ, which contains a small set of language constructions, but offers a flexible and useful environment for programming. The development of the language and research related to its properties and semantics takes place in cooperation with the University of Ljubljana (Mihelič et al., 2021).

Thanks to its simplicity, the language provides several possibilities, for example, a clear definition of semantics, the possibility of direct development of an interpreter, the design of tools for analyzing programs for the purpose of optimization or verification, etc.

To describe the syntax of expressions $E \in \mathbf{Expr}$ in the programming language KKJ, we use the well-known BNF-notation:

$$E ::= \varepsilon \mid i \mid n \mid \{E\} \mid E\,E.$$

Here, ε stands for an empty expression, a numeral $i \in \mathbf{IntNum}$ and a name $n \in \mathbf{Name}$ are considered as expressions as well. As an example of calculations in this language, we show in Fig. 8 an evaluation of the program

$$[\![\texttt{14 \{dup dup\} \{add add\} compose apply}]\!]\, s$$

in particular steps.

The attentive reader will quickly realize that it is a calculation of the value of the expression 3×14.

5.2 Semantics of domain-specific languages

The latest contribution to the teaching of semantic methods is the definition of the semantics of domain-specific languages, which represent the modern trend

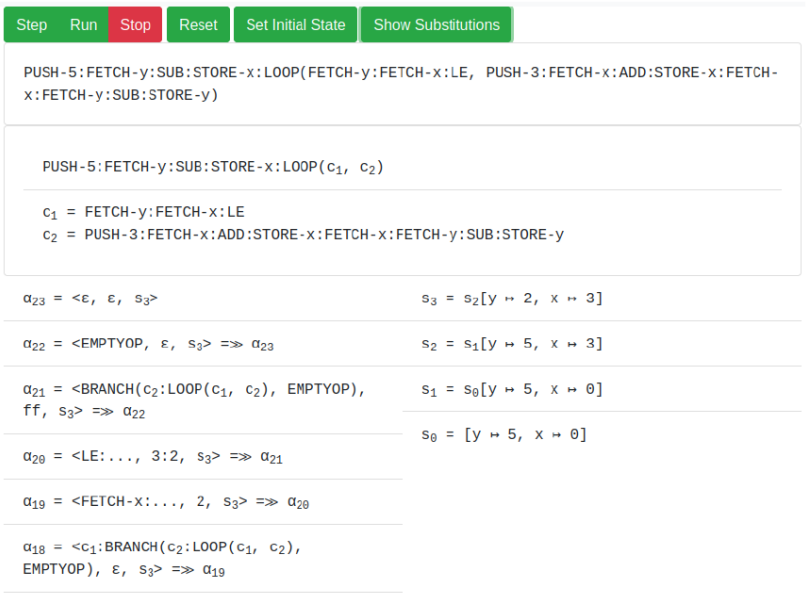

Figure 7: Window containing a visualization of abstract machine

$$
\begin{aligned}
&[14\ \{\text{dup dup}\}\ \{\text{add add}\}\ \text{compose apply}]\ s = \\
&\quad = [\text{apply}] \circ [\text{compose}] \circ [\{\text{add add}\}] \circ [\{\text{dup dup}\}] \circ [14]\ s \\
&\quad = [\text{apply}] \circ [\text{compose}] \circ \text{push}\ [\text{add add}] \circ \text{push}\ [\text{dup dup}] \circ [14]\ s \\
&\quad = [\text{apply}] \circ [\text{compose}]\ s{\cdot}14{\cdot}[\text{dup dup}]{\cdot}[\text{add add}] \\
&\quad = [\text{apply}]\ s{\cdot}14{\cdot}([\text{add add}] \circ [\text{dup dup}]) \\
&\quad = [\text{add add}] \circ [\text{dup dup}]\ s{\cdot}14 = [\text{add}] \circ [\text{add}] \circ [\text{dup}] \circ [\text{dup}]\ s{\cdot}14 \\
&\quad = [\text{add}] \circ [\text{add}]\ s{\cdot}14{\cdot}14{\cdot}14 = s{\cdot}42
\end{aligned}
$$

Figure 8: Step-wise evaluation of the calculation in KKJ language

in the development of programming languages. For teaching, we chose a simple domain-specific language for moving the robot along a rectangular grid (Figure 9).

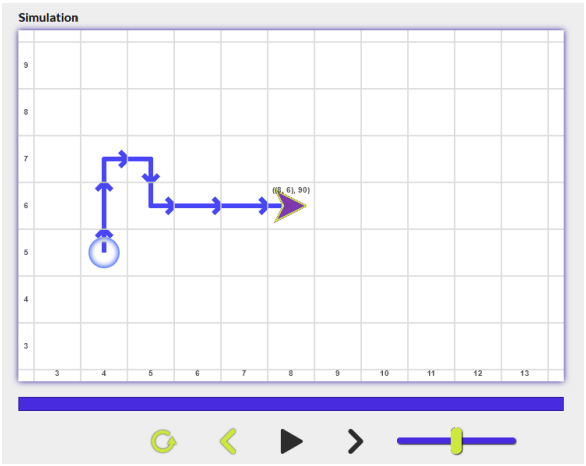

Figure 9: Visualization of robot movement in a grid

The language is quite easy itself. The structure of directions is defined by the production rule:

$$d ::= \textbf{left} \mid \textbf{right} \mid \textbf{up} \mid \textbf{down}$$

and we define the structure of statements as follows:

$$S ::= d \mid d\ n \mid \textbf{reset} \mid \textbf{skip} \mid S; S,$$

where

- $D\ n$ expresses n-steps of movement in given direction;
- **reset** defines movement to starting position;
- **skip** is the empty statement;
- $S; S$ is sequence of statements.

The second version of this language, which is closer to real languages, has the following syntax.

$$\begin{aligned} S ::= \quad & \textbf{turn left} \mid \textbf{turn right} \mid \textbf{forward} \\ & \textbf{forward}\ n \mid \textbf{reset} \mid \textbf{skip} \mid S; S. \end{aligned}$$

We explored and refined the definition of its denotational and structural operational semantics (formulated in (Horpácsi and Kőszegi, 2015)) and defined natural

semantics (Steingartner and Novitzká, 2021) and the first version of abstract machine for structural operational semantics (Steingartner et al., 2022) and thereby again showed students that semantic methods are applicable to such languages. An example of computation on an abstract machine is in Figure 10.

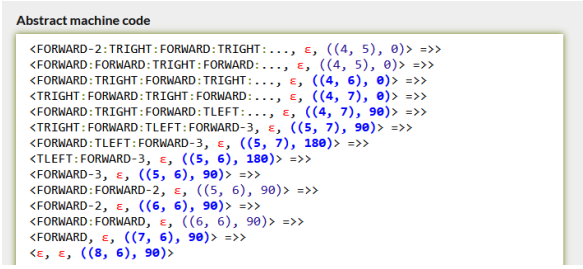

Figure 10: Example of computation on an abstract machine

Currently, we are extending this robot language with additional commands that allow to bypass the obstacle in the path of the robot and determine the size of the grid. Adding this problem to the language requires defining new commands (conditional, loop) so that the robot moves on a grid of a fixed size and knows how to avoid obstacles. A possible promising extension for the future is the addition of a third dimension in an orthogonal grid, allowing for semantic modeling of drone movement in space.

6 Conclusion

In this article, we presented the most important results of our research in the field of semantic methods. In our research, we extended the set of traditional semantic methods for imperative languages by two methods based on category theory, which were very positively received by students. Despite their mathematical nature, the categories are also positively received by students with a technical focus (with less mathematical background) due to their illustrative and expressive power. In our research, we further dealt with the extension of traditional methods for some other paradigms of languages, specifically for selected domain-specific languages and for concatenative (stack-oriented) languages. As a big positive, students receive software support for visualization and interactive processing of calculations using selected semantic methods. Hereby, we have presented a comprehensive view of research in the field of semantic methods and semantic modeling, as well as making them available to students in an interesting and interactive form.

Acknowledgments

This work was supported in the frame of the initiative project “Semantics-Based Rapid Prototyping of Domain-Specific Languages” under the bilateral program “Aktion Österreich – Slowakei, Wissenschafts- und Erziehungskooperation” granted by the Slovak Academic Information Agency and by the project KEGA 030TUKE-4/2023 – “Application of new principles in the education of IT specialists in the field of formal languages and compilers”, granted by the Cultural and Education Grant Agency of the Slovak Ministry of Education.

References

Hannan, J. and Miller, D. (1990). From operational semantics to abstract machines: Preliminary results. In Kahn, G., editor, *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*, pages 323–332. ACM.

Herzberg, D. and Reichert, T. (2009). Concatenative programming - an overlooked paradigm in functional programming. In *Proceedings of the 4th International Conference on Software and Data Technologies - Volume 1: ICSOFT,*, pages 257–262. INSTICC, SciTePress.

Horpácsi, D. and Kőszegi, J. (2015). Formal semantics. `https://dtk.tankonyvtar.hu/xmlui/handle/123456789/3736`. Accessed: Dec 14th, 2022.

Hyland, J. and Ong, C.-H. (2000). On Full Abstraction for PCF: I, II, and III. *Information and Computation*, 163(2):285–408.

Jacobs, B. (2016). *Introduction to Coalgebra: Towards Mathematics of States and Observation*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

Mihelič, J., Steingartner, W., and Novitzká, V. (2021). A denotational semantics of a concatenative/compositional programming language. *Acta Polytechnica Hungarica*, 18(4).

Parr, T. (2013). *The Definitive ANTLR 4 Reference*. Pragmatic Programmers, LLC, The, Raleigh.

Steingartner, W. (2020). Support for online teaching of the Semantics of Programming Languages course using interactive software tools. In *2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, pages 665–671.

Steingartner, W., Baraník, R., and Novitzká, V. (2022). Abstract machine for operational semantics of domain-specific language. In Chiusano, S.,

Cerquitelli, T., Wrembel, R., Nørvåg, K., Catania, B., Vargas-Solar, G., and Zumpano, E., editors, *New Trends in Database and Information Systems*, pages 413–424, Cham. Springer International Publishing.

Steingartner, W., Haratim, M., and Dostál, J. (2019). Software visualization of natural semantics of imperative languages - a teaching tool. In *2019 IEEE 15th International Scientific Conference on Informatics*, pages 509–514.

Steingartner, W. and Novitzká, V. (2021). Natural semantics for domain-specific language. In Bellatreche, L., Dumas, M., Karras, P., Matulevičius, R., Awad, A., Weidlich, M., Ivanović, M., and Hartig, O., editors, *New Trends in Database and Information Systems*, pages 181–192, Cham. Springer International Publishing.

Steingartner, W., Novitzká, V., Bačíková, M., and Š. Korečko (2017). New approach to categorical semantics for procedural languages. *Computing and Informatics*, 36(6).

Steingartner, W., Novitzká, V., and Schreiner, W. (2020). Coalgebraic operational semantics for an imperative language. *Computing and Informatics*, 38(5):1181–1209.

Steingartner, W. and Sivý, I. (2023). From high-level language to abstract machine code: An interactive compiler and emulation tool for teaching structural operational semantics. In *New Trends in Database and Information Systems: ADBIS 2023 Short Papers, workshop MADEISD*. forthcomming.