
A Categorical Model of State Transformations in Programming Languages

M. KOHÚTOVÁ, W. STEINGARTNER AND S. NOVOTNÝ

Abstract

Formal semantics of programming languages represents a basic formal method connecting theoretical foundations of computer science with the development of reliable software. It allows for rigorous analysis of the correctness and properties of programs independently of the implementation. When formulating it, it is necessary to respect the principles of the programming paradigm, which determines the basic abstractions of the language. At the same time, there are unifying formalisms that allow for systematic comparison of semantic definitions across paradigms. Category theory represents one of the most important such approaches. We formalize this perspective by introducing a general categorical model of state transformations and show how different programming paradigms can be uniformly represented and related within this framework via structure-preserving mappings.

Mathematics Subject Classification 2000: Primary 68Q55; Secondary 68N18, 68N30, 18C50

General Terms: Category Theory, Denotational Semantics, Formal Semantics, State-Based Models, Domain-Specific Languages, Imperative Languages, Stack-Based Languages

1. INTRODUCTION

The diversity of programming languages reflects the many ways in which people think about computation, state and their change, as discussed in modern treatments of programming language theory [Pierce 2002; Harper 2016; Milewski 2021]. Imperative languages model computation as sequences of commands that explicitly modify state, whereas declarative approaches describe properties of desired results, leaving the underlying state transformations implicit.

From one important perspective, many computational models can be understood in terms of state and its transformation [Baez et al. 2023]. However, the notion of state varies significantly: it may appear as a store of variables, a stack, a tree structure, a system of equations, or even spatial configurations such as positions in a geometric space. These differences reflect distinct abstractions rather than fundamentally different computational principles.

Interestingly, classical models such as the Random Access Machine (RAM) [Cook and Reckhow 1973] illustrate this point. Although typically viewed as abstract machines, they can also be interpreted as formal systems for describing state

10.2478/jamsi-2026-0004

©M. Kohútová, W. Steingartner and S. Novotný

This is an open access article licensed under the Creative Commons Attribution License (<http://creativecommons.org/licenses/by/4.0>).

transformations with explicit access to memory. Similarly, stack-based machines or algebraic formalisms can be seen as alternative representations of computational state.

Despite their surface differences, these models share a common structural aspect: they describe collections of states together with transformations between them. This observation naturally leads to more abstract formulations of computation, as studied in semantics and category theory [Barr and Wells 1990].

Category theory provides a unifying framework in which states (or types of states) can be represented as objects, and computations as morphisms between them [Riley et al. 2021]. In this setting, programs correspond to morphisms, while entire computational systems give rise to categories. Such formulations make it possible not only to describe state transformations, but also to relate different computational models via functors, as explored in [Spivak 2014].

The aim of this paper is to develop a conceptual framework in which diverse programming languages — ranging from low-level machine models to declarative domain-specific languages — can be understood within a unified categorical perspective on state. This approach suggests a pathway toward relating and potentially integrating different programming paradigms within a common abstract structure [Zhao et al. 2023].

The structure of the paper is as follows: Section 2 introduces formal models of state and surveys different semantic approaches, including operational, denotational, and categorical perspectives. Section 3 presents concrete examples of state-based categories derived from simple programming languages. In Section 4, we generalize these constructions and discuss their interpretation as categories associated with programming languages and formulate foundations of a unified framework for modeling state transformations in different programming languages. Section 5 reviews related work in categorical semantics and its recent developments. Finally, Section 6 concludes the paper and outlines directions for future research.

2. FORMAL MODELS OF STATE AND SEMANTICS

Having established the formal representation of program state across different paradigms, we now turn to the question of *semantics*—how these states acquire meaning through computation. Each programming language not only defines a space of possible states, but also a set of rules or transformations that give those states behavioral interpretation. To capture these transformations precisely, several

complementary semantic frameworks have been developed, ranging from operational semantics [Plotkin 1981] to denotational approaches [Schmidt 1986] and categorical models [Steingartner and Novitzká 2017].

2.1. Operational, Denotational, and Functorial Semantics

In programming language theory, formal semantics is commonly described along three well-known complementary perspectives: a transitional (operational) view concerned with state evolution, a mathematical (denotational) view assigning meanings (denotations) to programs, and a logical (axiomatic) view capturing correctness properties.

Operational semantics (in its small-step and big-step variants) characterizes computation via transitions between program states [Winskel 1993], while denotational semantics assigns mathematical meanings to programs, typically as functions over suitable domains [Stoy 1977; Scott 1970; Schmidt 1986]. Axiomatic semantics, in turn, expresses correctness in terms of relations between preconditions and postconditions [Hoare 1969].

Category theory provides a unifying abstraction for semantic approaches by modeling states as objects and computations as morphisms [Jacobs 1999]. In this setting, functorial semantics, following Lawvere [Lawvere 1963], interprets programs and languages as categorical structures and their relationships as functors preserving the computational structure.

This work is situated within a broader research program that progressively develops a functorial approach to semantics. This approach is motivated by the observation that functorial semantics provides a formal language for expressing when different languages admit a common *space of states*. In particular, the existence of suitable structure-preserving functors between their associated categories may be interpreted as indicating a form of semantic compatibility at the level of state representation.

2.2. State as Structure

Traditionally, state is viewed as a collection of variable values or a configuration of memory, as formalized for instance in denotational models of imperative languages [Schmidt 1986]. In the categorical setting, however, state is modeled abstractly as an object in a category, where its behavior is determined by morphisms representing state transformations. This interpretation allows us to generalize the notion of state beyond conventional models:

—In imperative languages, states are memory configurations, and morphisms

correspond to commands.

- In functional languages, computations are modeled as functions between values, with state either implicit or represented through suitable abstractions.
- In concatenative languages or stack-based languages, states are stacks, and morphisms are stack operations (see, e.g., KKJ in [Mihelič et al. 2021] or StackLang in [Steingartner and Schreiner 2025]).
- In RAM model and abstract machines, states are configurations of registers and memory, and morphisms correspond to instructions.
- In domain-specific languages (DSLs), states are elements of domain-specific structures (e.g., geometric positions, logical facts, or table configurations).

The unifying observation is that, in each case, a language determines a collection of states together with operations that transform them, a perspective closely aligned with categorical approaches to programming languages [Steingartner et al. 2017]. Formally, such systems can be modeled as categories, where states correspond to objects and transformations to morphisms. This viewpoint provides a foundation for a functorial approach in which different languages can be related through structure-preserving mappings, with the notion of state serving as a common abstraction across programming paradigms.

To serve as a common foundation across programming languages, the notion of state must be expressed independently of specific implementations or syntax, in a sufficiently abstract form. Category theory provides such a framework [Mac Lane 1998].

Category theory models systems in terms of objects and morphisms, emphasizing transformations between them rather than internal structure [Awodey 2010], a perspective well suited to describing computation as state transformation.

2.3. State as Object, Computation as Morphism

Categorical structures provide an abstract and unified framework for modeling programs, as developed in categorical semantics of computation [Barr and Wells 1990; Steingartner et al. 2017]. A programming language (or computational model) can then be described categorically as follows:

- a set of **objects** $\text{Ob}(\mathcal{L})$ representing possible *system states*,
- a set of **morphisms** $\text{Mor}(\mathcal{L})$ representing *computations or transitions* between states,

- a **composition** operation $\circ$ corresponding to the sequencing of computations, and
- an **identity morphism** id_s for each state s , representing a computation that leaves the state unchanged.

Formally,

$$\mathcal{L} = (\text{Ob}(\mathcal{L}), \text{Mor}(\mathcal{L}), \circ, id),$$

satisfying the usual axioms of associativity and identity [Mac Lane 1998].

This model allows us to view computation as a composition of morphisms, with states represented as the objects and computations as morphisms between them. Imperative, functional, and declarative languages can all be modeled within this same categorical framework, differing primarily in how their objects and morphisms are interpreted.

2.4. Formalization of State

In many semantic frameworks, state can be viewed as a schematic representation of a program's semantics: it captures one aspect of program execution. In operational models, state evolves during execution, forming a sequence of configurations that describes how computation unfolds. To make the notion of *state* explicit and comparable across different programming languages, we introduce a simple formal framework that captures the essential structure of computation. Let s be a state, an element of the semantic domain of states, $s \in \text{State}$. A computation step or program command can then be expressed as a function

$$f : \text{State} \rightarrow \text{State},$$

in a deterministic setting, where $f(s) = s'$ represents a transition from an initial state s to a resulting state s' . This definition abstracts away from any particular representation of state and focuses solely on the transformation relation between states.

2.5. A Categorical View of State Across Programming Paradigms

Building on the previous formalization of state and computation, we now move toward a unified categorical perspective that relates different programming paradigms. Different programming languages give rise to categories of states that share a common structural pattern. This suggests the existence of a unifying semantic domain capturing state transformations independently of specific language constructs. Within this framework, operations from language-specific categories can

be interpreted as morphisms and related via structure-preserving mappings, that is, functors, into a common category of state transformations. This perspective is illustrated in the following section through two concrete examples: a robot-control DSL and a stack-based language. The robot-control DSL [Steingartner et al. 2024] represents states as positions in an integer grid and primitive commands as movement transformations, while StackLang [Steingartner and Schreiner 2025] represents states as stack configurations and primitive operations such as push and pop as stack transformations.

Different paradigms instantiate this scheme in different ways: imperative languages use memory stores as states, stack-based languages use stack configurations, and domain-specific languages may use e.g. geometric positions or other structured configurations. Despite these differences, each induces a compositional system of state transformations. This motivates a functorial perspective in which language-specific categories are related by structure-preserving mappings into a shared category of state transformations, denoted $\mathcal{C}_{State}$.

For example, in a robot-control DSL, a state may be a coordinate $(x, y) \in \mathbb{Z}^2$, and commands such as `moveRight` or `moveUp` define morphisms between coordinates.

The following diagram (see Figure 1) illustrates this idea by showing how morphisms from various language categories can be mapped into a common category of state transformations, $\mathcal{C}_{State}$, through functorial relationships.

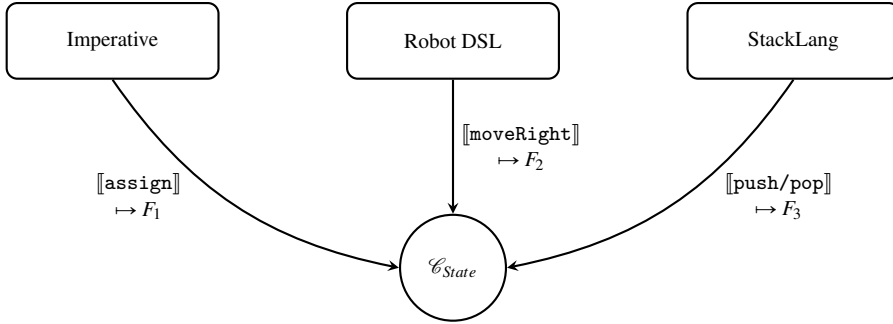


Fig. 1. Language-specific operations interpreted as morphisms and mapped via functors into a shared category of state transformations $\mathcal{C}_{State}$.

Figure 1 illustrates how operations from different languages are mapped into a common category of state transformations, denoted $\mathcal{C}_{State}$. The arrows represent functorial mappings that preserve the compositional structure of computations. Thus,

although the languages differ syntactically and operationally, their computations can be compared at the level of state transformations.

Building on this idea, we can now generalize the concept of state transformation into a single categorical framework. In such a setting, each individual computation $f : s \rightarrow s'$ is formalized as a *morphism*, and each possible configuration as an *object*. This abstraction allows different programming paradigms to be expressed within one coherent mathematical model of computation.

3. EXAMPLES OF STATE CATEGORIES

We now examine the robot-control DSL and StackLang in more detail as concrete instances of state categories. In both cases, objects represent states and morphisms represent state transformations.

3.1. A Robot DSL as a Category of States

Let $\mathcal{C}_{\text{Robot}}$ be a category whose objects are positions in the integer grid,

$$\text{Ob}(\mathcal{C}_{\text{Robot}}) = \mathbb{Z} \times \mathbb{Z}.$$

Morphisms are generated by primitive movement commands: r (right), l (left), u (up), and d (down), where for each $(x, y) \in \mathbb{Z} \times \mathbb{Z}$,

$$r_{(x,y)} : (x, y) \rightarrow (x + 1, y),$$

$$l_{(x,y)} : (x, y) \rightarrow (x - 1, y),$$

$$u_{(x,y)} : (x, y) \rightarrow (x, y + 1),$$

$$d_{(x,y)} : (x, y) \rightarrow (x, y - 1).$$

For each state (x, y) , the identity morphism $id_{(x,y)}$ represents an identity transformation (no-op transformation), leaving the position unchanged. In imperative languages, such behavior is typically captured by the `skip` statement. Composition corresponds to sequential execution of commands, and identity morphism represents a no-op transformation. Thus, robot programs are represented as paths in a category of grid states.

3.2. A Stack-Based Language as a Category of States

Let $\mathcal{C}_{\text{StackLang}}$ be a category whose objects are stack configurations $s = [a_1, \dots, a_n]$. Morphisms correspond to stack operations such as `push` and `pop`, each inducing a transition $f : s \rightarrow s'$ between configurations. Composition again corresponds to sequential execution, and identity morphisms represent no-op transformations. An

example of the execution trace, a directed path in a category for computation in StackLang, is presented in Figure 2.

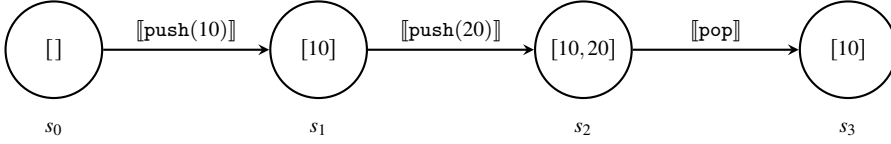


Fig. 2. A local fragment of the category $\mathcal{C}_{\text{StackLang}}$. Objects represent stack configurations, while morphisms correspond to stack operations interpreted as state transformations.

Although the robot and stack examples differ in the structure of their states, both induce categories in which computation is represented by morphisms between states. This common structure motivates the generalization developed in the next section, where language-specific state categories are related within a unified categorical framework.

4. A GENERAL CATEGORY OF STATE TRANSFORMATIONS

In this section, we generalize the categorical view of state introduced in the previous examples and formulate the foundations of a unified framework for modeling state transformations in different programming languages.

4.1. Generalization of State Categories

The robot DSL and stack-based language introduced in the previous sections serve as concrete examples of how programming languages can be expressed in categorical terms. In each case, a language produces a space of states together with operations that transform these states. These examples suggest that the categorical structure observed is not specific to particular languages, but reflects a more general pattern underlying state-based computation.

More generally, many programming languages can be described by categories whose objects represent system states and whose morphisms capture state transformations [Novitzká et al. 2019]. Let L denote a programming language, and let $\mathcal{C}_L$ be the associated category defined by:

- objects corresponding to possible states (or their abstractions) of the system,
- morphisms corresponding to elementary program operations,
- identity morphisms representing no-op transformations, and

—composition corresponding to sequential execution of operations.

Under this interpretation, programs correspond to compositions of morphisms in the category $\mathcal{C}_L$. From a graphical perspective, a category can be viewed as a directed graph, and a concrete program corresponds to a path in this graph. Different programming languages thus give rise to different categories of states, depending on the structure of their state spaces and available transformations.

This perspective enables the comparison of programming languages at an abstract structural level. It also motivates the study of how language-specific categories can be systematically related within a unified categorical framework.

4.2. A General Category of State Transformations

To capture this common structure explicitly, we introduce a general category of state transformations that abstracts from individual programming languages. From a categorical viewpoint, this leads naturally to the notion of a general category of state transformations.

Let S be a set of states. The *category of state transformations*, denoted $\mathcal{C}_{State}$, is defined by:

- objects given by elements $s \in S$,
- morphisms $f : s \rightarrow s'$ representing state transformations,
- composition corresponding to sequential execution of transformations,
- identity morphisms $id_s : s \rightarrow s$ representing no-op transformations.

This construction abstracts from any particular programming language and language features, and captures the essential structure of computation as transformations between states. Programs can then be understood as compositions of morphisms in $\mathcal{C}_{State}$. This category can be seen as a canonical semantic domain of state-based computation. Within this framework, categories associated with individual programming languages can be viewed as instances of this general structure.

From this perspective, a programming language induces a category generated by its primitive operations: elementary instructions correspond to generating morphisms, and programs correspond to their compositions. Equivalently, such a category can be viewed as the free category generated by a graph whose vertices are states and edges are primitive transitions [Mac Lane 1998].

This framework provides a uniform basis for comparing and relating

language-specific models. In particular, it motivates the use of functors as structure-preserving mappings between categories of states, which we explore in the following sections.

5. RELATED WORKS

Categorical approaches to semantics have a well-established foundation. Lawvere's notion of functorial semantics introduced the idea that syntactic structures can be interpreted via structure-preserving functors into semantic categories [Lawvere 1963]. This perspective was further developed in programming language semantics, where Tennent employed functor-category constructions to provide uniform denotational models [Tennent 1986], and Moggi introduced monads as a categorical framework for modeling computational effects [Moggi 1989].

Subsequent work extended categorical semantics toward richer accounts of syntax and computation. Fiore, Plotkin, and Turi formulated a categorical treatment of abstract syntax with variable binding based on initial semantics [Fiore et al. 1999]. Related developments have explored categorical structures for modeling programming languages and computational effects in a more compositional and modular way, particularly in the context of algebraic effects and handlers [Plotkin and Pretnar 2013]. In a related direction, Diskin and Kadish applied category-theoretic techniques to bridge abstract syntax and formal semantics in graph-based modeling frameworks [Diskin and Kadish 2003].

More recent research has reinforced the importance of compositionality and functorial interpretation across a broader range of semantic and computational settings. For example, compositional modeling based on decorated cospans and functorial semantics has been used to relate structured system descriptions to their dynamical interpretations, showing how categorical methods support modular semantic specification beyond traditional programming-language settings [Baez et al. 2023]. Similarly, category-theoretic approaches have been used in model-based systems engineering to formalize transformations between conceptual, graph-based, and operational representations of complex systems, again emphasizing the role of categories and functors as tools for relating heterogeneous models [Mordecai et al. 2021].

At the same time, recent work has continued to develop categorical semantic modeling more directly in relation to programming and formal systems. We provided a contemporary overview of semantic modeling in categories, with particular

attention to imperative programming, denotational modeling of memory states, and coalgebraic formulations of operational behavior [Steingartner 2025]. Beyond programming-language semantics in the narrow sense, newer categorical studies also investigate how semantic information can arise from structured transformations between syntax-like and meaning-like levels, as illustrated for example in enriched categorical approaches to language and semantic representation [Bradley et al. 2022]. Taken together, these works suggest that category theory continues to function both as a foundational semantic framework and as a flexible instrument for relating different levels of computational description.

In contrast to approaches that aim at highly expressive semantic meta-frameworks or broad cross-domain architectures, the present work adopts a deliberately lightweight categorical interpretation of computation. We model programming languages directly as categories of states, where objects represent concrete configurations and morphisms represent executable transitions. The primary contribution lies in using this state-based categorical perspective as a unifying abstraction for comparing different programming paradigms and expressing translations between them via functors.

6. CONCLUSION

In this article, we have presented the conceptual basis for using category theory as a unifying formal framework for describing the semantics of programming languages across different paradigms. Although the idea of category semantics is not new, existing approaches often require the introduction of more complex structures (e.g., arenas, institutions, or quantales), which leads to higher formal complexity and limited applicability in some contexts.

We have shown that by interpreting states as objects and dynamic state changes as morphisms, a simple yet sufficiently expressive model of computation can be obtained. Within this framework, different programming paradigms differ primarily in the structure of the state space, while the state transformation mechanism itself remains formally unified.

A further level of abstraction is represented by the description of semantics using functors between categories, which allow us to capture the relationships between different semantic levels, as well as between syntax and semantics. This functional perspective provides a natural way to compare and interpret programming languages within a common formal environment. In the next stages of the research, we plan to

focus on the systematic formalization of functional relations between categories representing different programming languages. The goal is to identify the conditions under which these representations preserve the semantic structure of computation, in particular the composition, identity, and equivalence of programs. The expected benefit of this approach is the possibility of formal comparison and classification of programming languages based on their structural properties, as well as the systematic description of translations and interpretations between languages. From a practical point of view, this framework can find application in the design and analysis of domain-specific languages, in the modeling of compilers, or in the formal verification of programs. In the next step, we will also extend the model to more advanced categorical constructs, such as Kleisli categories and coalgebraic approaches, which allow for a more precise capture of the effects and dynamic aspects of computational processes.

ACKNOWLEDGMENT

This work was supported by KEGA project 028TUKE-4/2026 – “Innovative approaches to teaching and researching formal methods in the context of modern software engineering”, granted by the Cultural and Education Grant Agency of the Slovak Ministry of Education.

REFERENCES

- AWODEY, S. 2010. *Category Theory*, 2 ed. Oxford Logic Guides, vol. 52. Oxford University Press, Oxford.
- BAEZ, J., LI, X., LIBKIND, S., OSGOOD, N. D., AND PATTERSON, E. 2023. Compositional modeling with stock and flow diagrams. *Electronic Proceedings in Theoretical Computer Science* 380, 77–96.
- BARR, M. AND WELLS, C. 1990. *Category Theory for Computing Science*. Prentice-Hall, Inc., USA.
- BRADLEY, T.-D., TERILLA, J., AND VLASSOPOULOS, Y. 2022. An enriched category theory of language: From syntax to semantics. *La Matematica* 1, 2, 551–580.
- COOK, S. A. AND RECKHOW, R. A. 1973. Time-bounded random access machines. *Journal of Computer and System Sciences* 7, 4, 354–375.
- DISKIN, Z. AND KADISH, B. 2003. Variable set semantics for keyed generalized sketches: Formal semantics for object identity and abstract syntax for conceptual modeling. *Data & Knowledge Engineering* 47, 1, 1–59.
- FIGLIORE, M. P., PLOTKIN, G. D., AND TURI, D. 1999. Abstract syntax and variable binding. In *Proceedings of the 14th Symposium on Logic in Computer Science (LICS 1999)*. IEEE Computer Society Press, Trento, Italy, 193–202.
- HARPER, R. 2016. *Practical Foundations for Programming Languages*, 2 ed. Cambridge University Press, Cambridge.

-
- HOARE, C. A. R. 1969. An axiomatic basis for computer programming. *Communications of the ACM* 12, 10, 576–580.
- JACOBS, B. 1999. *Categorical Logic and Type Theory*. Studies in Logic and the Foundations of Mathematics, vol. 141. North Holland, Elsevier, Amsterdam, The Netherlands.
- LAWVERE, F. W. 1963. Functorial semantics of algebraic theories. *Proceedings of the National Academy of Sciences of the United States of America* 50, 5, 869–872.
- MAC LANE, S. 1998. *Categories for the Working Mathematician*, 2 ed. Graduate Texts in Mathematics, vol. 5. Springer, New York.
- MIHELIČ, J., STEINGARTNER, W., AND NOVITZKÁ, V. 2021. A denotational semantics of a concatenative/compositional programming language. *Acta Polytechnica Hungarica* 18, 4, 231–250.
- MILEWSKI, B. 2021. *Category Theory for Programmers*. Blurb, San Francisco, CA.
- MOGGI, E. 1989. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS 1989)*. IEEE Computer Society Press, Pacific Grove, CA, USA, 14–23.
- MORDECAI, Y., FAIRBANKS, J. P., AND CRAWLEY, E. F. 2021. Category-theoretic formulation of the model-based systems architecting cognitive-computational cycle. *Applied Sciences* 11, 4, 1945.
- NOVITZKÁ, V., PERHÁČ, J., AND STEINGARTNER, W. 2019. Categorical approach to denotational and operational semantics. *IPSI Transactions on Internet Research* 15, 2, 45–51.
- PIERCE, B. C. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA.
- PLOTKIN, G. 1981. A Structural Approach to Operational Semantics. Tech. rep., Computer Science Department, Aarhus University.
- PLOTKIN, G. D. AND PRETNAR, M. 2013. Handling algebraic effects. *Logical Methods in Computer Science* 9, 4:23, 1–36.
- RILEY, M. ET AL. 2021. Category theory in computer science: Recent trends and applications. Bibliographic data unverified; replace with exact source before final submission.
- SCHMIDT, D. A. 1986. *Denotational Semantics: A Methodology for Language Development*. William C. Brown Publishers, USA.
- SCOTT, D. S. 1970. Outline of a mathematical theory of computation. In *Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems*. Department of Electrical Engineering, Princeton University, Princeton, NJ, 169–176.
- SPIVAK, D. I. 2014. *Category Theory for the Sciences*. MIT Press, Cambridge, MA.
- STEINGARTNER, W. 2025. Perspectives of semantic modeling in categories. *Journal of King Saud University Computer and Information Sciences* 37, 19.
- STEINGARTNER, W. AND NOVITZKÁ, V. 2017. Categorical semantics of programming languages. In *Selected Topics in Contemporary Mathematical Modeling*, A. Z. Grzybowski, Ed. 331. Publishing Office of Czestochowa University of Technology, Czestochowa, Poland, Chapter 11, 167–192. Series Monograph.
- STEINGARTNER, W., NOVITZKÁ, V., BAČÍKOVÁ, M., AND KOREČKO, Š. 2017. New approach to categorical semantics for procedural languages. *Computing and Informatics* 36, 6, 1385–1414.
- STEINGARTNER, W., NOVITZKÁ, V., AND SCHREINER, W. 2024. Proof of equivalence of semantic

- methods for a selected domain-specific language. *Journal of Applied Mathematics and Computational Mechanics* 23, 2, 79–92.
- STEINGARTNER, W. AND SCHREINER, W. 2025. Executable Semantics for Teaching Concatenative Stack-Based DSLs: The Case of StackLang. In *New Trends in Database and Information Systems, ADBIS 2025 Short Papers, Doctoral Consortium and Tutorials, Tampere, Finland, September 23–26, 2025, Proceedings*, P. K. Chrysanthis, K. Nørvåg, K. Stefanidis, Z. Zhang, E. Quintarelli, and E. Zumpano, Eds. Communications in Computer and Information Science (CCIS), vol. 2676. Springer, Cham, Switzerland, 248–263.
- STOY, J. E. 1977. *Denotational semantics : the Scott-Strachey approach to programming language theory*. Series in Computer Science, No. 1. MIT Press, Cambridge, MA.
- TENNENT, R. D. 1986. Functor-category semantics of programming languages and logics. In *Category Theory and Computer Programming*, D. Pitt, S. Abramsky, A. Poigné, and D. Rydeheard, Eds. Lecture Notes in Computer Science, vol. 240. Springer, Berlin, Heidelberg, 206–224.
- WINSKEL, G. 1993. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing. MIT Press, Cambridge, MA.
- ZHAO, H., CHEN, Y., AND FU, X. 2023. Comparison of machine learning based on category theory. *Journal of Web Engineering* 22, 1, 41–54.

Monika Kohútová
Faculty of Electrical Engineering and Informatics,
Technical University of Košice,
Letná 9, 042 00 Košice, Slovakia
E-mails: monika.kohutova@tuke.sk

William Steingartner
Faculty of Electrical Engineering and Informatics,
Technical University of Košice,
Letná 9, 042 00 Košice, Slovakia
E-mails: william.steingartner@tuke.sk

Samuel Novotný
Faculty of Electrical Engineering and Informatics,
Technical University of Košice,
Letná 9, 042 00 Košice, Slovakia
E-mails: samuel.novotny@tuke.sk