Faculty of Electrical Engineering and Informatics

Perspectives of semantic modeling in categories

William Steingartner

Technical University of Košice
Faculty of Electrical Engineering and Informatics

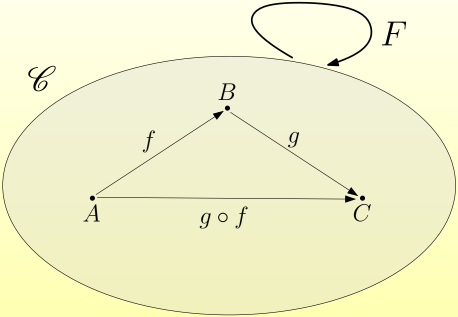
* * *

14 January – 16 January 2022

Categories

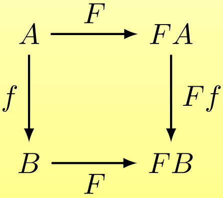
Category $\mathscr{C} = (\mathscr{C}_{obj}, \mathscr{C}_{morp})$ is a structure consisting of

- class of *objects* $\mathscr{C}_{obj}$,
- class of *morphisms* $\mathscr{C}_{morp}$,
- composition – binary operation $\circ$ defined on morphisms.



Functor $F : \mathscr{C} \to \mathscr{D}$ is a structure-preserving morphism between categories

$$F_0 : \mathscr{C}_{obj} \to \mathscr{D}_{obj}$$
$$F_1 : \mathscr{C}_{morp} \to \mathscr{D}_{morp},$$

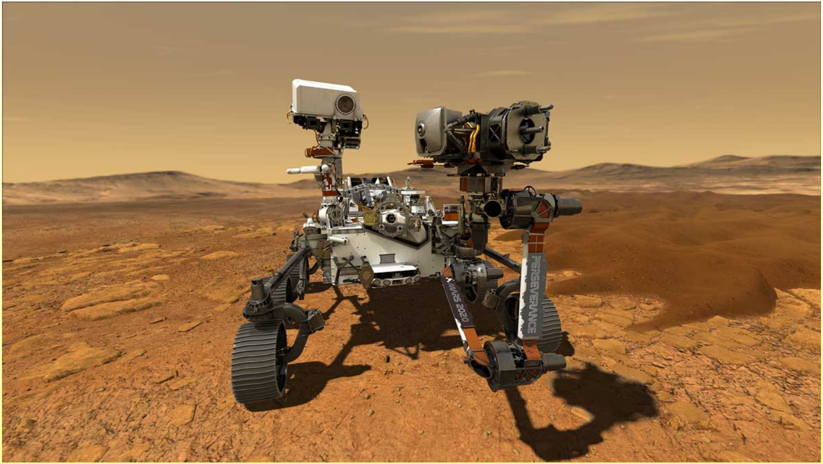


Categories

- mathematical structures consisting of objects and morphisms between them,
- objects can be various mathematical structures, data structures, types,
- categories have become useful for modeling computations, processes, programs, program systems,
- are basic structures for coalgebraic behavioral models.

Categories in teaching

- quite simple mathematical structures,
- graphical representations useful for illustration of examples,
- understandable for our students.



https://www.nasa.gov/consortium/CategoryTheory

Formal semantics

- provides unambiguous meaning of programs written in programming language,
- helps designers to prepare good and useful programming languages,
- serves for designers to design correct compilers,
- encourages users/programmers how to use language constructions properly.

Semantic methods

- denotational semantics,
- operational semantics,
- natural semantics,
- axiomatic semantics,
- action semantics,
- game semantics,
- ...

Categorical semantics

- denotational semantics uses category of types where objects are types and morphisms are functions,
- algebraic semantics uses institutions as complex structures based on categories of signatures,
- game semantics uses category of arenas.

Basic ideas of our approach

Why categorical semantics

- provides illustrative view of dynamics of states,
- provides simply understandable mathematical model of programs,
- appropriate for designers of compilers,
- serves for creating skills to work with formal methods.

Construction of category of states

- we consider simple imperative language,
- our language has only two implicit types,
- for now, we do not consider exception, jumps and recursion,
- we construct category of states,
- environment of procedures is constructed as category of categories,
- so simplified model is understandable without losing exactness.

Categorical denotational semantics of imperative languages

Categorical representation

- formulation of meaning indicates (determines) a construction of a categorical model for a given program,
- categorical model consists of:
  - objects – states during the program execution,
  - morphisms, which express the relations between objects – steps of computations.

The language *Jane*

$$
\begin{array}{ll}
n \in \mathbf{Num} & x \in \mathbf{Var} \\
e \in \mathbf{Expr} & b \in \mathbf{Bexpr} \\
S \in \mathbf{Statm} & D \in \mathbf{Decl}
\end{array}
$$

Steingartner, W., Novitzká, V., Bačíková, M., Korečko, Š., New approach to categorical semantics for procedural languages, Computing and Informatics, 36(6), 2017, pp. 1385–1414, DOI: 10.4149/cai_2017.6.1385

Language *Jane* – Syntax

The elements $n \in \mathbf{Num}$ and $x \in \mathbf{Var}$ have no internal structure from semantic point of view.

The syntactic domain $\mathbf{Expr}$ consists of all well-formed arithmetic expressions created by the following production rule

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

Boolean expression from $\mathbf{Bexpr}$ can be of the following structure:

$$b ::= \mathtt{false} \mid \mathtt{true} \mid e = e \mid e \leq e \mid \neg b \mid b \wedge b.$$

The variables used in programs have to be declared. We consider $D \in \mathbf{Decl}$ as a sequence of declarations:

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

As the statements $S \in \mathbf{Statm}$ we consider five Dijkstra's statements together with a block statement and an input statement:

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

Categorical denotational semantics of imperative languages

Categorical model

- we construct operational model of *Jane* as the category $\mathscr{C}_{State}$ of states,
- we assign to states their representation,
- because of block structure of *Jane*, we have to consider also a level of block nesting ($l \in \mathbf{Level}, \mathbf{Level} \subseteq \mathbf{N}$),
- representation of type $State$ has to express variable, its value with respect to the actual nesting level.

Specification of states

State

- can be considered as some abstraction of computer memory,
- change of state means change of a value in memory,
- because of block structure of *Jane*, we have to consider also a level of block nesting,
- every variable occurring in a program has to be allocated,
- we can assign and modify a value of allocated variable inducing change of state.

The signature Σ_{State} for states

$$
\begin{array}{lll}
\Sigma_{State} = & & \\
& \underline{types:} & State, Var, Value \\
& \underline{opns:} & init : \rightarrow State \\
& & alloc : Var, State \rightarrow State \\
& & get : Var, State \rightarrow Value \\
& & del : State \rightarrow State
\end{array}
$$

Categorical denotational semantics of imperative languages

States and their representation

- the state expresses an abstraction of memory: each step of program execution is characterized by the current state,
- the nesting level in the state allows us to create an environment of variables and distinguish locally declared variables from global ones,
- each state s is an element of the semantic domain $s \in \mathbf{State}$ and it is represented as a function

$$s : \mathbf{Var} \times \mathbf{Level} \rightarrow \mathbf{Value}$$

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

variable	level	value
x_1	1	v_1
$\vdots$		
x_n	l	v_n

Representation of operations

The operation $[\![init]\!]$

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

creates the initial state of a program with no declared variable.

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

The operation $[\![alloc]\!]$

$$[\![alloc]\!](x, s) = s \diamond ((x, l), \bot),$$

sets actual nesting level to declared variable. Because of undefined value of declared variable, the operation $[\![alloc]\!]$ does not change the state.

variable	level	value
$\vdots$	$\vdots$	$\vdots$
x	l	$\bot$

Representation of operations

The operation $[\![\, get \,]\!]$ returns a value of a variable declared on the highest nesting level,

$$[\![\, get \,]\!](x, \langle \ldots, ((x, l_i), v_i), \ldots, ((x, l_k), v_k), \ldots \rangle) = v_k,$$

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

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

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

variable	level	value
$\vdots$	$\vdots$	$\vdots$
x	l_i	v
x_i	l_j	v_k
$\vdots$	$\vdots$	$\vdots$
x_n	l_j	v_m

Declarations

Declarations

A declaration

$$\texttt{var}\ x$$

is represented as an endomorphism:

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

for a given state s and defined by

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

A sequence of declarations:

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

A declaration creates a new entry for declared variable with the actual level of nesting and an undefined value

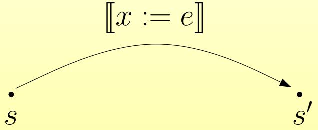
$$((x, l), \bot).$$

Categorical denotational semantics

Semantics of statements

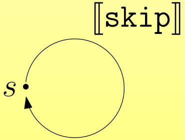
- variable assignment

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



- empty statement

$$[\![\text{skip}]\!] = \text{id}$$

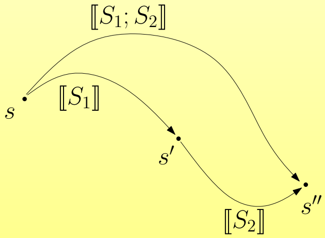


Categorical denotational semantics

Semantics of statements

- sequence of statements

$$[\![S_1; S_2]\!] s = \begin{cases} s', & \text{if } [\![S_1]\!] s = s'' \text{ and } [\![S_2]\!] s'' = s', \\ s_\perp, & \text{if } [\![S_1]\!] s = s_\perp, \text{ or} \\ & \text{if } [\![S_1]\!] s = s'' \text{ and } [\![S_2]\!] s'' = s_\perp. \end{cases}$$

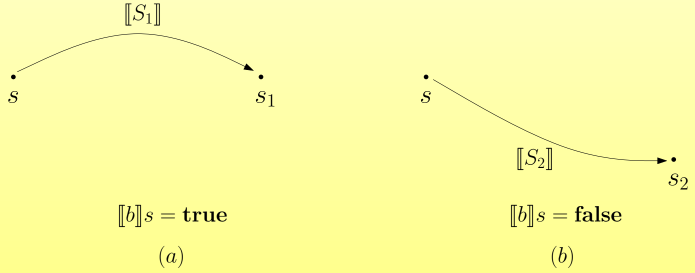


Categorical denotational semantics

Semantics of statements

- conditional statement

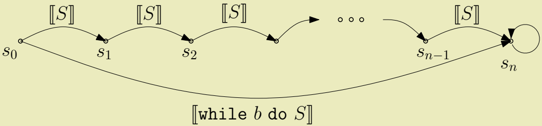
$$[\![\, \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \,]\!]s = \begin{cases} [\![\, S_1 \,]\!]s, & \text{if } [\![\, b \,]\!]s = \textbf{true}, \\ [\![\, S_2 \,]\!]s, & \text{if } [\![\, b \,]\!]s = \textbf{false}, \\ s_\perp, & \text{otherwise.} \end{cases}$$



Statements

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

$$[\![\, \texttt{input}\ x \,]\!] s = \begin{cases} s' = s\,[v/x]\,, & \text{for } \left((x, \max\ l)\,, v'\right) \in s, \\ s_{\perp}\,, & \text{otherwise.} \end{cases}$$



Block statement

$$\texttt{begin}\ D, S\ \texttt{end}$$

The following is a summary of the four steps used to execute of unnamed blocks.

- Nesting level l is incremented. We represent this step by fictive entry in state table

  $$((\texttt{begin}, l+1), \bot)$$

  i.e. endomorphism $\mathbf{State} \to \mathbf{State}$.
- Local declarations are elaborated on nesting level $l+1$.
- The body S of block is performed.
- Locally declared variables are forgotten at the end of block. We model this situation using operation $[\![\, del \,]\!]$.

The semantics:

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

Constructing the category

Now we can define the category $\mathscr{C}_{State}$ of states as follows:

- category objects are states as sequences of tuples for variables together with special state $s_\perp$,
- category morphisms are functions $[\![S]\!] : s \rightarrow s'$.

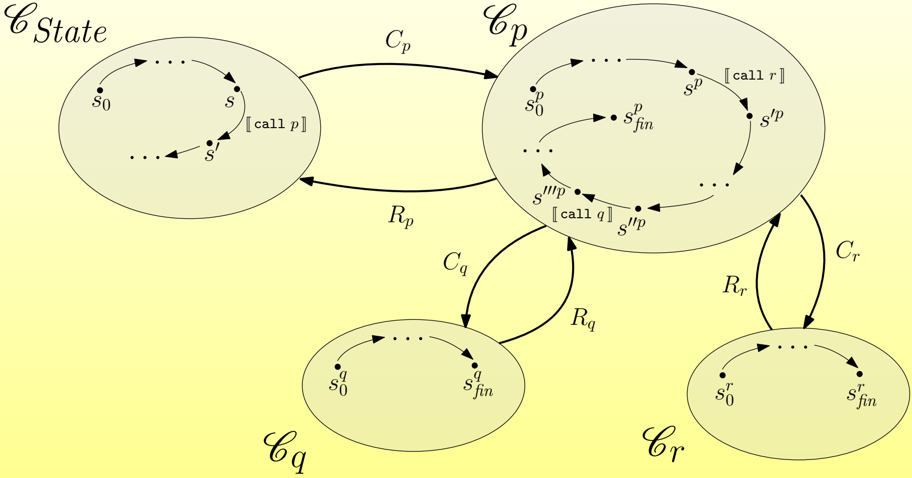
The category $\mathscr{C}_{State}$ has the following properties:

- the special object $s_\perp = \langle((\perp, \perp), \perp)\rangle$, an undefined state, is a terminal object of our category, from any object there is a unique morphism to this state,
- the initial state $s_0 = \langle((\perp, 1), \perp)\rangle$ is the initial object of our category,
- the category $\mathscr{C}_{State}$ has no products, because a program written in $Jane$ cannot be simultaneously in more than one state.

We can state that $\mathscr{C}_{State}$ is a category without products and with initial and terminal objects.

Categorical denotational semantics

Semantics of procedures

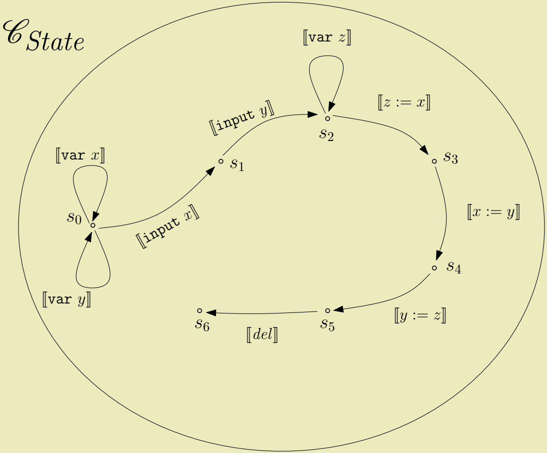


Example

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

We consider values $\mathbf{3}$ and $\mathbf{5}$ for variables x and y, resp.

Categorical representation of program



States during program execution

s_0		
x	1	$\perp$
y	1	$\perp$

s_1		
x	1	**3**
y	1	$\perp$

s_2		
x	1	**3**
y	1	**5**
z	2	$\perp$

s_3		
x	1	**3**
y	1	**5**
z	2	**3**

s_4		
x	1	**5**
y	1	**5**
z	2	**3**

s_5		
x	1	**5**
y	1	**3**
z	2	**3**

s_6		
x	1	**5**
y	1	**3**
~~z~~	~~2~~	~~$\perp$~~

Categorical operational semantics

Coalgebraic approach

- coalgebras are defined as arrows from the state space (X) to the image of state space in the endofunctor F (determined by the signature):

$$\langle [\![sel_1]\!], \ldots, [\![sel_n]\!] \rangle : X \to FX$$

- we define coalgebras above the base category, whose objects create a state space and whose morphisms are transitions,
- coalgebras provide observable properties and are one of the tools for modeling the behavior of dynamical systems,
- each individual step of program execution is expressed by the application of a polynomial endofunctor in the category of configurations.

Steingartner, W., Novitzká, V., Schreiner W., Coalgebraic Operational Semantics for an Imperative Language, Computing and Informatics, 38(5), 2019, pp. 1181-1209, DOI: 10.31577/cai_2019_5_1181

Categorical operational semantics

State space and its representation

- we consider the data type of configurations as the state space,
- memory expresses one moment of program execution:

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

- representation of state space is a set

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

where configuration is given as follows:

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

Semantics of statements

We define the execution of one step by morphism $[\![\, next \,]\!]$:

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

- variable assignment $x := e$:

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

where

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

- empty statement

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

Semantics of statements

- sequence of statements $S_1; S_2; S^*$:

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

- conditional statement

$$[\![\, next \,]\!]([\![\, \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2; S^* \,]\!], m, i^*, o^*) =$$

$$\begin{cases} ([\![\, S_1; S^* \,]\!], m, i^*, o^*), & \text{if } [\![\, b \,]\!]m = \mathbf{true}, \\ ([\![\, S_2; S^* \,]\!], m, i^*, o^*), & \text{if } [\![\, b \,]\!]m = \mathbf{false}, \\ m_\perp, & \text{otherwise} \end{cases}$$

Semantics of statements

- user input – statement `read`

$$[\![\, read \,]\!] : \mathbf{Config} \rightarrow \mathbf{Config}^{\mathbf{Value}}$$

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

where $m' = m[((x, Highest(m, x)), v) \mapsto ((x, Highest(m, x)), v')]$,

- user output – statement `print`

$$[\![\, print \,]\!] : \mathbf{Config} \rightarrow \mathbf{Value} \times \mathbf{Config}$$

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

Coalgebra for language *Jane*

Coalgebra for language *Jane*

Category of configurations $\mathscr{C}onfig$ consists of:

- objects – configurations $config = ([\![D^*; S^*]\!], m, i^*, o^*)$,
- arrows – morphisms $[\![next]\!], [\![read]\!], [\![print]\!]$ a $[\![abort]\!]$.

Polynomial endofunctor (over the category of configurations):

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

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

$$Q(config) = [\![abort]\!](config) \quad Q(config) = [\![next]\!](config)$$
$$Q(config) = [\![print]\!](config) \quad Q(config) = [\![read]\!](config)$$

Categorical operational semantics – Example

```
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;
```

For simplicity we introduce the following substitutions:

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

and we consider values $\mathbf{3}$ and $\mathbf{5}$ for variables x and y, resp.

Example

The initial configuration is

$$config_0 = ([\![D_1; D_2; S_1; S_2; S_3; S_4]\!], m_0, i^*, o^*).$$

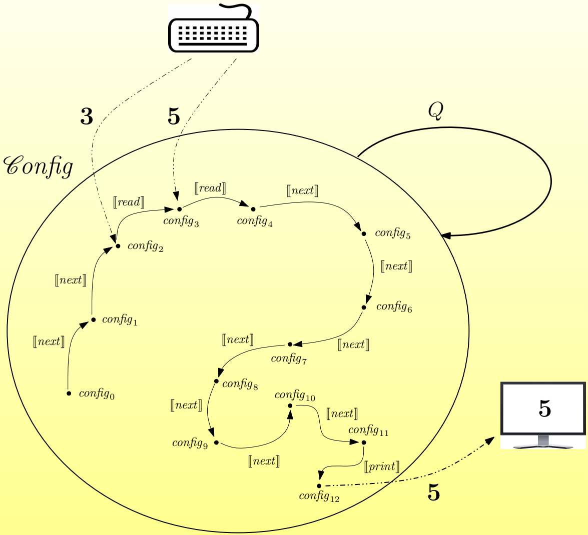
Each application of the endofunctor Q represents one step of program execution. First, the individual declarations and user inputs are processed in separate steps:

$$\begin{aligned}
Q(config_0) &= [\![next]\!](config_0) = config_1 = \\
&= ([\![D_2; S_1; S_2; S_3; S_4]\!], [\![\texttt{var}\ x]\!] m_0, (\mathbf{3}, \mathbf{5}), \varepsilon), \\
\\
Q(config_1) &= [\![next]\!](config_1) = config_2 = \\
&= ([\![S_1; S_2; S_3; S_4]\!], [\![\texttt{var}\ y]\!] m_1, (\mathbf{3}, \mathbf{5}), \varepsilon), \\
Q(config_2) &= [\![read]\!](config_2) = config_3 = \\
&= ([\![S_2; S_3; S_4]\!], m_3, (\mathbf{5}), \varepsilon), \\
\\
Q(config_3) &= [\![read]\!](config_3) = config_4 = \\
&= ([\![S_3; S_4]\!], m_4, \varepsilon, \varepsilon),
\end{aligned}$$

Example

$$\begin{aligned}
Q(config_4) &= [\![next]\!](config_4) = config_5 = \\
&= ([\![\texttt{begin var } z; z := x; x := y; y := z \textbf{ end}; S_4]\!], m_4, \varepsilon, \varepsilon), \\
Q(config_5) &= [\![next]\!](config_5) = config_6 = \\
&= ([\![\texttt{var } z; z := x; x := y; y := z \textbf{ end}; S_4]\!], m_5, \varepsilon, \varepsilon), \\
Q(config_6) &= [\![next]\!](config_6) = config_7 = \\
&= ([\![z := x; x := y; y := z \textbf{ end}; S_4]\!], m_6, \varepsilon, \varepsilon), \\
Q(config_7) &= [\![next]\!](config_7) = config_8 = \\
&= ([\![x := y; y := z \textbf{ end}; S_4]\!], m_7, \varepsilon, \varepsilon), \\
Q(config_8) &= [\![next]\!](config_8) = config_9 = \\
&= ([\![y := z \textbf{ end}; S_4]\!], m_8, \varepsilon, \varepsilon), \\
Q(config_9) &= [\![next]\!](config_9) = config_{10} = \\
&= ([\![\textbf{end}; S_4]\!], m_9, \varepsilon, \varepsilon), \\
Q(config_{10}) &= [\![next]\!](config_{10}) = config_{11} = \\
&= ([\![S_4]\!], [\![end]\!] m_9, \varepsilon, \varepsilon).
\end{aligned}$$

Example



3
5
Q
𝒞onfig
⟦read⟧
⟦read⟧
⟦next⟧
config3
config4
config5
config2
⟦next⟧
⟦next⟧
config6
config1
⟦next⟧
⟦next⟧
⟦next⟧
config7
config0
config8
config10
⟦next⟧
⟦next⟧
config11
config9
⟦next⟧
⟦print⟧
config12
5
5

Modeling of recursive computations

Modeling of recursive computations

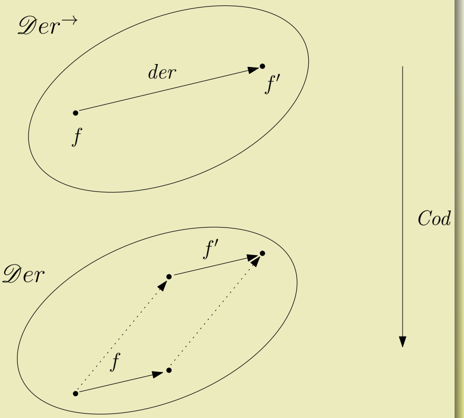
- we model recursion development and subsequent calculation using algebras and coalgebras and their properties,
- there is exactly one morphism from the initial algebra to any algebra (A, a) – *catamorphism*, in the calculation it acts as an iterator (deconstructor) – a function that provides elements of the structure,
- from any coalgebra (U, φ) there exists one unique morphism into the final coalgebra – *anamorphism*, in calculations it acts as a coiterator (constructor) – a function that creates a structure,
- the composition of catamorphism and anamorphism creates a new morphism – *hylomorphism*, which represents a recursive function – by creating complex data structures and then processing them.

Steingartner, W., Macko, P., Some New Approaches in Functional Programming Using Algebras and Coalgebras, Electronic Notes in Theoretical Computer Science, 279(3), 2011, pp. 41–62

Categorical modeling in differential calculus

- the relationship between functions and their derivatives expresses a categorical model,
- we model functions and their derivatives as objects in the category of morphisms $\mathscr{D}er^{\to}$ over the base category $\mathscr{D}er$,
- the relationship between the two categories is expressed by a codomain functor:

$$Cod : \mathscr{D}er^{\to} \to \mathscr{D}er$$



Steingartner, W., Galinec, D., The Rôle of Categorical Structures in Infinitesimal Calculus, Journal of Applied Mathematics and Computational Mechanics, 12(1), 2013, pp. 107–119

Modeling of component systems

1. interfaces and interactions between components (algebraic specifications) – **interface category** (objects are interfaces, morphisms are interactions),
2. contracts for component composition and interaction - (I) we extend interface specifications with assumptions and guarantees or (II) we express them as formulas in predicate linear logic,
3. dependencies – expressed as predicates in predicate linear logic.

3	dependencies
2	contracts
1	interfaces

Steingartner, W., Novitzká, V., Benčková, M., Prazňák, P., Considerations and Ideas in Component Programming – Towards to Formal Specification, 25th CECIIS, 2014, pp. 332–339

Denotational semantics of concatenative language

Research in the field of concatenative languages

- we designed a simple concatenative / compositional language KKJ,
- the language has a compositional character – the syntactic concatenation of programs corresponds to the semantic composition of functions,
- KKJ language syntax:

$$e ::= \varepsilon \mid i \mid n \mid \ \{e\} \ \mid e\ e$$

- the state of memory is expressed by the stack – the program gradually changes the contents of the stack,
- in our research we constructed classical denotational semantics for the KKJ language as the first step in research.

Mihelič, J., Steingartner, W., Novitzká, V., A denotational semantics of a concatenative / compositional programming language, Acta Politechnica Hungarica, 18(4), 2021, pp. 231–250, DOI: 10.12700/APH.18.4.2021.4.13

Application of research into teaching

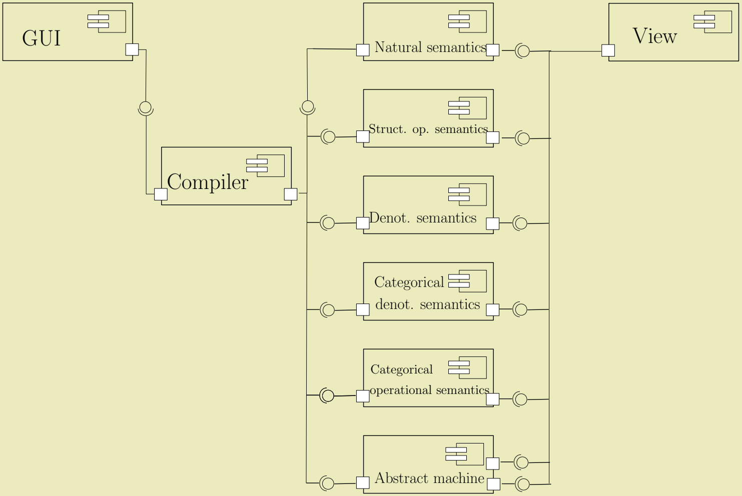
Software to support the teaching of formal semantics

- we also apply published results focused on categorical semantics and semantic modeling in the teaching process,
- our main goal is to implement and deploy a comprehensive learning environment – an interactive software package that will allow illustrative and understandable use of semantic methods,
- the mentioned software package will provide full-fledged modules for working with individual semantic methods and principles, which are presented in the teaching.

Steingartner, W., Novitzká, V., A survey of teaching tools for the course on the Semantics of Programming Languages, In: Mathematical Modelling in Physics and Engineering, Częstochowa, Poland, Politechnika Częstochowska, pp. 40–47. *Invited lecture*.

Application of research into teaching

Software to support the teaching of formal semantics



Conclusion

Achieved results

- categorical models for programming languages: semantic methods for imperative languages,
- semantic modeling of recursive computations, the relationship of functions and their derivatives, the properties of categorical models of components and the relationships between them and denotational semantics for a new concatenative language,
- application of achieved results to teaching.

Areas for future research

- modeling of component systems,
- semantic modeling for concatenative and some domain-specific languages,
- modeling of properties of mathematical objects,
- application of results in the field of construction of reliable programs.

Thank You for Your attention.