A new approach to operational semantics by coalgebras

William Steingartner

william.steingartner@tuke.sk

Faculty of Electrical Engineering and Informatics
Department of Computers and Informatics

Coalgebraic structural operational semantics

- Coalgebra allows to model behavior of program systems.
- Coalgebra defines structural operational semantics of programs written in some programming language.
- Coalgebra is constructed over category of state space by polynomial endofunctor.
- Every application of functor models one step of computation so as structural operational semantics.

Language *Jane*

- We briefly show the structural operational semantics of *Jane*.
- The following syntactic domains are introduced:

$n \in \mathbf{Num}$ — strings of digits,
$x \in \mathbf{Var}$ — names of variables,
$e \in \mathbf{Expr}$ — arithmetic expressions,
$b \in \mathbf{Bexpr}$ — Boolean expressions,
$S \in \mathbf{Statm}$ — statements,
$D \in \mathbf{Decl}$ — declarations of variables.

Syntactic domains $\mathbf{Num}$ and $\mathbf{Var}$ do not have internal structure from the semantic point of view.

Language *Jane*: Syntax

Syntactic domain $\mathbf{Expr}$ contains well-formed arithmetic expressions according the following syntactic rule:

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

Well-formed Boolean expressions from $\mathbf{Bexpr}$ can be of the following forms:

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

Statements $S \in \mathbf{Statm}$ are standard (basic) Dijkstra's constructs (also called D-diagrams) – variable assignment, empty statement, sequence of statements, conditional statement and loop statement:

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

State

- The main semantic domain in structural operational semantics is $\mathbf{State}$ which contains particular memory states.
- State $s \in \mathbf{State}$ is an abstraction of computer memory.
- Each state is considered as function:

$$s : \mathbf{Var} \times \mathbf{Level} \rightarrow \mathbf{Z}.$$

A change of memory content (cell) is represented as an actualization of a state:

$$s' = s[((x, l), v) \mapsto ((x, l), \mathbf{n})].$$

Semantics of expressions

Semantic functions for arithmetic and Boolean expressions, resp., are

$$[\![e]\!] : \mathbf{Expr} \to \mathbf{State} \to \mathbf{Value} \qquad [\![b]\!] : \mathbf{Bexpr} \to \mathbf{State} \to \mathbf{Bool}$$

$$[\![n]\!]s = \mathbf{n}$$

$$[\![x]\!]s = [\![get]\!](x, s)$$

$$[\![e_1 + e_2]\!]s = [\![e_1]\!]s \oplus [\![e_2]\!]s$$

$$[\![e_1 - e_2]\!]s = [\![e_1]\!]s \ominus [\![e_2]\!]s$$

$$[\![e_1 * e_2]\!]s = [\![e_1]\!]s \otimes [\![e_2]\!]s$$

$$[\![\mathtt{true}]\!]s = \mathbf{true}$$

$$[\![\mathtt{false}]\!]s = \mathbf{false}$$

$$[\![e_1 = e_2]\!]s = \begin{cases} \mathbf{true}, & \text{if } [\![e_1]\!]s{=}[\![e_2]\!]s, \\ \mathbf{false}, & \text{otherwise,} \end{cases}$$

$$[\![e_1 \leq e_2]\!]s = \begin{cases} \mathbf{true}, & \text{if } [\![e_1]\!]s{\leq}[\![e_2]\!]s, \\ \mathbf{false}, & \text{otherwise,} \end{cases}$$

$$[\![\neg b]\!]s = \begin{cases} \mathbf{true}, & \text{if } [\![\neg b]\!]s{=}\mathbf{false}, \\ \mathbf{false}, & \text{otherwise,} \end{cases}$$

$$[\![b_1 \wedge b_2]\!]s = \begin{cases} \mathbf{true}, & \text{if } [\![b_1]\!]s{\leq}[\![b_2]\!]s{=}\mathbf{true}, \\ \mathbf{false}, & \text{otherwise.} \end{cases}$$

Semantics of statements

Semantic function is defined as follows:

$$[\![S]\!] : \textbf{Statm} \to \textbf{State} \to \textbf{State}$$

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

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

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

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

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

Basic notions

- Coalgebra is mathematical structure, that is constructed over a base category by polynomial endofunctor.
- Category $\mathscr{C}$ is a structure, that consists of objects and morphisms between them.
- Functor is a kind of morphism between categories. Functor defined on one category is called endofunctor:

$$F : \mathscr{C} \to \mathscr{C}.$$

  The name polynomial is used for functor which has a form of polynomial:

$$FX = A_0 + A_1 \times X^{B_1} + A_2 \times X^{B_2} + \ldots + A_n \times X^{Bn},$$

  where A_i, B_i are sets, and X stands for a state space, a class of objects in base category.
- Then, a coalgebra is a mapping

$$c : X \to FX,$$

  where c is n-tuple of morphisms of category that causes change of state.

Language *E-Jane*

We extend the base language with new constructs.

- Declarations of variables (without initializations):

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

  where ε is an empty declaration.

- Block statement, user input and output statements:

$$S ::= \ldots \mid \mathtt{begin}\ D; S\ \mathtt{end} \mid \mathtt{input}\ x \mid \mathtt{print}\ e.$$

- A source program in *E-Jane* has a form

$$D_1; \ldots; D_m; S_1; \ldots; S_n,$$

  so it consists of a list of declarations and a list of statements.

Signatures for lists of declarations and statements

First, we specify signatures for the lists of declarations and statements:

$$
\begin{array}{l}
\Sigma_{Decl\_List} = List_{fin}\,[Decl] \\
types: \quad Decl\_List, Decl \\
opns: \\
\quad init_d :\to Decl\_List \\
\quad head_d : Decl\_List \to Decl \\
\quad tail_d : Decl\_List \to Decl\_List
\end{array}
$$

$$
\begin{array}{l}
\Sigma_{Statm\_List} = List_{fin}\,[Statm] \\
types: \quad Statm\_List, Statm \\
opns: \\
\quad init :\to Statm\_List \\
\quad head : Statm\_List \to Statm \\
\quad tail : Statm\_List \to Statm\_List
\end{array}
$$

Signature for memory

An abstraction of a memory we specify as a signature $Memory$:

$$\Sigma_{Memory} =$$

$$\begin{array}{ll} types: & Memory, Var, Value \\ opns: & init :\to Memory \\ & alloc : Var, Memory \to Memory \\ & get : Var, Memory \to Value \\ & del : Memory \to Memory \end{array}$$

- $init$ establishes an initial memory of a program,
- $alloc$ reserves a memory cell for declared variable,
- get returns (gets) an actual values of a variable,
- del deallocates (forgets) all memory cells defined on the highest nesting level.

Signature for configuration

In our approach, we will use *configuration* as a kind of state (snapshot of a memory) which we specify as follows:

$$\begin{aligned}\Sigma_{Config} = \quad & \Sigma_{Decl\_List} + \Sigma_{Statm\_List} + \Sigma_{Memory} + \\ & types : Config \\ & opns : next : Config \to Config \\ & \quad input : Config, Value \to Config \\ & \quad print : Config \to Value, Config\end{aligned}$$

- $next$ provides the next configuration,
- $input$ takes an input value and stores it into the given (concrete) memory cell,
- $print$ evaluates the value of an argument and returns it as an observable value.

Representation of types

We assign to particular specifications of types their representations as follows:

Type	**Representation**	
$Value$	$\mathbf{Value} = \mathbf{Z} \cup \{\perp\}$	
Var	$\mathbf{Var}$	
$Decl\_List$	$\mathbf{Decl\_List}$	$[\![D^*]\!] = [\![D_1; \ldots; D_n]\!]$
$Statm\_List$	$\mathbf{Statm\_List}$	$[\![S^*]\!] = [\![S_1; \ldots; S_n]\!]$
$Memory$	$\mathbf{Memory}$	
$Config$	$\mathbf{Config} = \mathbf{Program} \times \mathbf{Memory}$	

Representation of a program is :

$$\mathbf{Program} = [\![D^*; S^*]\!]$$

and an initial configuration of a program is:

$$config_0 = ([\![D^*; S^*]\!], m_0, i^*, o^*).$$

Representation of memory

We must to consider a level of nesting for a block statement:

$$l \in \mathbf{Level}, \quad \text{where} \;\; \mathbf{Level} \subseteq \mathbf{N}.$$

An actual memory $m \in \mathbf{Memory}$ we represent as a function:

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

This function can be defined also by graph of a function:

$$graph(m) = \{((x,l),v) \mid m(x,l) = v\}\,.$$

We need an auxiliary function which returns the highest nesting level:

$$\begin{array}{ll} maxlevel : & \mathbf{Memory} \to \mathbf{Level}, \\ maxlevel(m) = & L, \text{where } \forall l_j . L \geq l_j, \end{array}$$

for

$$graph(m) = \{((x_1,1),v_1),\ldots,((x_i,l_j),v_k)\}.$$

Operations on a memory

An operation for initialization of memory we define as follows:

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

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

actual memory

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

initial memory

Operations on a memory

The other operations in memory we define as follows:

$$
\begin{array}{ll}
[\![alloc]\!](x,m) = & graph(m) \cup \{((x, maxlevel(m))\,,\perp)\} \\
[\![get]\!](x,m) = & v, \quad \text{if } ((x, maxlevel(m)), v) \in m \\
[\![del]\!]m = & graph(m) \setminus \{((x_i, maxlevel(m)), v_i) \mid i \in \mathbf{N}\}
\end{array}
$$

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

allocation of memory

variable	level	value
$\vdots$	$\vdots$	$\vdots$
x	l_{j-1}	v
x_i	l_j	v_k
$\vdots$	$\vdots$	$\vdots$
x_n	l_j	v_m

forgetting of local variables

Representation of configurations

The last step is to define the representation of configurations. Configurations config are elements of the set $\mathbf{Config}$:

$$\mathbf{Config} = \mathbf{Program} \times \mathbf{Memory}.$$

An initial configuration before the execution of a program has a form:

$$\mathit{config}_0 = ([\![D_1; D_2; \ldots; D_m; S_1; S_2; \ldots; S_n]\!], m_0, i^*, o^*).$$

After each step of program execution:

- a configuration is changed, an operation $[\![\mathit{tail}]\!]$ is applied on a list of declarations and statements;
- memory can be changed, this depends on an elaborated declaration or executed statement.

Set $\mathbf{Config}$ is considered as a **state space** of coalgebra.

Semantics of declarations

Semantics of declarations is defined as a function on a memory:

$$[\![\mathtt{var}\ x]\!] : \mathbf{Memory} \to \mathbf{Memory}$$

as

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

We extend set of variables $\mathbf{Var}$ with special (dummy) variables $\mathtt{begin}$ and $\mathtt{end}$ to ensure that we can identify the beginning and the end of a block statement. Such declaration increments level of nesting and we define its elaboration as follows:

$$\begin{aligned} [\![begin]\!]m = &\quad graph(m) \cup \{((\mathtt{begin}, l+1), \perp)\}, \\ [\![end]\!]m = &\quad [\![del]\!]m. \end{aligned}$$

Extending the semantics of declarations for configurations

Because a state space of coalgebra is a set of configurations, we extend the definition for elaboration of declarations for configurations. We define a morphism:

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

and we define elaboration of declarations by the following functions:

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

This definition corresponds with the traditional definition for elaboration of declarations, where the declaration of a variable actualizes an environment of variables.

Semantics of statements

An execution of one step we define with the morphism $[\![next]\!]$:

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

For an assignment statement and for the statement `skip`, this morphism is defined as follows:

$$\begin{aligned}
[\![next]\!]([\![x := e, S^*]\!], m, i^*, o^*) &= ([\![S^*]\!], m[x \mapsto [\![e]\!]m], i^*, o^*),\\
[\![next]\!]([\![\texttt{skip}; S^*]\!], m, i^*, o^*) &= ([\![S^*]\!], m, i^*, o^*).
\end{aligned}$$

In the sequence of statements, any statement can be executed in one or more steps. Hence we must to consider the following situations:

$$[\![next]\!]([\![S_i; S_{i+1}; \ldots; S_n]\!], m, i^*, o^*) = \begin{cases} ([\![S_{i+1}; \ldots; S_n]\!], m', i^*, o^*), \\ \quad \text{if } \langle S_i, m \rangle \Rightarrow m', \\ \\ ([\![S_i'; S_{i+1}; \ldots; S_n]\!], m', i^*, o^*), \\ \quad \text{if } \langle S_i, m \rangle \Rightarrow \langle S_i', m' \rangle. \end{cases}$$

Semantics of statements

An execution of conditional statement depends on a value of Boolean condition. Hence we define the first step of execution as follows:

$$[\![next]\!]([\![\texttt{if}\ b\ \texttt{then}\ S_i'\ \texttt{else}\ S_i''; S_{i+1}; \ldots; S_n]\!], m, i^*, o^*) = \begin{cases} ([\![S_i'; S_{i+1}; \ldots; S_n]\!], m, i^*, o^*), & \text{if } [\![b]\!]m = \mathbf{true}, \\ ([\![S_i''; S_{i+1}; \ldots; S_n]\!], m, i^*, o^*), & \text{if } [\![b]\!]m = \mathbf{false}. \end{cases}$$

The first step of execution of a loop is a transformation to the semantically equivalent conditional statement, and we define this step as follows:

$$\begin{aligned} &[\![next]\!][\![\,(\texttt{while}\ b\ \texttt{do}\ S_i'; S_{i+1}; \ldots; S_n, m, i^*, o^*)\,]\!] \\ &\quad = [\![next]\!]\,([\![\texttt{if}\ b\ \texttt{then}\ S_i'; \texttt{while}\ b\ \texttt{do}\ S_i'\ \texttt{else}\ \texttt{skip}; S_{i+1}; \ldots; S_n]\!], m)\,, i^*, o^*) \end{aligned}$$

Semantics of statements

For the statements of user input and output, we define appropriate representations of the operations in signature for configurations:

$$[\![input]\!]([\![\texttt{input}\ x; S_{i+1}; \ldots; S_n]\!], m, i^*, o^*) = \begin{array}{l} \lambda v'.([\![S_{i+1}; \ldots; S_n]\!], m', tail(i^*), o^*), \\ ([\![S_{i+1}; \ldots; S_n]\!], m_\perp, tail(i^*), o^*), \end{array}$$

where m' is an actualized memory

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

where an input value v is stored into a memory cell allocated for the variable x on the highest nesting level.

The output statement does not change a memory, but it changes a configuration (output statement is removed from the sequence of statements):

$$[\![print]\!](\texttt{print}\ e; S_{i+1}; \ldots; S_n]\!], m, i^*, o^*) = ([\![e]\!]m, ([\![S_{i+1}; \ldots; S_n]\!], m, i^*, [\![e]\!]m; o^*))$$

Semantics of statements

During the execution of a block statement, an elaboration of a special variable `begin` is elaborated:

$$
\begin{array}{c}
[\![next]\!]([\![\texttt{begin}\ D^*; S_i'\ \texttt{end}; S_{i+1}; \ldots; S_n]\!], m, i^*, o^*) = \\
([\![D^*; S_i'\ \texttt{end}; S_{i+1}; \ldots; S_n]\!], [\![begin]\!]m, i^*, o^*)
\end{array}
$$

and an execution of a block statement continues by elaboration of real local variables (if present) and by execution of statements inside a block.

For an interruption of program execution we define a function

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

which immediately terminates running program which ends in an undefined configuration $config_\perp$:

$$
[\![abort]\!](config) = (\varepsilon, m_\perp, \varepsilon, \varepsilon).
$$

Base category for coalgebra

We construct a category $\mathscr{C}onfig$ where:

- configurations $config = ([\![D^*, S^*]\!], m, i^*, o^*)$ are category objects,
- mappings $[\![next]\!], [\![input]\!], [\![output]\!]$ and $[\![abort]\!]$ are category morphisms,
- we define an identity morphism for each configuration,
- composition of morphisms and associativity of composition holds.

An undefined configuration is also a terminal (final) object of a category:

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

because there exists a possibility to abrupt the program execution from any configuration.

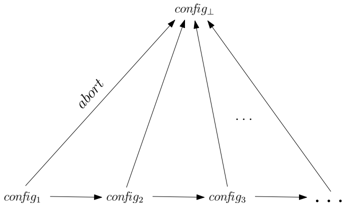
Colimit in base category

In case of infinite loop we must to guarantee that an execution of such a statement cannot be defined in category. Hence we require that each infinite path in category must to have a colimit:

- Let $config_1 \to config_2 \to config_3 \to \ldots$ be an infinite path (diagram $\mathcal{D}$) in category;
- object $config_\perp$ is a colimit of diagram $\mathcal{D}$

$$\text{colimit } \mathcal{D} = \bigcirc_{i \in N} [\![next]\!]([\![\texttt{while } b \texttt{ do } S; S^*]\!], m, i^*, o^*)$$

if from every object exists a morphism into this object.



Coalgebra

- We have constructed base category $\mathscr{C}onfig$, which objects are configurations $config$ and morphisms are $[\![next]\!], [\![input]\!], [\![output]\!]$.
- We construct an appropriate form of polynomial endofunctor which will characterize this kind of systems:

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

- Coalgebra is defined as $c : X \to FX$ in general, we assign our state space and morphisms, then a coalgebra for programs in language *E-Jane* has a form:

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

Coalgebra models the behavior of a program such that in each step one of the following alternatives occurs:

- $Q(\mathbf{Config}) = 1$ – program aborts,
- $Q(\mathbf{Config}) = \mathbf{Config}$ elaborates a declaration/executes a statement,
- $Q(\mathbf{Config}) = O \times \mathbf{Config}$ provides an output value,
- $Q(\mathbf{Config}) = \mathbf{Config}^I$ takes an input value.

Example

We consider a program in language *E-Jane*:

$$
\begin{array}{l}
\texttt{var } x; \texttt{var } y;\\
\texttt{input } x; \texttt{input } y;\\
\texttt{if } x <= y \texttt{ then}\\
\quad \texttt{begin}\\
\quad\; \texttt{var } z;\\
\quad\; z := x; x := y; y := z;\\
\quad \texttt{end}\\
\; \texttt{else skip};\\
\texttt{print } x
\end{array}
$$

For simplicity we introduce the following substitutions:

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

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

Example

An initial configuration is $\mathit{config}_0 = ([\![D_1; D_2; S_1; S_2; S_3; S_4]\!], m_0)$.
Every application of functor Q represents one step of program execution:

$$
\begin{aligned}
Q(\mathit{config}_0) &= [\![\mathit{next}]\!](\mathit{config}_0) = \mathit{config}_1 = \\
&= ([\![D_2; S_1; S_2; S_3; S_4]\!], [\![\mathtt{var}\ x]\!]m_0, (\mathbf{3}, \mathbf{5}), \varepsilon), \\[1em]
Q(\mathit{config}_1) &= [\![\mathit{next}]\!](\mathit{config}_1) = \mathit{config}_2 = \\
&= ([\![S_1; S_2; S_3; S_4]\!], [\![\mathtt{var}\ y]\!]m_1, (\mathbf{3}, \mathbf{5}), \varepsilon), \\[1em]
Q(\mathit{config}_2) &= [\![\mathit{read}]\!](\mathit{config}_2) = \mathit{config}_3 = \\
&= ([\![S_2; S_3; S_4]\!], m_3, (\mathbf{5}), \varepsilon), \\[1em]
Q(\mathit{config}_3) &= [\![\mathit{read}]\!](\mathit{config}_3) = \mathit{config}_4 = \\
&= ([\![S_3; S_4]\!], m_4, \varepsilon, \varepsilon),
\end{aligned}
$$

Example

$$\begin{aligned}
Q(\mathit{config}_4) &= [\![\mathit{next}]\!](\mathit{config}_4) = \mathit{config}_5 = \\
&= ([\![\texttt{begin var}\ z; z := x; x := y; y := z\ \textbf{end}; S_4]\!], m_4, \varepsilon, \varepsilon),
\end{aligned}$$

$$\begin{aligned}
Q(\mathit{config}_5) &= [\![\mathit{next}]\!](\mathit{config}_5) = \mathit{config}_6 = \\
&= ([\![\texttt{var}\ z; z := x; x := y; y := z\ \textbf{end}; S_4]\!], m_5,
\end{aligned}$$

$$\begin{aligned}
Q(\mathit{config}_6) &= [\![\mathit{next}]\!](\mathit{config}_6) = \mathit{config}_7 = \\
&= ([\![z := x; x := y; y := z\ \textbf{end}; S_4]\!], m_6, \varepsilon, \varepsilon),
\end{aligned}$$

$$\begin{aligned}
Q(\mathit{config}_7) &= [\![\mathit{next}]\!](\mathit{config}_7) = \mathit{config}_8 = \\
&= ([\![x := y; y := z\ \textbf{end}; S_4]\!], m_7, \varepsilon, \varepsilon),
\end{aligned}$$

Example

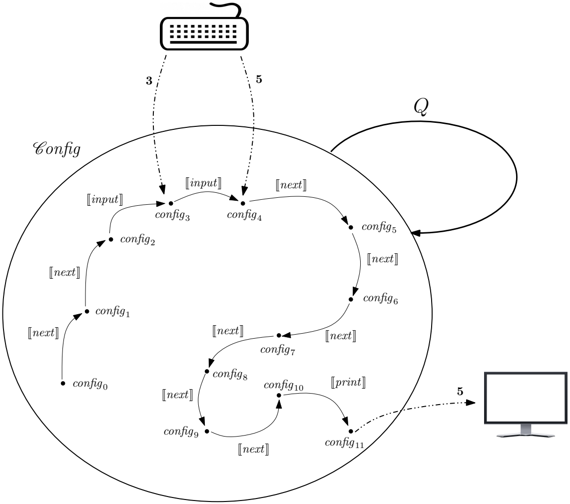
$$
\begin{aligned}
Q(\mathit{config}_8) \quad &= [\![\mathit{next}]\!](\mathit{config}_8) = \mathit{config}_9 = \\
&= ([\![y := z\ \texttt{end};\ S_4]\!], m_8, \varepsilon, \varepsilon),
\end{aligned}
$$

$$
\begin{aligned}
Q(\mathit{config}_9) \quad &= [\![\mathit{next}]\!](\mathit{config}_9) = \mathit{config}_{10} = \\
&= ([\![\texttt{end};\ S_4]\!], m_9, \varepsilon, \varepsilon),
\end{aligned}
$$

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

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

Example



3
5
Q
𝒞onfig
⟦input⟧
⟦input⟧
⟦next⟧
config$_3$
config$_4$
config$_5$
config$_2$
⟦next⟧
⟦next⟧
config$_6$
config$_1$
⟦next⟧
⟦next⟧
⟦next⟧
config$_7$
config$_8$
config$_0$
config$_{10}$
⟦print⟧
⟦next⟧
5
config$_9$
⟦next⟧
config$_{11}$