Ministerstwo Nauki
i Szkolnictwa Wyższego

Organizacja IX Konferencji Modelowanie Matematyczne
w Fizyce i Technice (MMFT 2017)
- zadanie finansowane w ramach umowy 829/P-DUN/2017
ze środków Ministra Nauki i Szkolnictwa Wyższego przeznaczonych
na działalność upowszechniającą naukę.

Coalgebras for modelling behaviour

Valerie Novitzká, William Steingartner

Faculty of Electrical Engineering and Informatics
Technical University of Košice, Slovakia

September 21st, 2017

Behaviour of program systems

- The aim of programing is to force the computer to execute some actions and to generate a desired behaviour;
- the most important concept is a **state** - an abstraction of computer memory;
- execution of a program means a change of states;
- states are often hidden from observer;
- the aim of behavioural theory is to determine a relation between internal states and observable values;
- as a formal model coalgebras are used to provide observable behaviour of program systems;

Coalgebra

A coalgebra is defined as a mapping

$$c : \mathbf{State} \to Q(\mathbf{State});$$

where

- $\mathbf{State}$ is the representation of states, state space;
- $Q : \mathscr{C}_{\mathrm{State}} \to \mathscr{C}_{\mathrm{State}}$ is a polynomial endofunctor over a category of state representations.

To construct a coalgebra of a system

- we start with a signature $State$ of a state space specifying types and operations;
- we construct a base category $\mathscr{C}_{\mathrm{State}}$ of state representations from a set $\mathbf{State}$, where morphisms are transitions (state changes);
- we construct a polynomial endofunctor Q over a category indicated by a given signature;
- we define a coalgebra $c : \mathbf{State} \to Q(\mathbf{State})$.

Simple language $\mathscr{J}ane$

We introduce for $\mathscr{J}ane$ the following syntactic domains:

- $n \in \mathbf{Num}$ for digit strings;
- $x \in \mathbf{Var}$ for variables names;
- $e \in \mathbf{Aexpr}$ for arithmetic expressions;
- $b \in \mathbf{Bexpr}$ for Boolean expressions;
- $S \in \mathbf{Statm}$ for statements.

Syntax:

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

State space

A basic concept in coalgebraic approach is a state specified by the **signature**:

$$\Sigma_{State} = \begin{array}{ll} \underline{types:} & State, Var, Value \\ \underline{opns:} & init :\to State \\ & get : Var, State \to Value \\ & next : Statm, State \to State \end{array}$$

Representation:

- we assign to the syntactic domain Val a set:

$$\mathbf{Value} = \mathbf{Z} \cup \{\bot\}\,;$$

- we assign to the type Var a countable set $\mathbf{Var}$ of variable names;
- our representation of an element of type $State$ has to express a variable name together with its value:

$$s : \mathbf{Var} \to \mathbf{Value};$$

  where

$$s = \langle (x, v_1), \ldots, (z, v_n) \rangle$$

- special states are the initial state $s_0 = [\![init]\!]$ and undefined state $s_\bot = \langle (\bot, \bot) \rangle$;
- state representations form the set $\mathbf{State}$ - state space.

Category

We construct a base category $\mathscr{C}_{State}$ of states, where

- category objects are state representations from $\mathbf{State}$; and
- category morphisms are transitions defining changes of states.

We define the representation of $next$, the transition function $[\![next]\!]$:

$$[\![next]\!] : \mathbf{Statm} \to (\mathbf{State} \to \mathbf{State}),$$

that returns for a statement S

$$[\![next]\!][\![S]\!] : \mathbf{State} \to \mathbf{State}$$

the next state obtained from the execution of the first step of a statement S.

To be $\mathscr{C}_{State}$ a category we require that every infinite path (composition of morphisms) has a colimit.

Transition function

Transition function is defined for statements of the language $\mathscr{J}ane$ as follows:

$$
[\![next]\!][\![S]\!](s) = \begin{cases}
s' = s\,[x \mapsto [\![e]\!]s] & \text{if } S = x := e; \\
s & \text{if } S = \texttt{skip} \\
 & \text{or } S = \texttt{while}\ b\ \texttt{do}\ S \text{ and } [\![b]\!]s = \textbf{false}; \\
[\![next]\!][\![S_1'; S_2]\!](s') & \text{if } S = S_1; S_2 \text{ and } \langle S_1; S_2, s\rangle \Rightarrow \left\langle S_1'; S_2, s'\right\rangle; \\
[\![next]\!][\![S_2]\!](s') & \text{if } S = S_1; S_2 \text{ and } \langle S_1; S_2, s\rangle \Rightarrow \left\langle S_2, s'\right\rangle; \\
[\![next]\!][\![S_1]\!](s) & \text{if } S = \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \text{ and } [\![b]\!]s = \textbf{true}; \\
[\![next]\!][\![S_2]\!](s) & \text{if } S = \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \text{ and } [\![b]\!]s = \textbf{false}; \\
[\![next]\!][\![S; \texttt{while}\ b\ \texttt{do}\ S]\!](s) & \text{if } S = \texttt{while}\ b\ \texttt{do}\ S\ \texttt{and}\ [\![b]\!]s = \textbf{true}; \\
abort(s) & \text{otherwise,}
\end{cases}
$$

where $abort$ is a unique morphism which sends any state to the undefined state $s_\perp$:

$$abort : s \dashrightarrow s_\perp$$

Polynomial endofunctor

Now we construct the polynomial endofunctor indicated by Σ_{State} as

$$Q : \mathscr{C}_{State} \rightarrow \mathscr{C}_{State}.$$

For our purposes we define a functor

$$Q(\mathbf{State}) = 1 + \mathbf{State}.$$

We define this functor for objects and morphisms in $\mathscr{C}_{State}$ as follows:

$$\begin{array}{ll} Q(s) = & s_{\perp} + [\![next]\!][\![S]\!]s, \\ Q([\![next]\!][\![S]\!]) = & abort + [\![next]\!][\![S]\!]. \end{array}$$

Q-coalgebra for $\mathscr{J}ane$

A Q-coalgebra, also called coalgebra of type Q or Q-system, is a pair

$$(\mathbf{State}, [\![next]\!][\![S]\!]),$$

where $\mathbf{State}$ is a state space of the coalgebra and $[\![next]\!][\![S]\!]$ is the structure map of the coalgebra on $\mathbf{State}$:

$$[\![next]\!][\![S]\!] : \mathbf{State} \to Q(\mathbf{State}).$$

Example

We consider a simple program in $\mathscr{J}ane$:

$$\begin{array}{l} z := 0; \\ \texttt{while } (y \le x) \texttt{ do } (z := z + 1; x := x - y); \end{array}$$

and let the initial state be $s_0 = [x \mapsto \mathbf{17}, y \mapsto \mathbf{5}]$.

We construct over a category $\mathscr{C}_{State}$ a polynomial endofunctor

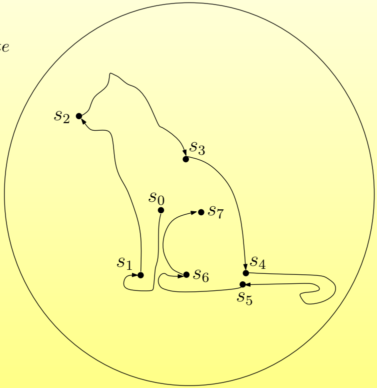
$$Q(\mathbf{State}) = 1 + \mathbf{State}$$

defined for objects and morphisms by

$$\begin{array}{ll} Q(s) = & s_\perp + [\![next]\!][\![S]\!]s, \\ Q([\![next]\!][\![S]\!]) = & abort + [\![next]\!][\![S]\!]. \end{array}$$

Example - continuation

$\mathscr{C}_{State}$



$$
\begin{aligned}
Q(s_0) = \quad & 1 + [\![next]\!][\![S_0]\!]s_0 \\
& = [\![next]\!][\![S_1; S_2]\!]s_0 \\
& = [\![next]\!][\![S_2]\!]s_1 \\
& = [\![next]\!][\![z := z + 1; x := x - y; S_2]\!]s_1 \\
& = [\![next]\!][\![x := x - y; S_2]\!]s_2 \\
& = [\![next]\!][\![S_2]\!]s_3 \\
& = [\![next]\!][\![z := z + 1; x := x - y; S_2]\!]s_3 \\
& = [\![next]\!][\![x := x - y; S_2]\!]s_4 \\
& = [\![next]\!][\![S_2]\!]s_5 \\
& = [\![next]\!][\![z := z + 1; x := x - y; S_2]\!]s_5 \\
& = [\![next]\!][\![x := x - y; S_2]\!]s_6 \\
& = [\![next]\!][\![S_2]\!]s_7 \\
& = s_7
\end{aligned}
$$

$$
\begin{aligned}
s_0 &= \langle (x, \mathbf{17}), (y, \mathbf{5}) \rangle & & \\
s_1 &= \langle (x, \mathbf{17}), (y, \mathbf{5}), (z, \mathbf{0}) \rangle & [\![y \le x]\!]s_1 &= \mathbf{true} \\
s_2 &= \langle (x, \mathbf{17}), (y, \mathbf{5}), (z, \mathbf{1}) \rangle & & \\
s_3 &= \langle (x, \mathbf{12}), (y, \mathbf{5}), (z, \mathbf{1}) \rangle & [\![y \le x]\!]s_3 &= \mathbf{true} \\
s_4 &= \langle (x, \mathbf{12}), (y, \mathbf{5}), (z, \mathbf{2}) \rangle & & \\
s_5 &= \langle (x, \mathbf{7}), (y, \mathbf{5}), (z, \mathbf{2}) \rangle & [\![y \le x]\!]s_5 &= \mathbf{true} \\
s_6 &= \langle (x, \mathbf{7}), (y, \mathbf{5}), (z, \mathbf{3}) \rangle & & \\
s_7 &= \langle (x, \mathbf{2}), (y, \mathbf{5}), (z, \mathbf{3}) \rangle & [\![y \le x]\!]s_7 &= \mathbf{false}
\end{aligned}
$$

Object oriented programming

Basic concepts in object oriented programming are:

- **classes:**
  - class specification is like a signature specifying methods;
  - it determines also an interface to a program;
  - for implementation of methods some constraints (assertions) are given;
  - the essentials are in a class specification;
  - the particulars are in a class implementation;
- **objects:**
  - deal with specific tasks;
  - coordination and communication is realized via sending of messages;
  - objects have private data accessible only by methods;
  - objects are grouped into classes;
  - have local states accessible by the object methods;
  - combine data structure with behaviour.

Class and object

Class

Object

Coalgebra for OOP

A class specification is a named structure consisting of a tuple of methods of the form:

$$m_i : X \times A_i \to B_i + C_i \times X, \text{ for } i = 1, \ldots, n,$$

where

- X is a state space specification;
- A_i are inputs;
- B_i and C_i are outputs of a method m_i.

A polynomial endofunctor has then a form:

$$Q(X) = \prod_{i=1}^{n} (B_i + C_i \times X)^{A_i}.$$

- If $C_i = \emptyset$, the associated method yields observable element from B_i, but does not change a local state;
- if $C_i \neq \emptyset$, the associated method changes a local state.

Let $\mathbf{State}$ be an interpretation of objects local states, with elements $o \in \mathbf{State}$.
A coalgebra

$$\mathbf{m} = \langle m_1, \ldots, m_n \rangle$$

is defined by:

$$\mathbf{m} : \mathbf{State} \to Q(\mathbf{State}),$$

Example

Consider a class for bank accounts with the methods:

$$\begin{aligned} balance: &\quad X \to \mathbb{R} \\ change: &\quad X \times \mathbb{R} \to X, \end{aligned}$$

with the assertion:

$$s.change(a).balance = s.balance + a$$

for $s \in X$ and $a \in \mathbb{R}$.

We interpret state space X as the set of finite sequences of reals $\mathbb{R}^*$, i.e. each element (object of this class) $o \in \mathbb{R}^*$ is an account of the form

$$o = \langle a_0, a_1, \ldots, a_n \rangle.$$

The polynomial endofunctor is

$$Q(\mathbb{R}^*) = \mathbb{R} \times (\mathbb{R}^*)^{\mathbb{R}}$$

The methods $balance$ and $change$ together form a coalgebra

$$\langle balance, change \rangle : \mathbb{R}^* \to Q(\mathbb{R}^*).$$

Then the methods for an element $o \in \mathbb{R}^*$ are

$$o.balance = a_0 + a_1 + \cdots + a_n \qquad o.change(a) = \langle a_0, a_1, \ldots, a_n, a \rangle.$$

Example -continuation

The assertion

$$o.change(a).balance = o.balance + a$$

is always valid.

The empty account is denoted by the empty sequence $\langle\rangle$.

Let now a state be the sequence

$$o = \langle 3.2, 5.3, -1.4\rangle.$$

Then

$$\begin{array}{lll} o.balance = & 3.2 + 5.3 - 1.4 = 7.1 & \text{and} \\ o.change(8.7) = & \langle 3.2, 5.3, -1.4, 8.7\rangle. & \end{array}$$

Example -continuation

We can consider another interpretation, which

- keeps a record of changes;
- makes additions immediately.

The interpretation of a state space X is a set $\mathbb{R}^+$ of non empty sequences of reals. For an element (object)

$$o' = \langle a_1, \dots, a_n \rangle \in \mathbb{R}^+$$

the methods are

$$o'.balance = a_n \quad \text{and} \quad o'.change(a) = \langle a_1, \dots, a_n, a_n + a \rangle.$$

The initial state is now $\langle 0.0 \rangle$.

A coalgebra is

$$\langle balance, change \rangle : \mathbb{R}^+ \to Q(\mathbb{R}^+)$$

and the methods also satisfy the assertion

$$o'.change(a).balance = o'.balance + a.$$

Component based programming

Component based programming is about

- how to create an application program from prefabricated components together;
- with new software providing both glue between the components and new functionality.

A component

- is an independent deployable entity;
- it interacts with the environment by typed ports specified in its interface;
- it has no observable state, its initial state is established after its deployment.
- can be generic, substitution of its parameters by appropriate arguments (of proper types) enable its using for different purposes.

The typed ports

- serve as end points interactions;
- they enable transfer of data of some type in required direction;
- cooperation between components can be performed only trough ports of corresponding types.

From components to an application

$\Downarrow$ composition

Coalgebra for components

To define coalgebras for components we denote by

- I a set of typed input ports;
- O a set of typed output ports.

Then an interface of a component is a pair

$$(I, O)$$

and a component is an arrow

$$comp : I \to O.$$

To ensure genericity, we use a strong monad B over a base category and then a coalgebra of a component is

$$c_{comp} : X_{comp} \times I_{comp} \to B(X_{comp} \times O)$$

For each state $s \in X_{comp}$ the behaviour is organized as a tree because it depends on the sequences of input values. In this tree:

- elements of O are nodes;
- elements of I are labels of edges.

Example

Consider a buffer $Buffer$ as a component that stores input data elements ($Message$) and returns them in responding to request. This component has one input port and one output port and operations:

$$
\begin{aligned}
put: &\quad Message \times Buffer \to Buffer \\
pick: &\quad Buffer \to Message \times Buffer
\end{aligned}
$$

Let

- M be a type of messages;
- M^* represents a buffer;
- $I = M + 1$ represents inputs;
- $O = 1 + M$ represents outputs, where 1 stands for nullary datatype.

The polynomial endofunctor is then

$$Q(M^*) = (M^* \times O)^I$$

and the coalgebra for this component is

$$c : M^* \to Q(M^*).$$

Thank you for your attention