*Bull. Nov. Comp. Center*, Comp. Science, 40 (2016), 15–35


# Method of the development of ontological operational semantics for imperative programming languages*

I.S. Anureev, I.V. Maryasov, I.N. Mikhailov

**Abstract.** The paper presents a method of the development of operational semantics for imperative programming languages. It is based on the ontological approach to formal programming language specification implemented by information transition systems and conceptual transition systems. The method is illustrated by a fragment of the C language.

**Keywords:** operational semantics, ontological operational semantics, information transition system, conceptual transition system, C.

## 1. Introduction

Currently, there are tens of thousands of computer languages (programming languages, specification languages, domain-specific languages, scripting languages, markup languages, modeling languages, knowledge representation languages, and so on), and the creation of new computer languages continues. Formal methods are a means to ensure the correct and effective use of computer languages [2]. Application of formal methods to texts in these languages requires a formalization of these texts. Therefore, the development of formal semantics for computer languages is an important problem.

Operational semantics describing the abstract machine (AM[PL] for short) executing the instructions of a programming language (PL) on a set of states is generally used to formalize the language. The methodology for the development of the ontological operational semantics of PLs [1] based on conceptual transition systems (CTSs) was proposed in [3]. Like abstract state machines [4] (ASMs), CTSs allow states to be described in detail, but both these formalisms do not allow transitions to be described in detail. The languages AsmL [5] and XasM [6] based on ASMs are general-purpose languages for the specification of computer systems. They are not DSLs oriented to the description of transitions in AMs specifying operational semantics of PLs.

In this paper, we propose a method to elaborate this methodology. The development of operational semantics of a PL based on the method consists of two main stages. In the first stage, AM[PL] is described in the form of

*Partially supported by RFBR under Grant 15-01-05974.

an information transition system [7] (ITS[PL]). ITSs are information models for a preliminary rough representation of the AMs structure. The purpose of the informal description is to classify the objects of AM[PL] as states, state objects, information queries, query objects, answers, and answer objects of ITS[PL]. The states, queries and answers of ITS[PL] describe the states, instructions and returning values of AM[PL], respectively. The state objects describe the objects observable in the states of AM[PL] (in particular, elements and substates of the states of AM[PL]). The query objects and answer objects of ITS[PL] describe the elements constituting the instructions and returning values of AM[PL], respectively. In the second stage, the formal conceptual information transition model [7] (CITM[PL]) of ITS[PL] in the language CTSL (Conceptual Transition System Language) [7] is defined. CITM[PL] includes representations of states, state objects, queries, query objects, answers, and answer objects in CTSL in the form of the conceptual structures (elements, conceptuals, concepts, attributes, individuals, conceptual states, and conceptual configurations) of CTSL, and an extension of CTSL [7] describing the operational semantics of query representations. Thus, the operational semantics of PL is defined in CTSL in conceptual (ontological) terms. Therefore, it is called the ontological operational semantics of PL [1].

The paper is organized as follows. Notions and denotations used in this paper are given in Section 2. The operational semantics method for programming languages based on CTSs is described in Section 3. Sections from 4 to 8 describe the stages of the development of operational semantics based on the method for a fragment of the C language (CF).

## 2. Preliminaries

Let $O_b$ be the set of objects considered in this paper. Let $S_t$ be a set of sets. Assume that $I_{nt}$, $N_t$, $N_{t0}$ and $B_l$ are the sets of integers, natural numbers, natural numbers with zero, and boolean values $true$ and $false$, respectively. Let the names of sets be represented by capital letters, possibly with subscripts, and the elements of sets be represented by the corresponding small letters, possibly with extended subscripts. For example, $i_{nt}$ and $i_{nt.1}$ are elements of $I_{nt}$.

Let $s_{t.(*)}$, $s_{t.\{*\}}$, and $s_{t.*}$ denote the sets of sequences of the forms $(o_{b.1}, \ldots, o_{b.n_{t0}})$, $\{o_{b.1}, \ldots, o_{b.n_{t0}}\}$, and $o_{b.1}, \ldots, o_{b.n_{t0}}$ from elements of $s_t$.

The terms used in the paper are context-dependent. Contexts have the form $[\![o_{b.*}]\!]$, where the elements of $o_{b.*}$ called embedded contexts have the form $l_b{:}o_b$, $l_b{:}$ or $o_b$. The elements of the set $L_b$ are called labels. Let $o_b[\![o_{b.*}]\!]$ denote the object $o_b$ in the context $[\![o_{b.*}]\!]$.

Let $und$ denote the undefined value. Let $F_n$ be a set of functions. Assume that $[f_n\ a_{rg.*}]$ is the application of $f_n$ to $a_{rg.*}$. Let $[support\ f_n]$ denote the support in $[\![f_n]\!]$, i.e., $[support\ f_n] = \{a_{rg} : [f_n\ a_{rg}] \neq und\}$.

Let $A_{rg}$ and $V_l$ be the sets of arguments and values. An object $u_p$ of the form $a_{rg} : v_l$ is called an update. The objects $a_{rg}$ and $v_l$ are called the argument and the value in $[\![u_p]\!]$. Let $U_p$ be a set of updates.

Let $[f_n\ u_p]$ denote the function $f_{n.1}$ such that $[f_{n.1}\ a_{rg}] = [f_n\ a_{rg}]$ if $a_{rg} \neq a_{rg}[\![u_p]\!]$ and $[f_{n.1}\ a_{rg}[\![u_p]\!]] = v_l[\![u_p]\!]$. Let $[f_n\ u_p, u_{p.*n_t}]$ be a shortcut for $[[f_n\ u_p]\ u_{p.*n_t}]$; $[f_n\ a_{rg}.a_{rg.1}. \ldots .a_{rg.n_t} : v_l]$, for $[f_n\ a_{rg} : [[f_n\ a_{rg}]\ a_{rg.1}. \ldots .a_{rg.n_t} : v_l]]$; and $[u_{p.*}]$ for $[f_n\ u_{p.*}]$, where $[support\ f_n] = \emptyset$.

Let $S_{tt}$, $O_{b.s}$, $Q_r$, $O_{b.q}$, $A_{ns}$ and $O_{b.a}$ be the sets of states, state objects, queries, query objects, answers and answer objects. Let $E_l$, $C_{nf}$ and $S_{tr}$ be the sets of elements, conceptual configurations and conceptual structures (elements, conceptuals, concepts, attributes, individuals, conceptual states, conceptual configurations) of CTSL. Let $L_n$ be a set of programming languages.

## 3. The method of the development of ontological operational semantics for imperative programming languages

The development of operational semantics of $l_n$ in CTSL includes the following stages:

1. Describe AM$[l_n]$ in the form of an ITS$[l_n]$.

   (a) Describe the sets of proper state objects. An object $o_{b.s}$ is a proper state object if $o_{b.s} \notin S_{tt}$.

   (b) Describe the sets of states.

   (c) Describe the sets of proper answer objects. An object $o_{b.a}$ is a proper answer object if $o_{b.a} \notin A_{ns}$.

   (d) Describe the sets of answers.

   (e) Describe the sets of proper query objects. An object $o_{b.q}$ is a proper query object if $o_{b.q} \notin Q_r$.

   (f) Describe the sets of queries.

2. Define CITM$[l_n]$ of ITS$[l_n]$. The CTS of CITM$[l_n]$ denoted by CTSL$[l_n]$ is an extension of CTSL. The extension defines the operational semantics of $l_n$ in CTSL, and the model describes the correspondence between the objects of AM$[l_n]$ and conceptual structures of CTSL.

   (a) Define the set of conceptual structures of CTSL representing proper state objects of ITS$[l_n]$ in CTSL$[l_n]$.

   (b) Define the set of conceptual configurations representing the states of ITS$[l_n]$ in CTSL$[l_n]$. The set of conceptual structures representing the state objects of ITS$[l_n]$ is called an ontology of $l_n$ in CTSL.

Therefore, the operational semantics of $l_n$ in CTSL is called an ontological operational semantics of $l_n$.

(c) Define the set of elements representing the proper answer objects of ITS[$l_n$] in CTSL[$l_n$].

(d) Define the set of elements representing the answers of ITS[$l_n$] in CTSL[$l_n$].

(e) Define the set of elements in $s_{s.t.c}$ representing the proper query objects of ITS[$l_n$] in CTSL[$l_n$].

(f) Define the set of defined elements in CTSL[$l_n$] representing the queries of ITS[$l_n$] in CTSL[$l_n$].

(g) Define the element interpretation order [8], exogenous transition order [7] and endogenous transition order [7] in CTSL[$l_n$]. They describe the order of execution of element interpretations and transitions.

Let $r_{p.s} \in O_{b.s} \to S_{tr}$, $r_{p.q} \in O_{b.q} \to E_l$ and $r_{p.a} \in O_{b.a} \to E_l$ be the representation functions of state objects, query objects and answer objects of ITS[$l_n$] in CTSL. Let $r^-_{p.s}$, $r^-_{p.q}$ and $r^-_{p.a}$ be inverse functions of $r_{p.s}$, $r_{p.q}$ and $r_{p.a}$.

In the following sections, we apply the method to the development of ontological operational semantics for a fragment of the C language defined by an abstract machine AM[C].

## 4. Description of ITS[C]

The set of proper state objects of ITS[C] includes the objects of AM[C] such as names, types, addresses and their values, variables and their attributes (names, values, types, addresses), functions and their attributes (names, parameters, parameter types, return values, bodies), call levels, relative variable scopes.

A call level specifies the number of nested function calls. A relative variable scope specifies the number of block nesting. The scope 0 is associated with global variables, and the other scopes are associated with local variables.

The state of ITS[C] specifies the current variable scope, the current call level and relations between the following objects: addresses and their values, variables and their attributes, functions and their attributes.

The set of answers of ITS[C] includes the values of C types, the jumps initiated by jump statements and the program error message.

The set of queries of ITS[C] includes the instructions of AM[C] such as statements, expressions (built of variables, literals, and operators), declarations, conversions, and programs.

AM[C] contains extra instructions in addition to C instructions. The extra instructions includes C expressions extended by new literals such as pointer literals, array literals, structure literals, union literals and function literals representing the values of pointer types, array types, structure types, union types, and function types, respectively, and also the dynamic memory management instructions *new* and *delete* from the C-light language [9].

## 5. Proper state objects in CTSL[C]

A name is represented by an instance of the concept *name* defined by the rule

$(rule\ \ (x\ \ is\ \ name)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ normal))$.

The syntax and semantics of rules are defined in [7]. The predefined CTSL element $(e_l\ is\ normal)$ specifies that $e_l$ is a normal element [8]. Thus, names are represented by normal elements. Let $N_m = [content\ name]$. The object $[content\ c_{ncp}]$ denotes the content (the set of instances) of the concept $c_{ncp}$.

A label is represented by an instance of the concept *label* defined by the rule

$(rule\ \ (x\ \ is\ \ label)\ \ var\ \ (x)\ \ abn\ \ then$
$\ (x\ \ matches\ \ y :: label\ \ var\ \ (y)\ \ where\ \ (y\ \ is\ \ name)))$.

The predefined CTSL element $(e_l\ matches\ p_{tt}\ var\ (v_{r.*})\ where\ c_{nd})$ specifies that the element $e_l$ matches the pattern $p_{tt}$ with the variables $v_{r.*}$ and the condition $c_{nd}$ is true for the corresponding values of these variables. Let $L_b = [content\ label]$.

A type is represented by an instance of the concept *type−literal* defined by the rule

$(rule\ \ (x\ \ is\ \ type{-}literal)\ \ var\ \ (x)\ \ abn\ \ then$
$\ (x\ \ is\ \ basic{-}type)\ \ or\ \ (x\ \ is\ \ derived{-}type{-}literal)))$.

The predefined CTSL element $(c_{nd.1}\ or\ c_{nd.2})$ specifies the disjunction of the conditions $c_{nd.1}$ and $c_{nd.2}$. The false and true values are defined in CTSL as follows: the element *und* is the false value, and any element distinct from *und* is the true value.

A basic type is represented by an instance of the concept *basic−type* defined by the rule

$(rule\ \ (x\ \ is\ \ basic{-}type)\ \ var\ \ (x)\ \ abn\ \ then$
$\ (x :: q\ \ in :: set\ \ (int,\ \ float,\ \ ...) :: q))$,

where $int, float, ...$ is a sequence of all basic types of C. The predefined CTSL element $(e_l\ in\ ::\ set\ (e_{l.*}))$ specifies that $e_l$ is an element of the sequence $e_{l.*}$. The element of the form $e_l\ ::\ q$ is called a quoted element.

The value of the quoted element $e_l :: q$ in CTSL is defined as $e_l$. Let $T_{p.b} = [content\ basic{-}type]$.

A derived type is represented by an instance of the concept *derived−type−literal* defined by the rule

$(rule\ (x\ is\ derived{-}type{-}literal)\ var\ (x)\ abn\ then$
$\ (x\ is\ pointer{-}type{-}literal)\ or\ (x\ is\ array{-}type{-}literal)\ or$
$\ (x\ is\ structure{-}type{-}literal)\ or\ (x\ is\ function{-}type{-}literal))$.

An array type is represented by an instance of the concept *array−type−literal* defined by the rule

$(rule\ (x\ is\ array{-}type{-}literal)\ var\ (x)\ abn\ then$
$\ (x\ matches\ (array\ y)\ where\ (y\ is\ type{-}literal))$.

A structure type is represented by an instance of the concept *structure−type−literal* defined by the rule

$(rule\ (x\ is\ structure{-}type{-}literal)\ var\ (x)\ abn$
$\ then\ (x\ matches\ y :: structure{-}type\ var\ (y)))$.

Let $T_{p.s} = [content\ structure{-}type{-}literal]$.

A pointer type is represented by an instance of the concept *pointer−type−literal* defined by the rule

$(rule\ (x\ is\ pointer{-}type{-}literal)\ var\ (x)\ abn\ then$
$\ (x\ matches\ (pointer\ y)\ where\ (y\ is\ type{-}literal))$.

A function type is represented by an instance of the concept *function−type−literal* defined by the rule

$(rule\ (x\ is\ function{-}type)\ var\ (x)\ abn\ then$
$\ (x\ matches\ (function\ y\ z)\ var\ (y,\ z)$
$\ \ where\ ((y\ is\ (sequence\ type{-}literal))\ and\ (z\ is\ type{-}literal))))$.

The predefined CTSL element $(c_{nd.1}\ and\ c_{nd.2})$ specifies the conjunction of the conditions $c_{nd.1}$ and $c_{nd.2}$.

A type in $[\![s_{tt}[\![\mathrm{ITS[C]}]\!]]\!]$ is represented by an instance of the concept *type* in $[\![[r_{p.s}\ s_{tt}]]\!]$ defined by the rule

$(rule\ (x\ is\ type)\ var\ (x)\ abn\ then$
$\ (x\ is\ basic{-}type)\ or\ (x\ is\ derived{-}type)))$.

A derived type in $[\![s_{tt}[\![\mathrm{ITS[C]}]\!]]\!]$ is represented by an instance of the concept *derived−type* in $[\![[r_{p.s}\ s_{tt}]]\!]$ defined by the rule

$(rule\ (x\ is\ derived{-}type)\ var\ (x)\ abn\ then$
$\ (x\ is\ pointer{-}type)\ or\ (x\ is\ array{-}type)\ or$
$\ (x\ is\ structure{-}type)\ or\ (x\ is\ function{-}type)))$.

An array type in $[\![s_{tt}[\![\text{ITS[C]}]\!]]\!]$ is represented by an instance of the concept $array\text{−}type$ in $[\![[r_{p.s}\ s_{tt}]]\!]$ defined by the rule

$(rule\ \ (x\ \ is\ \ array\text{−}type)\ \ var\ \ (x)\ \ abn\ \ then$
$\ (x\ \ matches\ \ (array\ \ y)\ \ where\ \ (y\ \ is\ \ type)).$

The conceptual $(0 : t_{p.s}, 1 : structure\text{−}type)$ in $[\![c_{nf}]\!]$ represents the structure type $t_{p.s}$ in $[\![[r_{p.s}\ c_{nf}]]\!]$. A structure type $t_{p.s}$ is a structure type in $c_{nf}$ if $[c_{nf}\ (0 : t_{p.s}, 1 : structure\text{−}type)] \neq und$. The conceptual $(-1 : body, 0 : t_{p.s}, 1 : structure\text{−}type)$ in $c_{nf}$ represents a body, fields and their types in $[\![t_{p.s}, [r_{p.s}\ c_{nf}]]\!]$. An element $b_d$ is a body in $[\![t_{p.s}, c_{nf}]\!]$ if $[c_{nf}\ (-1 : body, 0 : t_{p.s}, 1 : structure\text{−}type)] = b_d$. The element $b_d$ is an attribute element. The attribute element with the attributes $a_{tt.1}$, ..., $a_{tt.n_{t0}}$ has the form $(a_{tt.1} : v_{l.1}, ..., a_{tt.n_{t0}} : v_{l.n_{t0}})$, where $v_{l.1}$, ..., $v_{l.n_{t0}}$ are the values of the attributes $a_{tt.1}$, ..., $a_{tt.n_{t0}}$. The attribute element can be considered as a function mapping the attributes to their values. A field $f_l$ is a field in $[\![t_{p.s}, c_{nf}]\!]$ if $[[c_{nf}\ (-1 : body, 0 : t_{p.s}, 1 : structure\text{−}type)]\ f_l] \neq und$. A type $t_p$ is a type in $[\![f_l, t_{p.s}, c_{nf}]\!]$ if $[[c_{nf}\ (-1 : body, 0 : t_{p.s}, 1 : structure\text{−}type)]\ f_l] = t_p$.

A structure type in $[\![s_{tt}[\![\text{ITS[C]}]\!]]\!]$ is represented by an instance of the concept $structure\text{−}type$ in $[\![[r_{p.s}\ s_{tt}]]\!]$ defined by the rule

$(rule\ \ (x\ \ is\ \ structure\text{−}type)\ \ var\ \ (x)\ \ abn$
$\ then\ \ ((x\ \ is\ \ structure\text{−}type\text{−}literal)\ \ and\ \ (0 : x,\ \ 1 : structure\text{−}type))).$

A field in $[\![[[r^{-}_{p.s}\ t_{p.s}]]\!]$ in $[\![s_{tt}[\![\text{ITS[C]}]\!]]\!]$ is represented by an instance of the concept $(field\ in\ t_{p.s})$ in $[\![[r_{p.s}\ s_{tt}]]\!]$ defined by the rule

$(rule\ \ (x\ \ is\ \ (field\ \ in\ \ y))\ \ var\ \ (x,\ \ y)\ \ abn\ \ then$
$\ ((x\ \ is\ \ field\text{−}literal)\ \ and$
$\ \ ((-1 : body,\ \ 0 : y,\ \ 1 : structure\text{−}type)\ \ ..\ \ x))).$

The predefined CTSL element $(e_{l.a}\ ..\ a_{tt})$ specifies the value of the attribute $a_{tt}$ of the attribute element $e_{l.a}$.

A pointer type in $[\![s_{tt}[\![\text{ITS[C]}]\!]]\!]$ is represented by an instance of the concept $pointer\text{−}type$ in $[\![[r_{p.s}\ s_{tt}]]\!]$ defined by the rule

$(rule\ \ (x\ \ is\ \ pointer\text{−}type)\ \ var\ \ (x)\ \ abn\ \ then$
$\ (x\ \ matches\ \ (pointer\ \ y)\ \ where\ \ (y\ \ is\ \ type)).$

A function type in $[\![s_{tt}[\![\text{ITS[C]}]\!]]\!]$ is represented by an instance of the concept $function\text{−}type$ in $[\![[r_{p.s}\ s_{tt}]]\!]$ defined by the rule

$(rule\ \ (x\ \ is\ \ function\text{−}type)\ \ var\ \ (x)\ \ abn\ \ then$
$\ (x\ \ matches\ \ (function\ \ y\ \ z)\ \ var\ \ (y,\ \ z)$
$\ \ where\ \ ((y\ \ is\ \ (sequence\ \ type))\ \ and\ \ (z\ \ is\ \ type)))).$

The predefined CTSL element $(e_l\ is\ (sequence\ c_{ncp}))$ specifies that $e_l$ is a sequence of the instances of the concept $c_{ncp}$.

A variable is represented by an instance of the concept $variable{-}literal$ defined by the rule

$$(rule\ \ (x\ \ is\ \ variable{-}literal)\ \ abn\ \ var\ \ (x)\ \ then$$
$$(x\ \ matches\ \ y :: variable\ \ var\ \ (y)\ \ where\ \ (y\ \ is\ \ name))).$$

Let $V_r = [content\ variable{-}literal]$.

A relative variable scope is represented by an instance of the concept $scope$ defined by the rule

$$(rule\ \ (x\ \ is\ \ scope)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ nat0)).$$

The predefined CTSL element $(e_l\ is\ nat0)$ specifies that $e_l \in N_{t0}$. Let $S_{cp} = [content\ scope]$.

An array is represented by an instance of the concept $array{-}literal$ defined by the rule

$$(rule\ \ (x\ \ is\ \ array{-}literal)\ \ var\ \ (x)\ \ abn$$
$$then\ \ (x\ \ matches\ \ y :: array\ \ var\ \ (y)\ \ where\ \ (y\ \ is\ \ nat))).$$

The predefined CTSL element $(e_l\ is\ nat)$ specifies that $e_l \in N_t$. Let $A_{rr} = [content\ array{-}literal]$.

A structure is represented by an instance of the concept $structure{-}literal$ defined by the rule

$$(rule\ \ (x\ \ is\ \ structure{-}literal)\ \ var\ \ (x)\ \ abn$$
$$then\ \ (x\ \ matches\ \ y :: structure\ \ var\ \ (y)\ \ where\ \ (y\ \ is\ \ nat))).$$

Let $S_{trc} = [content\ structure{-}literal]$.

A field is represented by an instance of the concept $field{-}literal$ defined by the rule

$$(rule\ \ (x\ \ is\ \ field{-}literal)\ \ var\ \ (x)\ \ abn\ \ then$$
$$(x\ \ matches\ \ y :: field\ \ var\ \ (y))).$$

Let $F_l = [content\ field{-}literal]$.

A function is represented by an instance of the concept $function{-}literal$ defined by the rule

$$(rule\ \ (x\ \ is\ \ function{-}literal)\ \ var\ \ (x)\ \ abn\ \ then$$
$$(x\ \ matches\ \ y :: function\ \ var\ \ (y)\ \ where\ \ (y\ \ is\ \ name))).$$

Let $F_n = [content\ function{-}literal]$.

A formal argument of a function is represented by an instance of the concept $argument$ defined by the rule

$(rule\ \ (x\ \ is\ \ argument)\ \ var\ \ (x)\ \ abn\ \ then$
$\ (x\ \ is\ \ variable{-}literal))$.

Thus, formal function arguments are represented by variables. Let $A_{rg} = [content\ argument]$.

A call level is represented by an instance of the concept *call−level* defined by the rule

$(rule\ \ (x\ \ is\ \ call{-}level)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ nat0))$.

Let $L_{v.c} = [content\ call{-}level]$.

A pointer is represented by an instance of the concept *pointer−literal* defined by the rule

$(rule\ \ (x\ \ is\ \ pointer{-}literal)\ \ var\ \ (x)\ \ abn\ \ then$
$\ ((x\ \ is\ \ typed{-}pointer{-}literal)\ \ or\ \ (x\ \ is\ \ variable{-}pointer{-}literal)\ \ or$
$\ \ (x\ \ is\ \ function{-}pointer{-}literal)\ \ or\ \ (x\ \ is\ \ array{-}pointer{-}literal)\ \ or$
$\ \ (x\ \ is\ \ structure{-}pointer{-}literal)\ \ or\ \ (x :: q\ \ =\ \ null)))$.

Let $P_n = [content\ pointer{-}literal]$. These pointers are smart, i.e., they 'know' their types and their connections with variables, arrays, structures, unions, and functions.

The concept *typed−pointer−literal* is defined by the rule

$(rule\ \ (x\ \ is\ \ typed{-}pointer{-}literal)\ \ var\ \ (x)\ \ abn$
$\ then\ \ (x\ \ matches\ \ (id : y,\ \ type : z) :: pointer\ \ var\ \ (y)$
$\ \ where\ \ ((y\ \ is\ \ nat0)\ \ and\ \ (z\ \ is\ \ type))))$.

Let $P_{n.t} = [content\ typed{-}pointer{-}literal]$.

The concept *variable−pointer−literal* is defined by the rule

$(rule\ \ (x\ \ is\ \ variable{-}pointer{-}literal)\ \ var\ \ (x)\ \ abn\ \ then$
$\ \ (x\ \ matches\ \ (variable : y,\ \ scope : z,\ \ call{-}level : u) :: pointer$
$\ \ \ var\ \ (y,\ \ z,\ \ u)\ \ where\ \ ((y\ \ is\ \ variable{-}literal)\ \ and\ \ (z\ \ is\ \ scope)\ \ and$
$\ \ \ \ (u\ \ is\ \ call{-}level))))$.

Let $P_{n.v} = [content\ variable{-}pointer{-}literal]$.

The concept *array−pointer−literal* is defined by the rule

$(rule\ \ (x\ \ is\ \ array{-}pointer{-}literal)\ \ var\ \ (x)\ \ abn\ \ then$
$\ \ (x\ \ matches\ \ (array : y,\ \ index : z) :: pointer\ \ var\ \ (y,\ \ z)$
$\ \ \ where\ \ ((y\ \ is\ \ array{-}literal)\ \ and\ \ (z\ \ is\ \ nat0))))$.

Let $P_{n.a} = [content\ array{-}pointer{-}literal]$.

The concept *structure−pointer−literal* is defined by the rule

$(rule\ \ (x\ \ is\ \ structure{-}pointer{-}literal)\ \ var\ \ (x)\ \ abn\ \ then$
$\ \ (x\ \ matches\ \ (structure : y,\ \ field : z) :: pointer\ \ var\ \ (y,\ \ z)$
$\ \ \ where\ \ ((y\ \ is\ \ structure{-}literal)\ \ and\ \ (z\ \ is\ \ field - literal))))$.

Let $P_{n.s} = [content\ structure{-}pointer{-}literal]$.

The concept $function{-}pointer{-}literal$ is defined by the rule

$$(rule\ \ (x\ \ is\ \ function{-}pointer{-}literal)\ \ var\ \ (x)\ \ abn\ \ then$$
$$(x\ \ matches\ \ (function : y,\ \ types : z) :: pointer\ \ var\ \ (y,\ \ z)$$
$$where\ \ ((y\ \ is\ \ function{-}literal)\ \ and\ \ (z\ \ is\ \ (sequence\ \ type))))).$$

Let $P_{n.f} = [content\ function{-}pointer{-}literal]$.

The conceptual $(0 : p_{n.t}, 1 : pointer)$ in $[\![c_{nf}]\!]$ represents the pointer $p_{n.t}$ in $[\![[r^-_{p.s}\ c_{nf}]]\!]$. A pointer $p_{n.t}$ is a pointer in $[\![c_{nf}]\!]$ if $[c_{nf}\ (0 : p_{n.t}, 1 : pointer)] \neq und$. The conceptual $(-1 : value, 0 : p_{n.t}, 1 : pointer)$ in $[\![c_{nf}]\!]$ represents a value in $[\![p_{n.t}, [r^-_{p.s}\ c_{nf}]]\!]$. An element $v_l$ is a value in $[\![p_{n.t}, c_{nf}]\!]$ if $v_l = [c_{nf}\ (-1 : value, 0 : p_{n.t}, 1 : pointer)]$.

A pointer in $[\![s_{tt}[\![\mathrm{ITS[C]}]\!]]\!]$ is represented by an instance of the concept $pointer$ in $[\![[r_{p.s}\ s_{tt}]]\!]$ defined by the rule

$$(rule\ \ (x\ \ is\ \ pointer)\ \ var\ \ (x)\ \ abn\ \ then$$
$$((x\ \ is\ \ pointer{-}literal)\ \ and$$
$$((x\ \ is\ \ typed{-}pointer)\ \ =>\ \ (0 : x,\ \ 1 : pointer)))).$$

The predefined CTSL element $(c_{nd.1}\ \ =>\ \ c_{nd.2})$ specifies that $c_{nd.1}$ implies $c_{nd.2}$.

The conceptual $(0 : a_{rr}, 1 : array)$ in $[\![c_{nf}]\!]$ represents the array $a_{rr}$ in $[\![[r^-_{p.s}\ c_{nf}]]\!]$. An array $a_{rr}$ is an array in $[\![c_{nf}]\!]$ if $[c_{nf}\ (0 : a_{rr}, 1 : array)] \neq und$. The conceptual $(-1 : element{-}type, 0 : a_{rr}, 1 : array)$ in $[\![c_{nf}]\!]$ represents an element type and a type in $[\![a_{rr}, [r^-_{p.s}\ c_{nf}]]\!]$. A type $t_p$ is an element type in $[\![a_{rr}, c_{nf}]\!]$ if $[c_{nf}\ (-1 : element{-}type, 0 : a_{rr}, 1 : array)] = t_p$. A type $(array\ t_p)$ is a type in $[\![a_{rr}, c_{nf}]\!]$ if $t_p$ is an element type in $[\![a_{rr}, c_{nf}]\!]$. The conceptual $(-1 : body, 0 : a_{rr}, 1 : array)$ in $[\![c_{nf}]\!]$ represents elements in $[\![a_{rr}, [r^-_{p.s}\ c_{nf}]]\!]$. A sequence element $b_d$ is a body in $[\![a_{rr}, c_{nf}]\!]$ if $b_d = [c_{nf}\ (-1 : body, 0 : a_{rr}, 1 : array)]$. The element $e_l$ is an element in $[\![a_{rr}, c_{nf}, n_t]\!]$ if $[[c_{nf}\ (-1 : body, 0 : a_{rr}, 1 : array)]\ .\ [n_{t0} + 1]] = e_l$, and $0 \leq n_{t0} < [len\ [c_{nf}\ (-1 : body, 0 : a_{rr}, 1 : array)]]$. The element $e_l$ is an element in $[\![a_{rr}, c_{nf}]\!]$ if $e_l$ is an element in $[\![a_{rr}, c_{nf}, n_{t0}]\!]$ for some $n_{t0}$.

An array in $[\![s_{tt}[\![\mathrm{ITS[C]}]\!]]\!]$ is represented by an instance of the concept $array$ in $[\![[r_{p.s}\ s_{tt}]]\!]$ defined by the rule

$$(rule\ \ (x\ \ is\ \ array)\ \ var\ \ (x)\ \ abn$$
$$then\ \ ((x\ \ is\ \ array{-}literal)\ \ and\ \ (0 : x,\ \ 1 : array))).$$

The conceptual $(0 : s_{trc}, 1 : structure)$ in $[\![c_{nf}]\!]$ represents the structure in $[\![[r^-_{p.s}\ c_{nf}]]\!]$. A structure $s_{trc}$ is a structure in $[\![c_{nf}]\!]$ if $[c_{nf}\ (0 : s_{trc}, 1 : structure)] \neq und$. The conceptual $(-1 : type, 0 : s_{trc}, 1 : structure)$ in $[\![c_{nf}]\!]$ represents a type in $[\![s_{trc}, [r^-_{p.s}\ c_{nf}]]\!]$. A type $t_{p.s}$ is a type in $[\![s_{trc}, c_{nf}]\!]$ if $[c_{nf}\ (-1 : type, 0 : s_{trc}, 1 : structure)] = t_{p.s}$. The conceptual $(-1 : body, 0 : s_{trc}, 1 : structure)$ in $[\![c_{nf}]\!]$ represents a body, fields and their values in $[\![s_{trc}, [r^{-}_{p.s}\ c_{nf}]]\!]$. An element $b_d$ is a body in $[\![s_{trc}, c_{nf}]\!]$ if $[c_{nf}\ (-1 : \mathit{body}, 0 : s_{trc}, 1 : \mathit{structure})] = b_d$. A field $f_l$ is a field in $[\![s_{trc}, c_{nf}]\!]$ if $[[c_{nf}\ (-1 : \mathit{body}, 0 : s_{trc}, 1 : \mathit{structure})]\ f_l] \neq \mathit{und}$. An element $v_l$ is a value in $[\![f_l, t_{p.s}, c_{nf}]\!]$ if $[[c_{nf}\ (-1 : \mathit{body}, 0 : s_{trc}, 1 : \mathit{structure})]\ f_l] = v_l$.

A structure in $[\![s_{tt}[\![\mathrm{ITS[C]}]\!]]\!]$ is represented by an instance of the concept *structure* in $[\![[r_{p.s}\ s_{tt}]]\!]$ defined by the rule

$$(\mathit{rule}\ \ (x\ \ \mathit{is}\ \ \mathit{structure})\ \ \mathit{var}\ \ (x)\ \ \mathit{abn}\\ \ \mathit{then}\ \ ((x\ \ \mathit{is}\ \ \mathit{structure{-}literal})\ \ \mathit{and}\ \ (0 : x,\ \ 1 : \mathit{structure}))).$$

The information about functions is represented by the substate $\mathit{function}$ in configurations. The conceptual $(-1 : t_{p.(*)}, 0 : f_n, 1 : \mathit{function}) :: \mathit{state} :: \mathit{function}$ in $[\![c_{nf}]\!]$ represents the function $[r^{-}_{p.s}\ f_n]$ with the argument types $[r^{-}_{p.s}\ t_{p.(*)}]$ in $[\![[r^{-}_{p.s}\ c_{nf}]]\!]$. A function $f_n$ is a function in $[\![t_{p.(*)}, c_{nf}]\!]$ if $[c_{nf}\ (-1 : t_{p.(*)}, 0 : f_n, 1 : \mathit{function}) :: \mathit{state} :: \mathit{function}] \neq \mathit{und}$. An element $t_{p.(*)}$ is an argument type sequence in $[\![f_n, c_{nf}]\!]$ if $f_n$ is a function in $[\![t_{p.(*)}, c_{nf}]\!]$. The conceptual $(-2 : \mathit{arguments}, -1 : t_{p.(*)}, 0 : f_n, 1 : \mathit{function}) :: \mathit{state} :: \mathit{function}$ in $[\![c_{nf}]\!]$ represents arguments in $[\![[r^{-}_{p.s}\ f_n], [r^{-}_{p.s}\ c_{nf}]]\!]$. An element $a_{rg.(*)}$ is an argument sequence in $[\![f_n, c_{nf}]\!]$ if $[c_{nf}\ (-2 : \mathit{arguments}, -1 : t_{p.(*)}, 0 : f_n, 1 : \mathit{function}) :: \mathit{state} :: \mathit{function}] = a_{rg.(*)}$. The conceptual $(-2 : \mathit{return{-}type}, -1 : t_{p.(*)}, 0 : f_n, 1 : \mathit{function}) :: \mathit{state} :: \mathit{function}$ in $[\![c_{nf}]\!]$ represents the return type in $[\![[r^{-}_{p.s}\ f_n], [r^{-}_{p.s}\ c_{nf}]]\!]$. A type $t_p$ is the return type in $[\![f_n, c_{nf}]\!]$ if $[c_{nf}\ (-2 : \mathit{return{-}type}, -1 : t_{p.(*)}, 0 : f_n, 1 : \mathit{function}) :: \mathit{state} :: \mathit{function}] = t_p$. The conceptual $(-2 : \mathit{body}, -1 : t_{y.(*)}, 0 : f_n, 1 : \mathit{function}) :: \mathit{state} :: \mathit{function}$ in $[\![c_{nf}]\!]$ represents a body in $[\![[r^{-}_{p.s}\ f_n], [r^{-}_{p.s}\ c_{nf}]]\!]$. An element $b_d$ is a body in $[\![f_n, c_{nf}]\!]$ if $[c_{nf}\ (-2 : \mathit{body}, -1 : t_{y.(*)}, 0 : f_n, 1 : \mathit{function}) :: \mathit{state} :: \mathit{function}] = b_d$.

The conceptuals $(0 : \mathit{level}) :: \mathit{state} :: \mathit{function}$ and $(0 : \mathit{type}) :: \mathit{state} :: \mathit{function}$ specify the current call level and the return type in it, respectively.

The conceptual $(-2 : l_{v.c}, -1 : s_{cp}, 0 : v_r, 1 : \mathit{variable})$ in $[\![c_{nf}]\!]$ represents the variable $[r^{-}_{p.s}\ v_r]$ in $[\![l_{v.c}, s_{cp}, [r^{-}_{p.s}\ c_{nf}]]\!]$. A variable $v_r$ is a variable in $[\![l_{v.c}, s_{cp}, c_{nf}]\!]$ if $[c_{nf}\ (-2 : l_{v.c}, -1 : s_{cp}, 0 : v_r, 1 : \mathit{variable})] \neq \mathit{und}$. The conceptual $(-3 : \mathit{pointer}, -2 : l_{v.c}, -1 : s_{cp}, 0 : v_r, 1 : \mathit{variable})$ in $[\![c_{nf}]\!]$ represents an address in $[\![[r^{-}_{p.s}\ v_r], l_{v.c}, s_{cp}, [r^{-}_{p.s}\ c_{nf}]]\!]$. A pointer $p_{n.v}$ is a pointer in $[\![v_r, l_{v.c}, s_{cp}, c_{nf}]\!]$ if $[c_{nf}\ (-3 : \mathit{pointer}, -2 : l_{v.c}, -1 : s_{cp}, 0 : v_r, 1 : \mathit{variable})] = p_{n.v}$. The conceptual $(-3 : \mathit{type}, -2 : l_{v.c}, -1 : s_{cp}, 0 : v_r, 1 : \mathit{variable})$ in $[\![c_{nf}]\!]$ represents a type in $[\![[r^{-}_{p.s}\ v_r], l_{v.c}, s_{cp}, [r^{-}_{p.s}\ c_{nf}]]\!]$. A type $t_p$ is a type in $[\![v_r, l_{v.c}, s_{cp}, c_{nf}]\!]$ if $[c_{nf}\ (-3 : \mathit{type}, -2 : l_{v.c}, -1 : s_{cp}, 0 : v_r, 1 : \mathit{variable})] = t_p$. The conceptual $(-3 : \mathit{value}, -2 : l_{v.c}, -1 : s_{cp}, 0 : v_r, 1 : \mathit{variable})$ in $[\![c_{nf}]\!]$ represents a value in $[\![[r^{-}_{p.s}\ v_r], l_{v.c}, s_{cp}, [r^{-}_{p.s}\ c_{nf}]]\!]$. An element $v_l$ is a value in $[\![v_r, l_{v.c}, s_{cp}, c_{nf}]\!]$ if $[c_{nf}\ (-3 : \mathit{value}, -2 : l_{v.c}, -1 : s_{cp}, 0 : v_r, 1 : \mathit{variable})] = v_l$.

The information about blocks is represented by the substate *block* in configurations. The conceptual $(0 : \mathit{scope}) :: \mathit{block}$ specifies the current relative scope.

A name $n_m$ can have a set of possible values in $[\![c_{nf}]\!]$. For example, possible values of the name $v_r$ of the form $n_{m.1}$ :: *variable* in $[\![c_{nf}]\!]$ are the variables with the name $n_{m.1}$ of scopes from 0 to the current scope in $[\![c_{nf}]\!]$. To choose the right value in $[\![name : n_m, c_{nf}]\!]$ and, thus, to resolve the name conflict, these possible values are indexed. For example, indices in $[\![name : v_r, c_{nf}]\!]$ are scopes from 0 to the current scope in $[\![c_{nf}]\!]$. Then the name resolution problem is reduced to the choice of a right index in $[\![name : n_m, c_{nf}]\!]$.

A variable in $[\![s_{tt}[\![\text{ITS[C]}]\!]]\!]$ is represented by an instance of the concept *variable* in $[\![[r_{p.s}\ s_{tt}]]\!]$ defined by the rule

$(rule\ \ (x\ \ is\ \ variable)\ \ var\ \ (x)\ \ abn\ \ then\ \ (index\ \ in\ \ x))$.

The element $(index\ in\ v_r)$ returning the right index in $[\![v_r, c_{nf}]\!]$ is defined by the rules

$(rule\ \ (index\ \ in\ \ x)\ \ var\ \ (x)\ \ abn\ \ where\ \ (x\ \ is\ \ variable{-}literal)$
$\ then\ \ (let :: seq\ \ (w1,\ \ w2)\ \ be\ \ (current{-}scope,\ \ current{-}call{-}level)\ \ in$
$\ \ (index\ \ in\ \ x\ \ in\ \ w1,\ \ w2)))$;
$(rule\ \ (x\ \ is\ \ index\ \ in\ \ y,\ \ z)\ \ var\ \ (x,\ \ y,\ \ z)\ \ abn\ \ then$
$\ (if\ \ ({-2} : y,\ \ {-1} : z,\ \ 0 : x,\ \ 1 : variable)\ \ then\ \ z :: q\ \ else$
$\ \ (if\ \ (z\ \ =\ \ 0)\ \ then\ \ und\ \ else$
$\ \ \ (let\ \ w\ \ be\ \ (z\ \ -\ \ 1)\ \ in\ \ (index\ \ in\ \ x\ \ in\ \ y,\ \ w)))))$.

The predefined CTSL element $(let :: seq\ (v_{r.*})\ be\ (e_{l.*})\ in\ b_d)$, where $b_d \in E_{l.*}$, replaces the variables $v_{r.*}$ in the body $b_d$ by the values of the corresponding elements of $e_{l.*}$ and executes the resulting body.

The elements *current−scope* and *current−call−level* are defined by the rules

$(rule\ \ current{-}scope\ \ then\ \ (0 : scope) :: state :: block)$;
$(rule\ \ current{-}scope\ \ then\ \ (0 : call{-}level) :: state :: function)$.

A variable $v_r$ is global in $[\![c_{nf}]\!]$ if the right index in $[\![v_r, c_{nf}]\!]$ equals 0. A variable $v_r$ is local in $[\![c_{nf}]\!]$ if the right index in $[\![v_r, c_{nf}]\!]$ is greater than 0. The type and value of a global variable $v_r$ in $[\![c_{nf}]\!]$ is specified by the conceptuals $(-3 : type, -2 : 0, -1 : 0, 0 : v_r, 1 : variable)$ and $(-3 : value, -2 : 0, -1 : 0, 0 : v_r, 1 : variable)$, respectively.

A value is represented by an instance of the concept *value−literal* defined by the rule

$(rule\ \ (x\ \ is\ \ value{-}literal)\ \ var\ \ (x)\ \ abn\ \ then$
$\ ((x\ \ is\ \ int)\ \ or\ \ (x\ \ is\ \ float)\ \ or\ \ ...\ \ or$
$\ \ (x\ \ is\ \ pointer{-}literal)\ \ or\ \ (x\ \ is\ \ array{-}literal)\ \ or$
$\ \ (x\ \ is\ \ structure{-}literal)\ \ or\ \ (x\ \ is\ \ function{-}literal)))$,

where ... are disjuncts of the form $(x\ is\ t_{p.b})$ for all $t_{p.b}$. Let $V_l = [content\ value{-}literal]$.

A value in $[\![s_{tt}[\![\mathrm{ITS[C]}]\!]]\!]$ is represented by an instance of the concept *value* in $[\![[r_{p.s}\ s_{tt}]]\!]$ defined by the rule

$$
\begin{array}{l}
(rule\ \ (x\ \ is\ \ value)\ \ var\ \ (x)\ \ abn\ \ then\\
\ ((x\ \ is\ \ int)\ \ or\ \ (x\ \ is\ \ float)\ \ or\ \ ...\ \ or\\
\ \ (x\ \ is\ \ pointer)\ \ or\ \ (x\ \ is\ \ array)\ \ or\\
\ \ (x\ \ is\ \ structure)\ \ or\ \ (x\ \ is\ \ function))),
\end{array}
$$

where ... are disjuncts of the form $(x\ is\ t_{p.b})$ for all $t_{p.b}$.

## 6. States in CTSL[C]

States are represented by configurations including the substates *block* and *function* and conceptuals defined in Section 5. These substates model information associated with blocks and functions, respectively.

## 7. Answers in CTSL[C]

The element of the form $e_l :: ex$ is called an exception. The value of the exception $e_l :: ex$ in CTSL is defined as $e_l :: ex$. The exceptions $(type : break) :: ex$, $(type : continue) :: ex$ and $(type : goto, label : l_b) :: ex$ represent the execution of the *break* statement, the *continue* statement, and the *goto* statement with the label $l_b$, respectively. The exceptions $(type : return, value : v_l) :: ex$ and $(type : return) :: ex$ represent the execution of the *return* statement. These exceptions are called jumps.

The values of C types are represented by instances of the concept *value−literal*, the extra literals of AM[C] are represented by instances of the corresponding concepts, jumps initiated by jump statements are represented by jump exceptions, and the program error message is represented by *und*.

## 8. Defined elements in CTSL[C]

### 8.1. Statements and blocks

The elements *break*, *continue* and $(goto\ l_b)$ representing the queries *break*;, *continue*; and *goto* $[r_{p.q}\ l_b]$; are defined by the rules

$$
\begin{array}{l}
(rule\ \ break\ \ abn\ \ then\ \ (type : break) :: ex);\\
(rule\ \ continue\ \ abn\ \ then\ \ (type : continue) :: ex);\\
(rule\ \ (goto\ \ x)\ \ var\ \ (x)\ \ abn\ \ where\ \ (x\ \ is\ \ label)\ \ then\\
\ (type : goto,\ \ label : x) :: ex).
\end{array}
$$

The element $n_m :: label$ representing queries of the form $[r_{p.q}\ n_m : s_{ttm};]$ as $(seq\ n_m :: label\ [r_{p.q}\ s_{ttm}])$ is defined by the rule

$$(rule\ \ x :: label\ \ var\ \ (x)\ \ und\ \ then\ \ (catch\ \ w$$
$$(if\ \ w\ \ matches\ \ (type : goto,\ \ label : y :: label)\ \ var\ \ (y)$$
$$where\ \ (y :: q\ \ =\ \ x :: q)\ \ then\ \ else\ \ (throw\ \ w :: q)))).$$

The predefined CTSL element ($throw\ e_l$)) assigns the value of the element $e_l$ to a special conceptual $(0 : ()) :: state :: value$ specifying the current value of the CTS CTSL[C]. The predefined CTSL element ($catch\ v_r\ e_{l.*}$) replaces the variable $v_r$ in the sequence $e_l$ by the value of the conceptual $(0 : ()) :: state :: value$ and executes the resulting sequence. The predefined CTSL element ($if\ e_l\ matches\ p_{tt}\ var\ (v_{r.*})\ where\ c_{nd}\ then\ e_{l.*.1}\ else\ e_{l.*.2}$) specifies that if the element $e_l$ matches the pattern $p_{tt}$ with the variables $v_{r.*}$ and the condition $c_{nd}$ is true for the corresponding values of these variables, then the sequence $e_{l.*.1}$ is executed for these values of the variables. If $e_l$ does not match $p_{tt}$, then the sequence $e_{l.*.2}$ is executed. The predefined CTSL element ($e_{l.1}\ =\ e_{l.2}$) specifies that $e_{l.1}$ and $e_{l.2}$ are equal.

The elements ($return\ e_l$) and ($return$) representing the queries *return* $[r^-_{p.q}\ e_l]$; and *return*; are defined by the rules

$$(rule\ \ (return\ \ x)\ \ var\ \ (x)\ \ abn\ \ val\ \ (x)\ \ then$$
$$(let\ \ w1\ \ be\ \ (0 : type) :: state :: function\ \ in$$
$$(if\ \ (w1 :: q\ \ =\ \ void :: q)\ \ then\ \ und\ \ else$$
$$(let\ \ w2\ \ be\ \ (cast\ \ x :: * :: q\ \ w1)\ \ in$$
$$(type : return,\ \ value : w2 :: q) :: ex))));$$
$$(rule\ \ (return)\ \ abn\ \ then$$
$$(if\ \ ((0 : type) :: state :: function\ \ =\ \ void :: q)$$
$$then\ \ (type : return) :: ex\ \ else\ \ und)).$$

The predefined CTSL element ($if\ c_{nd}\ then\ e_{l.*.1}\ else\ e_{l.*.2}$) specifies that if the value of $c_{nd}$ is true, then the sequence $e_{l.*.1}$ is executed. Otherwise, the sequence $e_{l.*.2}$ is executed. The predefined CTSL element ($let\ v_r\ be\ e_l\ in\ e_{l.*}$) is a shortcut for ($let :: seq\ (v_r)\ be\ (e_l)\ in\ e_{l.*}$).

The element ($block\ e_{l.*}$) representing the query $\{[r^-_{p.q}\ e_{l.*}]\}$ is defined by the rule

$$(rule\ \ (block\ \ x)\ \ var\ \ (x)\ \ abn\ \ then$$
$$enter{-}block,\ \ (let :: seq\ \ (w1,\ \ w2)$$
$$be\ \ ((block{-}variables\ \ in\ \ (x)),\ \ (block{-}labels\ \ in\ \ (x)))\ \ in$$
$$x,\ \ (continue{-}block\ \ in\ \ w2,\ \ (x))$$
$$(catch :: u\ \ w\ \ (exit{-}block\ \ in\ \ w1),\ \ (throw\ \ w)))).$$

The element $enter{-}block$ specifying the actions executed when the current configuration enters the block is defined by the rule

$$(rule\ \ enter{-}block\ \ abn\ \ then\ \ current{-}scope++).$$

The element $current{-}scope++$ is defined by the rule

$(rule\ \ current{-}scope+\!+\ \ abn\ \ then$
$\ ((0 : scope) :: state :: block\ \ \ ::=\ \ \ ((0 : scope) :: state :: block\ \ \ +\ \ \ 1)))\,.$

The predefined CTSL element $(c_{nptl}\ \ ::=\ \ e_l)$ assigns the value of $e_l$ to the conceptual $c_{nptl}$. The predefined CTSL element $(e_{l.1}\ \ +\ \ e_{l.2})$ specifies the sum of $e_{l.1}$ and $e_{l.2}$.

The element $(block{-}variables\ in\ (e_{l.*}))$ returning the sequence of the local variables defined in declaration statements that are the elements of $e_{l.*}$ is defined by the rules

$(rule\ \ (block{-}variables\ \ in\ \ ((var\ \ x\ \ y)\ \ z))\ \ var\ \ (x,\ \ y)\ \ seq\ \ (z)\ \ abn$
$\ where\ \ ((x\ \ is\ \ variable{-}literal)\ \ and\ \ (y\ \ is\ \ type))\ \ then$
$\ \ (x :: q\ \ .+\ \ \ (block{-}variables\ \ in\ \ (z))))\,;$
$(rule\ \ (block{-}variables\ \ in\ \ (x\ \ y))\ \ var\ \ (x)\ \ seq\ \ (y)\ \ abn\ \ then$
$\ (block{-}variables\ \ in\ \ (y)))\,;$
$(rule\ \ (block{-}variables\ \ in\ \ ())\ \ abn\ \ then\ \ ())\,.$

The predefined CTSL element $(e_l\ \ .+\ \ (e_{l.*}))$ adds the element $e_l$ to the head of the sequence $(e_{l.*})$.

The element $(block{-}labels\ in\ (e_{l.*}))$ returning the sequence of the labels that are the elements of $e_{l.*}$ is defined by the rules

$(rule\ \ (block{-}labels\ \ in\ \ (x :: label\ \ y))\ \ var\ \ (x)\ \ seq\ \ (y)\ \ abn\ \ then$
$\ (x :: label :: q\ \ .+\ \ \ (block{-}labels\ \ in\ \ (y))))\,;$
$(rule\ \ (block{-}labels\ \ in\ \ (x\ \ y))\ \ var\ \ (x)\ \ seq\ \ (y)\ \ abn\ \ then$
$\ (block{-}labels\ \ in\ \ (y)))\,;$
$(rule\ \ (block{-}labels\ \ in\ \ ())\ \ abn\ \ then\ \ ())\,.$

The element $(continue{-}block\ in\ l_{b.(*)},\ \ e_{l.(*)})$ handling *goto* exceptions when the current configuration reaches the end of the block is defined by the rule

$(rule\ \ (continue{-}block\ \ in\ \ x,\ \ (y))\ \ var\ \ (x)\ \ seq\ \ (y)\ \ und\ \ then$
$\ (catch\ \ w$
$\ \ (if\ \ w\ \ matches\ \ (type : goto,\ \ label : z) :: ex\ \ var\ \ (z)$
$\ \ \ where\ \ (z :: q\ \ in :: set\ \ x)$
$\ \ \ then\ \ (throw\ \ w),\ \ y,\ \ (continue{-}block\ \ in\ \ x,\ \ (y))\ \ else\ \ (throw\ \ w))))\,.$

The element $(exit{-}block\ in\ v_{r.(*)})$ specifying the actions executed when the current configuration exits the block is defined by the rule

$(rule\ \ (exit{-}block\ \ in\ \ x)\ \ var\ \ (x)\ \ und\ \ then\ \ (catch\ \ w$
$\ (delete{-}variables\ \ in\ \ x),\ \ current{-}scope-\!-,\ \ (throw\ \ w :: q)))\,.$

The element $current{-}scope-\!-$ is defined by the rule

$(rule\ \ current{-}scope-\!-\ \ abn\ \ then$
$\ ((0 : scope) :: state :: block\ \ \ ::=\ \ \ ((0 : scope) :: state :: block\ \ \ -\ \ \ 1)))\,.$

The predefined CTSL element $(e_{l.1} - e_{l.2})$ specifies the difference between $e_{l.1}$ and $e_{l.2}$.

The element $(delete{-}variables\ in\ (v_{r.*}))$ deleting the local variables $v_{r.*}$ in $[\![current{-}scope, current{-}call{-}level]\!]$ is defined by the rule

$$
\begin{array}{l}
(rule\ \ (delete{-}variables\ \ in\ \ x)\ \ var\ \ (x)\ \ abn\ \ then \\
\ (let :: seq\ \ (w1,\ \ w2)\ \ be\ \ (current{-}scope,\ \ current{-}call{-}level)\ \ in \\
\ \ (foreach\ \ y\ \ in\ \ x :: q\ \ do \\
\ \ \ ((-3 : w2,\ \ -2 : w1,\ \ -1 : ponter,\ \ 0 : y,\ \ 1 : variable)\ \ \ ::=), \\
\ \ \ ((-3 : w2,\ \ -2 : w1,\ \ 0 : y,\ \ 1 : variable)\ \ \ ::=)))) .
\end{array}
$$

The predefined CTSL element $(c_{ncptl}\ \ ::=)$ is a shortcut for $(c_{ncptl}\ \ ::=\ und)$. The predefined CTSL element $(foreach\ v_r\ in\ (e_{l.*})\ do\ b_d)$, where $b_d \in E_{l.*}$, executes sequentially the body $b_d$ for each value of the variable $v_r$ taken from the sequence $e_{l.*}$ from left to right.

The elements $(e_l\ ";")$ and $";"$ representing the queries $[r^-_{p.q}\ e_l]$; and ; are defined by the rules

$$
\begin{array}{l}
(rule\ \ (x\ \ ";")\ \ var\ \ (x)\ \ abn\ \ then\ \ x); \\
(rule\ \ ";"\ \ var\ \ (x)\ \ abn\ \ then)
\end{array}
$$

The element $(if\ :: \ C\ \ c_{nd}\ \ then\ \ e_{l.1}\ \ else\ \ e_{l.2})$ representing the query $if\ [r^-_{p.q}\ (c_{nd}])\ then\ [r^-_{p.q}\ e_{l.1}]\ else\ [r^-_{p.q}\ e_{l.2}]$ is defined by the rule

$$
\begin{array}{l}
(rule\ \ (if :: C\ \ x\ \ then\ \ y\ \ else\ \ z)\ \ var\ \ (x,\ \ y,\ \ z)\ \ abn \\
\ then\ \ (if\ \ ((cast\ \ x\ \ int)\ \ != \ \ 0)\ \ then\ \ (block\ \ y)\ \ else\ \ (block\ \ z))) .
\end{array}
$$

Other switch statements are defined in a similar way.

The element $(while :: C\ c_{nd}\ do\ e_l)$ representing the query $(while\ ([r^-_{p.q}\ c_{nd}])\ [r^-_{p.q}\ e_l])$ is defined by the rule

$$
\begin{array}{l}
(rule\ \ (while :: C\ \ x\ \ do\ \ y)\ \ var\ \ (x,\ \ y)\ \ abn\ \ then \\
\ (while\ \ ((cast\ \ x\ \ int)\ \ != \ \ 0) \\
\ \ do\ \ (block\ \ y,\ \ (delete{-}exception\ \ continue))), \\
\ (delete{-}exception\ \ break)) .
\end{array}
$$

The predefined CTSL element $(while\ c_{nd}\ do\ b_d)$, where $b_d \in E_{l.*}$, executes the body $b_d$ until the condition $c_{nd}$ becomes the false value. Other iteration statements are defined in a similar way.

### 8.2. Declarations

The element $(function\ f_n\ (a_{rg.1} : t_{p.1},\ ...,\ a_{rg.n_{t.0}} : t_{p.n_{t.0}}) : t_p\ b_d)$ representing the query $[r^-_{p.q}\ t_p]\ [r^-_{p.q}\ f_n]([r^-_{p.q}\ t_{p.1}]\ [r^-_{p.q}\ a_{rg.1}], ...,\ [r^-_{p.q}\ t_{p.n_{t.0}}]\ [r^-_{p.q}\ a_{rg.n_{t.0}}])\ \{[r^-_{p.q}\ b_d]\}$ is defined by the rule

$(rule\ \ (function\ \ x\ \ y : z\ \ u)\ \ var\ \ (x,\ \ y,\ \ z)\ \ seq\ \ (u)\ \ abn$
$\ \ where\ \ ((x\ \ is\ \ function{-}literal)\ \ and\ \ (y\ \ is\ \ attribute{-}element)\ \ and$
$\ \ \ \ (z\ \ is\ \ type))$
$\ \ then\ \ (let\ \ w\ \ be\ \ (values\ \ in\ \ y)\ \ in$
$\ \ \ \ (if\ \ ((w\ \ is\ \ (sequence\ \ type))\ \ and$
$\ \ \ \ \ \ (not\ \ (-1 : w,\ \ 0 : x,\ \ 1 : function) :: function))\ \ then$
$\ \ \ \ \ \ ((-2 : arguments,\ \ -1 : w,\ \ 0 : x,\ \ 1 : function) :: function\ \ ::=$
$\ \ \ \ \ \ \ \ (attributes\ \ in\ \ y)),$
$\ \ \ \ \ \ ((-2 : return{-}type,\ \ -1 : w,\ \ 0 : x,\ \ 1 : function) :: function\ \ ::=$
$\ \ \ \ \ \ \ \ z :: q),$
$\ \ \ \ \ \ ((-2 : body,\ \ -1 : w,\ \ 0 : x,\ \ 1 : function) :: function\ \ ::=\ \ (u) :: q),$
$\ \ \ \ \ \ ((-1 : w,\ \ 0 : x,\ \ 1 : function) :: function\ \ ::=\ \ true)))$
$\ \ \ \ else\ \ und))\,.$

The predefined CTSL element ($e_l$ *is attribute−element*) specifies that $e_l$ is an attribute element. The predefined CTSL elements (*attributes in* $e_{l.a}$) and (*values in* $e_{l.a}$) specify the sequence of attributes in the attribute element $e_{l.a}$ and the sequence of their values, respectively. The predefined CTSL element (*not* $c_{nd}$) specifies the negation of $c_{nd}$.

The element (*var* $v_r$ $t_p$) representing the query $[r^-_{p.q}\ t_p]\ [r^-_{p.q}\ v_r]$; is defined by the rule

$(rule\ \ (var\ \ x\ \ y)\ \ var\ \ (x,\ \ y)\ \ abn$
$\ \ where\ \ ((x\ \ is\ \ variable{-}literal)\ \ and\ \ (y\ \ is\ \ type))$
$\ \ then\ \ (let :: seq\ \ (w1,\ \ w2)\ \ be\ \ (current{-}scope,\ \ current{-}call{-}level)\ \ in$
$\ \ \ \ (if\ \ (-2 : w1,\ \ -1 : w2,\ \ 0 : x,\ \ 1 : variable)\ \ then\ \ und\ \ else$
$\ \ \ \ \ \ ((-3 : type,\ \ -2 : w1,\ \ -1 : w2,\ \ 0 : x,\ \ 1 : variable)\ \ ::=\ \ y :: q),$
$\ \ \ \ \ \ ((-2 : w1,\ \ -1 : w2,\ \ 0 : x,\ \ 1 : variable)\ \ ::=\ \ true)))\,.$

The element (*struct* $t_{p.s}$ $(f_{l.1} : t_{p.1},\ \ldots,\ f_{l.n_t} : t_{p.n_t})$) representing the query *struct* $[r^-_{p.q}\ t_{p.s}]\ \{[r^-_{p.q}\ t_{p.1}]\ [r^-_{p.q}\ f_{l.1}];\ \ldots;\ [r^-_{p.q}\ t_{p.n_t}]\ [r^-_{p.q}\ f_{l.n_t}]\}$ is defined by the rule

$(rule\ \ (struct\ \ x\ \ y)\ \ var\ \ (x,\ \ y)\ \ abn\ \ where$
$\ \ \ \ ((x\ \ is\ \ structure{-}type{-}literal)\ \ and\ \ (y\ \ is\ \ attribute)\ \ and$
$\ \ \ \ \ \ (let\ \ w\ \ be\ \ (attributes\ \ in\ \ y)\ \ in\ \ (w\ \ is\ \ (sequence\ \ field{-}literal)))$
$\ \ \ \ \ \ and\ \ (let\ \ w\ \ be\ \ (attribute{-}values\ \ in\ \ y)\ \ in\ \ (w\ \ is\ \ (sequence\ \ type))))$
$\ \ then\ \ (if\ \ (x\ \ is\ \ structure{-}type)\ \ then\ \ und$
$\ \ \ \ \ \ \ \ else\ \ ((-1 : body,\ \ 0 : x,\ \ 1 : structure{-}type)\ \ ::=\ \ y :: q),$
$\ \ \ \ \ \ \ \ \ \ \ \ ((0 : x,\ \ 1 : structure{-}type)\ \ ::=\ \ true)))\,.$

### 8.3. Expressions

The element $v_l$ representing the query $v_l$ is defined by the rule

$(rule\ \ x\ \ var\ \ (x)\ \ abn\ \ where\ \ (x\ \ is\ \ value{-}literal)\ \ then\ \ (throw\ \ x :: q))\,.$

The element $v_r$ representing the query $[r^-_{p.q}\ v_r]$ is defined by the rule

$$(rule\ x\ var\ (x)\ abn\ then\ (let :: seq\ (w1,\ w2)\ be\\ ((index\ in\ x),\ (current{-}call{-}level\ in\ w1))\ in\\ (-3 : value,\ -2 : w2,\ -1 : w1,\ 0 : x,\ 1 : variable))).$$

The element $(current{-}call{-}level\ in\ s_{cp})$ is defined by the rule

$$(rule\ (current{-}call{-}level\ in\ x)\ var\ (x)\ abn\ where\ (x\ is\ scope)\ then\\ (if\ (x :: q\ =\ 0)\ then\ 0\ else\ current{-}call{-}level)).$$

The element $(e_l\ [\ e_{l.1}\ ])$ representing the query $[r^-_{p.q}\ e_l][[r^-_{p.q}\ e_{l.1}]]$ is defined by the rule

$$(rule\ (x\ [\ y\ ])\ var\ (x,\ y)\ abn\ val\ (y,\ x)\ where\ (x :: *\ is\ array)\\ then\ (let\ w\ be\ (cast\ y :: *\ int)\ in\ (if\ (w\ and\ (w :: q\ >=\ 0))\\ then\ ((-1 : body,\ 0 : x :: *,\ 1 : array)\ .\ (w\ +\ 1))\ else\ und))).$$

The predefined CTSL element $(e_{l.1}\ >=\ e_{l.2})$ specifies that $e_{l.1}$ is greater than or equal to $e_{l.2}$. The predefined CTSL element $((e_{l.*})\ .\ n_t)$ specifies the $n_t$-th element of the sequence $e_{l.*}$.

The element $(e_l\ .::C\ f_l)$ representing the query $[r^-_{p.q}\ e_l].[r^-_{p.q}\ f_l]$ is defined by the rule

$$(rule\ (x\ .::C\ y)\ var\ (x,\ y)\ abn\ val\ (x)\ where\\ (x :: *\ is\ structure)\ and\ (y\ is\ field{-}literal))\ then\\ ((-1 : body,\ 0 : x :: *,\ 1 : structure)\ ..\ y)).$$

The element $(e_l\ :=\ e_{l.1})$ representing the query $[r^-_{p.q}\ e_l]\ :=\ [r^-_{p.q}\ e_{l.1}]$ is defined by the rule

$$(rule\ (x\ :=\ y)\ var\ (x,\ y)\ abn\ val\ (y)\ then\\ (let :: seq\ (w1,\ w2,\ w3,\ w4)\ be\ ((left{-}hand\ in\ x),\\ (w1\ ..\ left),\ (w1\ ..\ type),\ (cast\ y :: * :: q\ w3))\ in\\ (if\ (w1\ and\ w4)\ then\ (w2\ ::=\ w4 :: q)\ else\ und))).$$

The element $(left{-}hand\ in\ e_l)$ is defined by the rule

$$(rule\ (left{-}hand\ in\ x)\ var\ (x)\ abn\ then\ (let\ w1\ be\ (index\ in\ x)\ in\\ (if\ w1\ then\ (let :: seq\ (w2,\ w3)\ be\ ((current{-}call{-}level\ in\ w1),\\ (-3 : type,\ -2 : w2,\ -1 : w1,\ 0 : x,\ 1 : variable))\ in\\ (left : (-3 : value,\ -2 : w2,\ -1 : w1,\ 0 : x,\ 1 : variable),\\ type : w3) :: q)\\ elseif\ x\ matches\ (*\ y)\ var\ (y)\ val\ (y)\ then\\ (if\ (y :: *\ is\ typed{-}pointer)\ then\ (let\ w2\ be\ (y :: * :: q\ ..\ type)\ in\\ (left : (-1 : value,\ 0 : y :: *,\ 1 : pointer),\ type : w2) :: q))\\ elseif\ (y :: *\ is\ variable{-}pointer)\ then\\ (let :: seq\ (w2,\ w3,\ w4,\ w5)$$

$$
\begin{array}{l}
\quad\quad be\ ((y :: * \ ..\ variable),\ (y :: * \ ..\ scope),\ (y :: * \ ..\ call{-}level),\\
\quad\quad\ \ (-3 : type,\ \ -2 : w4,\ \ -1 : w3,\ \ 0 : w2,\ \ 1 : variable))\ in\\
\quad\quad (left : (-3 : value,\ \ -2 : w4,\ \ -1 : w3,\ \ 0 : w2,\ \ 1 : variable),\\
\quad\quad\ \ type : w5) :: q)\\
\quad elseif\ \ (y :: * \ is\ \ array{-}pointer)\ then\ (let :: seq\ (w2,\ w3,\ w4)\\
\quad\quad be\ ((y :: * \ ..\ array),\ (y :: * \ ..\ index),\\
\quad\quad\quad (-1 : element{-}type,\ \ 0 : w2,\ \ 1 : array))\ in\\
\quad\quad (left : ((-1 : body,\ \ 0 : w2,\ \ 1 : array)\ .\ w3 :: q),\ \ type : w4) :: q)\\
\quad elseif\ \ (y :: * \ is\ \ structure{-}pointer)\ then\\
\quad\ \ (let :: seq\ (w2,\ w3,\ w4,\ w5)\\
\quad\quad be\ ((y :: * \ ..\ structure),\ (y :: * \ ..\ field),\\
\quad\quad\quad (-1 : type,\ \ 0 : w2,\ \ 1 : structure),\\
\quad\quad\quad ((-1 : body,\ \ 0 : w4,\ \ 1 : structure{-}type)\ ..\ w3))\ in\\
\quad\quad (left : ((-1 : body,\ \ 0 : w2,\ \ 1 : structure)\ ..\ w3),\ \ type : w5) :: q)\\
\quad else\ \ und)\\
elseif\ \ x\ \ matches\ \ (y\ \ . :: C\ \ z)\ \ var\ \ (y,\ z)\ \ val\ \ (y)\\
\ \ where\ \ (y :: * \ is\ \ structure)\ then\ (let :: seq\ (w2,\ w3)\ be\\
\quad\quad ((-1 : type,\ \ 0 : y :: *,\ \ 1 : structure),\\
\quad\quad\ \ ((-1 : body,\ \ 0 : w2,\ \ 1 : structure - type)\ ..\ z))\ in\\
\quad\ (if\ \ w3\ \ then\\
\quad\quad (left : ((-1 : body,\ \ 0 : y :: *,\ \ 1 : structure)\ ..\ z),\ \ type : w3) :: q\\
\quad\quad else\ \ und))\\
elseif\ \ x\ \ matches\ \ (y\ \ [\ z\ ])\ \ var\ \ (y,\ z)\ \ val\ \ (z,\ y)\\
\ \ where\ \ (y :: * \ is\ \ array)\ then\ (let :: seq\ (w2,\ w3,\ w4)\ be\\
\quad\quad ((-1 : element{-}type,\ \ 0 : y :: *,\ \ 1 : array),\ \ (cast\ \ z :: * \ int),\\
\quad\quad\ \ (-1 : length,\ \ 0 : y :: *,\ \ 1 : array))\ in\\
\quad\ (if\ \ ((w2\ \ is\ \ type)\ \ and\ \ w3\ \ and\ \ (w3 :: q\ \ >=\ \ 0)\ \ and\\
\quad\quad\quad (w3 :: q\ \ <\ \ w4 :: q))\ \ then\\
\quad\quad (left : ((-1 : body,\ \ 0 : y :: *,\ \ 1 : array)\ .\ z :: * :: q),\\
\quad\quad\ \ type : w4) :: q\ \ else\ \ und))\\
else\ \ und))).
\end{array}
$$

The predefined CTSL element $(e_{l.1}\ <\ e_{l.2})$ specifies that $e_{l.1}$ is less than $e_{l.2}$. The predefined CTSL element $(if\ c_{nd}\ then\ e_{l.*}\ elseif\ c_{nd.1}\ e_{l.*.1})$ is a shortcut for $(if\ c_{nd}\ then\ e_{l.*}\ else\ (if\ c_{nd.1}\ e_{l.*.1}))$.

Let $e_{l.*}\ \#\ c_{nf}$ be a shortcut for $[c_{nf}\ (0 : ()) :: state :: program : (e_{l.*})]$. Let $e_{l.*}\ \#\ v_l\ \#\ c_{nf}$ be a shortcut for $[c_{nf}\ (0\ :\ ())\ ::\ state\ ::\ program\ :\ (e_{l.*}), (0 : ()) :: state :: value : v_l]$. The atoms *program* and *value* are the names of the substates of configurations in CTSL specifying the information about programs and the returned values in CTSL [7]. The conceptuals $(0 : ()) :: state :: program$ and $(0 : ()) :: state :: value$ store the current program and the returned value, respectively.

The element $(cast\ e_l\ t_p)$ representing the query $([r^-_{p.q}\ t_p])\ [r^-_{p.q}\ e_l]$ is defined as follows:

$(rule\ \ (cast\ \ x\ \ y)\ \ var\ \ (x,\ \ y)\ \ abn\ \ val\ \ (x)\ \ where\ \ (y\ \ is\ \ type)$
$\ then\ \ (cast\ \ x :: *\ \ y) :: atm)$;
$(transition\ \ (cast\ \ x\ \ y) :: atm\ \ var\ \ (x,\ \ y)\ \ then\ \ f_n)$,

where

- if $[r^-_{p.a}\ v_l]$ is a result of conversion of $[r^-_{p.q}\ x_0]$ to $[r^-_{p.q}\ y_0]$, then $(cast\ x_0\ y_0) :: atm,\ e_{l.*}\ \#\ c_{nf} \to_{f_n,(x:x_0,y:y_0)} e_{l.*}\ \#\ v_l\ \#\ c_{nf}$.

The syntax and semantics of atomic transitions is defined in [7].

The element $(e_l\ + :: C\ e_{l.1})$ representing the query $[r^-_{p.q}\ e_l]\ +\ [r^-_{p.q}\ e_{l.1}]$ which specifies the sum of numbers $[r^-_{p.q}\ e_l]$ and $[r^-_{p.q}\ e_{l.1}]$ of the type *int* is defined as follows:

$(rule\ \ (x\ + :: C\ \ y)\ \ var\ \ (x,\ \ y)\ \ abn\ \ val\ \ (x)$
$\ where\ \ (x\ \ is\ \ int)\ \ and\ \ (y\ \ is\ \ int)\ \ then\ \ (x\ +\ ::\ C\ \ y)\ ::\ (int\ \ +\ \ int)\ ::$
$atm)$;
$(transition\ \ (x\ + :: C\ \ y) :: (int\ \ +\ \ int) :: atm\ \ var\ \ (x,\ \ y)\ \ then\ \ f_n)$,

where

- if $[r^-_{p.a}\ v_l]$ is the result of addition of $[r^-_{p.q}\ x_0]$ and $[r^-_{p.q}\ y_0]$ returned by AM[C], then $(x_0\ +\ ::\ C\ y_0)\ ::\ (int\ \ +\ \ int)\ ::\ atm,\ e_{l.*}\ \#\ c_{nf} \to_{f_n,(x:x_0,y:y_0)} e_{l.*}\ \#\ v_l\ \#\ c_{nf}$.

The elements $(new\ t_p)$ and $(delete\ p_{n.t})$ representing the dynamic memory management queries are defined by the rules

$(rule\ \ (new\ \ x)\ \ var\ \ (x)\ \ abn\ \ where\ \ (x\ \ is\ \ type)\ \ then$
$\ (let\ \ w\ \ be\ \ (new\ \ pointer{-}id)\ \ in$
$\ \ ((0 : (id : w,\ \ type : x) :: pointer,\ \ 1 : pointer)\ \ ::=\ \ true),$
$\ \ \ (id : w,\ \ type : x) :: q))$;
$(rule\ \ (delete\ \ x)\ \ var\ \ (x)\ \ abn\ \ val\ \ (x)$
$\ where\ \ (x :: *\ \ is\ \ typed{-}pointer{-}literal)\ \ then$
$\ \ ((-1 : value,\ \ 0 : x :: *,\ \ 1 : pointer)\ \ ::=),$
$\ \ \ ((0 : x :: *,\ \ 1 : pointer)\ \ ::=))$.

The element $(new{-}cc\ c_{ncp.c})$ generates a new instance of the countable concept $c_{ncp.c}$ [7]. The element *pointer−id* is a countable concept specifying unique identifiers of addresses.

We have considered the ways of constructing the definitions for C expressions by the examples of some C operators. The construction of a definition for a function call can be found in [3]. The definitions for other C operators are constructed in a similar way.

### 8.4. Programs

The element sequence $e_{l.1}$ ... $e_{l.n_t}$ represents the program $[r^-_{p.q}\ e_{l.1}]$ ... $[r^-_{p.q}\ e_{l.n_t}]$ in C.

## 9. Conclusion

The method presented in this paper describes the stepwise well-defined process of operational semantics development for imperative programming languages. Therefore, it can became a basis of the technology of operational semantics development for this class of languages.

The fragment of the C language used as the case study for this method covers a representative set of constructs of procedural programming languages. Thus, the paper can be also considered as a cookbook on the development of operational semantics for procedural programming languages.

## References

[1] Anureev I.S. Operational ontological approach to formal programming language specification // Programming and Computer Software. – 2009. – Vol. 35, No. 1. – P. 35–42.

[2] Parnas D.L. Really rethinking formal methods // Computer. IEEE Computer Society. – 2010. – Vol. 43, No. 1. – P. 28–34.

[3] Anureev I.S. Operational semantics development for procedural programming languages based on conceptual transition systems // Bulletin NCC. Series: Computer Science. – 2015. – Vol. 38. – P. 1–28.

[4] Gurevich Y. Abstract state machines: an overview of the project. foundations of information and knowledge systems (FoIKS) // Lect. Notes Comput. Sci. – 2004. – Vol. 2942. – P. 6–13.

[5] AsmL: The Abstract State Machine Language. Reference Manual, 2002. – URL: http://research.microsoft.com/en-us/projects/asml/.

[6] Matthias Anlauff. XasM – An Extensible, Component-Based Abstract state Machines Language. – URL: http://xasm.sourceforge.net/XasmAnl00/XasmAnl00.html.

[7] Anureev I.S. Formalisms for conceptual design of information systems // System Informatics. – 2016. – No. 8. – P. 53–88.

[8] Anureev I.S. Formalisms for conceptual design of closed information systems // System Informatics. – 2016. – No. 7. – P. 69–148.

[9] Nepomniaschy V.A., Anureev I.S., Mikhailov I.N., Promskii A.V. Towards verification of C programs. C-light language and its formal semantics // Programming and Computer Software. – 2002. – Vol. 28, No. 6. – P. 314–323.