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 $S_i$ be a set of sets. Assume that $I_{nt}$, $N_i$, $N_{i0}$ and $B_i$ 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 $i_{nt1}$ are elements of $I_{nt}$.

Let $s_{t,()}$, $s_{t,\{\}}$, and $s_{t,*}$ denote the sets of sequences of the forms $(o_{b,1}, \ldots, o_{b,n_{i0}})$, $\{o_{b,1}, \ldots, o_{b,n_{i0}}\}$, and $o_{b,1}, \ldots, o_{b,n_{i0}}$ from elements of $s_t$.

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

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., $\text{[support } f_n] = \{a_{rg} : [f_n a_{rg}] \neq \text{und}\}$.

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 $[[f_n u_p, u_{p,s,nt}]]$ be a shortcut for $[[f_n u_p]] u_{p,s,nt}$; $[f_n a_{rg}, a_{rg.1}, \ldots, a_{rg.n_t} : v_l]]$, for $[f_n a_{rg} : [[f_n a_{rg}] a_{rg.1} \ldots a_{rg.n_t} : v_l]]$; and $[u_p, s]$ for $[[f_n u_p]]$, where $\text{[support } f_n] = \emptyset$.

Let $S_t$, $O_{b,s}$, $Q_r$, $O_{b,q}$, $A_{ns}$ 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:


   (a) Describe the sets of proper state objects. An object $o_{b,s}$ is a proper state object if $o_{b,s} \notin S_t$.
   (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 \( r_{p.s} \in O_{b.s} \rightarrow S_{tr} \), \( r_{p.q} \in O_{b.q} \rightarrow E_l \) and \( r_{p.a} \in O_{b.a} \rightarrow E_l \) 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

\[(\text{rule} \ (x \text{ is name}) \ \text{var} \ (x) \ \text{abn} \ \text{then} \ (x \text{ is normal}).)\]

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

\[(\text{rule} \ (x \text{ is label}) \ \text{var} \ (x) \ \text{abn} \ \text{then} \ (x \text{ matches } y :: \text{label var} \ (y) \ \text{where} \ (y \text{ is name}))).\]

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

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

\[(\text{rule} \ (x \text{ is type−literal}) \ \text{var} \ (x) \ \text{abn} \ \text{then} \ (x \text{ is basic−type}) \ \text{or} \ (x \text{ is derived−type−literal}))).\]

The predefined CTSL element \((c_{nd,1} \text{ or } c_{nd,2})\) 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

\[(\text{rule} \ (x \text{ is basic−type}) \ \text{var} \ (x) \ \text{abn} \ \text{then} \ (x :: q \ \text{in} :: \text{set} \ (\text{int}, \ \text{float}, \ ...) :: q)),\]

where \(\text{int, float, ...}\) is a sequence of all basic types of C. The predefined CTSL element \((e_l \text{ in } :: \text{set} \ (e_{l,s}))\) specifies that \(e_l\) is an element of the sequence \(e_{l,s}\). The element of the form \(e_l :: q\) is called a quoted element.
The value of the quoted element $e_i :: q$ in CTSL is defined as $e_i$. Let $T_{p,b} = [\text{content basic-type}].$

A derived type is represented by an instance of the concept $\text{derived-type-literal}$ defined by the rule

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

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

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

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

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

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

A pointer type is represented by an instance of the concept $\text{pointer-type-literal}$ defined by the rule

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

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

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

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 $[[s_{it}[\text{ITS[C]]}]]$ is represented by an instance of the concept $\text{type}$ in $[[r_{p,s} s_{it}]]$ defined by the rule

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

A derived type in $[[s_{it}[\text{ITS[C]}]]]$ is represented by an instance of the concept $\text{derived-type}$ in $[[r_{p,s} s_{it}]]$ defined by the rule

$$(\text{rule } (x \text{ is derived-type}) \text{ var } (x) \text{ abn then } (x \text{ is pointer-type}) \text{ or } (x \text{ is array-type}) \text{ or } (x \text{ is structure-type}) \text{ or } (x \text{ is function-type}))).$$
An array type in $[s_{Lt}[ITS[C]]]$ is represented by an instance of the concept array-type in $[r_{p,s} s_{Lt}]$ defined by the rule

\[(\text{rule }} (x \text{ is array-type}) \text{ var } (x) \text{ abn then} \]
\[(x \text{ matches } (\text{array } y) \text{ where } (y \text{ is type})).\]

The conceptual $(1 : t_{p,s}, 1 : \text{structure-type})$ 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 $[c_{nf} (0 : t_{p,s}, 1 : \text{structure-type})] \neq \text{und}$. The conceptual $(-1 : body, 0 : t_{p,s}, 1 : \text{structure-type})$ 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 $[c_{nf} (-1 : body, 0 : t_{p,s}, 1 : \text{structure-type})] = b_d$. The element $b_d$ is an attribute element. The attribute element with the attributes $a_{Lt,1}, ..., a_{Lt,n_0}$ has the form $(a_{Lt,1} : v_{Lt,1}, ..., a_{Lt,n_0} : v_{Lt,n_0})$, where $v_{Lt,1}, ..., v_{Lt,n_0}$ are the values of the attributes $a_{Lt,1}, ..., a_{Lt,n_0}$. 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 $[c_{nf} (-1 : body, 0 : t_{p,s}, 1 : \text{structure-type})] f_l \neq \text{und}$. A type $t_p$ is a type in $[f_l, t_{p,s}, c_{nf}]$ if $[c_{nf} (-1 : body, 0 : t_{p,s}, 1 : \text{structure-type})] f_l = t_p$.

A structure type in $[s_{Lt}[ITS[C]]]$ is represented by an instance of the concept structure-type in $[r_{p,s} s_{Lt}]$ defined by the rule

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

A field in $[[[r_{p,s} t_{p,s}]]]$ in $[s_{Lt}[ITS[C]]]$ is represented by an instance of the concept (field in $t_{p,s}$) in $[r_{p,s} s_{Lt}]$ defined by the rule

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

The predefined CTSL element $(e_{Lt,a} .. a_{Lt})$ specifies the value of the attribute $a_{Lt}$ of the attribute element $e_{Lt,a}$.

A pointer type in $[s_{Lt}[ITS[C]]]$ is represented by an instance of the concept pointer-type in $[r_{p,s} s_{Lt}]$ defined by the rule

\[(\text{rule }} (x \text{ is pointer-type}) \text{ var } (x) \text{ abn then} \]
\[(x \text{ matches } (\text{pointer } y) \text{ where } (y \text{ is type})).\]

A function type in $[s_{Lt}[ITS[C]]]$ is represented by an instance of the concept function-type in $[r_{p,s} s_{Lt}]$ defined by the rule

\[(\text{rule }} (x \text{ is function-type}) \text{ var } (x) \text{ abn then} \]
\[(x \text{ matches } (\text{function } y z) \text{ var } (y, z) \text{ where } ((y \text{ is (sequence type)}) \text{ and } (z \text{ is type}))).\]
The predefined CTSL element \((e_l \text{ is } \text{sequence } c_{ncp})\) 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 \(\text{variable–literal}\) defined by the rule

\[
\text{(rule } (x \text{ is variable–literal}) \text{ abn var } (x) \text{ then}\n\text{ (x matches y :: variable var } (y) \text{ where } (y \text{ is name}))\text{).}
\]

Let \(V_r = [\text{content variable–literal}]\).

A relative variable scope is represented by an instance of the concept \(\text{scope}\) defined by the rule

\[
\text{(rule } (x \text{ is scope}) \text{ var } (x) \text{ abn then } (x \text{ is nat0})\text{).}
\]

The predefined CTSL element \((e_l \text{ is nat0})\) specifies that \(e_l \in N_{0}\). Let \(S_{cp} = [\text{content scope}]\).

An array is represented by an instance of the concept \(\text{array–literal}\) defined by the rule

\[
\text{(rule } (x \text{ is array–literal}) \text{ var } (x) \text{ abn then } (x \text{ matches y :: array var } (y) \text{ where } (y \text{ is nat}))\text{).}
\]

The predefined CTSL element \((e_l \text{ is nat})\) specifies that \(e_l \in N_{1}\). Let \(A_{rr} = [\text{content array–literal}]\).

A structure is represented by an instance of the concept \(\text{structure–literal}\) defined by the rule

\[
\text{(rule } (x \text{ is structure–literal}) \text{ var } (x) \text{ abn then } (x \text{ matches y :: structure var } (y) \text{ where } (y \text{ is nat}))\text{).}
\]

Let \(S_{strc} = [\text{content structure–literal}]\).

A field is represented by an instance of the concept \(\text{field–literal}\) defined by the rule

\[
\text{(rule } (x \text{ is field–literal}) \text{ var } (x) \text{ abn then } (x \text{ matches y :: field var } (y))\text{).}
\]

Let \(F_{f} = [\text{content field–literal}]\).

A function is represented by an instance of the concept \(\text{function–literal}\) defined by the rule

\[
\text{(rule } (x \text{ is function–literal}) \text{ var } (x) \text{ abn then } (x \text{ matches y :: function var } (y) \text{ where } (y \text{ is name}))\text{).}
\]

Let \(F_{n} = [\text{content function–literal}]\).

A formal argument of a function is represented by an instance of the concept \(\text{argument}\) defined by the rule
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 $L_{c,l} =$ [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 $P_{n,s} = [\text{content structure:pointer:literal}]$.

The concept $\text{function:pointer:literal}$ 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 $P_{n,f} = [\text{content function:pointer:literal}]$.

The conceptual $(0 : p_{n,t}, 1 : \text{pointer})$ 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 $[c_{nf} (0 : p_{n,t}, 1 : \text{pointer})] \neq \text{und}$. The conceptual $(−1 : \text{value}, 0 : p_{n,t}, 1 : \text{pointer})$ 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 $v_l = [c_{nf} (−1 : \text{value}, 0 : p_{n,t}, 1 : \text{pointer})]$.

A pointer in $[[s_{tt}[\text{ITS}]]]$ is represented by an instance of the concept $\text{pointer}$ 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 $(0 : a_{rr}, 1 : \text{array})$ 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 $[c_{nf} (0 : a_{rr}, 1 : \text{array})] \neq \text{und}$. The conceptual $(−1 : \text{element:} \text{type}, 0 : a_{rr}, 1 : \text{array})$ 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 $[c_{nf} (−1 : \text{element:} \text{type}, 0 : a_{rr}, 1 : \text{array})] = t_p$. A type $(\text{array } t_p)$ is a type in $[[a_{rr}, c_{nf}]]$ if $t_p$ is an element type in $[[a_{rr}, c_{nf}]]$. The conceptual $(−1 : \text{body}, 0 : a_{rr}, 1 : \text{array})$ in $[c_{nf}]$ represents elements in $[[a_{rr}, r_{p,s}, c_{nf}]]$. A sequence element $b_l$ is a body in $[[a_{rr}, c_{nf}]]$ if $b_l = [c_{nf} (−1 : \text{body}, 0 : a_{rr}, 1 : \text{array})]$. The element $e_l$ is an element in $[[a_{rr}, c_{nf}, n_{l}]$ if $[c_{nf} (−1 : \text{body}, 0 : a_{rr}, 1 : \text{array}), n_{l}, 0 ≤ n_{l} < \text{len} [c_{nf} (−1 : \text{body}, 0 : a_{rr}, 1 : \text{array})]]$. The element $e_l$ is an element in $[[a_{rr}, c_{nf}, n_{l}]]$ if $e_l$ is an element in $[[a_{rr}, c_{nf}, n_{l}]]$ for some $n_{l}$.

An array in $[[s_{tt}[\text{ITS}]]]$ is represented by an instance of the concept $\text{array}$ 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 $(0 : s_{trc}, 1 : \text{structure})$ in $[c_{nf}]$ represents the structure in $[[r_{p,s}, c_{nf}]]$. A structure $s_{trc}$ is a structure in $[c_{nf}]$ if $[c_{nf} (0 : s_{trc}, 1 : \text{structure})] \neq \text{und}$. The conceptual $(−1 : \text{type}, 0 : s_{trc}, 1 : \text{structure})$ 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 $[c_{nf} (−1 : \text{type}, 0 : s_{trc}, 1 : \text{structure})] = t_{p,s}$. The conceptual $(−1 : \text{body}, 0 : s_{trc}, 1 : \text{structure})$ 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 
\([c_{nf} (−1 : body, 0 : s_{trc}, 1 : structure) = b_d\). A field \(f_i\) is a field in \([s_{trc}, c_{nf}]\) if 
\([c_{nf} (−1 : body, 0 : s_{trc}, 1 : structure)] f_i \neq \text{und}\). An element \(v_l\) is a value in 
\([f_i, t_{p,s}, c_{nf}]\) if \([c_{nf} (−1 : body, 0 : s_{trc}, 1 : structure)] f_i \approx v_l\).

A structure in \([s_{trc} \text{ITS}(C)]\) is represented by an instance of the concept 
structure in \([r_{p,s} s_{trc}]\) defined by the rule

\[(\text{rule } (x \text{ is structure}) \var{abn} \text{ then } ((x \text{ is structure–literal}) \text{ or } (0 : x, 1 : structure))).\]

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

The concepts \((0 : level) :: state :: function\) and \((0 : type) :: state :: function\) specify the current call level and the return type in it, respectively.

The conceptual \((−2 : l_{v,c}, −1 : s_{cp}, 0 : v_r, 1 : variable)\) 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 \([c_{nf} (−2 : l_{v,c}, −1 : s_{cp}, 0 : v_r, 1 : variable)] \neq \text{und}\). The conceptual \((−3 : pointer, −2 : l_{v,c}, −1 : s_{cp}, 0 : v_r, 1 : variable)\) 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 \([c_{nf} (−3 : pointer, −2 : l_{v,c}, −1 : s_{cp}, 0 : v_r, 1 : variable)] = p_{n,v}\). The conceptual \((−3 : type, −2 : l_{v,c}, −1 : s_{cp}, 0 : v_r, 1 : variable)\) 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 \([c_{nf} (−3 : type, −2 : l_{v,c}, −1 : s_{cp}, 0 : v_r, 1 : variable)] = t_p\). The conceptual \((−3 : value, −2 : l_{v,c}, −1 : s_{cp}, 0 : v_r, 1 : variable)\) 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 \([c_{nf} (−3 : value, −2 : l_{v,c}, −1 : s_{cp}, 0 : v_r, 1 : variable)] = v_l\).

The information about blocks is represented by the substate block
in configurations. The conceptual \((0 : scope) :: block\) specifies the current
A name \( n_m \) can have a set of possible values in \([n_f]\). For example, possible values of the name \( v_r \) of the form \( n_m, v \) in \([n_f]\) are the variables with the name \( n_m, v \) of scopes from 0 to the current scope in \([n_f]\). 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 of variables with the name \( n \) possible values of the name \( J \) in \([n_f]\) are scopes from 0 to the current scope in \([n_f]\). Then the name resolution problem is reduced to the choice of a right index in \([name : n_m, c_{nf}]\).

A variable in \([s_{it}, ITS[C]]\) is represented by an instance of the concept \texttt{variable} in \([r_{ps}, su]\] defined by the rule

\[
\text{(rule (x is variable) var (x) abn then (index in x)).}
\]

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

\[
\text{(rule (index in x) var (x) abn where (x is variable–literal) then (let :: seq (w1, w2) be (current–scope, current–call–level) in (index in x in w1, w2)))};
\]

\[
\text{(rule (x is index in y, z) var (x, y, z) abn then (if ~2 : y, ~1 : z, 0 : x, 1 : variable) then z :: q else (if (z = 0) then and else (let w be (z - 1) in (index in x in y, w)))).}
\]

The predefined CTSL element \((let :: seq (v_r, a) be (e_{1,s}) in b_d)\), where \( b_d \in E_{1,s} \), replaces the variables \( v_{r,s} \) in the body \( b_d \) by the values of the corresponding elements of \( e_{1,s} \) and executes the resulting body.

The elements \texttt{current–scope} and \texttt{current–call–level} are defined by the rules

\[
\text{(rule current–scope then (0 : scope :: state :: block));}
\]

\[
\text{(rule current–scope then (0 : call–level :: state :: function)).}
\]

A variable \( v_r \) is global in \([n_f]\) if the right index in \([v_r, c_{nf}]\) equals 0. A variable \( v_r \) is local in \([n_f]\) if the right index in \([v_r, c_{nf}]\) is greater than 0. The type and value of a global variable \( v_r \) in \([n_f]\) 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 \texttt{value–literal} defined by the rule

\[
\text{(rule (x is value–literal) var (x) abn then ((x is int) or (x is float) or ... or (x is pointer–literal) or (x is array–literal) or (x is structure–literal) or (x is function–literal))).}
\]
where ... are disjuncts of the form \( (x \text{ is } t_{p,b}) \) for all \( t_{p,b} \). Let \( V_l = [\text{content value} - \text{literal}] \).

A value in \( [s_{tt}][\text{ITS[C]}] \) is represented by an instance of the concept \text{value} in \( [[r_p,s_{tt}]] \) defined by the rule

\[
\text{(rule} \ (x \text{ is value}) \ \text{var} \ (x) \ \text{abn then} \ \text{((x is int) or (x is float) or ... or (x is pointer) or (x is array) or (x is structure) or (x is function)))},
\]

where ... are disjuncts of the form \( (x \text{ is } t_{p,b}) \) 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 \( (\text{type} : \text{break}) :: ex, (\text{type} : \text{continue}) :: ex \) and \( (\text{type} : \text{goto}, \text{label} : l_b) :: ex \) represent the execution of the break statement, the continue statement, and the goto statement with the label \( l_b \), respectively. The exceptions \( (\text{type} : \text{return}, \text{value} : v_l) :: ex \) and \( (\text{type} : \text{return}) :: ex \) 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 \( (\text{goto} \ l_b) \) representing the queries break;; continue; and goto \( r_{p,q} \ l_b \); are defined by the rules

\[
\text{(rule} \ \text{break} \ \text{abn then} \ (\text{type} : \text{break}) :: ex); \\
\text{(rule} \ \text{continue} \ \text{abn then} \ (\text{type} : \text{continue}) :: ex); \\
\text{(rule} \ (\text{goto} \ x) \ \text{var} \ (x) \ \text{abn where} \ (x \text{ is label}) \ \text{then} \ (\text{type} : \text{goto}, \ \text{label} : x) :: ex).
\]

The element \( n_m :: \text{label} \) representing queries of the form \( [r_{p,q} \ n_m : s_{ttm}] \) as \( (\text{seq} \ n_m :: \text{label} \ [r_{p,q} s_{ttm}]) \) is defined by the rule
The predefined CTSL element \((\text{throw } e_l)\) assigns the value of the element \(e_l\) to a special conceptual \((0 : ()) \:: \text{state} :: \text{value}\) specifying the current value of the CTS CTSL[C]. The predefined CTSL element \((\text{catch } v_r \ e_{l,s})\) replaces the variable \(v_r\) in the sequence \(e_l\) by the value of the conceptual \((0 : ()) \:: \text{state} :: \text{value}\) and executes the resulting sequence. The predefined CTSL element \((\text{if } e_l \text{ matches } p_u \text{ var } (v_r,s) \text{ where } c_{nd} \text{ then } e_{l,s,1} \text{ else } e_{l,s,2})\) specifies that if the element \(e_l\) matches the pattern \(p_u\) with the variables \(v_{r,s}\) and the condition \(c_{nd}\) is true for the corresponding values of these variables, then the sequence \(e_{l,s,1}\) is executed for these values of the variables. If \(e_l\) does not match \(p_u\), then the sequence \(e_{l,s,2}\) 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 \((\text{return } e_l)\) and \((\text{return;})\) representing the queries \(\text{return } [r_{p,q} e_l];\) and \(\text{return;}\) are defined by the rules

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

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

The element \((\text{block } e_{l,s})\) representing the query \(\{[r_{p,q} e_{l,s}]\}\) is defined by the rule

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

The element \(\text{enter–block}\) specifying the actions executed when the current configuration enters the block is defined by the rule

\[
\text{(rule enter–block abn then current–scope ++).}
\]

The element \(\text{current–scope ++}\) is defined by the rule
The predefined CTSL element ($cnptl ::= e$) assigns the value of $e$ to the conceptual $cnptl$. The predefined CTSL element ($e_1 + e_2$) specifies the sum of $e_1$ and $e_2$.

The element ($block:variables in (e)$) returning the sequence of the local variables defined in declaration statements that are the elements of $e$ is defined by the rules

\begin{verbatim}
(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 ()).
\end{verbatim}

The predefined CTSL element ($e_1 + (e_2)$) adds the element $e_1$ to the head of the sequence $(e_2)$.

The element ($block:labels in (e)$) returning the sequence of the labels that are the elements of $e$ is defined by the rules

\begin{verbatim}
(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 ()).
\end{verbatim}

The element ($continue:block in x; (y)$) handling goto exceptions when the current configuration reaches the end of the block is defined by the rule

\begin{verbatim}
(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)))).
\end{verbatim}

The element ($exit:block in x; type$) specifying the actions executed when the current configuration exits the block is defined by the rule

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

The element current:scope -- is defined by the rule

\begin{verbatim}
(rule current:scope -- abn then
  ((0 : scope) :: state :: block ::= ((0 : scope) :: state :: block -- 1))).
\end{verbatim}
The predefined CTSL element \((e_{l1} - e_{l2})\) specifies the difference between \(e_{l1}\) and \(e_{l2}\).

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

\[(\text{rule } (\text{delete}\,\text{-variables in } x) \text{ var } (x) \text{ abn then} \]
\[\text{(let }::=\text{ seq } (w_1, w_2) \text{ be } (\text{current\,-scope, current\,-call\,-level}) \text{ in} \]
\[\text{(foreach } y \text{ in } x :: q \text{ do} \]
\[((-3 : w_2, -2 : w_1, -1 : \text{pointer}, 0 : y, 1 : \text{variable}) ::=), \]
\[((-3 : w_2, -2 : w_1, 0 : y, 1 : \text{variable}) ::=)).\]

The predefined CTSL element \((c_{ncp1} ::=)\) is a shortcut for \((c_{ncp1} ::= \text{ und})\). The predefined CTSL element \((\text{foreach } v_r \text{ in } (e_{l1,*}) \text{ do } b_d)\), 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

\[(\text{rule } (\text{;}) \text{ var } (x) \text{ abn then } x);\]
\[(\text{rule } ";" \text{ var } (x) \text{ abn then})\]

The element \((\text{if } :: C \text{ cnd then } e_{l1} \text{ else } e_{l2})\) representing the query \(\text{if } [r_{p,q} (c_{nd})] \text{ then } [r_{p,q} e_{l1}] \text{ else } [r_{p,q} e_{l2}]\) is defined by the rule

\[(\text{rule } (\text{if } :: C x \text{ then } y \text{ else } z) \text{ var } (x, y, z) \text{ abn} \]
\[\text{then } (\text{if } (\text{cast } x \text{ int}) \text{ !} = 0) \text{ then } (\text{block } y) \text{ else } (\text{block } z)).\]

Other switch statements are defined in a similar way.

The element \((\text{while } :: C \text{ cnd do } e_l)\) representing the query \((\text{while } ([r_{p,q} c_{nd}]) [r_{p,q} e_l])\) is defined by the rule

\[(\text{rule } (\text{while } :: C x \text{ do } y) \text{ var } (x, y) \text{ abn then} \]
\[(\text{while } ((\text{cast } x \text{ int}) \text{ !} = 0) \text{ do } (\text{block } y, (\text{delete\,-exception continue}))), \]
\[(\text{delete\,-exception break})).\]

The predefined CTSL element \((\text{while } c_{nd} \text{ do } b_d)\), 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 \((\text{function } f_n (a_{rg,1} : t_{p,1}, ..., a_{rg,n_{t,0}} : t_{p,n_{t,0}}) : t_p \text{ b}_d)\) representing the query \([r_{p,q} t_p][r_{p,q} f_n](t_{p,q} t_{p,1}) [r_{p,q} a_{rg,1}];, ..., [r_{p,q} t_{p,n_{t,0}}] [r_{p,q} a_{rg,n_{t,0}}])\) \([r_{p,q} b_d]\) 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 (el is attribute–element) specifies that el is an
attribute element. The predefined CTSL elements (attributes in el,a) and
(values in el,a) specify the sequence of attributes in the attribute element
eel,a and the sequence of their values, respectively. The predefined CTSL
element (not cnd) specifies the negation of cnd.

The element (var vr tp) representing the query [r−p,q tp][r−p,q vr]; 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 tp,s (f1,1 : tp,1, ..., fn,m : tp,n)) representing the
query struct [r−p,q tp,1][r−p,q f1,1]; ...; [r−p,q tp,n][r−p,q fn,m]) 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 v1 representing the query v1 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 \([r_{p,q}^\neg v_r]\) is defined by the rule

\[
\text{rule } x \text{ var } (x) \text{ abn then (let } :: \text{ seq (w1, w2) be }
\n\begin{align*}
&((\text{index in } x), (\text{current-call-level in } w1)) \text{ in } \\
&(-3 : \text{value}, -2 : w2, -1 : w1, 0 : x, 1 : \text{variable})).
\end{align*}
\]

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

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

The element \((e_l [e_{l,1}] )\) representing the query \([r_{p,q}^\neg e_l][r_{p,q}^\neg e_{l,1}]\) is defined by the rule

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

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 \(((e_{l,*}) . n_l)\) specifies the \(n_l\)-th element of the sequence \(e_{l,*}\).

The element \((e_l :: C f_l)\) representing the query \([r_{p,q}^\neg e_l][r_{p,q}^\neg f_l]\) is defined by the rule

\[
\text{rule } (x :: C y) \text{ var } (x, y) \text{ abn val } (x) \text{ where } (x :: * \text{ is structure) and (y is field-literal) then } \\
\begin{align*}
&((-1 : \text{body}, 0 : x :: *, 1 : \text{structure} .. y)).
\end{align*}
\]

The element \((e_l := e_{l,1})\) representing the query \([r_{p,q}^\neg e_l] := [r_{p,q}^\neg e_{l,1}]\) is defined by the rule

\[
\text{rule } (x := y) \text{ var } (x, y) \text{ abn val } (y) \text{ then (let } :: \text{ seq (w1, w2, w3, w4) be ((left-hand in } x),
\n\begin{align*}
&(w1 .. left), (w1 .. type), (cast y :: * :: q w3)) \text{ in (if (w1 and } w4) \text{ then (w2 := w4 :: q) else und})).
\end{align*}
\]

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

\[
\text{rule } (\text{left-hand in } x) \text{ var } (x) \text{ abn then (let } w1 \text{ be (index in } x) \text{ in (if } w1 \text{ then (let } :: \text{ seq (w2, w3) be ((current-call-level in } w1),}
\n\begin{align*}
&(-3 : \text{type}, -2 : w2, -1 : w1, 0 : x, 1 : \text{variable}) \text{ in (left : (-3 : value, -2 : w2, -1 : w1, 0 : x, 1 : variable),}
\n&\text{type : w3 :: q})
\end{align*}
\]

else if \(x \text{ matches } (* y) \text{ var } (y) \text{ val } (y) \text{ then (if } (y :: * \text{ is typed-pointer) then (let } w2 \text{ be (y :: * :: q .. type) in (left : (-1 : value, 0 : y :: *, 1 : pointer), type : w2 :: q))}
\nelse if \((y :: * \text{ is variable-pointer) then (let } :: \text{ seq (w2, w3, w4, w5})
\]
be (\((y :: * .. \text{variable}), (y :: * .. \text{scope}), (y :: * .. \text{call-level}),
(\!-\!3 : \text{type}, -2 : w4, -1 : w3, 0 : w2, 1 : \text{variable})\)) in
(left : (\((-3 : \text{value}, -2 : w4, -1 : w3, 0 : w2, 1 : \text{variable}),
\text{type} : w5) :: q)\)

\text{elseif} (y :: * \text{is array-pointer}) \text{then} (let :: seq (w2, w3, w4)
be ((y :: * .. \text{array}), (y :: * .. \text{index}),
(1 : \text{element-type}, 0 : w2, 1 : \text{array})) in
(left : ((-1 : \text{body}, 0 : w2, 1 : \text{array}) . w3 :: q), \text{type} : w4 :: q)\)

\text{elseif} (y :: * \text{is structure-pointer}) \text{then}
(let :: seq (w2, w3, w4, w5)
be ((y :: * .. \text{structure}), (y :: * .. \text{field}),
(1 : \text{type}, 0 : w2, 1 : \text{structure}),
((-1 : \text{body}, 0 : w4, 1 : \text{structure-type}) .. w3)) in
(left : ((-1 : \text{body}, 0 : w2, 1 : \text{structure}) .. w3), \text{type} : w5 :: q)\)
else und\))

\text{elsif} x \text{matches} (y :: C z) \text{var} (y, z) \text{val} (y)
where (y :: * \text{is structure}) \text{then} (let :: seq (w2, w3)
be ((-1 : \text{type}, 0 : y :: *, 1 : \text{structure}),
((-1 : \text{body}, 0 : w2, 1 : \text{structure-type}) .. z)) in
(if w3 \text{then}
(left : ((-1 : \text{body}, 0 : y :: *, 1 : \text{structure}) .. z), \text{type} : w3) :: q
\text{else und})\))

\text{elsif} x \text{matches} (y \ [z \ ] \text{var} (y, z) \text{val} (y)
where (y :: * \text{is array}) \text{then} (let :: seq (w2, w3, w4)
be ((-1 : \text{element-type}, 0 : y :: *, 1 : \text{array}), (\text{cast} z :: * \text{int}),
(-1 : \text{length}, 0 : y :: *, 1 : \text{array})) in
(if ((w2 \text{is type}) \text{and} w3 \text{and} (w3 :: q >= 0) \text{and}
\text{(w3 :: q < w4 :: q)) then
(left : ((-1 : \text{body}, 0 : y :: *, 1 : \text{array}) . z :: * :: q),
\text{type} : w4 :: q) \text{else und})\))
else und\)).

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 \(c_{nd}\) \text{then} \(e_{l,s}\) \text{elseif} \(c_{nd,1} e_{l,s,1}\)) is a shortcut for (if \(c_{nd}\) \text{then} \(e_{l,s}\) \text{else} (if \(c_{nd,1} e_{l,s,1}\))).

Let \(e_{l,s} \# c_{nf}\) be a shortcut for \([c_{nf} \ (0 : () :: \text{state} :: \text{program} : (e_{l,s})]\).
Let \(e_{l,s} \# v_i \# c_{nf}\) be a shortcut for \([c_{nf} \ (0 : () :: \text{state} :: \text{program} : (e_{l,s}), (0 : () :: \text{state} :: \text{value} : v_i])\). The atoms \text{program} and \text{value} are the names of the substates of configurations in CTSL specifying the information about programs and the returned values in CTSL \cite{7}. The conceptuals \(0 : () :: \text{state} :: \text{program}\) and \(0 : () :: \text{state} :: \text{value}\) store the current program and the returned value, respectively.
The element \((\text{cast } e_1 t_p)\) representing the query \(\left[ r_{p,q}^- t_p \right] \left[ r_{p,q}^- e_1 \right]\) is defined as follows:

\[
(\text{rule } (\text{cast } x y) \text{ var } (x, y) \text{ abn val } (x) \text{ where } (y \text{ is type}) \text{ then } (\text{cast } x :: \ast y) :: \text{atm});
\]

\[
(\text{transition } (\text{cast } x y) :: \text{atm} \text{ var } (x, y) \text{ then } f_n),
\]

where

- if \(r_{p,q}^- v_l\) is a result of conversion of \(r_{p,q}^- x_0\) to \(r_{p,q}^- y_0\), then \((\text{cast } x_0 y_0) :: \text{atm}, e_{l,s} \# c_{nf} \rightarrow f_n, (x:x_0,y:y_0) e_{l,s} \# v_l \# c_{nf}\).

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

The element \((e_1 + :: C e_{l,1})\) representing the query \(r_{p,q}^- e_1 + r_{p,q}^- e_{l,1}\) which specifies the sum of numbers \(r_{p,q}^- e_1\) and \(r_{p,q}^- e_{l,1}\) of the type int is defined as follows:

\[
(\text{rule } (x + :: C y) \text{ var } (x, y) \text{ abn val } (x) \text{ where } (x \text{ is int}) \text{ and } (y \text{ is int}) \text{ then } (x + :: C y) :: (\text{int} + \text{ int}) :: \text{atm});
\]

\[
(\text{transition } (x + :: C y) :: (\text{int} + \text{ int}) :: \text{atm} \text{ var } (x, y) \text{ then } f_n),
\]

where

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

The elements \((\text{new } t_p)\) and \((\text{delete } p_{n,1})\) representing the dynamic memory management queries are defined by the rules

\[
(\text{rule } (\text{new } x) \text{ var } (x) \text{ abn where } (x \text{ is type}) \text{ then }
\]

\[
(\text{let } w \text{ be } (\text{new pointer-id}) \text{ in }
\]

\[
((0 : (\text{id} : w, \text{ type } : x) :: \text{pointer}, 1 : \text{pointer}) ::= \text{true}),
\]

\[
(\text{id} : w, \text{ type } : x) :: q));
\]

\[
(\text{rule } (\text{delete } x) \text{ var } (x) \text{ abn val } (x) \text{ where } (x :: \ast \text{ is typed-pointer-literal}) \text{ then }
\]

\[
((-1 : \text{value}, 0 : x :: *, 1 : \text{pointer}) ::=),
\]

\[
((0 : x :: *, 1 : \text{pointer}) ::=)).
\]

The element \((\text{new-cc } c_{ncp,c})\) generates a new instance of the countable concept \(c_{ncp,c}\) [7]. The element \(\text{pointer-id}\) 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 \(e_{l,1} \ldots e_{l,n_l}\) represents the program \([r_{p,q}^- e_{l,1}] \ldots [r_{p,q}^- e_{l,n_l}]\) 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


