Bull. Nov. Comp. Center, Comp. Science, 38 (2015), 1–28

Operational semantics development for procedural programming languages based on conceptual transition systems*

I. S. Anureev

Abstract. The methodology of the operational semantics development for programming languages based on the operational ontological approach, conceptual transition systems and CTSL, the language for the specification of such systems, is proposed. The development of operational semantics is illustrated by an example of procedural programming languages from the family MPL of model programming languages. Each target language covers a certain type of the procedural language constructs. Thus, the paper can be also considered as a cookbook on the development of operational semantics of procedural programming languages.

Keywords: operational semantics, conceptual transition systems, procedural programming languages, operational ontological approach, conceptuals, CTSL, MPL.

1. Introduction

Currently, there are tens of thousands of computer languages (programming languages, specificaton 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 one of the means to ensure the correct and effective use of computer languages. Application of formal methods to the texts in these languages requires the formalization of these texts. Therefore, the problem of the development of formal specifications for computer languages arises.

As a rule, specifications based on operational semantics are used for executable computer languages. The operational semantics of computer languages is usually based on the state transition systems. The methodology of the application of transition systems to the development of formal semantics of a class of computer languages such as programming languages – the method of structural operational semantics – was proposed in [1]. However, because of the conceptual poverty of the formalism of transition systems based only on two concepts, a state and a transition, this methodology can not take into account the conceptual structure of programming languages, while modern programming languages have quite a complex conceptual structure, including hundreds of concepts. This results in cumber-

*Partially supported by RFBR under Grant №15-01-05974.

some specifications in which it is easy to make mistakes and difficult to find them.

The logical algebraic approach to this problem was proposed in [2, 3]. It is based on abstract state machines – the transition systems in which states are algebras. The choice of an appropriate algebra for the specification of a computer language solves the problem of modeling the conceptual structure of the language partially.

The operational ontological approach [4] to this problem is based on the specification of the conceptual structure using an ontology [5]. The formalisms earlier used for the implementation of this approach, such as ontological transition systems [6, 7] and domain-specific transition systems [8], can specify only a restricted number of the kinds of ontological elements. The new formalism of conceptual transition systems [9, 10] (CTSs for short) seems promising for the implementation of this approach, since it is quite universal to specify typical ontological elements (concepts, attributes, concept instances, relations, relation instances, individuals, types, and so on). In addition, it gives quite a complete classification of ontological elements which allows us to define new kinds and subkinds of ontological elements.

In this paper, we propose the methodology for the development of operational semantics of programming languages based on the specification language CTSL (Conceptual Transition System Language) [10]. Using the specialized language allows us to raise the development of formal semantics of programming languages to a much higher level compared to the conventional description of the semantics in a natural language by inference rules. We get rid of the ambiguity of the natural language. Not all units (terms, functions, predicates, etc.) occurring in the inference rules are usually defined in detail. Furthermore, in our experience, the size of the detailed definition of the semantics in the natural language is at least not less than the size of its specification in CTSL. We virtually 'program' the semantics of a programming language in CTSL in a natural imperative style, using a usual terminology of the programming language, encoded in its ontology, and we can also 'test' the semantics (if there is a CTSL implementation). Using the language for describing semantics, we can develop a common methodology for the development of the semantics of the classes of related programming languages. In our experience, the associated ontological elements and the structure of CTSL rules for many constructs of the related languages are identical. The accumulated techniques, idioms, and components describing the semantics for certain constructs of programming languages can be reused to develop the semantics of new languages. On the basis of a unified formalism, we can carry out a comparative analysis of the semantics of programming languages, study and prove the properties of their semantics. And finally, a CTSL specification of the semantics of a programming language is a strict and complete documentation for the compiler of the language.

The methodology is illustrated by an example of the model procedural programming language MPL. It is defined as a family of programming languages in which each subsequent language is obtained from the previous language by the introduction of new constructs and/or complication of the semantics of constructs.

The paper is organized as follows. Notions and denotations used in this paper are given in Section 2. The methodology of the development of operational semantics of programming languages based on CTSs is described in Section 3. The elements common for all languages of the family MPL are defined in Section 4. The languages of the family MPL are defined in sections from 5 to 11. The MPL_1 language (Section 5) includes a minimal set of basic types such as integer and boolean types with operations on them and the statements of imperative programming languages such as conditional statements, block statements, assignment statements and while statements. The MPL_2 language (Section 6) adds variable scopes. The MPL_3 language (Section 7) adds functions, procedures, and the return statement. The MPL_4 language (Section 8) adds pointers. The MPL_5 language (Section 9) adds transfer-of-control statements such as *break*, *continue* and *goto*. The MPL_6 language (Section 10) adds compound types such as arrays and structures. The MPL_7 language (Section 11) adds functional and procedural types and variables.

2. Preliminaries

The names of sets begin with a capital letter. The elements of a set are represented by the name of the set with a small first letter provided possibly with indexes and primes. For example, the elements of the set X_α are represented by x_α, $x_{\alpha 1}$, ${x_\alpha}^2$, ${x_\alpha}'$, ${x_\alpha}''$, and so on.

Let $B_{oo} = \{true, false\}$. Let $I_{n..r}$, N_{at} and $N_{at.0}$ denote the sets of integers, natural numbers and natural numbers with zero; O_b, F_u, S_{et}, L_{ab}, A_{rg}, and V_{al} denote the sets of objects, functions, sets, labels, function arguments, and function values. Let $sup(f_u)$ and $v_{al.u}$ denote the support of f_u and the undefined value of f_u. Let $s_{et.(*)}$, $s_{et.[*]}$, $s_{et.\{*\}}$, and $s_{et.*}$ denote the sets of sequences of the forms $(o_{b.1}, \ldots, o_{b.n_{at.0}})$, $[o_{b.1}, \ldots, o_{b.n_{at.0}}]$, $\{o_{b.1}, \ldots, o_{b.n_{at.0}}\}$, and $o_{b.1}, \ldots, o_{b.n_{at.0}}$ from elements of s_{et}. For example, $I_{n..r.(*)}$ is a set of sequences of the form $(i_{n..r.1}, \ldots, i_{n..r.n_{at.0}})$, and $i_{n..r.*}$ is a sequence of the form $i_{n..r.1}, \ldots, i_{n..r.n_{at.0}}$. Let S_{eq} be a set of sequences. Let $len(s_{eq})$ denote the length of s_{eq}. Let $s_{eq}(n_{at})$ denote the n_{at}-th element of s_{eq}. If $len(s_{eq}) < n_{at}$, then $s_{eq}(n_{at}) = v_{al.u}$.

Let B_{od}, $C_{o..n}$, and V_{ar} be the sets of elements called bodies, conditions, and variables.

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: $l_{ab}{:}o_b$, $l_{ab}{:}$ or o_b. The context in which some embedded contexts are omitted is called a partial context. All omitted embedded contexts are considered bound by the existential quantifier, unless otherwise specified.

Let $o_b[\![o_{b.*}]\!]$ denote the object o_b in the context $[\![o_{b.*}]\!]$.

Let $S_{y.t.c}$ denote a set of conceptual transition systems [9]. Let A_{to}, E_l, $E_{l.s}$, $C_{o..l}$, $C_{o..pt}$, A_{tt}, S_{ta} and T_r denote the sets of atoms, elements, structural elements, conceptuals, concepts, attributes, conceptual states and transitions in $[\![s_{y.t.c}]\!]$. Let $t_r = (s_{ta}, s_{t.c.1})$.

Let $B_{oo.u} = \{true, v_{al.u.e}\}$. The element $v_{al.u.e}$ is called the element undefined value.

3. The methodology of operational semantics development for programming languages based on CTSs

Let $L_{an.p}$ be a set of programming languages, and $C_{o..ct.l.p}$ be the set of constructs of $l_{an.p}$. The construction of $s_{y.t.c}$ which specifies the operational semantics of $l_{an.p}$ consists of five stages:

1. Define A_{to} in $[\![s_{y.t.c}]\!]$. The atoms of A_{to} specify the elementary units of $l_{an.p}$.
2. Define the conceptuals and elements (concepts, attributes, individuals) of the conceptual states in $[\![s_{y.t.c}]\!]$, which specify the conceptual structure and states of $l_{an.p}$. An object o_b is called a state of $l_{an.p}$ if o_b is a specific content of the conceptual structure of $l_{an.p}$.
3. Define a one-to-one mapping of constructs of $l_{an.p}$ into special kinds of elements in $[\![s_{y.t.c}]\!]$. An element e_l is called a model of $c_{o..ct.l.p}$ if e_l represents $c_{o..ct.l.p}$. While execution of $c_{o..ct.l.p}$ changes the state of $l_{an.p}$, execution of e_l changes the corresponding conceptual state in $[\![s_{y.t.c}]\!]$.
4. The constructs of $l_{an.p}$ are divided into two groups: expressions the main semantics of which is to return values, and statements the main semantics of which is to change states. Expressions can also change states, but this feature is not their main semantics.
5. Define the operational semantics of the models of expressions and statements of $l_{an.p}$ by transition rules in $[\![s_{y.t.c}]\!]$.

4. Description of languages of the family MPL

Since MPL is a new language and its syntax is undefined, we define MPL as a sublanguage of CTSL [10] at the syntactic level for simplicity. In this case, the models of MPL constructs coincide with these constructs. This feature provides the extensibility of MPL.

Let $l_{an.p}$ be a language of the family MPL, and $s_{y.t.c}$ be a CTS specifying the operational semantics of $l_{an.p}$. In this section, we define the constructs

of $l_{an.p}$, and conceptuals and executable elements in $[\![s_{y.t.c}]\!]$. The language $l_{an.p}$ includes:

- a set L_i of literals that are syntactic representations of the values of types in $l_{an.p}$;
- a set I_d of identifiers such that $I_d = A_{to} \setminus L_i$, where A_{to} is a set of atoms in CTSL;
- a set $C_{o..pt.r}[\![s_{ta}]\!]$ of rule-defined concepts [10] that specify the types and kinds of objects in $l_{an.p}$ and their values (instances) in $[\![s_{ta}]\!]$. The set $C_{o..pt.r}$, in addition to the concepts of CTSL includes the concepts $literal$ and $identifier$ specifying literals and identifiers in $l_{an.p}$, respectively;
- a set $T_y[\![s_{ta}]\!] \subseteq C_{o..pt.r}[\![s_{ta}]\!]$ of types in $[\![s_{ta}]\!]$ in $l_{an.p}$;
- a set $V_{ar}[\![s_{ta}]\!]$ of variables in $[\![s_{ta}]\!]$ such that their names are identifiers and their types are from $T_y[\![s_{ta}]\!]$;
- expressions consisting of literals, variables and operations; and
- statements.

Let $C_{o..pt.r}$, T_y, and V_{ar} denote $C_{o..pt.r}[\![s_{ta}]\!]$, $T_y[\![s_{ta}]\!]$, and $V_{ar}[\![s_{ta}]\!]$ for the current state s_{ta}, respectively.

5. MPL$_1$: basic operations and statements

The MPL$_1$ language includes the types *int* and *bool* of CTSL [9], the set L_i of literals such that $L_i = I_{n..r} \cup B_{oo}$, the operations = and != on elements, the integer operations +, −, *., *div* and *mod*, the integer relations <, >, <= and >=, the boolean operations *and*, *or*, *not*, => and <=>, variable declarations, assignments, if statements, while statements and block statements.

5.1. Conceptual states

In this section, we list the conceptuals of the conceptual states of MPL$_1$.

The conceptual $\{-1{:}type, 0{:}v_{ar}, 1{:}variable\}$ specifies the variable v_{ar} and its type. An identifier v_{ar} is a variable of a type t_y in $[\![s_{ta}]\!]$ if $s_{ta}(\{-1{:}type, 0{:}v_{ar}, 1{:}variable\}) = t_y$. Let V_{ar} be a set of variables. The following property holds for MPL$_1$: if $s_{ta}(\{-1{:}type, 0{:}v_{ar}, 1{:}variable\}) \neq v_{al.u.e}$, then $s_{ta}(\{-1{:}type, 0{:}v_{ar}, 1{:}variable\}) \in T_y$. The conceptual $\{-1{:}value, 0{:}v_{ar}, 1{:}variable\}$ specifies the value of v_{ar}. A variable v_{ar} has the value v_{al} in $[\![s_{ta}]\!]$ if $s_{ta}(\{-1{:}value, 0{:}v_{ar}, 1{:}variable\}) = v_{al}$.

5.2. Expressions

In this section, we define the MPL_1 expressions and their semantics.

The element $(*.e_l\ is\ *.e_{l.1})$ specifying rule-defined concepts and their instances is defined by the rule

$(rule\ (*.x\ is\ *.y)\ var\ (x,\ y)\ where\ (*\ x,\ y)\ then\ (*.x\ is\ *.y))$.

The element $(e_l\ is\ type)$ specifying that $e_l \in T_y$ is defined by the rules

$(rule\ (int\ is\ type)\ then\ true)$;
$(rule\ (bool\ is\ type)\ then\ true)$.

The element $(e_l\ is\ literal)$ specifying that $e_l \in L_i$ is defined by the rules

$(rule\ (x\ is\ literal)\ var\ (x)\ then\ (x\ is\ int))$;
$(rule\ (x\ is\ literal)\ var\ (x)\ then\ (x\ is\ bool))$.

The element $(e_l\ is\ identifier)$ specifying that $e_l \in I_d$ is defined by the rule

$(rule\ (x\ is\ identifier)\ var\ (x)$
$then\ ((x\ is\ atom)\ and\ (not\ (x\ is\ literal))))$.

The elements $(type\ of\ b_{oo})$ and $(type\ of\ i_{n..r})$ returning the types of b_{oo} and $i_{n..r}$, respectively, are defined by the rules

$(rule\ (type\ of\ x)\ var\ (x)\ where\ (x\ is\ bool)\ then\ \texttt{'}.bool)$;
$(rule\ (type\ of\ x)\ var\ (x)\ where\ (x\ is\ int)\ then\ \texttt{'}.int)$.

The element $(e_l\ is\ variable)$ specifying that $e_l \in V_{ar}$ is defined by the rule

$(rule\ (x\ is\ variable)\ var\ (x)\ then\ ((x\ is\ identifier)\ and$
$(\{-1{:}type,\ 0{:}x,\ 1{:}variable\}\ \texttt{!=}\ v_{al.u.e})))$.

The element v_{ar} specifying the value of v_{ar} is defined by the rule

$(rule\ x\ var\ (x)\ where\ (x\ is\ variable)$
$then\ \{-1{:}value,\ 0{:}x,\ 1{:}variable\}$.

The element $(type\ of\ v_{ar})$ specifying the type of v_{ar} is defined by the rule

$(rule\ (type\ of\ x)\ var\ (x)\ where\ (x\ is\ identifier)$
$then\ \{-1{:}type,\ 0{:}x,\ 1{:}variable\})$.

MPL_1 inherits the following operations of CTSL: $+$, $-$, $*$, div, mod, $=$, $!=$, $<$, $<=$, $>$, $>=$, and, or, not, $=>$, and $<=>$.

The element $(e_l\ is\ embedded\text{-}statement)$ specifying that e_l is an embedded statement, i.e. a statement which can be included in compound statements at the top level, is defined by the rule

$(rule\ (x\ is\ embedded\text{-}statement)\ var\ (x)\ then\ true)$.

Thus, all MPL_1 statements are embedded.

5.3. Statements

In this section, we define the MPL_1 statements and their semantics.

The element $(var\ v_{ar}\ t_y)$ specifying the declaration of the variable v_{ar} of the type t_y is defined by the rule

$(rule\ (var\ x\ y)\ var\ (x,\ y)\ where\ ((x\ is\ identifier)\ and\ (y\ is\ type))$
$\ then\ (if\ (x\ is\ variable)\ then\ v_{al.u.e}$
$\ else\ (\{-1{:}type,\ 0{:}x,\ 1{:}variable\}\ ::=\ \texttt{'.}y))).$

The element $(v_{ar}\ :=\ e_l)$ specifying the assignment of the value of e_l to v_{ar} is defined by the rule

$(rule\ (x\ :=\ y)\ var\ (x,\ y)\ where\ ((*\ y)\ and\ (x\ is\ variable))\ then$
$\ (if\ (\texttt{*.}y\ is\ \texttt{*.}(type\ of\ x))$
$\ \ then\ (\{-1{:}value,\ 0{:}x,\ 1{:}variable\}\ ::=\ \texttt{'.*.}y)\ else\ v_{al.u.e})).$

The elements $(if\text{-}m\ c_{o..n}\ then\ e_{l.1}\ else\ e_{l.2})$ and $(if\text{-}m\ c_{o..n}\ then\ e_{l.2})$ specifying the if statement with the condition $c_{o..n}$, *then*-branch $e_{l.1}$ and *else*-branch $e_{l.2}$ are defined by the rules:

$(rule\ (if\text{-}m\ x\ then\ y\ else\ z)\ var\ (x,\ y,\ z)$
$\ where\ ((y\ is\ embedded\text{-}statement)\ and\ (z\ is\ embedded\text{-}statement))$
$\ then\ (if\ x\ then\ y\ else\ z));$
$(rule\ (if\text{-}m\ x\ then\ y)\ var\ (x,\ y)\ where\ (y\ is\ embedded\text{-}statement)$
$\ then\ (if\ x\ then\ y)).$

The element $(while\text{-}m\ c_{o..n}\ do\ b_{od})$ specifying the while statement with the condition $c_{o..n}$ and the body b_{od} is defined by the rule

$(rule\ (while\text{-}m\ x\ do\ y)\ var\ (x,\ y)$
$\ where\ (y\ is\ embedded\text{-}statement)\ then\ (while\ x\ do\ y)).$

The element $(block\ e_{l.*})$ specifying the block statement with the body $e_{l.*}$ is defined by the rule

$(rule\ (block\ \texttt{.::}\ x)\ var\ (x)\ then\ (seq\ \texttt{.::}\ x)).$

6. MPL_2: variable scopes

The MPL_2 language adds variable scopes. The scope of v_{ar} occuring in $c_{o..ct.l.p}$ is the number of blocks surrounding this occurence of v_{ar} in $c_{o..ct.l.p}$. The value and type of v_{ar} depend on its scope. The variable v_{ar} can be global (with the scope 0) and local. The following example illustrates variable scopes:

$seq \setminus\setminus x = v_{al.u.e},\ y = v_{al.u.e},\ scope = 0$
$(var\ x\ int) \setminus\setminus x = v_{al.u.e},\ y = v_{al.u.e},\ scope = 0$
$(x := 0) \setminus\setminus x = 0,\ y = v_{al.u.e},\ scope = 0$
$(var\ y\ bool) \setminus\setminus x = 0,\ y = v_{al.u.e},\ scope = 0$
$(y := true) \setminus\setminus x = 0,\ y = true,\ scope = 0$
$(block \setminus\setminus x = 0,\ y = true,\ scope = 1$
$(var\ x\ bool) \setminus\setminus x = v_{al.u.e},\ y = true,\ scope = 1$
$(x := false) \setminus\setminus x = false,\ y = true,\ scope = 1$
$(block \setminus\setminus x = false,\ y = true,\ scope = 2$
$(var\ x\ int) \setminus\setminus x = v_{al.u.e},\ y = true,\ scope = 2$
$(x := 2) \setminus\setminus x = 2,\ y = true,\ scope = 2$
$) \setminus\setminus x = false,\ y = true,\ scope = 1$
$(var\ y\ int) \setminus\setminus x = false,\ y = v_{al.u.e},\ scope = 1$
$(y := 1) \setminus\setminus x = false,\ y = 1,\ scope = 1$
$) \setminus\setminus x = 0,\ y = true,\ scope = 0$
).

6.1. Conceptual states

In this section, we list the specific conceptuals of the conceptual states of MPL_2.

Let S_c be a set of variable scopes represented by the elements of $N_{at.0}$. The conceptual $\{0{:}scope\}$ called a scope specifier and denoted by $s_{c.s..r}$ specifies the scope in which $s_{y.t.c}$ is being executed.

The conceptual $\{-2{:}s_c, -1{:}type, 0{:}v_{ar}, 1{:}variable\}$ specifies a variable v_{ar} and its type in $[\![s_c]\!]$. An identifier v_{ar} is a variable of a type t_y in $[\![s_c, s_{ta}]\!]$ if $s_{ta}(\{-2{:}s_c, -1{:}type, 0{:}v_{ar}, 1{:}variable\}) = t_y$. The variable v_{ar} is global if $s_c = 0$. The variable v_{ar} is local if $s_c > 0$. The following property holds for MPL_2: if $s_{ta}(\{-2{:}s_c, 0{:}v_{ar}, 1{:}variable\}) \neq v_{al.u.e}$, then $s_{ta}(\{-2{:}s_c, -1{:}type, 0{:}v_{ar}, 1{:}variable\}) \in T_y$. The conceptual $\{-2{:}s_c, -1{:}value, 0{:}v_{ar}, 1{:}variable\}$ specifies the value of v_{ar} in $[\![s_c]\!]$. The variable v_{ar} has the value v_{al} in $[\![s_c, s_{ta}]\!]$ if $s_{ta}(\{-2{:}s_c, -1{:}value, 0{:}v_{ar}, 1{:}variable\}) = v_{al}$.

6.2. Expressions

In this section, we define the MPL_2 expressions and their semantics.

The element $(index\ of\ i_d)$ is defined by the rules

$(rule\ (index\ of\ x)\ var\ (x)\ where\ (x\ is\ identifier)$
$then\ (index\ of\ x\ in\ *.s_{c.s..r}));$
$(rule\ (index\ of\ x\ in\ y)\ var\ (x,\ y)\ then$
$(if\ (\{-2{:}y,\ -1{:}type,\ 0{:}x,\ 1{:}variable\}\ \texttt{!=}\ v_{al.u.e})\ then\ \texttt{'}.y$
$else\ (if\ (y = 0)\ then\ v_{al.u.e}\ else\ (index\ of\ x\ in\ *.(y - 1)))).$

To resolve the name conflict, a unique index is associated with each value of the name. In the case of the above rule, the index of v_{ar} is the scope of a variable with the name v_{ar}.

The element (e_l *is variable*) is defined by the rule

$(rule\ (x\ is\ variable)\ var\ (x)\ then\ ((index\ of\ x)\ \texttt{!=}\ v_{al.u.e}))$

The element v_{ar} is defined by the rule

$(rule\ x\ var\ (x)\ hvar\ (w)\ then\ (seq\ (w\ ::=\ (index\ of\ x))$
$(if\ (w\ =\ v_{al.u.e})\ then\ v_{al.u.e}$
$else\ \{-2{:}*.w,\ -1{:}value,\ 0{:}x,\ 1{:}variable\}))).$

The element (*type of* v_{ar}) is defined by the rule

$(rule\ (type\ of\ x)\ var\ (x)\ where\ hvar\ (w)\ then\ (seq$
$(w\ ::=\ (index\ of\ x))$
$(if\ (w\ =\ v_{al.u.e})\ then\ v_{al.u.e}$
$else\ \{-2{:}*.w,\ -1{:}type,\ 0{:}x,\ 1{:}variable\}))).$

The element (e_l *is embedded-statement*) is defined by the rule

$(rule\ (x\ is\ embedded\text{-}statement)\ var\ (x)\ then$
$(not\ (x\ matches\ (var\ u\ v)\ var\ (u,\ v)$
$where\ ((u\ is\ identifier)\ and\ (v\ is\ type))))).$

Thus, only variable declarations are not embedded in MPL_2.

6.3. Statements

In this section, we define the MPL_2 statements and their semantics.

The variable declaration is defined by the rule

$(rule\ (var\ x\ y)\ var\ (x,\ y)\ where\ ((x\ is\ identifier)\ and\ (y\ is\ type))$
$then\ (if\ (\{-2{:}s_{c.s..r},\ -1{:}type,\ 0{:}x,\ 1{:}variable\}\ \texttt{!=}\ v_{al.u.e})$
$then\ v_{al.u.e}$
$else\ (\{-2{:}*.s_{c.s..r},\ -1{:}type,\ 0{:}x,\ 1{:}variable\}\ ::=\ \texttt{'}.y))).$

The assignment statement is defined by the rule

$(rule\ (x\ :=\ y)\ var\ (x,\ y)\ where\ (*\ y)\ hvar\ (w)\ then\ (seq$
$(w\ :=\ (index\ of\ x))$
$(if\ ((w\ \texttt{!=}\ v_{al.u.e})\ and$
$(y\ is\ *.\{-2{:}*.w,\ -1{:}type,\ 0{:}x,\ 1{:}variable\}))$
$then\ (\{-2{:}z,\ -1{:}value,\ 0{:}x,\ 1{:}variable\}\ ::=\ \texttt{'}.*.y)$
$else\ v_{al.u.e}))).$

The block statement is defined by the rule

$(rule\ (block\ .::\ x)\ var\ (x)\ then$
$\ (seq\ (enter\text{-}block)\ (seq\ .::\ x)\ (exit\text{-}block))).$

The element (*enter-block*) specifying the actions executed when $s_{y.t.c}$ enters a block is defined by the rule

$(rule\ (enter\text{-}block)\ then\ (s_{c.s..r}\ ::=\ (s_{c.s..r}\ +\ 1))).$

The element (*exit-block*) specifying the actions executed when $s_{y.t.c}$ exits a block is defined by the rule

$(rule\ (exit\text{-}block)\ where\ \#\ catch\ w\ then$
$\ (seq\ (delete\text{-}local\text{-}variables)\ (s_{c.s..r}\ ::=\ (s_{c.s..r}\ -\ 1))\ (throw\ w))).$

The element (*delete-local-variables*) specifying deletion of local variables of the current scope is defined by the rule

$(rule\ (delete\text{-}local\text{-}variables)\ then$
$\ (foreach\text{-}match\ \{-2{:}*.s_{c.s..r},\ -1{:}type,\ 0{:}x,\ 1{:}variable\}\ var\ (x)$
$\ \ do\ (seq\ (\{-2{:}*.s_{c.s..r},\ -1{:}type,\ 0{:}*.x,\ 1{:}variable\}\ ::=)$
$\ \ \ \ \ \ (\{-2{:}*.s_{c.s..r},\ -1{:}value,\ 0{:}*.x,\ 1{:}variable\}\ ::=)))).$

7. MPL$_3$: functions and procedures

The MPL$_3$ language adds declarations and calls of overloaded functions and procedures, and the return statement.

7.1. Conceptual states

In this section, we list the specific conceptuals of the conceptual states of MPL$_3$.

The conceptual $\{-1{:}argument\text{-}types, 0{:}f_u, 1{:}function\}$ specifies the function f_u and its argument types. An identifier f_u is a function with argument types $t_{y.[*]}$ in $[\![s_{ta}]\!]$ if $s_{ta}(\{-1{:}argument\text{-}types, 0{:}f_u, 1{:}function\}) = t_{y.[*]}$. Let F_u be a set of functions in $[\![s_{ta}]\!]$. The conceptual $\{-2{:}t_{y.[*]}, -1{:}return\text{-}type, 0{:}f_u, 1{:}function\}$ specifies the return type of f_u. A function f_u has the return type t_y in $[\![s_{ta}]\!]$ if $s_{ta}(\{-2{:}t_{y.[*]}, -1{:}return\text{-}type, 0{:}f_u, 1{:}function\}) = t_y$. A function f_u with the return type t_y in $[\![s_{ta}]\!]$ is a procedure in $[\![s_{ta}]\!]$ if $t_y = void$. The conceptual $\{-2{:}t_{y.[*]}, -1{:}parameters, 0{:}f_u, 1{:}function\}$ specifies the parameters of f_u. The function f_u has the parameters $i_{d.[*]}$ in $[\![s_{ta}]\!]$ if $s_{ta}(\{-2{:}t_{y.[*]}, -1{:}parameters, 0{:}f_u, 1{:}function\}) = i_{d.[*]}$. The conceptual $\{-2{:}t_{y.[*]}, -1{:}body, 0{:}f_u, 1{:}function\}$ specifies the body of f_u. The function f_u has the body b_{od} in $[\![s_{ta}]\!]$ if $s_{ta}(\{-2{:}t_{y.[*]}, -1{:}body, 0{:}f_u, 1{:}function\}) = b_{od}$. The following property holds for MPL$_3$: if $s_{ta}(\{-1{:}argument\text{-}types, 0{:}f_u, 1{:}function\}) \neq v_{al.u.e}$, then there exist $t_{y.[*]}$, t_y and $i_{d.[*]}$ such that $s_{ta}(\{-1{:}argument\text{-}types, 0{:}f_u, 1{:}function\}) = t_{y.[*]}$, $s_{ta}(\{-2{:}$

$t_{y.[*]}, -1{:}return\text{-}type, 0{:}f_u, 1{:}function\}) = t_y$, $\{-2{:}t_{y.[*]}, -1{:}parameters, 0{:}f_u, 1{:}function\} = i_{d.[*]}$, $\{-2{:}t_{y.[*]}, -1{:}body, 0{:}f_u, 1{:}function\} \neq v_{al.u.e}$, and $len(t_{y.[*]}) = len(i_{d.[*]})$.

Let $L_{e.c}$ be a set of call levels represented by the elements of N_{at}. The conceptual $\{0{:}call\text{-}depth\}$ called a call level specifier and denoted by $l_{e.c.s..r}$ specifies the level of function calls, i.e. the number of embedded function calls in which $s_{y.t.c}$ is being executed.

The conceptual $\{-3{:}l_{e.c}, -2{:}s_c, -1{:}type, 0{:}v_{ar}, 1{:}variable\}$ specifies a variable v_{ar} and its type in $[\![l_{e.c}, s_c]\!]$. An identifier v_{ar} is a variable of a type t_y in $[\![l_{e.c}, s_c, s_{ta}]\!]$ if $s_{ta}(\{-3{:}l_{e.c}, -2{:}s_c, -1{:}type, 0{:}v_{ar}, 1{:}variable\}) = t_y$. The following property holds for MPL$_3$: if $s_{ta}(\{-3{:}l_{e.c}, -2{:}s_c, -1{:}type, 0{:}v_{ar}, 1{:}variable\}) \neq v_{al.u.e}$, then $s_{ta}(\{-3{:}l_{e.c}, -2{:}s_c, -1{:}type, 0{:}v_{ar}, 1{:}variable\}) \in T_y$. The conceptual $\{-3{:}l_{e.c}, -2{:}s_c, -1{:}value, 0{:}v_{ar}, 1{:}variable\}$ specifies the value of v_{ar} in $[\![l_{e.c}, s_c]\!]$. The variable v_{ar} has the value v_{al} in $[\![l_{e.c}, s_c, s_{ta}]\!]$ if $s_{ta}(\{-3{:}l_{e.c}, -2{:}s_c, -1{:}value, 0{:}v_{ar}, 1{:}variable\}) = v_{al}$. The type and value of the global variable v_{ar} in $[\![l_{e.c}, s_c, s_{ta}]\!]$ are specified by the conceptuals $\{-3{:}0, -2{:}0, -1{:}type, 0{:}v_{ar}, 1{:}variable\}$ and $\{-3{:}0, -2{:}0, -1{:}value, 0{:}v_{ar}, 1{:}variable\}$, respectively.

The conceptual $\{-1{:}return\text{-}type, 0{:}l_{e.c.s..r}\}$ denoted by $t_{y.r.s..r}$ specifies the return type of the function executed in $[\![l_{e.c.s..r}, s_{ta}]\!]$.

The exceptions $\{-1{:}return, 0{:}v_{al}, 1{:}exception\}$ and $\{-1{:}return, 1{:}exception\}$ specify the execution of the return statement with the return value v_{al} and without the return value, respectively.

7.2. Expressions

In this section, we define the MPL$_3$ expressions and their semantics.

The element (*void is type*) specifying that $void \in T_y$ is defined by the rules

(*rule* (*void is type*) *then true*).

The element (*index of* i_d) is defined by the rules

(*rule* (*index of x*) *var* (*x*) *where* (*x is identifier*)
then (*index of x in* $*.l_{e.c.s..r}$, $*.s_{c.s..r}$));
(*rule* (*x is index in y, z*) *var* (*x, y, z*) *then*
(*if* ({−3:*y*, −2:*z*, −1:*type*, 0:*x*, 1:*variable*} != $v_{al.u.e}$) *then* '.*z*
else (*if* (*z* = 0) *then* $v_{al.u.e}$
else (*index of x in y*, $*.(z - 1)$))))).

The element (e_l *is variable*) is defined by the rule

(*rule* (*x is variable*) *var* (*x*) ((*index of x*) != $v_{al.u.e}$)).

The element v_{ar} is defined by the rule

$(rule\ x\ var\ (x)\ hvar\ (w)\ then\ (seq\ (w\ ::=\ (index\ of\ x))$
$(if\ (w\ =\ v_{al.u.e})\ then\ v_{al.u.e}$
$else\ \{-3{:}y,\ -2{:}{*}.w,\ -1{:}value,\ 0{:}x,\ 1{:}variable\}))).$

The element $(type\ of\ v_{ar})$ is defined by the rule

$(rule\ (type\ of\ x)\ var\ (x)\ hvar\ (w)\ then\ (seq\ (w\ ::=\ (index\ of\ x))$
$(if\ (w\ =\ v_{al.u.e})\ then\ v_{al.u.e}$
$else\ \{-3{:}y,\ -2{:}{*}.w,\ -1{:}type,\ 0{:}x,\ 1{:}variable\}))).$

The element $(fun\text{-}call\ f_u\ (e_{l.*}))$ calling f_u with the arguments $e_{l.*}$ is defined by the rule

$(rule\ (fun\text{-}call\ x\ y)\ var\ (x,\ y)$
$where\ (y\ is\ evaluable\text{-}ordered\text{-}element)\ hvar\ (u,\ t,\ s)\ then\ (seq$
$(u\ ::=\ (execute\text{-}arguments\ y))\ (t\ ::=\ (argument\text{-}types\ of\ {*}.u))$
$(if\ (\{-1{:}argument\text{-}types,\ 0{:}x,\ 1{:}function\}\ =\ t)\ then\ (seq$
$(l_{e.c.s..r}\ ::=\ (l_{e.c.s..r}\ +\ 1))\ (s\ ::=\ s_{c.s..r})\ (s_{c.s..r}\ ::=\ 1)$
$(create\text{-}local\text{-}variables$
${*}.\{-2{:}{*}.t,\ -1{:}parameters,\ 0{:}x,\ 1{:}function\}\ {*}.t\ {*}.u)$
$(t_{y.r.s..r}\ ::=\ \{-2{:}{*}.t,\ -1{:}return\text{-}type,\ 0{:}x,\ 1{:}function\})$
${*}.\{-2{:}{*}.t,\ -1{:}body,\ 0{:}x,\ 1{:}function\}$
$(catch\ w\ (seq$
$(if\ ({*}.w\ is\ not\text{-}admissible\text{-}function\text{-}exception)\ then\ v_{al.u.e})$
$(if\ ((not\ ({*}.w\ is\ exception))\ and\ (t_{y.r.s..r}\ \texttt{!=}\ void))\ then\ v_{al.u.e})$
$(exit\text{-}block)\ (t_{y.r.s..r}\ ::=)\ (l_{e.c.s..r}\ ::=\ (l_{e.c.s..r}\ -\ 1))$
$(s_{c.s..r}\ ::=\ s)$
$(if\ ((w\ ..\ -1)\ =\ \texttt{'}.return)\ then\ (w\ ::=\ (w\ ..\ 0)))$
$(throw\ w))))$
$else\ v_{al.u.e}))).$

The element $(argument\text{-}types\ of\ e_{l.(*)})$ returning the types of argument values $e_{l.(*)}$ is defined by the rules

$(rule\ (argument\text{-}types\ of\ ())\ then\ ());$
$(rule\ (argument\text{-}types\ of\ (x\ .::\ y))\ var\ (x,\ y)$
$then\ ({*}.(type\ of\ x)\ .+\ {*}.(argument\text{-}types\ of\ y))).$

The element $(e_x\ is\ not\text{-}admissible\text{-}function\text{-}exception)$ specifies that e_x is an exception which is not admissible when a function call exits. The absence of definition of this element means that all elements in MPL_3 are admissible when a function call exits.

7.3. Statements

In this section, we define MPL_3 statements and their semantics.

The element $(execute\text{-}arguments\ e_{l.(*)})$ executing the function arguments $e_{l.(*)}$ is defined by the rule

$(rule\ (execute\text{-}arguments\ (x\ .::\ y))\ var\ (x,\ y)\ where\ (*\ x)$
$\ then\ ((execute\text{-}arguments\ y)\ +.\ *.x));$
$(rule\ (execute\text{-}arguments\ ())\ then\ (skip)).$

The element $(create\text{-}local\text{-}variables\ e_{l.(*)}\ e_{l.(*).1})$ creating the local variables corresponding to the function parameters $e_{l.(*)}$ with the values $e_{l.(*).1}$ is defined by the rules

$(rule\ (create\text{-}local\text{-}variables\ (x\ .::\ y)\ (t\ .::\ z)\ (u\ .::\ v))$
$\ var\ (x,\ y,\ t,\ z,\ u,\ v)$
$\ then\ (seq\ (var\ x\ t)\ (x\ :=\ '.u)\ (create\text{-}local\text{-}variables\ y\ z\ v)));$
$(rule\ (create\text{-}local\text{-}variables\ ()\ ()\ ())\ then\ (skip)).$

The element $(delete\text{-}local\text{-}variables)$ is defined by the rule

$(rule\ (delete\text{-}local\text{-}variables)\ then$
$\ (foreach\text{-}match$
$\ \{-3{:}*.l_{e.c.s..r},\ -2{:}*.s_{c.s..r},\ -1{:}type,\ 0{:}x,\ 1{:}variable\}\ var\ (x)\ do$
$\ (seq\ (\{-3{:}*.l_{e.c.s..r},\ -2{:}*.s_{c.s..r},\ -1{:}type,\ 0{:}*.x,\ 1{:}variable\}\ ::=)$
$\ (\{-3{:}*.l_{e.c.s..r},\ -2{:}*.s_{c.s..r},\ -1{:}value,\ 0{:}*.x,\ 1{:}variable\}\ ::=)))).$

The variable declaration is defined by the rule

$(rule\ (var\ x\ y)\ var\ (x,\ y)\ where\ ((x\ is\ identifier)\ and\ (y\ is\ type))$
$\ then$
$\ (if\ (*.\{-3{:}l_{e.c.s..r},\ -2{:}s_{c.s..r},\ -1{:}type,\ 0{:}x,\ 1{:}variable\}\ !=\ v_{al.u.e})$
$\ then\ v_{al.u.e}$
$\ else\ (\{-3{:}*.l_{e.c.s..r},\ -2{:}*.s_{c.s..r},\ -1{:}type,\ 0{:}x,\ 1{:}variable\}\ ::=$
$\ '.y))).$

The assignment is defined by the rules

$(rule\ (x\ :=\ y)\ var\ (x,\ y)\ where\ (*\ y)\ hvar\ (w)\ then\ (seq$
$\ (w\ ::=\ (index\ of\ x))$
$\ (if\ (w\ =\ v_{al.u.e})\ then\ v_{al.u.e}$
$\ else\ \{-3{:}z,\ -2{:}*.w,\ -1{:}value,\ 0{:}x,\ 1{:}variable\}\ ::=\ '.*.y))).$

The element $(function\ f_u\ ((i_{d.*}(1)\ t_{y.*}(1)),\ \ldots,\ (i_{d.*}(n_{at.0})\ t_{y.*}(n_{at.0})))$ $t_y\ b_{od})$, where $len(i_{d.*}) = len(t_{y.*}) = n_{at.0}$, specifying the declaration of the function f_u with the parameters $i_{d.*}$ of the types $t_{y.*}$, the return type t_y, and the body b_{od} is defined by the rule

$(rule\ (function\ x\ y\ z\ u)\ var\ (x,\ y,\ z,\ u)\ hvar\ (t)$
$\ where\ ((x\ is\ identifier)\ and\ (y\ is\ evaluable\text{-}ordered\text{-}element)\ and$
$\qquad (z\ is\ type))$
$\ then\ (seq$
$\ (t\ ::=\ (extract\text{-}types\ y))$
$\ (if\ ((t\ =\ v_{al.u.e})\ or$
$\qquad (\{-1{:}argument\text{-}types,\ 0{:}x,\ 1{:}function\}\ \texttt{!=}\ v_{al.u.e}))\ then\ v_{al.u.e}$
$\ else\ (seq\ (\{-2{:}*.t,\ -1{:}return\text{-}type,\ 0{:}x,\ 1{:}function\}\ ::=\ \texttt{'}.z)$
$\ (\{-2{:}*.t,\ -1{:}parameters,\ 0{:}x,\ 1{:}function\}\ ::=$
$\ (extract\text{-}names\ y))$
$\ (\{-2{:}*.t,\ -1{:}body,\ 0{:}x,\ 1{:}function\}\ ::=\ \texttt{'}.u)$
$\ (\{-1{:}argument\text{-}types,\ 0{:}x,\ 1{:}function\}\ ::=\ t))))).$

The element $(extract\text{-}types\ ((i_{d.*}(1)\ t_{y.*}(1)),\ \ldots,\ (i_{d.*}(n_{at.0})\ t_{y.*}(n_{at.0}$)))) returning types $(t_{y.*})$ is defined by the rules

$(rule\ (extract\text{-}types\ ())\ then\ ());$
$(rule\ (extract\text{-}types\ ((x\ t)\ \texttt{.::}\ y))\ var\ (x,\ t,\ y)$
$\ where\ ((x\ is\ identifier)\ and\ (t\ is\ type))$
$\ then\ (\texttt{'}.t\ \texttt{.+}\ \texttt{*.}(extract\text{-}types\ y))).$

The element $(extract\text{-}names\ ((i_{d.*}(1)\ t_{y.*}(1)),\ \ldots,\ (i_{d.*}(n_{at.0})\ t_{y.*}(n_{at.0}$)))) returning names $(i_{d.*})$ is defined by the rules

$(rule\ (extract\text{-}names\ ())\ then\ ());$
$(rule\ (extract\text{-}names\ ((x\ t)\ \texttt{.::}\ y))\ var\ (x,\ t,\ y)\ then$
$\ (\texttt{'}.x\ \texttt{.+}\ \texttt{*.}(extract\text{-}names\ y))).$

The elements $(return\ e_l)$ and $(return)$ specifying the return statement are defined by the rules

$(rule\ (return\ x)\ var\ (x)\ where\ (*\ x)\ then$
$\ (if\ ((t_{y.r.s..r}\ \texttt{!=}\ \texttt{'}.void)\ and\ (\texttt{*.}x\ is\ \texttt{*.}t_{y.r.s..r}))$
$\ then\ [-1{:}return,\ 0{:}\texttt{*.}x,\ 1{:}exception]\ else\ v_{al.u.e}));$
$(rule\ (return)\ var\ (x)\ where\ (*\ x)\ then$
$\ (if\ (t_{y.r.s..r}\ =\ \texttt{'}.void)\ then\ [-1{:}return,\ 1{:}exception]\ else\ v_{al.u.e})).$

8. MPL$_4$: pointers

The MPL$_4$ language adds the pointer types, a pointer access, a variable address access, a pointer creation, a pointer assignment, and a pointer deletion.

8.1. Conceptual states

The specific conceptuals of the conceptual states of MPL$_4$ are listed below.

A conceptual $[0{:}n_{at}, 1{:}pointer]$ is called a pointer literal. Let P_{oi} be a set of pointer literals. A literal p_{oi} is a pointer of the type $[pointer\ of\ t_y]$ in

$[\![s_{ta}]\!]$ if $s_{ta}(\{-1{:}element\text{-}type, 0{:}p_{oi}, 1{:}pointer\}) = t_y$. The following property holds for MPL$_4$: if $s_{ta}(\{-1{:}element\text{-}type, 0{:}p_{oi}, 1{:}pointer\}) \neq v_{al.u.e}$, then $s_{ta}(\{-1{:}element\text{-}type, 0{:}p_{oi}, 1{:}pointer\}) \in T_y$.

The conceptual $\{-1{:}value, 0{:}p_{oi}, 1{:}pointer\}$ specifies the value of p_{oi}. A value v_{al} of the type t_y is the value of p_{oi} of the type $[pointer\ of\ t_y]$ in $[\![s_{ta}]\!]$ if $v_{al} = s_{ta}(\{-1{:}value, 0{:}p_{oi}, 1{:}pointer\})$ and $s_{ta}(\{-1{:}element\text{-}type, 0{:}p_{oi}, 1{:}pointer\}) = t_y$.

The conceptual $\{-3{:}l_{e.c}, -2{:}s_c, -1{:}pointer, 0{:}v_{ar}, 1{:}variable\}$ specifies the variable v_{ar}, its type, and the pointer which v_{ar} represents in $[\![l_{e.c}, s_c]\!]$. An identifier v_{ar} is a variable of the type t_y which represents the pointer p_{oi} and has the value v_{al} in $[\![l_{e.c}, s_c, s_{ta}]\!]$ if $p_{oi} = s_{ta}(\{-3{:}l_{e.c}, -2{:}s_c, -1{:}pointer, 0{:}v_{ar}, 1{:}variable\})$, $t_y = s_{ta}(\{-1{:}element\text{-}type, 0{:}p_{oi}, 1{:}pointer\})$, and $v_{al} = s_{ta}(\{-1{:}value, 0{:}p_{oi}, 1{:}pointer\})$. The following property holds for MPL$_4$: if $s_{ta}(\{-3{:}l_{e.c}, -2{:}s_c, -1{:}pointer, 0{:}v_{ar}, 1{:}variable\}) \neq v_{al.u.e}$, then $s_{ta}(\{-3{:}l_{e.c}, -2{:}s_c, -1{:}pointer, 0{:}v_{ar}, 1{:}variable\})$ is a pointer in $[\![s_{ta}]\!]$. The pointer represented by the global variable v_{ar} in $[\![l_{e.c}, s_c, s_{ta}]\!]$ is specified by the conceptual $\{-3{:}0, -2{:}0, -1{:}pointer, 0{:}v_{ar}, 1{:}variable\}$.

For simplicity, we do not distinguish the cases of stack and heap.

8.2. Expressions

In this section, we define MPL$_4$ expressions and their semantics.

The element $([pointer\ of\ t_y]\ is\ type)$ specifying that $[pointer\ of\ t_y] \in T_y$ is defined by the rule

$(rule\ ([pointer\ of\ x]\ is\ type)\ var\ (x)\ then\ (x\ is\ type))$.

The element $(e_l\ is\ pointer\text{-}literal)$ specifying that e_l is a pointer literal is defined by the rule

$(rule\ ([0{:}x,\ 1{:}pointer]\ is\ pointer\text{-}literal)\ var\ (x)\ then\ (x\ is\ nat))$.

The element $(e_l\ is\ literal)$ specifying that $e_l \in L_i$ is defined by the rule

$(rule\ (x\ is\ literal)\ var\ (x)\ then\ (x\ is\ pointer\text{-}literal))$.

The element $(p_{oi}\ is\ pointer)$ specifying that p_{oi} is a pointer is defined by the rule

$(rule\ (x\ is\ pointer)\ var\ (x)\ then\ ((x\ is\ pointer\text{-}literal)\ and$
$(\{-1{:}element\text{-}type,\ 0{:}x,\ 1{:}pointer\}\ \texttt{!=}\ v_{al.u.e})))$.

The element $(*\ e_l)$ specifying the value of p_{oi}, where p_{oi} is the value of e_l, is defined by the rule

$(rule\ (*\ x)\ var\ (x)\ where\ ((*\ x)\ and\ (*.x\ is\ pointer))\ then$
$\{-1{:}value,\ 0{:}*.x,\ 1{:}pointer\})$.

The element (*element-type of* p_{oi}) specifying the element type of p_{oi} is defined by the rule

(*rule* (*element-type of* x) *var* (x) *where* (x *is pointer-literal*) *then*
{-1:*element-type*, 0:x, 1:*pointer*}).

The element (*type of* p_{oi}) specifying the type of p_{oi} is defined by the rule

(*rule* (*type of* x) *var* (x) *where* (x *is pointer*) *then*
[*pointer of* *.{-1:*element-type*, 0:x, 1:*pointer*}]).

The element (e_l *is* [*pointer of* t_y]) specifying that e_l is a pointer of the type [*pointer of* t_y] is defined by the rule

(*rule* (x *is* [*pointer of* y]) *var* (x, y) *then* ((x *is pointer*) *and*
('.y = {-1:*element-type*, 0:x, 1:*pointer*}))).

The element (*index of* i_d) is defined by the rules

(*rule* (*index of* x) *var* (x) *where* (x *is identifier*)
then (*index of* x *in* *.$l_{e.c.s..r}$, *.$s_{c.s..r}$));
(*rule* (x *is index in* y, z) *var* (x, y, z) *then*
(*if* ({-3:y, -2:z, -1:*pointer*, 0:x, 1:*variable*} != $v_{al.u.e}$)
then '.z
else (*if* (z = 0) *then* $v_{al.u.e}$
else (*index of* x *in* y, *.(z − 1))))).

The element (& v_{ar}) specifying the pointer represented by v_{ar} is defined by the rule

(*rule* (& x) *var* (x) *hvar* (w) *then* (*seq* (w ::= (*index of* x))
(*if* (w = $v_{al.u.e}$) *then* $v_{al.u.e}$
else {-3:y, -2:*.w, -1:*pointer*, 0:x, 1:*variable*}))).

The element v_{ar} is defined by the rule

(*rule* x *var* (x) *hvar* (w) *then* (*seq* (w ::= (*index of* x))
(*if* (w = $v_{al.u.e}$) *then* $v_{al.u.e}$ *else*
{-1:*value*, 0:*.{-3:y, -2:*.w, -1:*pointer*, 0:x, 1:*variable*},
1:*pointer*}))).

The element (*type of* v_{ar}) is defined by the rule

(*rule* x *var* (x) *hvar* (w) *then* (*seq* (w ::= (*index of* x))
(*if* (w = $v_{al.u.e}$) *then* $v_{al.u.e}$ *else*
{-1:*element-type*,
0:*.{-3:y, -2:*.w, -1:*pointer*, 0:x, 1:*variable*},
1:*pointer*}))).

The element (*new-pointer* t_y) specifying a new pointer of the type [*pointer of* t_y] is defined by the rule

(*rule* (*new-pointer* x) *var* (x) *where* (x *is type*) *hvar* (w) *then* (*seq*
(w ::= (*new-count pointer*))
({−1:*element-type*, 0:∗.w, 1:*pointer*} ::= '.x)
w)).

8.3. Statements

In this section, we define MPL_4 statements and their semantics.

The variable declaration is defined by the rule

(*rule* (*var* x y) *var* (x, y) *where* ((x *is identifier*) *and* (y *is type*))
hvar(w) *then*
(*if* ({−3:∗.$l_{e.c.s..r}$, −2:∗.$s_{c.s..r}$, −1:*pointer*, 0:x, 1:*variable*} **!=**
$v_{al.u.e}$) *then* $v_{al.u.e}$
else (*seq* (w ::= (*new-pointer* y))
({−3:∗.$l_{e.c.s..r}$, −2:∗.$s_{c.s..r}$, −1:*pointer*, 0:x, 1:*variable*} ::= w)
({−1:*element-type*, 0:∗.w, 1:*pointer*} ::= '.y)))).

The assignment is defined by the rule

(*rule* (x := y) *var* (x, y) *where* (∗ y) *hvar* (w, p) *then* (*seq*
(w ::= (*index of* x))
(*if* (w = $v_{al.u.e}$) *then* $v_{al.u.e}$ *else* (*seq*
(p ::= {−3:∗.$l_{e.c.s..r}$, −2:∗.w, −1:*pointer*, 0:x, 1:*variable*})
(*if* (∗.y *is* ∗.{−1:*element-type*, 0:∗.p, 1:*pointer*})
then ({−1:*value*, 0:∗.p, 1:*pointer*} ::= '.∗.y) *else* $v_{al.u.e}$))))).

The element (* $e_{l.1}$:= $e_{l.2}$) specifying the assignment of v_{al} to p_{oi}, where p_{oi} and v_{al} are the values of $e_{l.1}$ and $e_{l.2}$, respectively, is defined by the rule

(*rule* (∗ x := y) *var* (x, y) *where* ((∗ x, y) *and* (∗.x *is pointer*))
then (*if* (∗.y *is* ∗.{−1:*element-type*, 0:∗.x, 1:*pointer*})
then ({−1:*value*, 0:∗.x, 1:*pointer*} ::= '.∗.y) *else* $v_{al.u.e}$)).

The element (*delete-local-variables*) is defined by the rule

(*rule* (*delete-local-variables*) *hvar* (w) *then* (*seq*
(*foreach-match* {−3:∗.$l_{e.c.s..r}$, −2:∗.$s_{c.s..r}$, −1:*pointer*, 0:x,
1:*variable*} *var* (x) *do* (*seq*
(w ::=
{−3:∗.$l_{e.c.s..r}$, −2:∗.$s_{c.s..r}$, −1:*pointer*, 0:∗.x, 1:*variable*})
({−1:*element-type*, 0:∗.w, 1:*pointer*} ::=)
({−1:*value*, 0:∗.w, 1:*pointer*} ::=)
({−3:∗.$l_{e.c.s..r}$, −2:∗.$s_{c.s..r}$, −1:*pointer*, 0:∗.x, 1:*variable*} ::=)
)))).

The element (*delete-pointer* e_l) deleting the pointer p_{oi}, where p_{oi} is the value of e_l, is defined by the rule

(*rule* (*delete-pointer* x) *var* (x) *where* ((∗ x) *and* (∗.x *is pointer*))
then (*seq* ({−1:*element-type*, 0:∗.x, 1:*pointer*} ::=)
({−1:*value*, 0:∗.x, 1:*pointer*} ::=))).

9. MPL$_5$: break, continue, goto

The MPL$_5$ language adds the break statement, continue statement, goto statement, and label statement. Let L_{ab} be a set of the labels of label statements.

9.1. Conceptual states

In this section, we list the specific conceptuals of the conceptual states of MPL$_5$.

The exception [−1:*break*, 1:*exception*] specifies the execution of the break statement in $[\![l_{e.c}]\!]$.

The exception [−1:*continue*, 1:*exception*] specifies the execution of the continue statement in $[\![l_{e.c}]\!]$.

The exception [−1:*goto*, 0:l_{ab}, 1:*exception*] specifies the execution of the goto statement with the label l_{ab} in $[\![l_{e.c}]\!]$.

The conceptual {−2:$l_{e.c}$, −1:s_c, 0:l_{ab}, 1:*label*} specifies the label l_{ab} as the label of the label statement executed in $[\![l_{e.c}, s_c]\!]$.

9.2. Expressions

In this section, we define the MPL$_5$ expressions and their semantics.

The element (e_x *is not-admissible-function-exception*) is defined by the rule

(*rule* (x *is not-admissible-function-exception*) *var* (x) *where* (# x)
then ((x *is exception*) *and*
((('.x .. −1) = '.*break*) *or* (('.x .. −1) = '.*continue*) *or*
(('.x .. −1) = '.*goto*)))).

The element (e_l *is embedded-statement*) is defined by the rule

(*rule* (x *is embedded-statement*) *var* (x) *then*
(*not* ((x *matches* (*var* u v) *var* (u, v)
where ((u *is identifier*) *and* (v *is type*))) *or*
(x *matches* (*label* u) *var* (u) *where* (u *is identifier*)))))).

Thus, only variable declarations and label statements are not embedded in MPL$_5$.

9.3. Statements

In this section, we define MPL_5 statements and their semantics.

The element $(label\ l_{ab})$ specifying the label statement with the label l_{ab} is defined by the rule

$$(rule\ (label\ x)\ var\ (x)\ where\ (\#\ and\ (x\ is\ identifier))\ catch\ w\ then\ (seq\ (\{-2{:}*.l_{e.c.s..r},\ -1{:}*.s_{c.s..r},\ 0{:}x,\ 1{:}label\}\ ::=\ true)\ (if\ (((w\ ..\ 1)\ =\ '.exception)\ and\ ((w\ ..\ -1)\ =\ '.goto)\ and\ ((w\ ..\ 0)\ =\ '.x))\ then\ (w\ ::=\ true))\ (throw\ w)).$$

The element $(break)$ specifying the break statement is defined by the rule

$$(rule\ (break)\ then\ [-1{:}break,\ 1{:}exception]).$$

The element $(continue)$ specifying the continue statement is defined by the rule

$$(rule\ (continue)\ then\ [-1{:}continue,\ 1{:}exception]).$$

The element $(goto\ l_{ab})$ specifying the goto statement with the label l_{ab} is defined by the rule

$$(rule\ (goto\ x)\ var\ (x)\ where\ (x\ is\ identifier)\ then\ [-1{:}goto,\ 0{:}x,\ 1{:}exception]).$$

The while statement is defined by the rules

$$(rule\ (while\text{-}m\ x\ do\ y)\ var\ (x,\ y)\ then\ (seq\ (while1\ x\ do\ y)\ (delete\text{-}exception\ break)));$$
$$(rule\ (while1\ x\ do\ y)\ var\ (x,\ y)\ then\ (if\ x\ then\ (seq\ y\ (delete\text{-}exception\ continue)\ (while1\ x\ do\ y)))).$$

The block statement is defined by the rules

$$(rule\ (block\ .::\ x)\ var\ (x)\ then\ (seq\ (enter\text{-}block)\ (block1\ .::\ x)\ (exit\text{-}block)));$$
$$(rule\ (block1\ .::\ x)\ var\ (x)\ then\ (seq\ (seq\ .::\ x)\ (catch\ w\ (if\ ((*.w\ is\ exception)\ and\ ((w\ ..\ -1)\ =\ '.goto)\ and\ (\{-2{:}*.l_{e.c.s..r},\ -1{:}*.s_{c.s..r},\ 0{:}*.(w\ ..\ 0),\ 1{:}label\}\ =\ true))\ then\ (seq\ (throw\ w)\ (block1\ .::\ x))\ else\ (throw\ w)))).$$

The element $(exit\text{-}block)$ is defined by the rule

(*rule* (*exit-block*) *where* # *catch w then* (*seq*
(*delete-local-variables*) (*delete-labels*) ($s_{c.s..r}$ ::= ($s_{c.s..r}$ − 1))
(*throw w*))).

The element (*delete-labels*) deleting the labels of the label statements in $[\![l_{e.c.s..r}, s_{c.s..r}]\!]$ is defined by the rule

(*rule* (*delete-labels*) *then*
(*foreach-match* {−2:∗.$l_{e.c.s..r}$, −1:∗.$s_{c.s..r}$, 0:x, 1:*label*}
var (*x*) *do* ({−2:∗.$l_{e.c.s..r}$, −1:∗.$s_{c.s..r}$, 0:∗.x, 1:*label*} ::=))).

10. MPL$_6$: arrays and structures

The MPL$_6$ language adds the array and structure types, access to array elements and structure fields, structure declarations, array element and structure field assignments.

10.1. Conceptual states

In this section, we list the specific conceptuals of the conceptual states of MPL$_6$.

An element of the form [0:n_{at}, 1:*array*] is called an array literal. Let A_{rr} be a set of array literals. A literal a_{rr} is an array of the type [*array of* t_y] in $[\![s_{ta}]\!]$ if s_{ta}({−1:*element-type*, 0:a_{rr}, 1:*array*}) = t_y. The following property holds for MPL$_6$: if s_{ta}({−1:*element-type*, 0:a_{rr}, 1:*array*}) ≠ $v_{al.u.e}$, then s_{ta}({−1:*element-type*, 0:a_{rr}, 1:*array*}) ∈ T_y. The conceptual {−1:n_{at}, 0:a_{rr}, 1:*array*} specifies the value of the n_{at}-th element of a_{rr}. The value v_{al} of the type t_y is the value of the n_{at}-th element of a_{rr} of the type [*array of* t_y] in $[\![s_{ta}]\!]$ if $v_{al} = s_{ta}$({−1:n_{at}, 0:a_{rr}, 1:*array*} and s_{ta}({−1:*element-type*, 0:a_{rr}, 1:*array*}) = t_y.

Let $T_{y.s}$ and F_i be the sets of identifiers called structure types and fields. The conceptual {0:$t_{y.s}$, 1:*structure-type*} specifies the structure type $t_{y.s}$. The identifier $t_{y.s}$ is a structure type in $[\![s_{ta}]\!]$ if s_{ta}({0:$t_{y.s}$, 1:*structure-type*}) ≠ $v_{al.u.e}$. Let $T_{y.s}[\![s_{ta}]\!]$ be a set of structure types in $[\![s_{ta}]\!]$. The conceptual {−1:f_i, 0:$t_{y.s}$, 1:*structure-type*} specifies the type of the field f_i of the structure type $t_{y.s}$. A structure type $t_{y.s}$ has the field f_i of the type t_y in $[\![s_{ta}]\!]$ if s_{ta}({−1:f_i, 0:$t_{y.s}$, 1:*structure-type*}) = t_y. The following property holds for MPL$_6$: if s_{ta}({−1:f_i, 0:$t_{y.s}$, 1:*structure-type*}) ≠ $v_{al.u.e}$, then s_{ta}({−1:f_i, 0:$t_{y.s}$, 1:*structure-type*}) ∈ T_y.

An element of the form [0:n_{at}, 1:*structure*] is called a structure literal. Let S_{tr} be a set of structure literals. A literal s_{tr} is a structure of the type $t_{y.s}$ in $[\![s_{ta}]\!]$ if s_{ta}({−1:*type*, 0:s_{tr}, 1:*structure*}) = $t_{y.s}$. The following property holds for MPL$_6$: if s_{ta}({−1:*type*, 0:s_{tr}, 1:*structure*}) ≠ $v_{al.u.e}$, then s_{ta}({−1:*type*, 0:s_{tr}, 1:*structure*}) ∈ $T_{y.s}$. The conceptual {−1:f_i, 0:s_{tr}, 1:*structure*} specifies the value of f_i of s_{tr}. The value v_{al} of the type t_y

is the value of f_i of s_{tr} of the type $t_{y.s}$ in $[\![s_{ta}]\!]$ if $v_{al} = s_{ta}(\{-1{:}f_i, 0{:}s_{tr}, 1{:}\ structure\})$, $s_{ta}(\{-1{:}type, 0{:}s_{tr}, 1{:}structure\}) = t_{y.s}$, and $t_{y.s}$ is a structure type which has the field f_i of the type t_y in $[\![s_{ta}]\!]$.

10.2. Expressions

In this section, we define the MPL_6 expressions and their semantics.

The element $([array\ of\ t_y]\ is\ type)$ specifying that $[array\ of\ t_y] \in T_y$ is defined by the rule

$(rule\ ([array\ of\ x]\ is\ type)\ var\ (x)\ then\ (x\ is\ type)).$

The element $(e_l\ is\ array\text{-}literal)$ specifying that e_l is an array literal is defined by the rule

$(rule\ ([0{:}x,\ 1{:}array]\ is\ array\text{-}literal)\ var\ (x)\ then\ (x\ is\ nat)).$

The element $(a_{rr}\ is\ array)$ specifying that a_{rr} is an array is defined by the rule

$(rule\ (x\ is\ array)\ var\ (x)\ then\ ((x\ is\ array\text{-}literal)\ and$
$(\{-1{:}element\text{-}type,\ 0{:}x,\ 1{:}array\}\ \texttt{!=}\ v_{al.u.e}))).$

The element $(element\text{-}type\ of\ a_{rr})$ specifying the element type of a_{rr} is defined by the rule

$(rule\ (element\text{-}type\ of\ x)\ var\ (x)\ where\ (x\ is\ array)\ then$
$\{-1{:}element\text{-}type,\ 0{:}x,\ 1{:}array\}).$

The element $(type\ of\ a_{rr})$ specifying the type of a_{rr} is defined by the rule

$(rule\ (type\ of\ x)\ var\ (x)\ where\ (x\ is\ array)\ then$
$[array\ of\ *.\{-1{:}element\text{-}type,\ 0{:}x,\ 1{:}array\}]).$

The element $(e_l\ is\ [array\ of\ t_y])$ specifying that e_l is an array of the type $[array\ of\ t_y]$ is defined by the rule

$(rule\ (x\ is\ [array\ of\ y])\ var\ (x,\ y)\ then\ ((x\ is\ array)\ and$
$('.y\ =\ \{-1{:}element\text{-}type,\ 0{:}x,\ 1{:}array\}))).$

The element $(e_l\ "["\ e_{l.1}\ "]")$ specifying the value of the n_{at}-th element of a_{rr}, where a_{rr} and n_{at} are the values of e_l and $e_{l.1}$, respectively, is defined by the rule

$(rule\ (x\ "["\ y\ "]")\ var\ (x,\ y)$
$where\ ((*\ x,\ y)\ and\ (*.x\ is\ array)\ and\ (*.y\ is\ nat))$
$then\ \{-1{:}*.y,\ 0{:}*.x,\ 1{:}array\}).$

The element ($t_{y.s}$ *is structure-type*) specifying that $t_{y.s}$ is a structure type is defined by the rule

(*rule* (*x is structure-type*) *var* (*x*) *then* ((*x is identifier*) *and*
({0:*x*, 1:*structure-type*} != $v_{al.u.e}$))).

The element ($t_{y.s}$ *is type*) specifying that $t_{y.s} \in T_y$ is defined by the rule

(*rule* (*x is type*) *var* (*x*, *y*) *then* (*x is structure-type*)).

The element (e_l *is structure-literal*) specifying that e_l is a structure literal is defined by the rule

(*rule* ([0:*x*, 1:*structure*] *is structure-literal*) *var* (*x*)
then (*x is nat*)).

The element (e_l *is literal*) specifying that $e_l \in L_i$ is defined by the rules

(*rule* (*x is literal*) *var* (*x*) *then* (*x is array-literal*));
(*rule* (*x is literal*) *var* (*x*) *then* (*x is structure-literal*)).

The element (s_{tr} *is structure*) specifying that s_{tr} is a structure is defined by the rule

(*rule* (*x is structure*) *var* (*x*) *then* ((*x is structure-literal*) *and*
({−1:*type*, 0:*x*, 1:*structure*} != $v_{al.u.e}$))).

The element (*type of* s_{tr}) specifying the type of s_{tr} is defined by the rule

(*rule* (*type of x*) *var* (*x*) *where* (*x is structure*) *then*
{−1:*type*, 0:*x*, 1:*structure*}).

The element (e_l *is* $t_{y.s}$) specifying that e_l is a structure of the type $t_{y.s}$ is defined by the rule

(*rule* (*x is y*) *var* (*x*, *y*) *then* ((*y is structure-type*) *and*
('.*y* = {−1:*type*, 0:*x*, 1:*structure*}))).

The element (e_l "." f_i) specifying the value of the field f_i of s_{tr}, where s_{tr} is the value of e_l, is defined by the rule

(*rule* (*x* "." *y*) *var* (*x*, *y*) *where* ((∗ *x*) *and* (∗.*x is structure*)) *then*
(*if* ({−1:*y*, 0:∗.{−1:*type*, 0:*x*, 1:*structure*}, 1:*structure-type*} !=
$v_{al.u.e}$)
then {−1:*y*, 0:∗.*x*, 1:*structure*} *else* $v_{al.u.e}$)).

10.3. Statements

In this section, we define the MPL_6 statements and their semantics.

The element $(struct\ t_{y.s}\ ((f_{i.*}(1)\ t_{y.*}(1))\ \ldots,\ (f_{i.*}(n_{at})\ t_{y.*}(n_{at}))))$, where $len(f_{i.*}) = len(t_{y.*}) = n_{at}$, specifying the declaration of the structure type $t_{y.s}$ with the fields $f_{i.*}$ of the types $t_{y.*}$ is defined by the rules

$(rule\ (struct\ x\ y)\ var\ (x,\ y)$
$\ where\ ((x\ is\ identifier)\ and\ (y\ is\ evaluable\text{-}ordered\text{-}element))\ then$
$\ (if\ (x\ is\ structure\text{-}type)\ then\ v_{al.u.e}$
$\ \ else\ (seq\ (\{0{:}x,\ 1{:}structure\text{-}type\}\ ::=\ true)\ (struct1\ x\ y))));$
$(rule\ (struct1\ x\ ((y\ z)\ .::\ u))\ var\ (x,\ y,\ z,\ u)$
$\ where\ ((y\ is\ identifier)\ and\ (z\ is\ type))\ then$
$\ (seq\ (\{-1{:}y,\ 0{:}x,\ 1{:}structure\text{-}type\}\ ::=\ '.z)\ (struct1\ x\ u)));$
$(rule\ (struct1\ x\ ())\ var\ (x)\ then\ (skip)).$

The element $(e_{l.1}\ "["\ e_{l.2}\ "]"\ :=\ e_{l.3})$ specifying the assignment of v_{al} to the n_{at}-th element of a_{rr}, where a_{rr}, n_{at}, and v_{al} are the values of $e_{l.1}$, $e_{l.2}$, and $e_{l.3}$, respectively, is defined by the rule

$(rule\ (x\ "["\ y\ "]"\ :=\ z)\ var\ (x,\ y,\ z)\ where\ (*\ x,\ y,\ z)\ then$
$\ (if\ ((*.x\ is\ array)\ and\ (*.y\ is\ nat)\ and$
$\ \ \ \ (*.z\ is\ *.(element\text{-}type\ of\ *.x)))$
$\ \ then\ (\{-1{:}*.y,\ 0{:}*.x,\ 1{:}array\}\ ::=\ '.*.z)\ else\ v_{al.u.e})).$

The element $(e_{l.1}\ "."\ f_i\ :=\ e_{l.2})$ specifies the assignment of v_{al} to the field f_i of s_{tr}, where s_{tr} and v_{al} are the values of $e_{l.1}$ and $e_{l.2}$, respectively, is defined by the rule

$(rule\ (x\ "."\ y\ :=\ z)\ var\ (x,\ y,\ z)$
$\ where\ ((*\ x)\ and\ (*.x\ is\ structure))$
$\ hvar\ (t,\ w)\ then\ (seq$
$\ \ (t\ ::=$
$\ \ \ \{-1{:}y,\ 0{:}*.\{-1{:}type,\ 0{:}*.x,\ 1{:}structure\},\ 1{:}structure\text{-}type\})$
$\ \ (if\ (t\ =\ v_{al.u.e})\ then\ v_{al.u.e}\ else\ (seq\ (w\ ::=\ z)$
$\ \ \ \ (if\ (*.w\ is\ *.t)$
$\ \ \ \ \ then\ (\{-1{:}y,\ 0{:}*.x,\ 1{:}structure\}\ ::=\ w)\ else\ v_{al.u.e}))))).$

11. MPL_7: functional and procedural types and variables

The MPL_7 language adds the functional and procedural types and variables.

11.1. Conceptual states

In this section, we list the specific conceptuals of the conceptual states of MPL_7.

An element $[0{:}n_{at}, 1{:}function]$ is called a function literal. Let F_u be a set of function literals. A literal f_u is a function in $[\![s_{ta}]\!]$ with argument types $t_{y.[*]}$ if $s_{ta}(\{-1{:}argument\text{-}types, 0{:}f_u, 1{:}function\}) = t_{y.[*]}$. The conceptual $\{-1{:}return\text{-}type, 0{:}f_u, 1{:}function\}$ specifies the return type of f_u. A type t_y is the return type of f_u in $[\![s_{ta}]\!]$ if $s_{ta}(\{-1{:}return\text{-}type, 0{:}f_u, 1{:}function\}) = t_y$. A function f_u with the return type t_y in $[\![s_{ta}]\!]$ is a procedure in $[\![s_{ta}]\!]$ if $t_y = void$. The conceptual $\{-1{:}parameters, 0{:}f_u, 1{:}function\}$ specifies the parameters of f_u. The function f_u has the parameters $i_{d.[*]}$ in $[\![s_{ta}]\!]$ if $s_{ta}(\{-1{:}parameters, 0{:}f_u, 1{:}function\}) = i_{d.[*]}$. The conceptual $\{-1{:}body, 0{:}f_u, 1{:}function\}$ specifies the body of f_u. The function f_u has the body b_{od} in $[\![s_{ta}]\!]$ if $s_{ta}(\{-1{:}body, 0{:}f_u, 1{:}function\}) = b_{od}$. The following property holds for MPL$_7$: if $s_{ta}(\{-1{:}argument\text{-}types, 0{:}f_u, 1{:}function\}) \neq v_{al.u.e}$, then there exist $t_{y.[*]}$, t_y and $i_{d.[*]}$ such that $s_{ta}(\{-1{:}argument\text{-}types, 0{:}f_u, 1{:}function\}) = t_{y.[*]}$, $s_{ta}(\{-1{:}return\text{-}type, 0{:}f_u, 1{:}function\}) = t_y$, $\{-1{:}parameters, 0{:}f_u, 1{:}function\} = i_{d.[*]}$, $\{-1{:}body, 0{:}f_u, 1{:}function\} \neq v_{al.u.e}$, and $len(t_{y.[*]}) = len(i_{d.[*]})$. The function f_u has the type $[function\ of\ t_{y.[*]}\ t_y]$ in $[\![s_{ta}]\!]$ if $s_{ta}(\{-1{:}argument\text{-}types, 0{:}f_u, 1{:}function\}) = t_{y.[*]}$, and $s_{ta}(\{-1{:}return\text{-}type, 0{:}f_u, 1{:}function\}) = t_y$.

The conceptual $\{-1{:}t_{y.[*]}, 0{:}i_d, 1{:}function\text{-}name\}$ specifies the function with the parameter types $t_{y.[*]}$ represented by the name i_d. The identifier i_d represents the function f_u with the argument types $t_{y.[*]}$ in $[\![s_{ta}]\!]$ if $s_{ta}(\{-1{:}t_{y.[*]}, 0{:}i_d, 1{:}function\text{-}name\}) = f_u$, and f_u is a function with the argument types $t_{y.[*]}$ in $[\![s_{ta}]\!]$. Let $N_{am.f}$ be a set of function names. The following property holds for MPL$_7$: if $s_{ta}(\{-1{:}t_{y.[*]}, 0{:}i_d, 1{:}function\text{-}name\}) \neq v_{al.u.e}$, then $s_{ta}(\{-1{:}t_{y.[*]}, 0{:}i_d, 1{:}function\text{-}name\})$ is a function with the argument types $t_{y.[*]}$ in $[\![s_{ta}]\!]$.

11.2. Expressions

In this section, we define the MPL$_7$ expressions and their semantics.

The element $([function\ of\ t_{y.[*]}\ t_y]\ is\ type)$ specifying that $[function\ of\ t_{y.[*]}\ t_y] \in T_y$ is defined by the rule

$(rule\ ([function\ of\ x\ y]\ is\ type)\ var\ (x,\ y)$
$\ then\ ((x\ is\ type\text{-}sequence)\ and\ (y\ is\ type)))$.

The element $(e_l\ is\ type\text{-}sequence)$ specifying that $e_l \in T_{y.[*]}$ is defined by the rules

$(rule\ ([x\ ::\ y]\ is\ type\text{-}sequence)\ var\ (x,\ y)$
$\ then\ ((x\ is\ type)\ and\ (y\ is\ type\text{-}sequence)))$;
$(rule\ ([\ ]\ is\ type\text{-}sequence)\ then\ true)$.

The element $(e_l\ is\ function\text{-}literal)$ specifying that e_l is a function literal is defined by the rule

$(rule\ ([0{:}x,\ 1{:}function]\ is\ function\text{-}literal)\ var\ (x)$
$\ then\ (x\ is\ nat)).$

The element $(e_l\ is\ literal)$ specifying that $e_l \in L_i$ is defined by the rules

$(rule\ (x\ is\ literal)\ var\ (x)\ then\ (x\ is\ function\text{-}literal)).$

The element $(f_u\ is\ function)$ specifying that f_u is a function is defined by the rule

$(rule\ (x\ is\ function)\ var\ (x)\ then\ ((x\ is\ function\text{-}literal)\ and$
$\ (\{-1{:}argument\text{-}types,\ 0{:}x,\ 1{:}function\}\ \texttt{!=}\ v_{al.u.e}))).$

The element $(type\ of\ f_u)$ specifying the type of f_u is defined by the rule

$(rule\ (type\ of\ x)\ var\ (x)\ where\ (x\ is\ function)\ then$
$\ [function\ of$
$\ \ *.\{-1{:}argument\text{-}types,\ 0{:}x,\ 1{:}function\}$
$\ \ *.\{-1{:}return\text{-}type,\ 0{:}x,\ 1{:}function\}]).$

The element $(e_l\ is\ [function\ of\ t_{y.[*]}\ t_y])$ specifying that e_l is a function of the type $[function\ of\ t_{y.[*]}\ t_y]$ is defined by the rule

$(rule\ (x\ is\ [function\ of\ y\ z])\ var\ (x,\ y,\ z)$
$\ then\ ((x\ is\ function)\ and$
$\ \ ('.y\ =\ \{-1{:}argument\text{-}types,\ 0{:}x,\ 1{:}function\})\ and$
$\ \ ('.z\ =\ \{-1{:}return\text{-}type,\ 0{:}x,\ 1{:}function\})))$

The element $(fun\text{-}call\ e_l\ (e_{l.*}))$ specifying the call of f_u with the arguments $e_{l.*}$, where f_u is the value of e_l, is defined by the rule

$(rule\ (fun\text{-}call\ x\ y)\ var\ (x,\ y)$
$\ where\ (y\ is\ evaluable\text{-}ordered\text{-}element)\ hvar\ (u,\ t,\ s,\ f)\ then\ (seq$
$\ \ (u\ ::=\ (execute\text{-}arguments\ y))\ (t\ ::=\ *.(argument\text{-}types\ *.u))$
$\ \ (if\ ((x\ is\ identifier)\ and$
$\ \ \ \ (\{-1{:}*.t,\ 0{:}x,\ 1{:}function\text{-}name\}\ \texttt{!=}\ v_{al.u.e}))\ then$
$\ \ \ (f\ ::=\ \{-1{:}*.t,\ 0{:}x,\ 1{:}function\text{-}name\})\ else\ (f\ ::=\ x))).$
$\ \ (if\ (\{-1{:}argument\text{-}types,\ 0{:}f,\ 1{:}function\}\ =\ t)\ then\ (seq$
$\ \ \ (l_{e.c.s..r}\ ::=\ (l_{e.c.s..r}\ +\ 1))\ (s\ ::=\ s_{c.s..r})\ (s_{c.s..r}\ ::=\ 1)$
$\ \ \ (create\text{-}local\text{-}variables$
$\ \ \ \ *.\{-1{:}parameters,\ 0{:}*.f,\ 1{:}function\}\ *.t\ *.u)$
$\ \ \ (t_{y.r.s..r}\ ::=\ \{-1{:}return\text{-}type,\ 0{:}f,\ 1{:}function\})$
$\ \ \ *.\{-1{:}body,\ 0{:}*.f,\ 1{:}function\}$
$\ \ \ (catch\ w\ (seq$
$\ \ \ \ (if\ (*.w\ is\ not\text{-}admissible\text{-}function\text{-}exception)\ then\ v_{al.u.e})$
$\ \ \ \ (if\ ((not\ (*.w\ is\ exception))\ and\ (t_{y.r.s..r}\ \texttt{!=}\ void))\ then\ v_{al.u.e})$
$\ \ \ \ (exit\text{-}block)\ (t_{y.r.s..r}\ ::=)\ (l_{e.c.s..r}\ ::=\ (l_{e.c.s..r}\ -\ 1))$

$(s_{c.s..r}\ ::=\ s)$
$(if\ ((w\ ..\ -1)\ =\ \text{'}.return)\ then\ (w\ ::=\ (w\ ..\ 0)))$
$(throw\ w)))$
$else\ v_{al.u.e})))).$

For simplicity, we impose a constraint on function calls that function names can not be arguments. This constraint is not essential, for we can assign a function name to a function variable and use the function variable as an argument.

11.3. Statements

In this section, we define the MPL_7 statements and their semantics.

The function declaration is defined by the rule

$(rule\ (function\ x\ y\ z\ u)\ var\ (x,\ y,\ z,\ u)\ hvar\ (w,\ f,\ t)$
$where\ ((x\ is\ identifier)\ and\ (y\ is\ evaluable\text{-}ordered\text{-}element)\ and$
$(z\ is\ type))\ then\ (seq$
$(t\ ::=\ (extract\text{-}types\ y))$
$(if\ ((t\ =\ v_{al.u.e})\ or$
$(\{-1{:}*.t,\ 0{:}x,\ 1{:}function\text{-}name\}\ \texttt{!=}\ v_{al.u.e}))\ then\ v_{al.u.e}\ else$
$(f\ ::=\ (new\text{-}count\ function))$
$(\{-1{:}return\text{-}type,\ 0{:}*.f,\ 1{:}function\}\ ::=\ \text{'}.z)$
$(\{-1{:}parameters,\ 0{:}*.f,\ 1{:}function\}\ ::=\ (extract\text{-}names\ y))$
$(\{-1{:}body,\ 0{:}*.f,\ 1{:}function\}\ ::=\ \text{'}.u)$
$(\{-1{:}argument\text{-}type,\ 0{:}*.f,\ 1{:}function\}\ ::=\ t)$
$(\{-1{:}*.t,\ 0{:}x,\ 1{:}function\text{-}name\}\ ::=\ f)))).$

The assignment $(e_{l.1}\ :=\ e_{l.2})$ is defined by the rule

$(rule\ (x\ :=\ y)\ var\ (x,\ y)\ hvar\ (w,\ p,\ t,\ v)\ then\ (seq$
$(w\ ::=\ (index\ of\ x))$
$(if\ (w\ =\ v_{al.u.e})\ then\ v_{al.u.e}\ else\ (seq$
$(p\ ::=\ \{-3{:}*.l_{e.c.s..r},\ -2{:}*.w,\ -1{:}pointer,\ 0{:}x,\ 1{:}variable\})$
$(t\ ::=\ \{-1{:}element\text{-}type,\ 0{:}*.p,\ 1{:}pointer\})$
$(v\ ::=\ (value\ of\ y\ in\ t))\ (if\ (v\ =\ v_{al.u.e})\ then\ (v\ ::=\ y))$
$(if\ (*.v\ is\ *.t)$
$then\ (\{-1{:}value,\ 0{:}*.p,\ 1{:}pointer\}\ ::=\ v)\ else\ v_{al.u.e})))))).$

The assignment $(value\ of\ i_d\ in\ t_y)$ is defined by the rule

$(rule\ (value\ of\ x\ in\ [function\ of\ y\ z])\ var\ (x,\ y,\ z)$
$where\ ((x\ is\ identifier)\ and\ (y\ is\ type\text{-}sequence)\ and\ (z\ is\ type))$
$then\ \{-1{:}y,\ 0{:}x,\ 1{:}function\text{-}name\}).$

The assignment $(*\ e_{l.1}\ :=\ e_{l.2})$ is defined by the rule

$(rule\ (*\ x\ :=\ y)\ var\ (x,\ y)$
$\ where\ ((*\ x)\ and\ (*.x\ is\ pointer))\ hvar\ (t,\ v)$
$\ then\ (seq\ (t\ ::=\ \{-1{:}element\text{-}type,\ 0{:}*.x,\ 1{:}pointer\})$
$\ (v\ ::=\ (value\ of\ y\ in\ t))\ (if\ (v\ =\ v_{al.u.e})\ then\ (v\ ::=\ y))$
$\ (if\ (*.v\ is\ *.t)$
$\ then\ (\{-1{:}value,\ 0{:}*.x,\ 1{:}pointer\}\ ::=\ v)\ else\ v_{al.u.e}))).$

The assignment ($e_{l.1}$ "[" $e_{l.2}$ "]" := $e_{l.3}$) is defined by the rule

$(rule\ (x\ "["\ y\ "]"\ :=\ z)\ var\ (x,\ y,\ z)$
$\ where\ ((*\ x,\ y)\ and\ (*.x\ is\ array)\ and\ (*.y\ is\ nat))\ hvar\ (t,\ v)$
$\ then\ (seq$
$\ (t\ ::=\ \{-1{:}element\text{-}type,\ 0{:}*.x,\ 1{:}pointer\})$
$\ (v\ ::=\ (value\ of\ y\ in\ t))\ (if\ (v\ =\ v_{al.u.e})\ then\ (v\ ::=\ y))$
$\ (if\ (*.v\ is\ *.t)$
$\ then\ (\{-1{:}*.y,\ 0{:}*.x,\ 1{:}array\}\ ::=\ v)\ else\ v_{al.u.e}))).$

The assignment ($e_{l.1}$ "." f_i := $e_{l.2}$) is defined by the rule

$(rule\ (x\ "."\ y\ :=\ z)\ var\ (x,\ y,\ z)$
$\ where\ ((*\ x)\ and\ (*.x\ is\ structure))\ hvar\ (w,\ t,\ v)\ then\ (seq$
$\ (t\ ::=$
$\ \{-1{:}y,\ 0{:}*.\{-1{:}type,\ 0{:}*.x,\ 1{:}structure\},\ 1{:}structure\text{-}type\})$
$\ (if\ (t\ =\ v_{al.u.e})\ then\ v_{al.u.e}\ else\ (seq$
$\ (v\ ::=\ (value\ of\ z\ in\ t))\ (if\ (v\ =\ v_{al.u.e})\ then\ (v\ ::=\ z))$
$\ (if\ (*.v\ is\ *.t)$
$\ then\ (\{-1{:}y,\ 0{:}*.x,\ 1{:}structure\}\ ::=\ v)\ else\ v_{al.u.e}))))).$

12. Conclusion

Although the methodology proposed in this paper has been applied only to procedural programming languages, it can be extended to other kinds of computer languages (object-oriented programming languages, executable specification languages, and parallel programming languages).

In the near future, we plan to extend the methodology to object-oriented programming languages, to add typical object-oriented constructs to MPL, and to apply the methodology to real programming languages.

References

[1] Plotkin G.D. A Structural Approach to Operational Semantics. – Aarhus, Denmark. 1981. – (Tech. Rep. / Computer Science Department, Aarhus University; DAIMI FN-19).

[2] Gurevich Y. Abstract state machines: an overview of the project // Foundations of Information and Knowledge Systems. – Lect. Notes Comput. Sci. – 2004. – Vol. 2942. – P. 6–13.

[3] Gurevich Y. Evolving algebras. Lipari guide // Specification and Validation Methods. – Oxford University Press, 1995. – P. 9–36.

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

[5] Gruber T.R. Toward principles for the design of ontologies used for knowledge sharing // Internat. J. Human-Computer Studies. – 1995. – Vol. 43(5-6). – P. 907–928.

[6] Anureev I.S. Ontological Transition Systems // Bulletin NCC. Series: Computer Science. – 2007. – Iss. 26. – P. 1–17.

[7] Anureev I.S. A language of actions in ontological transition systems // Bulletin NCC. Series: Computer Science. – 2007. – Iss. 26. – P. 19–38.

[8] Anureev I.S. Domain-specific transition systems and their application to a formal definition of a model programming language // Bulletin NCC. Series: Computer Science. – 2014. – Iss. 34. – P. 23–42.

[9] Anureev I.S. Conceptual transition systems // System Informatics. – 2015. – No. 5. – P. 1–38. – URL: http://www.system-informatics.ru/en/article/77 (accessed: 07.12.2015).

[10] Anureev I.S. Kinds and language of conceptual transition systems // System Informatics. – 2015. – No. 5. – P. 55–74. – URL: http://www.system-informatics.ru/en/article/80 (accessed: 07.12.2015).