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, 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 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-
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 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 language (Section 6) adds variable scopes. The MPL language (Section 7) adds functions, procedures, and the return statement. The MPL language (Section 8) adds pointers. The MPL language (Section 9) adds transfer-of-control statements such as break, continue and goto. The MPL language (Section 10) adds compound types such as arrays and structures. The MPL 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 \(X_n\) are represented by \(x_n, x_{n+1}, x_{n+2}, x_{n}', x_{n}''\), and so on.

Let \(B_{oo} = \{true, false\}\). Let \(I_{n,x}, N_{at}\) and \(N_{nat,0}\) denote the sets of integers, natural numbers and natural numbers with zero; \(Ob, F_u, S_{et}, L_{ab}, A_{rg}, V_{al}\), and \(V_{at}\) 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 \(s_{et,(s)}, s_{et,[s]}, s_{et,*}\), and \(s_{et,s}\) denote the sets of sequences of the forms \((o_{b,1}, \ldots, o_{b,nat,0}), [o_{b,1}, \ldots, o_{b,nat,0}], \{o_{b,1}, \ldots, o_{b,nat,0}\}\), and \(o_{b,1, \ldots, o_{b,nat,0}}\) from elements of \(s_{et}\). For example, \(I_{n,x,r}(*, s)\) is a set of sequences of the form \(i_{n,x,1, \ldots, i_{n,x,nat,0}}\), and \(i_{n,x,s}\) is a sequence of the form \(i_{n,x,1, \ldots, i_{n,x,nat,0}}\). 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 \([o_{b,s}]\), where the elements of \(o_{b,s}\) called embedded contexts have the
form: \(l_{ab}:o_b, l_{ab}:o_b\), 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 \(o_b[[o_b,s]]\) denote the object \(o_b\) in the context \([o_b,s]\].

Let \(S_{y,t,c}\) denote a set of conceptual transition systems [9]. Let \(A_{to}, E_{l}, E_{l,s}, C_{o,d}, C_{o,pt}, A_{it}, S_{ta}\) 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 \(B_{oo,u} = \{true, val:u:e\}\). The element \(val:u:e\) is called the element undefined value.

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

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 conceptualss 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 \(C_{o,p.t.r}[s_{ta}]\) 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 \(C_{o,p.t.r}\), in addition to the concepts of CTSL includes the concepts literal and identifier specifying literals and identifiers in \(l_{an,p}\), respectively;
- a set \(T_y[s_{ta}] \subseteq C_{o,p.t.r}[s_{ta}]\) 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 \(C_{o,p.t.r}, T_y\) and \(V_{ar}\) denote \(C_{o,p.t.r}[s_{ta}], T_y[s_{ta}],\) 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 \(\text{int}\) and \(\text{bool}\) 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 <, >, <\(\leq\) and \(\geq\), 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 \(\{-1: \text{type}, 0:v_{ar}, 1:\text{variable}\}\) specifies the variable \(v_{ar}\) and its type. An identifier \(v_{ar}\) is a variable of a type \(t_y\) in \([s_{ta}]\) if \(s_{ta}(\{-1: \text{type}, 0: v_{ar}, 1:\text{variable}\}) = t_y\). Let \(V_{ar}\) be a set of variables. The following property holds for MPL\(_1\): if \(s_{ta}(\{-1: \text{type}, 0:v_{ar}, 1:\text{variable}\}) \neq v_{al,u.e.}\), then \(s_{ta}(\{-1: \text{type}, 0:v_{ar}, 1:\text{variable}\}) \in T_y\). The conceptual \(\{-1: \text{value}, 0:v_{ar}, 1:\text{variable}\}\) specifies the value of \(v_{ar}\). A variable \(v_{ar}\) has the value \(v_{al}\) in \([s_{ta}]\) if \(s_{ta}(\{-1: \text{value}, 0:v_{ar}, 1:\text{variable}\}) = v_{al}\).
5.2. Expressions

In this section, we define the MPL₁ expressions and their semantics.

The element \((\cdot.e₁ \text{ is } \cdot.e₁)\) specifying rule-defined concepts and their instances is defined by the rule
\[(\text{rule } (\cdot.x \text{ is } \cdot.y) \text{ var } (x, y) \text{ where } (* x, y) \text{ then } (*, x \text{ is } *, y)).\]

The element \((e₁ \text{ is type})\) specifying that \(e₁ \in T_y\) is defined by the rules
\[(\text{rule } (\text{int is type}) \text{ then true});\]
\[(\text{rule } (\text{bool is type}) \text{ then true}).\]

The element \((e₁ \text{ is literal})\) specifying that \(e₁ \in T_y\) is defined by the rules
\[(\text{rule } (\text{x is literal}) \text{ var } (x) \text{ then } (x \text{ is int}));\]
\[(\text{rule } (\text{x is literal}) \text{ var } (x) \text{ then } (x \text{ is bool})).\]

The element \((e₁ \text{ is identifier})\) specifying that \(e₁ \in I_d\) is defined by the rule
\[(\text{rule } (\text{x is identifier}) \text{ var } (x) \text{ then } ((x \text{ is atom}) \text{ and } \text{not } (x \text{ is literal}))).\]

The elements \((\text{type of } b_{oo})\) and \((\text{type of } i_{n,r})\) returning the types of \(b_{oo}\) and \(i_{n,r}\), respectively, are defined by the rules
\[(\text{rule } (\text{type of } x) \text{ var } (x) \text{ where } (x \text{ is bool}) \text{ then } '\cdot bool);\]
\[(\text{rule } (\text{type of } x) \text{ var } (x) \text{ where } (x \text{ is int}) \text{ then } '\cdot int).\]

The element \((e₁ \text{ is variable})\) specifying that \(e₁ \in V_{ar}\) is defined by the rule
\[(\text{rule } (\text{x is variable}) \text{ var } (x) \text{ then } ((\text{x is identifier}) \text{ and } \text{not } (\text{x is literal}))).\]

The element \(v_{ar}\) specifying the value of \(v_{ar}\) is defined by the rule
\[(\text{rule } x \text{ var } (x) \text{ where } (x \text{ is variable}) \text{ then } \{-1: \text{value}, 0:x, 1: \text{variable}\}).\]

The element \((\text{type of } v_{ar})\) specifying the type of \(v_{ar}\) is defined by the rule
\[(\text{rule } (\text{type of } x) \text{ var } (x) \text{ where } (x \text{ is identifier}) \text{ then } \{-1: \text{type}, 0:x, 1: \text{variable}\}).\]

MPL₁ inherits the following operations of CTSL: \(+, -, *, \text{ div}, \text{ mod}, =, \neq, <, <=, >, >=, \text{ and}, \text{ or}, \text{ not}, =>, \text{ and } <=>.\)

The element \((e₁ \text{ is embedded-statement})\) specifying that \(e₁\) is an embedded statement, i.e. a statement which can be included in compound statements at the top level, is defined by the rule
\[(\text{rule } (\text{x is embedded-statement}) \text{ var } (x) \text{ then true}).\]

Thus, all MPL₁ statements are embedded.
5.3. Statements

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

The element $(\text{var } v \text{ t}_y)$ specifying the declaration of the variable $v$ of the type $t_y$ is defined by the rule

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

The element $(v_\text{ar} := e_1)$ specifying the assignment of the value of $e_1$ to $v_\text{ar}$ is defined by the rule

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

The elements $(\text{if-m } c_{o.n} \text{ then } e_{1.1} \text{ else } e_{1.2})$ and $(\text{if-m } c_{o.n} \text{ then } e_{1.2})$ specifying the if statement with the condition $c_{o.n}$, then-branch $e_{1.1}$ and else-branch $e_{1.2}$ are defined by the rules:

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

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

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

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

The element $(\text{block } e_{1.s})$ specifying the block statement with the body $e_{1.s}$ is defined by the rule

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

6. MPL$_2$: variable scopes

The MPL$_2$ language adds variable scopes. The scope of $v_\text{ar}$ occurring in $c_{o.c.t.l.p}$ is the number of blocks surrounding this occurrence of $v_\text{ar}$ in $c_{o.c.t.l.p}$. The value and type of $v_\text{ar}$ depend on its scope. The variable $v_\text{ar}$ can be global (with the scope 0) and local. The following example illustrates variable scopes:
In this section, we define the MPL

\( \text{MPL} \)

In this section, we list the specific conceptual states of the conceptual states of

6.1. Conceptual states

In this section, we list the specific conceptual states of the conceptual states of MPL2.

Let \( S_c \) be a set of variable scopes represented by the elements of \( N_{af.0} \).

The conceptual \( \{ 0 : \text{scope} \} \) 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 \( \{ -2 : s_c, -1 : \text{type}, 0 : v_{ar}, 1 : \text{variable} \} \) 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 \( s_{ta}(\{ -2 : s_c, -1 : \text{type}, 0 : v_{ar}, 1 : \text{variable} \}) = t_y \). 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 MPL2: if \( s_{ta}(\{ -2 : s_c, 0 : v_{ar}, 1 : \text{variable} \}) \neq v_{al,u,e} \), then \( s_{ta}(\{ -2 : s_c, -1 : \text{type}, 0 : v_{ar}, 1 : \text{variable} \}) \in T_y \). The conceptual \( \{ -2 : s_c, -1 : \text{value}, 0 : v_{ar}, 1 : \text{variable} \} \) specifies the value of \( v_{ar} \) in \([s_c]\). The variable \( v_{ar} \) has the value \( v_{at} \) in \([s_c, s_{ta}]\) if \( s_{ta}(\{ -2 : s_c, -1 : \text{value}, 0 : v_{ar}, 1 : \text{variable} \}) = v_{al} \).

6.2. Expressions

In this section, we define the MPL2 expressions and their semantics.

The element \( (\text{index of } i_d) \) is defined by the rules

\[
\begin{align*}
\text{(rule (index of } x) \text{ var } (x) \text{ where } (x \text{ is identifier}) & \text{ then } (\text{index of } x \text{ in } *, s_{c,s,r})); \\
\text{(rule (index of } x \text{ in } y) \text{ var } (x, y) \text{ then } & (\text{if } \{-2 : y, -1 : \text{type}, 0 : x, 1 : \text{variable} \} \not= v_{al,u,e} \text{ then } ^{\cdot} y \\
& \text{else if } (y = 0) \text{ then } v_{al,u,e} \text{ else } (\text{index of } x \text{ in } *, (y - 1))))).
\end{align*}
\]
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 \( v_{arr} \) is the scope of a
variable with the name \( v_{arr} \).

The element \((e_1 \text{ is variable})\) is defined by the rule

\[
\text{rule} \ (x \text{ is variable}) \  \text{var} \ (x) \ \text{then} \ ((\text{index of } x) \neq v_{al.u.e})
\]

The element \( v_{arr} \) is defined by the rule

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

The element \((\text{type of } v_{arr})\) is defined by the rule

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

The element \((e_1 \text{ is embedded-statement})\) is defined by the rule

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

Thus, only variable declarations are not embedded in MPL2.

### 6.3. Statements

In this section, we define the MPL2 statements and their semantics.

The variable declaration is defined by the rule

\[
\text{rule} \ (\text{var } x \ y) \ \text{var} \ (x, y) \ \text{where} \ ((x \text{ is identifier}) \ \text{and} \ (y \text{ is type})
\]
\[
\text{then} \ (\text{if} \ ((\{-2:s_{e.x}, -1:type, 0:x, 1:variable\} \neq v_{al.u.e})
\]
\[
\text{then} \ v_{al.u.e}
\]
\[
\text{else} \ ((\{-2:*e, -1:type, 0:x, 1:variable\} \ := \ ',y)\}))
\]

The assignment statement is defined by the rule

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

The block statement is defined by the rule
\( (\text{rule (block} :: x) \text{ var} (x) \text{ then} \)
\( (\text{seq (enter-block) (seq} :: x) \text{ (exit-block)})) \).

The element (enter-block) specifying the actions executed when \( s_{y,t,c} \)
enters a block is defined by the rule
\( (\text{rule (enter-block) then} (s_{c,s,r} ::= (s_{c,s,r} + 1))) \).

The element (exit-block) specifying the actions executed when \( s_{y,t,c} \)
exits a block is defined by the rule
\( (\text{rule (exit-block) where} \# \text{ catch} w \text{ then} \)
\( (\text{seq (delete-local-variables) (s}_{c,s,r} ::= (s_{c,s,r} - 1)) \text{ (throw} w)) \).

The element (delete-local-variables) specifying deletion of local variables
of the current scope is defined by the rule
\( (\text{rule (delete-local-variables) then} \)
\( \text{ (foreach-match} \{ -2::*s_{c,s,r}, -1::\text{type}, 0::x, 1::\text{variable} \} \text{ var} (x) \)
\( \text{ do} (\text{seq} (\{ -2::*s_{c,s,r}, -1::\text{type}, 0::*x, 1::\text{variable} \} ::=) \)
\( (\{ -2::*s_{c,s,r}, -1::\text{value}, 0::*x, 1::\text{variable} \} ::=))) \).

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 conceptual states of the conceptual states of
MPL\(_3\).

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

In this section, we define the MPL₃ expressions and their semantics.

The element \(\text{void is type}\) specifying that \(\text{void} \in T_y\) is defined by the rules

\[\text{(rule (void is type) then true)}\]

The element \(\text{(index of } i_d)\) is defined by the rules

\[\text{(rule (index of } x) \text{ var } (x) \text{ where } (x \text{ is identifier}) then (index of } x \text{ in } *, l_{e,c,s,r}, *, s_{e,c,s,r})\];
\[\text{(rule } x \text{ is index in } y, z \text{ var } (x, y, z) \text{ then (if } \{-3: y, -2: z, -1: type, 0: x, 1: variable\} !\in v_{al, a,e} \text{ then } \ldots \text{ else (index of } x \text{ in } y, *, (z - 1)))\).

The element \(\text{(e₁ is variable)}\) is defined by the rule

\[\text{(rule } x \text{ is variable) var } (x) \text{ ((index of } x) !\in v_{al, a,e})\).

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

\[t_{y,[s]}: -1: \text{return-type}, 0: f_u, 1: \text{function}) = t_y, \{-2: t_{y,[s]}, -1: \text{parameters}, 0: f_u, 1: \text{function}\} = i_d[s], \{-2: t_{y,[s]}, -1: \text{body}, 0: f_u, 1: \text{function}\} \neq v_{al, a,e}, \text{ and len}(t_{y,[s]}) = \text{len}(i_d[s]).\]

Let \(L_{c,e}\) be a set of call levels represented by the elements of \(N_{al}\). The conceptual \(\{0: \text{call-depth}\}\) 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 \(\{-3: l_{e,c}, -2: s_c, -1: \text{type}, 0: v_{ar}, 1: \text{variable}\}\) 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 \([l_{e,c}, s_c, s_{al}]\) if \(s_{al}(\{-3: l_{e,c}, -2: s_c, -1: \text{type}, 0: v_{ar}, 1: \text{variable}\}) = t_y\). The following property holds for MPL₃: if \(s_{al}(\{-3: l_{e,c}, -2: s_c, -1: \text{type}, 0: v_{ar}, 1: \text{variable}\}) \neq v_{al, a,e}\), then \(s_{al}(\{-3: l_{e,c}, -2: s_c, -1: \text{type}, 0: v_{ar}, 1: \text{variable}\}) \in T_y\). The conceptual \(\{-3: l_{e,c}, -2: s_c, -1: \text{value}, 0: v_{ar}, 1: \text{variable}\}\) specifies the value of \(v_{ar}\) in \([l_{e,c}, s_c]\). The variable \(v_{ar}\) has the value \(v_{ar}\) in \([l_{e,c}, s_c, s_{al}]\) if \(s_{al}(\{-3: l_{e,c}, -2: s_c, -1: \text{value}, 0: v_{ar}, 1: \text{variable}\}) = v_{al}\). The type and value of the global variable \(v_{ar}\) in \([l_{e,c}, s_c, s_{al}]\) are specified by the conceptual rules \(\{-3: 0, -2: 0, -1: \text{type}, 0: v_{ar}, 1: \text{variable}\}\) and \(\{-3: 0, -2: 0, -1: \text{value}, 0: v_{ar}, 1: \text{variable}\}\), respectively.

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

The exceptions \(\{-1: \text{return}, 0: v_{al}, 1: \text{exception}\}\) and \(\{-1: \text{return}, 1: \text{exception}\}\) specify the execution of the return statement with the return value \(v_{al}\) and without the return value, respectively.
(rule \( x \) \( \text{var} \) \( (x) \) \( \text{hvar} \) \( (w) \) then (seq \( (w := \text{index of } x) \))
(\( \text{if} \ (w = \text{val:u:e}) \) then \( \text{val:u:e} \)
\( \text{else} \ \{-3:y, -2:*.,w, -1:val, 0:x, 1:variable\}\))).

The element \((\text{type of } \text{var})\) is defined by the rule

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

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

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

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

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

The element \((e_x \text{ is not-admissible-function-exception})\) 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 MPL3 are admissible when a function call exits.
7.3. Statements

In this section, we define MPL3 statements and their semantics.

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

\[
\text{rule (execute-arguments} (x \ldots y)) \var (x, y) \text{ where } (* x)
\text{ then } ((\text{execute-arguments} y) \text{ +} \text{ *}.x));
\]

\[
\text{rule (execute-arguments} () \text{ then } (\text{skip})).
\]

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

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

\[
\text{rule (create-local-variables} () () () \text{ then } (\text{skip})).
\]

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

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

The variable declaration is defined by the rule

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

The assignment is defined by the rules

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

The element \( \text{function} f_u ((i_{d,s}(1) t_y,*(1)), \ldots, (i_{d,s}(n_{a,t,0}) t_y,s(n_{a,t,0})))
\text{t_y b_{od}} \), where \( \text{len}(i_{d,s}) = \text{len}(t_y) = n_{a,t,0} \), specifying the declaration of the function \( f_u \) with the parameters \( i_{d,s} \) of the types \( t_y, * \), the return type \( t_y \), and the body \( b_{od} \) is defined by the rule
The specific conceptual states of MPL 4

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. 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 conceptual states of the conceptual states of MPL 4 are listed below. A conceptual [0:nat, 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
in this section, we define MPL 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-literal) specifying that \( e_l \) is a pointer literal is defined by the rule

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

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

\[(rule \ (x \ is \ literal) \ var \ (x) \ then \ (x \ is \ pointer-literal)).\]

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

\[(rule \ (x \ is \ pointer) \ var \ (x) \ then \ ((x \ is \ pointer-literal) \ and \ ([−1:element-type, \ 0:x, \ 1:pointer] \ eq \ v_{al,u.e})).)\]

The element (* \( e_l \)) specifying the value of \( p_{oa} \), where \( p_{oa} \) 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

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

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

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

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

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

The element (index of \( i_{d} \)) is defined by the rules

\[
\text{(rule (index of x) var (x) where (x is identifier) then (index of x in \(*.l_{c,s,r}, \ *,.s_{c,s,r}\));}
\]

\[
\text{(rule (x is index in y, z) var (x, y, z) then}
\]

\[
\text{(if \( \{-3:y, -2:z, -1:pointer, 0:x, 1:variable\} \neq v_{al.u.e} \) then \* .z}
\]

\[
\text{else (index of x in y, \ *,(z - 1)))).}
\]

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

\[
\text{(rule (& x) var (x) hvar (w) then (seq (w ::= (index of x)))}
\]

\[
\text{(if (w = v_{al.u.e}) then v_{al.u.e}
\text{else \{-3:y, -2:*,w, -1:pointer, 0:x, 1:variable\}})).}
\]

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

\[
\text{(rule x var (x) hvar (w) then (seq (w ::= (index of x)))}
\]

\[
\text{(if (w = v_{al.u.e}) then v_{al.u.e}
\text{else \{-1:value, 0:*,\{-3:y, -2:*,w, -1:pointer, 0:x, 1:variable\},
\text{1:pointer\}})).}
\]

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

\[
\text{(rule x var (x) hvar (w) then (seq (w ::= (index of x)))}
\]

\[
\text{(if (w = v_{al.u.e}) then v_{al.u.e}
\text{else \{-1:element-type,}
\text{0:*,\{-3:y, -2:*,w, -1:pointer, 0:x, 1:variable\},
\text{1:pointer\}})).}
\]
In this section, we define MPL statements and their semantics.

The variable declaration is defined by the rule

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

**8.3. Statements**

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

The variable declaration is defined by the rule

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

The assignment is defined by the rule

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

The element \((* e_{1.1} := e_{1.2})\) specifying the assignment of \(v_{\text{al}}\) to \(p_{\text{oi}}\), where \(p_{\text{oi}}\) and \(v_{\text{al}}\) are the values of \(e_{1.1}\) and \(e_{1.2}\), respectively, is defined by the rule

\[
\text{(rule \((* x := y)\) \(\text{var} (x, y)\) \(((* x, y) \text{ and } (* x \text{ is pointer})\) \text{then if} \ ((*y \text{ is } *{},\{-1:\text{element-type}, 0:*x, 1:\text{pointer}\}) \text{then} ((\{-1:\text{value}, 0:*x, 1:\text{pointer}\} := 'y) \text{ else } v_{\text{al.u.e}}))))\).}
\]

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

\[
\text{(rule \(\text{delete-local-variables}\) \(\text{hvar} (w)\) \(\text{then} \ (\text{seq} \{−3:*l_{\text{c.s.r}}, −2:*s_{\text{c.s.r}}, −1:\text{pointer}, 0:x, 1:\text{variable}\} \text{ var} \ (x) \text{ do} \ (\text{seq} (w := \{-3:*l_{\text{c.s.r}}, −2:*s_{\text{c.s.r}}, −1:\text{pointer}, 0:*x, 1:\text{variable}\} \{(−1:\text{element-type}, 0:*w, 1:\text{pointer}) :=\} \{(−1:\text{value}, 0:*w, 1:\text{pointer}) :=\} \{(−3:*l_{\text{c.s.r}}, −2:*s_{\text{c.s.r}}, −1:\text{pointer}, 0:*x, 1:\text{variable}\} :=\}))))\).}
\]
The element \((\text{delete-pointer } e)\) deleting the pointer \(p_{oi}\), where \(p_{oi}\) is the value of \(e\), is defined by the rule

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

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 conceptual states of the conceptual states of MPL\(_5\).

The exception \([\text{--1:break, 1:exception}]\) specifies the execution of the break statement in \([l_{e,c}]\).

The exception \([\text{--1:continue, 1:exception}]\) specifies the execution of the continue statement in \([l_{e,c}]\).

The exception \([\text{--1:goto, 0:l}_{ab}, 1:exception]\) specifies the execution of the goto statement with the label \(l_{ab}\) in \([l_{e,c}]\).

The conceptual \(\{\{-2:l_{e,c}, \text{--1:s}_{c}, 0:l_{ab}, 1:\text{label}\}\}\) 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 \((e_x \text{ is not-admissible-function-exception})\) is defined by the rule

\[
\text{(rule } (x \text{ is not-admissible-function-exception}) \text{ var } (x) \text{ where } (# x)
\]
then \(((x \text{ is exception}) \text{ and }
\)
\(((*(x .. --1)) = (break)) \text{ or } ((*(x .. --1)) = (continue)) \text{ or }
\)
\(((x .. --1) = (goto)))))\).

The element \((e_l \text{ is embedded-statement})\) is defined by the rule

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

Thus, only variable declarations and label statements are not embedded in MPL\(_5\).
9.3. Statements

In this section, we define MPL\textsubscript{5} statements and their semantics.

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

\[
\text{(rule} (\text{label } x) \var x \text{ where } (\# \text{ and } x \text{ is identifier}) \text{ catch } w \text{ then} \begin{align*}
&\{ -2: . l_{c,s,r}, -1: . s_{c,s,r}, 0: x, 1: \text{label} \} \overset{\text{true}}{=} \\
&(\text{if} \ ((w \cdot 1) = ','.exception) \text{ and } ((w \cdot -1) = ','.goto) \text{ and} \\
&( (w \cdot 0) = ','.x)) \text{ then} (w \overset{\text{true}}{=}) \\
&(\text{throw } w)).
\end{align*}
\]

The element \((\text{break})\) specifying the break statement is defined by the rule

\[
\text{(rule} (\text{break}) \text{ then} \begin{align*}
&\begin{align*}
&\{ -1: \text{break}, 1: \text{exception} \}.
\end{align*}
\end{align*}
\]

The element \((\text{continue})\) specifying the continue statement is defined by the rule

\[
\text{(rule} (\text{continue}) \text{ then} \begin{align*}
&\begin{align*}
&\{ -1: \text{continue}, 1: \text{exception} \}.
\end{align*}
\end{align*}
\]

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

\[
\text{(rule} (\text{goto } x) \var x \text{ where } x \text{ is identifier} \text{ then} \begin{align*}
&\{ -1: \text{goto}, 0: x, 1: \text{exception} \}.
\end{align*}
\]

The while statement is defined by the rules

\[
\text{(rule} (\text{while-}\mathit{m} \ x \ \text{do } y) \var x, y \text{ then} \begin{align*}
&(\text{seq} (\text{while1 } x \ \text{do } y) \text{ (delete-exception break)})); \\
&(\text{rule} (\text{while1 } x \ \text{do } y) \var x, y \text{ then} \\
&(\text{if} x \text{ then} \begin{align*}
&(\text{seq } y \text{ (delete-exception continue) (while1 } x \ \text{do } y)))).
\end{align*}
\]

The block statement is defined by the rules

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

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

\[
\text{(rule} (\text{exit-block})) \text{ then} \begin{align*}
&\begin{align*}
&\{ -2: . l_{c,s,r}, -1: . s_{c,s,r}, 0: . (w \cdot 0), 1: \text{label} \} \overset{\text{true}}{=} \\
&(\text{throw } w)).
\end{align*}
\end{align*}
\]

\[
\text{Operational semantics development for procedural programming languages ... 19}
\]
The element (delete-labels) deleting the labels of the label statements in $\llbracket e.c.s.r, s.c.s.r \rrbracket$ is defined by the rule

\begin{equation}
\text{(rule (delete-labels) then)}
\end{equation}

\begin{equation}
\text{foreach-match}\{-2:* e.c.s.r, -1:* s.c.s.r, 0:* x, 1:* label\}
\end{equation}

\text{var (x) do \{(-2:* 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 $[0:n_{al}, 1:array]$ 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 $s_{ta}(-1:element-type, 0:a_{rr}, 1:array) \neq v_{al.u.e}$, then $s_{ta}(-1:element-type, 0:a_{rr}, 1:array) \in T_y$. The conceptual $(-1:n_{al}, 0:a_{rr}, 1:array)$ specifies the value of the $n_{al}$th element of $a_{rr}$. The value $v_{al}$ of the type $t_y$ is the value of the $n_{al}$th element of $a_{rr}$ of the type $[array$ of $t_y]$ in $s_{ta}$ if $v_{al} = s_{ta}(-1:n_{al}, 0:a_{rr}, 1:array)$ 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 $s_{ta}\{0:t_{y,s}, 1:structure-type\} \neq v_{al.u.e}$. 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 $s_{ta}\{\{-1:f_i, 0:t_{y,s}, 1:structure-type\} \neq v_{al.u.e}$, then $s_{ta}\{\{-1:f_i, 0:t_{y,s}, 1:structure-type\} \in T_y$. An element of the form $[0:n_{al}, 1:structure]$ 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 $s_{ta}\{\{-1:type, 0:s_{tr}, 1:structure\} \neq v_{al.u.e}$, then $s_{ta}\{\{-1:type, 0:s_{tr}, 1:structure\} \in T_y$. 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 $t_y$
Operational semantics development for procedural programming languages ...

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 expressions and their semantics.

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

$(\text{rule (array of } x) \text{ is type}) \text{ var } (x) \text{ then } (x \text{ is type})$.

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

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

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

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

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

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

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

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

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

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

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

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

The element \((t_{y,s} \text{ is structure-type})\) specifying that \(t_{y,s}\) is a structure type is defined by the rule

\[
\text{(rule } (x \text{ is structure-type}) \text{ var } (x) \text{ then } ((x \text{ is identifier}) \text{ and } \{0:x, 1:structure-type\} \neq v_{al.u.e}))\].

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

\[
\text{(rule } (x \text{ is type}) \text{ var } (x, y) \text{ then } (x \text{ is structure-type}))\].

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

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

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

\[
\text{(rule } (x \text{ is literal}) \text{ var } (x) \text{ then } (x \text{ is array-literal});)
\]
\[
\text{(rule } (x \text{ is literal}) \text{ var } (x) \text{ then } (x \text{ is structure-literal}))\].

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

\[
\text{(rule } (x \text{ is structure}) \text{ var } (x) \text{ then } ((x \text{ is structure-literal}) \text{ and } \{−1:1:structure, 0:x, 1:structure\} \neq v_{al.u.e}))\].

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

\[
\text{(rule } (type of x) \text{ var } (x) \text{ where } (x \text{ is structure}) \text{ then } \{−1:1:structure, 0:x, 1:structure\})\].

The element \((e_l \text{ is } t_{y,s})\) specifying that \(e_l\) is a structure of the type \(t_{y,s}\) is defined by the rule

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

The element \((e_l \text{ ."} f_i)\) 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

\[
\text{(rule } (x \text{ ."} y) \text{ var } (x, y) \text{ where } ((x) \text{ and } (x \text{ is structure})) \text{ then } (\text{if } \{−1:y, 0:*:\{−1:type, 0:x, 1:structure\}, 1:structure-type\} \neq v_{al.u.e})
\]
\[
\text{then } \{−1:y, 0:*:x, 1:structure\} \text{ else } v_{al.u.e})\).}
10.3. Statements

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

The element \((\text{struct } t_{y,s} \quad ((f_{i,s}(1) \quad t_{y,s}(1)) \; \ldots \; (f_{i,s}(n_{at}) \quad t_{y,s}(n_{at})))))\),

where \(len(f_{i,s}) = len(t_{y,s}) = n_{at}\), specifying the declaration of the structure

\(t_{y,s}\) with the fields \(f_{i,s}\) of the types \(t_{y,s}\) is defined by the rules

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

The element \(\langle e_{l,1} \; "[" \; e_{l,2} \; "]\rangle := e_{l,3}\) 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

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

The element \(\langle e_{l,1} \; ";" \; f_i \; := \; e_{l,2}\rangle\) 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

\[
\begin{align*}
\text{(rule } \; x \; ";" \; y \; := \; z) \; \text{var} \; (x, \; y, \; z) \\
\text{where} \; ((\star \; x) \; \text{and} \; (\star \; x \; \text{is structure}) \\
\text{hvar} \; (t, \; w) \; \text{then (seq} \\
\text{if} \; (\; t := \; \{-1:y, \; 0:␲\{-1:type, \; 0:␡\; x, \; 1:structure}, \; 1:structure-type\}) \\
\text{if} \; (\text{when} \; (\; \text{val}\; (\; \text{var}\; (\; t) \; \text{new}) \; \text{else (seq} \; (w := \; z)) \\
\text{if} \; (\star \; w \; = \; \star \; t) \\
\text{then } (\; \{-1:y, \; 0:␡\; x, \; 1:structure\} \; ::= \; w) \; \text{else } v_{al,u,e})))}))
\end{align*}
\]

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 conceptual states 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 \(t_{y,[s]}\) if \(s_{ta}\{(-1: argument-types, 0:f_u, 1: function)\} = t_{y,[s]}\). The conceptual \(\{(-1: return-type, 0:f_u, 1: function)\}\) specifies the return type of \(f_u\). A type \(t_y\) is the return type of \(f_u\) in \([s_{ta}]\) if \(s_{ta}\{(-1: return-type, 0:f_u, 1: function)\} = t_{y}\). A function \(f_u\) with the return type \(t_y\) in \([s_{ta}]\) is a procedure in \([s_{ta}]\) if \(t_y = \text{void}\). The conceptual \(\{(-1: parameters, 0:f_u, 1: function)\}\) specifies the parameters of \(f_u\). The function \(f_u\) has the parameters \(i_{d,[s]}\) in \([s_{ta}]\) if \(s_{ta}\{(-1: parameters, 0:f_u, 1: function)\} = i_{d,[s]}\). 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 MPL7: if \(s_{ta}\{(-1: argument-types, 0:f_u, 1: function)\} \neq v_{al,u,e}\), then there exist \(t_{y,[s]}, t_{y}\) and \(i_{d,[s]}\) such that \(s_{ta}\{(-1: argument-types, 0:f_u, 1: function)\} = t_{y,[s]}, s_{ta}\{(-1: return-type, 0:f_u, 1: function)\} = t_{y}, \{(-1: parameters, 0:f_u, 1: function)\} = i_{d,[s]}, \{(-1: body, 0:f_u, 1: function)\} \neq v_{al,u,e},\) and \(\text{len}(t_{y,[s]}) = \text{len}(i_{d,[s]})\). The function \(f_u\) has the type \([\text{function of } t_{y,[s]} \longrightarrow t_{y} \text{ in } [s_{ta}]\) if \(s_{ta}\{(-1: argument-types, 0:f_u, 1: function)\} = t_{y,[s]}, \) and \(s_{ta}\{(-1: return-type, 0:f_u, 1: function)\} = t_{y}\).

The conceptual \(\{(-1:t_{y,[s]}, 0:i_{d}, 1: function-name)\}\) specifies the function with the parameter types \(t_{y,[s]}\) represented by the name \(i_{d}\). The identifier \(i_{d}\) represents the function \(f_u\) with the argument types \(t_{y,[s]}\) in \([s_{ta}]\) if \(s_{ta}\{(-1:t_{y,[s]}, 0:i_{d}, 1: function-name)\} = f_u\), and \(f_u\) is a function with the argument types \(t_{y,[s]}\) in \([s_{ta}]\). Let \(N_{am,f}\) be a set of function names. The following property holds for MPL7: if \(s_{ta}\{(-1:t_{y,[s]}, 0:i_{d}, 1: function-name)\} \neq v_{al,u,e}\), then \(s_{ta}\{(-1:t_{y,[s]}, 0:i_{d}, 1: function-name)\}\) is a function with the argument types \(t_{y,[s]}\) in \([s_{ta}]\).

11.2. Expressions

In this section, we define the MPL7 expressions and their semantics.

The element \((\text{function of } t_{y,[s]} \longrightarrow t_{y})\) is type specifying that \([\text{function of } t_{y,[s]} \longrightarrow t_{y}] \in T_y\) is defined by the rule

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

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

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

\[(\text{rule} \ ([] \text{ is type-sequence}) \ \text{then} \ \text{true}).\]

The element \((e_l \text{ is function-literal})\) specifying that \(e_l\) is a function literal is defined by the rule
(rule ([0:x, 1:function] is function-literal) var (x)
then (x is nat)).

The element (e_l is literal) specifying that e_l ∈ L is defined by the rules

(rule (x is literal) var (x) then (x is function-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-literal) and
((−1:argument-types, 0:x, 1:function} ! = 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-types, 0:x, 1:function}
    *.{−1:return-type, 0:x, 1:function}]).

The element (e_l is [function of t_{y,[s]} t_{y}]) specifying that e_l is a function of the type [function of t_{y,[s]} 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-types, 0:x, 1:function]} and
(∗.z = {[−1:return-type, 0:x, 1:function]})).

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

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

Operational semantics development for procedural programming languages ... 25
(s_{c.s,r} := s)
(if (\(w \ldots -1\) = 'return) then (w := (w \ldots 0)))
(throw w))
else v_{al,a,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 statements and their semantics.

The function declaration is defined by the rule

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

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

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

The assignment \((\text{value of } i_d \text{ in } t_y)\) is defined by the rule

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

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


