Operational conceptual transition systems and their application to development of conceptual operational semantics of programming languages

Anureev I.S. (Institute of Informatics Systems)

In the paper the notion of the conceptual operational semantics of a programming language is proposed. This formalism represents operational semantics of a programming language in terms of its conceptual model based on conceptual transition systems. The special kind of conceptual transition systems, operational conceptual transition systems, oriented to specification of conceptual operational semantics of programming languages is defined, the extension of the language of conceptual transition systems CTSL for operational conceptual transition systems is described, and the technique of the use of the extended CTSL as a domain-specific language of specification of conceptual operational semantics of programming languages is proposed. The conceptual operational semantics for the family of sample programming languages illustrate this technique.

Ключевые слова: operational semantics, conceptual transition system, programming language, conceptual model, domain-specific language, conceptual operational semantics

1. Introduction

This paper relates to the development of operational semantics of programming languages. Following [1], we distinguish two parts of the operational semantics of a programming language. The structural part defines how the elements of the language relate to runtime elements that an abstract machine of the programming language can use at runtime. The structural part is called instantiation semantics or structure-only semantics [2]. The dynamic part describes the actual state changes that take place at runtime.

The notion of the conceptual model of a programming language is proposed in [3]. This formalism describes the instantiation semantics at the conceptual level. The conceptual model is specified in terms of conceptual transition systems (CTSs) in the language of conceptual transition systems CTSL [3].

* Partially supported by RFBR under grants 15-01-05974 and 17-07-01600 and SB RAS interdisciplinary integration project No.15/10.
In this paper, we introduce the notion of the conceptual operational semantics of a programming language. This formalism describes the operational semantics of a programming language in terms of its conceptual model. The dynamic part of the operational semantics is defined in terms of the special kind of CTS, operational CTSs, in the extension of CTSL for operational CTSs. Thus, CTSL acts as a domain-specific language oriented to specification of conceptual operational semantics of programming languages.

The paper has the following structure. The preliminary concepts and notation are given in section 2. The concepts and definitions related to the pattern matching which operational CTSs are based on is given in section 3. The operational CTSs is defined in section 4. The extension of the CTSL language for operational CTSs is described in section 5. Semantics of basic executable elements in CTSL is defined in section 6. The definition of the conceptual operational semantics of a programming language is introduced, and the technique of development of conceptual operational semantics of programming languages is illustrated by the sample programming language examples in section 7.

2. Preliminaries

The preliminary concepts and notation are given in this section.

2.1. Sets and sequences

Let $w$, $w1$, $w2$, ... denote elements of the sort $w$, where $w$ is a word, and $$$w$ denote the set of all elements of the sort $w$. For example, if $n$ is a sort of natural numbers, then $n$, $n1$, ... are natural numbers, and $$$n$ is the set of all natural numbers.

Let $$$o$ and $$$set$ be sets of objects and sets considered in this paper. Let $$$i$, $$$n$, and $$$bo$ be sets of integers, natural numbers (with zero), and boolean values true and false.

Let $$$se$ denote the set of finite sequences of the form $o1$ ... $o$n. Let $$$w*$ denote the set of finite sequences of the form $w1$ ... $w$n, and $w*$, $w*1$, $w*2$, and so on denote the elements of the set $$$w*$. Let [es] denote the empty sequence. Let $$$w+$ denote the set of finite nonempty sequences of the form $w1$ ... $w$n, and $w+$, $w+1$, $w+2$, and so on denote the elements of the set $$$w+$.

Let [repeat $o$ $n$] denote the sequence consisting of $n$-th occurrences of the object $o$.

Let [$o$ $\in$ $se$] and [se1 $\subseteq$ se2] denote $o$ $\in$ {$se$} and {se1} $\subseteq$ {se2}. Let [len $se$] denote the length of $se$. Let und denote the undefined value. Let [se .. $n$] denote the $n$-th element of $se$. If [len $se$] $<$ $n$, then [se .. $n$] = und. Let [se .. $n$ := $o$] denote the result $se1$ of
replacement of $n$-th element in $se$ by $o$. If $n > \lfloor \text{len } se \rfloor$, then $se1 = se \text{ repeat und } (\lfloor \text{len } se \rfloor - n - 1) \cdot o$.

Let $[so \in se]$ and $[se1 \subseteq se2]$ denote $so \in \{se\}$ and $\{se1\} \subseteq \{se2\}$. Let $\lfloor \text{len } se \rfloor$ denote the length of $se$. Let $\text{und}$ denote the undefined value. Let $[se .. n]$ denote the $n$-th element of $se$. If $\lfloor \text{len } se \rfloor < n$, then $[se .. n] = \text{und}$. Let $[se .. n := o]$ denote the result $se1$ of replacement of $n$-th element in $se$ by $o$. If $n = \lfloor \text{len } se \rfloor + 1$, then $se1 = se \cdot o$. If $n > \lfloor \text{len } se \rfloor + 1$, then $se1 = \text{und}$.

Let $[so1 < [se]] so2$ denote the fact that there exist $so^1, so^2$ and $so^3$ such that $se = so^1 \cdot so1 \cdot so^2 \cdot so^2 \cdot so^3$.

Let $[so \leftrightarrow so1]$ denote the result of replacement of all occurrences of $so1$ in $so$ by $so2$. Let $[se \leftrightarrow so1]$ denote the result of replacement of each element $so2$ in $se$ by $[so \leftrightarrow so1]$. For example, $[a b x \leftrightarrow (f x)]$ denotes $(f a)(f b)$.

Let $so, so2 \in $$se \cup $$set$. Then $[so1 = set so2]$ denote that the sets of elements of $so1$ and $so2$ coincide, and $[so1 = set so2]$ denote that the multisets of elements of $so1$ and $so2$ coincide.

The above defined operations on the set $se$ are also applied to the set $\{se \mid se \in se\}$. The results of $\lfloor (se .. n) \rfloor$, $[so \in (se)]$, $[se1 \subseteq (se2)]$, $[so1 < [se]] so2$, $\lfloor (se) so \leftrightarrow so1 \rfloor$, $\lfloor \text{len } (se) \rfloor$, $\lfloor (se .. n := so) \rfloor$ and $\text{and } (se)$ are $[se .. n]$, $[so \in se]$, $[se1 \subseteq se2]$, $[so1 < [se] so2]$, $[se so \leftrightarrow so1]$, $\lfloor \text{len } se \rfloor$, $\lfloor se .. n := so \rfloor$ and $\text{and } se$.

Let $[(o^*) + (so^1)]$, $[so \cdot + (o^*)]$ and $[(o^*) + so]$ denote $(o^* so^1), (o^* so^1)$ and $(o^* so)$.

2.2. Contexts

The terms used in the paper can be context-dependent. A context has the form $[so^*]$. The elements of $so^*$ are called embedded contexts. 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 $so[so^*]$ denote the object $so$ in the context $[so^*]$. The expression ‘in $[so1, so^*]$’ can be rewritten as ‘in $[so1]$ in $[so^*]$’, if this does not lead to ambiguity.

2.3. Functions

Let $f o a$ be a set of functions. Let $a$ and $v$ be sets of objects called arguments and values. Let $[f a^*]$ denote the result of application of $f$ to $a$. Let $[\text{support } f]$ denote the support in $[f]$, i.e. $[\text{support } f] = \{a \mid [f a] \neq \text{und}\}$. Let $[image f set]$ denote the image in $[f, set]$, i.e. $[image f set] = \{[f a] : a \in set\}$. Let $[image f f]$ denote the image in $[f, [\text{support } f]]$. 
Let $\text{narrow } f \text{ } set$ denote the function $f_1$ such that $\text{support } f_1 = [\text{support } f] \cap \text{set}$, and $[f_1 \text{ } a] = [f \text{ } a]$ for each $a \in [\text{support } f_1]$. The function $f_1$ is called a narrowing of $f$ to $\text{set}$. Let $[\text{support } f_1] \cap [\text{support } f_2] = \emptyset$. Let $f_1 \cup f_2$ denote the union of $f_1$ and $f_2$ such that $[f \text{ } a] = [f_1 \text{ } a]$ for each $a \in [\text{support } f_1]$, and $[f \text{ } a] = [f_2 \text{ } a]$ for each $a \in [\text{support } f_2]$. Let $f_1 \subseteq f_2$ denote the fact that $[\text{support } f_1] \subseteq [\text{support } f_2]$, and $[f_1 \text{ } a] = [f_2 \text{ } a]$ for each $a \in [\text{support } f_1]$.

An object $u$ of the form $a := v$ is called an update. The objects $a$ and $v$ are called an argument and values in $\langle u \rangle$. Let $\langle u \rangle$ be a set of updates.

Let $[f \langle u \rangle]$ denote the function $f_1$ such that $[f_1 \text{ } a] = [f \text{ } a]$ if $a \neq a[\langle u \rangle]$, and $[f_1 \text{ } a[\langle u \rangle]] = v[\langle u \rangle]$. Let $[f \langle u \rangle u^*]$ be a shortcut for $[\langle f \rangle \langle u \rangle, u^*]$. Let $[f \text{ } a, \text{ } a1, \ldots, a_n := v]$ be a shortcut for $[f \text{ } a := [[f \text{ } a] \text{ } a1, \ldots, a_n := v]]$. Let $[u^*]$ be a shortcut for $[f \langle u \rangle^*]$, where $[\text{support } f] = \emptyset$.

Let $[\text{if } con \text{ then } o_1 \text{ else } o_2]$ denote the object $o$ such that $o = o_1$ for $con = true$, and $o = o_2$ for $con = false$.

### 3. Pattern matching

General CTSs only defines the state structure. The special kinds of CTSs also refine the structure of the transition relation. The refinement of operational CTSs is based on the pattern matching on the state structure.

A function $\langle u \rangle \in \langle cs \rangle \rightarrow \langle cs \rangle^*$ is called a substituton. Let $\langle su \rangle$ be a set of substitutions. A function $\text{sub} \in \langle su \rangle \times \langle cs \rangle^* \rightarrow \langle cs \rangle^*$ is a substitution function if it is defined by the following rules (the first proper rule is applied):

- if $cs \in \langle \text{support } su \rangle$, then $[\text{sub } su \text{ } cs] = [su \text{ } cs]$;
- $[\text{sub } su \text{ } sato] = sato$;
- $[\text{sub } su \text{ } cs; \{t^+\}] = [\text{sub } su \text{ } cs]; ([\text{sub } su \text{ } t^+])$;
- $[\text{sub } su \text{ } cs; :\{t^+\}] = [\text{sub } su \text{ } cs]; ([\text{sub } su \text{ } t^+])$;
- $[\text{sub } su \text{ } (cs^*]) = ([\text{sub } su \text{ } cs^*]);$
- $[\text{sub } su \text{ } cs^*] = [cs^* \leftrightarrow [su \text{ } e]]$.

A structure $p$ is a pattern in $\langle cs, su \rangle$ if $[\text{sub } su \text{ } p] = cs$. A structure $p$ is a pattern in $\langle cs, su \rangle$ if $p$ is a pattern in $\langle cs, su \rangle$ for some $su$. Let $\langle p \rangle$ be a set of patterns. A structure $in$ is an instance in $\langle p, su \rangle$ if $[\text{sub } su \text{ } p] = in$. A structure $in$ is an instance in $\langle p \rangle$ if $in$ is an instance in $\langle p \rangle$ for some $su$. Let $\langle in \rangle$ be a set of instances.
A structure $cs1$ is weakly equal to a structure $cs2$ ($cs1 =_w cs2$) if the following properties hold:

- if $cs1 \in $$$ato$, then $cs2 \in $$$ato$, and $cs1 = cs2$;
- if $cs1 = v1::\{t+1\}$, then $cs2 = v2::\{t+2\}$, $v1 =_w v2$, and $\{t+1\} =_w \{t+2\}$ for some $va2$ and $t+2$;
- if $cs1 = v1::\{t+1\}$, then $cs2 = v2::\{t+2\}$, $v1 =_w v2$, and $\{t+1\} =_w \{t+2\}$ for some $va2$ and $t+2$;
- if $cs1 \in $$$cs$, then $cs2 \in $$$cs$, $\{len\ cs1\} = [len\ cs2]$, and $\{cs1..\ n\} =_w cs2$.. $n\} for each $1 \leq n \leq [len\ cs1]$.

A structure set $set1$ is weakly equal to a structure set $set2$ ($set1 =_w set2$) if the following properties hold:

- if $set1 = \emptyset$, then $set2 = \emptyset$;
- if $cs1 \in set1$, then there exists $cs2 \in set2$ such that $cs1 =_w cs2$, and $set1 \setminus \{cs1\} =_w set2 \setminus \{cs2\}$.

An element $e$ is linear in $\[e^*\]$ if $e$ occurs in $e$ exactly once for each element $e1$ of $e^*$. An element $ps$ of the form $(p (va^*) (sv^*))$ is a pattern specification if the elements of the sequence $va^* sv^*$ are pairwise distinct, and $p$ is linear in $\[va^* sv^*\]$. The elements $p$, $(va^*)$, and $(sv^*)$ are called a pattern, state variable specification and sequence variable specification in $\[ps\]$. The elements of $va^*$ and $sv^*$ are called state variables and sequence variables in $\[ps\]$. Let $\[ps\]$ be a set of pattern specifications. Let $\[va\]$ and $\[sv\]$ be sets of state and sequence variables.

A structure $in$ is an instance in $\[ps, su\]$ if $\[support\ su\] = $va^*$ $\cup$ $sv^*$, $\[su\ va\] \in $$s$ for $va \in $va^*$, $\[su\ sv\] \in $$s$ for $sv \in $sv^*$, and $in = w [sub\ p[ps]\ su]$. A structure $in$ is an instance in $\[ps\]$ if $in$ is an instance in $\[ps, su\]$ for some $su$.

A function $mt \in $$s \times $$ps \rightarrow $$su$ is a matching tactic if $\{mt \in $$su\}$ implies that $s$ is an instance in $\[ps, su\]$. A structure $in$ is an instance in $\[ps, mt, su\]$ if $\{mt \in $$in\ in\ su\} = $$su$. A structure $in$ is an instance in $\[ps, mt\]$ if $in$ is an instance in $\[ps, mt, su\]$ for some $su$.

A substitution $su$ is a matching result in $\[ps, mt, in\]$ if $in$ is an instance in $\[ps, mt, su\]$. A substitution $su$ is a matching result in $\[ps, mt\]$ if $su$ is a matching result in $\[ps, mt, in\]$ for some $in$. A value $v$ is a matching result in $\[v, ps, mt, su, in\]$ if $in$ is an instance in $\[ps, mt, su\]$, and $v = [su \ va]$. A value $v$ is a matching result in $\[v, ps, mt, in\]$ if $v$ is a matching result in $\[v, ps, mt, su, in\]$ for some $su$. Let $\mr$ be a set of matching results.
4. Operational conceptual transition systems

Operational CTSs (OCTSs) are the special kind of CTSs used to describe operational semantics of program systems and programming languages. Let $\mathcal{OCTS}$ be a set of OCTSs.

The structure of $\mathcal{OCTS}$ is based on program transition relations, program transition rules and atomic transition relations. Program transition relations and program transition rules include a specification of a pattern and their application is based on the matching of the first element of a program with the pattern. Atomic transition relations are applied in the case of the empty program.

An object $\text{ptr}$ of the form $(\text{ps}, f)$ is a program transition relation in $\mathcal{OCTS}$ if $f \in \{\text{su} \cup \{\text{cstate}: \text{ss}, \text{cvalue}: v[ss]\} \mid \text{su} \in \text{mr}[\text{ps}, \text{mt}], \text{ss} \in \text{ss}\} \rightarrow \text{tr}$. Thus, $\text{ptr}$ specifies a parametric transition relation, where the values of the parameter are the results of the pattern matching. The special elements $\text{cstate}$ and $\text{cvalue}$ refer to the current state and the value in the current state. Let $\mathcal{PTR}$ be a set of program transition relation. The objects $(\text{p} \parallel \text{ps}), (\text{va} \parallel \text{ps}), (\text{sv} \parallel \text{ps})$ and $f$ are called a pattern, state variable specification, sequence variable specification and value in $\mathcal{PTR}$. The elements of $\text{va}$ and $\text{sv}$ are called state and sequence variables in $\mathcal{PTR}$. If $\text{su}$ is the result of the first element $e$ of the program $\text{p} \parallel \text{ss}$ with the pattern of $\text{ptr}$, then a transition from $\text{ss}$ to $\text{ss1}$ initiated by $\text{ptr}$ and denoted by $\text{ss} \Leftarrow \text{ptr} \parallel \text{su} \parallel \text{ss1}$ is defined as $(\text{ss}, \text{ss1}) \in [\text{f} \parallel \text{su} \cup \{\text{cstate}: \text{ss}, \text{cvalue}: v[ss]\}]$. Let $\text{ss} \Leftarrow \text{ptr} \parallel \text{su} \parallel \text{ss1}$ denote $\text{ss} \Leftarrow \text{ptr} \parallel \text{su} \parallel \text{ss1}$ for some $\text{su}$.

A partial function $\text{ptrs} \in \text{e} \rightarrow \mathcal{PTR}$ is a program transition relation specification if $\text{support} \text{ptrs}$ is finite. A relation $\text{ptr}$ is a relation in $\mathcal{PTR}$ if $\text{ptrs} \parallel \text{na} = \text{ptr}$ for some $\text{na} \in \text{e}$. An element $\text{na}$ is a name in $\mathcal{PTR}$ if $\text{ptrs} \parallel \text{na} = \text{atr}$. An element $\text{na}$ is a name in $\mathcal{PTR}$ if $\text{na}$ is a name in $\mathcal{PTR}$ for some $\text{atr}$. Thus, $\mathcal{PTR}$ defines a finite set of named program transition relations.

An element $r$ of the form $(\text{ps} \parallel b)$ is called a (program) transition rule. The objects $(\text{p} \parallel \text{ps}), (\text{va} \parallel \text{ps}), (\text{sv} \parallel \text{ps})$ and $b$ are called a pattern, state variable specification, sequence variable specification and body in $\mathcal{R}$. The elements of $\text{va}$ and $\text{sv}$ are called state and sequence variables in $\mathcal{R}$. Let $\mathcal{R}$ be a set of transition rules. If $\text{su}$ is the result of the first element $e$ of the program $\text{p} \parallel \text{ss}$ with the pattern of $r$, then a transition from $\text{ss}$ initiated by $r$ replaces $e$ in $\text{p}$ by $\text{sub} \parallel \text{su} \cup \{\text{cstate}: \text{ss}, \text{cvalue}: v[ss]\} \parallel b$.

A structure $\text{rs}$ is a rule specification if $\text{rs} \parallel \{\text{na}\} \in \mathcal{R} \cup \{\text{und}\}$ for each $\text{na} \in \text{e}$. A rule $r$ is a rule in $\mathcal{R}$ if $\text{rs} \parallel \{\text{na}\} = r$ for some $\text{na} \in \text{e}$. An element $\text{na}$ is a name in
A structure \( rs \) is a rule specification in \( [s] \) if \( rs = [s \cdot \{\text{rules}\}] \), and \([rs \cdot \{na\}] \in \{rs \cup \{\text{und}\}\} \) for each \( na \in \{e\}. It specifies the set of named transition rules in \( [s] \).

Let \([\text{support } ptrs] \cap [\text{support } rs] = \emptyset\).

A structure \( ptrs \) of the form \((na^*)\) is a program transition order in \([ptrs, rs]\) if \((na^*) \subseteq [\text{support } ptrs] \cup [\text{support } rs]\), and the elements of \( na^* \) are pairwise distinct. It specifies the order of application of program transition relations and transition rules, i.e. the order of matching the first element of the program with their patterns.

A structure \((na^*)\) is a program transition order in \([s]\) if \((na^*) = [s \cdot \{(\text{program transition order})\}]\), and the elements of \( na^* \) are pairwise distinct. It specifies the order of application of program transition relations and transition rules in \([s]\).

A relation \( atr \in \{s\} \) is called an atomic transition relation. Let \( \{s\} atr \) be a set of atomic transition relations. If \( p[\{s\}] = () \), then a transition from \( s \) to \( s1 \) initiated by \( atr \) and denoted by \( s \leftrightarrow[atr] s1 \) is defined as \( (s, s1) \in atr \).

A partial function \( atrs \in \{s\} \rightarrow \{s\} atr \) is an atomic transition relation specification if \([\text{support } atrs]\) is finite. A relation \( atr \) is a relation in \([\{s\} atrs]\) if \([\{s\} atrs na] = atr \) for some \( na \in \{e\}. An element \( na \) is a name in \([\{s\} atr, \{s\} atrs]\) if \([\{s\} atrs nm] = atr \). An element \( na \) is a name in \([\{s\} atrs]\) if \( na \) is a name in \([\{s\} atr, \{s\} atrs]\) for some \( atr \). Thus, \( atrs \) defines a finite set of named atomic transition relations.

A structure \((na^*)\) is an atomic transition order in \([ptrs]\) if \((na^*) \subseteq [\text{support } atrs]\), and the elements of \( na^* \) are pairwise distinct. It specifies the order of application of atomic transition relations.

A structure \((na^*)\) is an atomic transition order in \([s]\) if \((na^*) = [s \cdot \{(\text{atomic transition order})\}]\), and the elements of \( na^* \) are pairwise distinct. It specifies the order of application of atomic transition relations in \([s]\).

Let \( \{s\} bi \) be a set of elements called backtracking invariants.

Let \( se^\ast \# s\# v \# s \) denote \([s] \text{program}:(se^\ast) \text{value}:v \). Let \( se^\ast \# s \) denote \([s] \text{program}:(se^\ast)\).

A tuple \( octs \) of the form \((ato, \{s\} is, ptrs, rs, pto, atrs, ator, \{s\} bi, \{s\} mt\) is an operational CTS if the system \( cts \) of the form \( \{s\} at, \{s\} is, \rightarrow \) is a CTS with backtracking in \([\{s\} bi]\) and with direct stop, \([\text{support } ptrs] \cap [\text{support } rs] = \emptyset\), \( s \) is consistent for each \( s \in \{rs\}[\rightarrow] \), and \( \rightarrow \) is defined by the following rules (the first proper rule is applied):
• if $\text{ptr} = [\text{ptrs $na$}]$, $e$ is an instance in $\llbracket [\text{ps} [\text{ptr}], \text{mt}, \text{su}] \rrbracket$, and $s \leftarrow [\text{ptr, su}] \# s1$, then
  
  $((\text{execute program element}) \ e \ (\text{na $na^\ast$})) \# s \leftarrow$

  (backtracking $s$ ((execute program element) $e$ ($na^\ast$))) \# $e^\ast \# s1$;

• if $\text{ptr} = [\text{ptrs $na$}]$, and $e$ is not an instance in $\llbracket [\text{ps} [\text{ptr}], \text{mt}] \rrbracket$, then
  
  $((\text{execute program element}) \ e \ (\text{na $na^\ast$})) \# s \leftarrow$

  $((\text{execute program element}) \ e$ ($\text{na}^\ast$)) \# $e^\ast \# s$;

• if ($r = [\text{rs} . \{\text{na}\}]$, or $r = [[\text{s} . \{\text{rules}\}] . \{\text{na}\}]$, and $e$ is an instance in $\llbracket [\text{ps} [\text{r}], \text{mt}, \text{su}] \rrbracket$, then
  
  $((\text{execute program element}) \ e \ (\text{na $na^\ast$})) \# s$ $ightharpoonup [\text{sub $su$ $(\text{cstate}$:$s, \text{cvalue}$:$s) \# b$ $[\text{rs}]$}]

  (backtracking $s$ ((execute program element) $e$ ($\text{na}^\ast$))) \# $e^\ast \# s$;

• if ($r = [\text{rs} . \{\text{na}\}]$, or $r = [[\text{s} . \{\text{rules}\}] . \{\text{na}\}]$, and $e$ is not an instance in $\llbracket [\text{ps} [\text{r}], \text{mt}] \rrbracket$, then
  
  $((\text{execute program element}) \ e \ (\text{na}^\ast)) \# s \leftarrow$

  $((\text{execute program element}) \ e$ ($\text{na}^\ast$)) \# $e^\ast \# s$;

• $((\text{execute program element}) \ e$ ($\text{()}) \# s \leftarrow $ $e^\ast \# s \ # \text{ und} \ # s$;

• if $\text{atr} = [\text{atars $na$}]$, $s \leftarrow [\text{atars}] \# s1$, and $p [[s] 1] \neq ()$, then
  
  $((\text{execute atomic transition}) \ (\text{na} \ (\text{na}^\ast)) \# s \leftarrow s1$;

• if $\text{atr} = [\text{atars $na$}]$, $s \leftarrow [\text{atars}] \# v \ # s1$, and $p [[s] 1] = ()$, then
  
  $((\text{execute atomic transition}) \ (\text{na} \ (\text{na}^\ast)) \# s \leftarrow$

  (backtracking $s$ ((execute atomic transition) ($\text{na}^\ast$))) \# $s1$;

• $((\text{execute atomic transition}) \ () \ # s \leftarrow [\text{s} \ true: \{\text{stop}\}]$;

• $e \ # e^\ast \ # s \leftarrow$

  $((\text{execute program element}) \ e \ [\text{su} \ . \ (\text{program transition order})]) \ # e^\ast \ # s$;

• $\# s \leftarrow ((\text{execute atomic transition}), \ [\text{s} \ . \ (\text{atomic transition order})]) \ # s$.

A state $s$ is consistent in $\llbracket $cts$ \rrbracket$ if the following properties hold:

• the set of bbacktracking invariants in $\llbracket s \rrbracket$ is finite;

• $[\text{support } \text{ptrs}] \cap [\text{support } \text{s . (rules)}] = \emptyset$;

• $[\text{support } \text{rs}] \cap [\text{support } \text{s . (rules)}] = \emptyset$;

• if $\text{na}1 < \llbracket \text{ptrs} \rrbracket \text{na}2$, $\text{na}1 \in \llbracket \text{s} . \ (\text{program transition order}) \rrbracket$, and $\text{na}2 \in$

  $\llbracket \text{s} . \ (\text{program transition order}) \rrbracket$, then $\text{na}1 > \llbracket \text{s} . \ (\text{program transition order}) \rrbracket\text{na}2$;
if $na1 <[[sator]] na2$, $na1 \in [s \cdot ((atomic \ transition \ order))]$, and $na2 \in [s \cdot ((atomic \ transition \ order))]$, then $na1 <[[s \cdot ((atomic \ transition \ order))] na2$.

5. The CTSL language

The CTSL language is extended for operational CTSs by adding transition rules and extended transition rules.

The transition rule ((p, (va*), (sv*)), b) with the name $na$ is represented in CTSL[0] by the state (rule $p \ var \ (va^*) \ seq \ (sv^*) \ then \ b$) : {$na$}.

Extended transition rules are transition rules enriched by the mechanisms of evaluation of pattern variable matching results, imposition of constraints on pattern variable matching results and propagation of abnormal values (undefined values and exceptions) from pattern variable matching results and the attribute value.

Let {$eva^*$} $\subseteq \ {va^*}$, the elements of the sequence $eva^*$ are pairwise disjoint, $set = \ {eva :: \ {\ast} | eva \in eva^*}$, $\{va^1\} \cup \{va^2\} \cup \{va^3\} \subseteq \{va^*\} \cup \{set\}$, the elements of the sequence $va^1$ $va^2$ $va^3$ are pairwise disjoint, and $se \in \{[se], \ und, \ exc, \ abn\}$. The state (rule $p \ var \ (va^*) \ seq \ (sv^*) \ val \ (eva^*) \ abn \ (va^1) \ und \ (va^2) \ exc \ (va^3) \ se$ where $co$ then $b$) is called an extended rule. It is defined as follows:

- If $co \neq true$, then
  (rule $p \ var \ (va^*) \ seq \ (sv^*) \ val \ (eva^*) \ und \ (va^1) \ exc \ (va^2) \ abn \ (va^3) \ se$ where $co$ then $b$)
  is a shortcut for
  (rule $p \ var \ (va^*) \ seq \ (sv^*) \ val \ (eva^*) \ und \ (va^1) \ exc \ (va^2) \ abn \ (va^3) \ se$ where true then (if $co$ then $b$ else und)).

- The rule
  (rule $p \ var \ (va^*) \ seq \ (sv^*) \ val \ (eva^*) \ und \ (va^1) \ exc \ (va^2)$
  $abn \ (va^3) \ eva :: \ {\ast}$ $\{va^3, se \ where \ true \ then \ b\}$)
  is a shortcut for
  (rule $p \ var \ (va^*) \ seq \ (sv^*) \ val \ (eva^*) \ und \ (va^1) \ exc \ (va^2)$
  $abn \ (va^3) \ eva :: \ {\ast} \ else \ b$).

- If $\{va^3\} \cap \{set\} = \emptyset$, then
  (rule $p \ var \ (va^*) \ seq \ (sv^*) \ val \ (eva^*) \ und \ (va^1)$

exc ($va^*21 \$eva::\{\} $va^*22) abn ($va^*3) \$se where true then $b

is a shortcut for

(rule $p \text{ var} ($va^*) \text{ seq} ($sv^*) \text{ val} ($seva^*) \text{ und} ($va^*1) \text{ exc} ($va^*21 $va^*22)

abn ($va^*3) \$se where true then (if ($seva::\{\} \text{ is exception}) then $seva::\{\} \text{ else } $b)).

• If ($\{va^*2\} \cup \{va^*3\}) \cap \text{ set} = \emptyset, then

(rule $p \text{ var} ($va^*) \text{ seq} ($sv^*) \text{ val} ($seva^*) \text{ und} ($va^*11 $seva :: \{\} $va^*12)

exc ($va^*2) abn ($va^*3) \$se where true then $b

is a shortcut for

(rule $p \text{ var} ($va^*) \text{ seq} ($sv^*) \text{ val} ($seva^*) \text{ und} ($va^*1 $seva :: \{\} $va^*12)

exc ($va^*2) abn ($va^*3) \$se where true then (if ($seva::\{\} \text{ is undefined}) then $seva::\{\} \text{ else } $b)).

• If ($\{va^*1\} \cup \{va^*2\} \cup \{va^*3\}) \cap \text{ set} = \emptyset, then

(rule $p \text{ var} ($va^*) \text{ seq} ($sv^*) \text{ val} ($seva^* $seva :: \{\}) \text{ und} ($va^*1 $exc ($va^*2)

abn ($va^*3) \$se where true then (let w be $seva in (\text{subst} ($seva :: \{\}:w) $b))),

where w is a new state that does not occur in the initial form.

• If ($\{va^*1\} \cup \{va^*2\} \cup \{va^*31, va, va^*32\}) \cap \text{ set} = \emptyset, then

(rule $p \text{ var} ($va^*) \text{ seq} ($sv^*) \text{ val} () \text{ und} ($va^*1 $exc ($va^*2)

abn ($va^*31 $va $va^*32) \$se where true then $b

is a shortcut for

(rule $p \text{ var} ($va^*) \text{ seq} ($sv^*) \text{ val} () \text{ und} ($va^*1 $exc ($va^*2)

abn ($va^*31 $va $va^*32) \$se where true then (if ($va is abnormal) then $va \text{ else } $b)).

• If ($\{va^*1\} \cup \{va^*21, va, va^*22\}) \cap \text{ set} = \emptyset, then

(rule $p \text{ var} ($va^*) \text{ seq} ($sv^*) \text{ val } () \text{ und} ($va^*1 $exc ($va^*21 $va $va^*22)

abn () \$se where true then $b

is a shortcut for

(rule $p \text{ var} ($va^*) \text{ seq} ($sv^*) \text{ val } () \text{ und} ($va^*1 $exc ($va^*21 $va $va^*22) abn () \$se

where true then (if ($va is exception) then $va \text{ else } $b)).

• If $\{va^*11, va, va^*12\} \cap \text{ set} = \emptyset, then

(rule $p \text{ var} ($va^*) \text{ seq} ($sv^*) \text{ val } () \text{ und} ($va^*11 $va $va^*12 $exc () abn () \$se

where true then $b

is a shortcut for
(rule $p$ var ($va^*$) seq ($sv^*$) val () und ($va^*$11 $va^*$12) exc () abn () $se$
where true then (if ($va$ is undefined) then $va$ else $b$)).

• The rule
  
  (rule $p$ var ($va^*$) seq ($sv^*$) val () und () exc () abn () abn where true then $b$)
  is a shortcut for

  (rule $p$ var ($va^*$) seq ($sv^*$) val () und () exc () abn () where true
  then (if (cvalue is abnormal) then else $b$)).

• The rule

  (rule $p$ var ($va^*$) seq ($sv^*$) val () und () exc () abn () exc where true then $b$)
  is a shortcut for

  (rule $p$ var ($va^*$) seq ($sv^*$) val () und () exc () abn () where true
  then (if (cvalue is exception) then else $b$).

• The rule

  (rule $p$ var ($va^*$) seq ($sv^*$) val () und () exc () abn () und where true then $b$)
  is a shortcut for

  (rule $p$ var ($va^*$) seq ($sv^*$) val () und () exc () abn () where true
  then (if (cvalue is undefined) then else $b$).

A pattern variable $va$ is evaluated if the matching result for $va$ is evaluated. The sequence $eva^*$ contains evaluated pattern variables. The special variable $eva: :[*]$ references to the value of the matching result for $va$. A pattern variable $va$ is quoted if the matching result for $va$ is not evaluated.

The state $co$ imposes of constraints on the values of the variables $va^*$, $sv^*$, $eva: :[*]$.

The undefined value $und$ is propagated through the variables $v^*1$. Exceptions are propagated through the variables $v^*2$. Abnormal values are propagated through the variables $v^*3$.

The sequence $se$ specifies propagation of abnormal values through the attribute value. The undefined value is propagated through the attribute value when $se = und$. Exceptions are propagated through the attribute value when $se = exc$. Abnormal values are propagated through the attribute value when $se = abn$.

The executable elements (if $con$ then $e^*1$ else $e^*2$) and (let $va$ be $e^*1 in $e^*2) are defined in section 6.7. The executable elements ($e$ is abnormal), ($e$ is exception) and ($e$ is undefined) are defined in section 7.4.

Let $$$er$ be a set of extended transition rules.
The objects \texttt{var} ($va^*$), \texttt{seq} ($sv^*$), \texttt{val} ($ev^*$), \texttt{abn} ($va^*1$), \texttt{und} ($va^*2$), \texttt{exc} ($va^*3$) and where \texttt{co} in extended transition rules can be omitted. The omitted objects correspond to \texttt{var} (), \texttt{seq} (), \texttt{val} (), \texttt{abn} (), \texttt{und} (), \texttt{exc} () and where \texttt{true}.

6. Semantics of executable elements in CTSL

To define operational semantics of executable elements in CTSL the special denotations for program and atomic transition relations are introduced.

Let \texttt{(transition \$p\ var\ ($va^*$)\ seq\ ($sv^*$)\ then\ \$f):\{$na\}} denote the program transition relation \texttt{((p, ($va^*), ($sv^*)), \$f)} with the name \$na. The objects \texttt{var} ($va^*$) and \texttt{seq} ($sv^*$) can be omitted. The omitted objects correspond to \texttt{var} () and \texttt{seq} ().

Let \texttt{(atomic\ transition\ \$f):\{$na\}} denotes the atomic transition relation defined by the characteristic function \$f \in \$s \times \$s \rightarrow \$b with the name \$na.

For simplicity, we omit the names of transition relations and transition rules.

6.1. Values

The executable elements handling the transition value are defined in this section.

An element \texttt{$e$} of the form \texttt{$v::\{q\}} is called a quoted element. It is defined as follows:

\( \texttt{(rule \ v::\{q\} \ var\ (v) \ abn \ then\ v::\{q\}::\{transition\});} \)

\( \texttt{(transition \ v::\{q\}::\{transition\} \ var\ (v) \ then\ \$f).} \)

where \texttt{$v::\{q\}::\{transition\};\$e^\# \$s \leftrightarrow_{[$\$f$]} \$e^\# \$v \# \$s.}

The value \texttt{v} is called a quoted value in \$[\$e\$]. The element \texttt{$e$} returns the quoted value \texttt{v}.

The element \texttt{und} is defined by the rule

\( \texttt{(rule \ und\ abn \ then\ und::\{q\}).} \)

The element \texttt{ex} is defined by the rule

\( \texttt{(rule \ v::\{exc\} \ var\ (v) \ abn \ then\ v::\{exc\}::\{q\}).} \)

The element \texttt{($e$ is undefined)} specifies that \$e equals \texttt{und}. It is defined by the rule

\( \texttt{(rule \ (e\ is\ undefined)\ var\ (e)\ abn\ then\ (e::\{q\} = \texttt{und})).} \)

The element \texttt{($e$ is defined)} specifies that \$e does not equal \texttt{und}. It is defined by the rule

\( \texttt{(rule \ (e\ is\ defined)\ var\ (e)\ abn\ then\ (e::\{q\} \neq \texttt{und}).} \)

The element \texttt{($e$ is exception)} specifies that \$e is an exception. It is defined by the rule

\( \texttt{(rule \ (e\ is\ exception)\ var\ (e)\ abn\ then\ (e\ is\ exception)::\{transition\});} \)

\( \texttt{(transition \ (e\ is\ exception)::\{transition\} \ var\ (e) \ then\ \$f),} \)

where
(e is exception) ::= {transition}; $e^* \cdot s \leftarrow [s\} \\
$e^* \# [if [$e \in \$exc \] then true else und] \# s.

The element ($e$ is abnormal) specifies that $e$ is abnormal. It is defined by the rule

(rule ($e$ is abnormal) var $(e)$ abn then ((($e$ is undefined) or ($e$ is exception))));

The element ($e$ is normal) specifying that $e$ is normal. It is defined by the rule

(rule ($e$ is normal) var $(e)$ abn then ((($e$ is defined) and (not ($e$ is exception))));

The element $e$ of the form (catch:::{und} $va$ $e^*$) is called a value handler. It is defined as follows:

(transition (catch:::{und} va e_s) var $(va)$ seq $(e_s)$ then $f$).

where (catch:::{und} $va$ $e^*$); $e^*1 \# $v \# $s \leftarrow [s\} (sub (va: $v$) $e^*$) $e^*1 \# true \# s$.

The elements $va$ and $e^*1$ are called a variable and body in $[e\}$. The element $e$ replaces all occurrences of the variable $va$ in the body $e^*1$ by the current value, resets the current value to true and executes the modified body.

The element $e$ of the form (catch $va$ $e^*$) is called an exception handler. It is defined as follows:

(rule (catch va e_s) var $(va)$ seq $(e_s)$ und then (catch:::{und} va e_s)),

The elements $va$ and $e^*$ are called a variable and body in $[e\}$. If the current value is defined, the element $e$ replaces all occurrences of the variable $va$ in the body $e^*1$ by the current value, resets the current value to true and executes the modified body. It propagates und.

The element (current value) returns the current value. It is defined by the rule

(rule (current value) abn then cvalue:::{q}).

The element ((to value) $e$) replaces the current value to $v$, where $v$ is the value of $e$. It is defined as follows:

(rule ((to value) e) var $(e)$ val $(e)$ then ((to value) e:::{*}):::{transition});

(transition ((to value) $v$):::{transition} var $(v)$ then $f$),

where ((to value) $v$):::{transition}; $e^* \# $s \leftarrow [s\} $e^* \# $v \# $s$.

The element ((catch exception) $t$) catches an exception of the type $t$. It is defined by the rule

(rule ((catch exception) $t$) var $(t)$ und \\
thenn (catch va ( \\
if ((va is exception) and ((va:::{q}. {type}) = $t:::{q})) \\
then ((to value) true) else ((to value) va:::{q}))).

6.2. Integers
The executable elements handling integers are defined in this section.

The element \( (s \text{ is } n \text{ at}) \) specifies that \( s \) is a natural number. It is defined as follows:

\[
\text{(rule } (s \text{ is } n \text{ at}) \text{ var } (e) \text{ abn then } (e \text{ is } n \text{ at}): \{\text{transition}\});
\]

\[
\text{(transition } (e \text{ is } n \text{ at}): \{\text{transition} \} \text{ var } (e) \text{ then } f),
\]

where

\[
(s \text{ is } n \text{ at}): \{\text{transition}\} s^* s \leftrightarrow [s] s^* # [if \ [s] e \in s\{n\] then \text{true} \text{ else und}] \# s.
\]

The element \( (s \text{ is } i \text{ nt}) \) specifies that \( s \) is an integer. It is defined as follows:

\[
\text{(rule } (s \text{ is } i \text{ nt}) \text{ var } (e) \text{ abn then } (e \text{ is } i \text{ nt}): \{\text{transition}\});
\]

\[
\text{(transition } (e \text{ is } i \text{ nt}): \{\text{transition} \} \text{ var } (e) \text{ then } f),
\]

where

\[
(s \text{ is } i \text{ nt}): \{\text{transition}\} s^* # s \leftrightarrow [s] s^* # [if \ [s] e \in s\{i\in\text{n}\] then \text{true} \text{ else und}] \# s.
\]

The element \( i \) is defined by the rules

\[
\text{(rule } i \text{ var } (i) \text{ abn where } (i \text{ is } i \text{ nt}) \text{ then } i: \{\text{q}\}).
\]

If \( s_v1 \) and \( s_v2 \) are values of \( s_1 \) and \( s_2 \), then the element \( (s_1 + s_2) \) returns \( [s_v1 + s_v2] \). It is defined as follows:

\[
\text{(rule } (e_1 + e_2) \text{ var } (e_1, e_2) \text{ val } (e_1, e_2) \text{ abn}
\]

\[
\text{then } (e_1: \{\} +: \{\text{integer}\} e_2: \{\}): \{\text{transition}\});
\]

\[
\text{(transition } (i_1 +: \{\text{integer}\} i_2): \{\text{transition} \} \text{ var } (i_1, i_2) \text{ then } f),
\]

where \( (s_1 +: \{\text{integer}\} s_2): \{\text{transition}\}; s^* # s \leftrightarrow [s] s^* # [s_1 + s_2] \# s.\)

The elements \( (s_1 \text{ op } s_2) \), where \( s_\text{op} \in \{-, *, \text{ div}, \text{ mod}\} \), specifying the integer operations \(-, *, \text{ div} \) and \text{ mod}, are defined in the similar way.

If \( s_v1 \) and \( s_v2 \) are values of \( s_1 \) and \( s_2 \), then the element \( (s_1 < s_2) \) specifies that \( [s_v1 < s_v2] \). It is defined as follows:

\[
\text{(rule } (e_1 < e_2) \text{ var } (e_1, e_2) \text{ val } (e_1, e_2) \text{ abn}
\]

\[
\text{then } (e_1: \{\} <: \{\text{integer}\} e_2: \{\}): \{\text{transition}\});
\]

\[
\text{(rule } (e_1: \{\} <: \{\text{integer}\} e_2: \{\}): \{\text{transition} \} \text{ var } (e_1, e_2) \text{ then } f),
\]

where \( (s_1 <: \{\text{integer}\} s_2): \{\text{transition}\}; s^* # s \leftrightarrow [s] s^* # [s_1 < s_2] \# s.\)

The elements \( (s_1 \text{ op } s_2) \), where \( s_\text{op} \in \{\leq, >, >\} \), specifying the integer relations \( \leq, > \) and \( \geq \), are defined in the similar way.

### 6.3. Boolean values

The executable elements handling boolean values are defined in this section.

The element \text{true} is defined by the rule:
(rule true abn then true::\{q\}).

If $v_1$ and $v_2$ are values of $e_1$ and $e_2$, then the element ($e_1$ and $e_2$) specifies the
conjunction of $v_1$ and $v_2$. It is defined by the rule:

(rule ($e_1$ and $e_2$) var ($e_1$, $e_2$) abn then (if $e_1$ then $e_2$ else und)).

If $v_1$ and $v_2$ are values of $e_1$ and $e_2$, then the elements ($e_1$ $op$ $e_2$), where $op \in \{or, = >, <=>\}$ specifying the disjunction, implication and equivalence of $v_1$ and $v_2$ are defined in the
similar way.

If $v_1$, $v_2$, ..., $v$n are values of $e_1$, $e_2$, ..., $e$n, then the element
($e_1$ and $e_2$ and ... and $e$n) specifies the conjunction of $v_1$, $v_2$, ..., $v$n. It is defined by
the rule

(rule ($e_1$ and $e_2$ and $e\_s$) var ($e_1$, $e_2$) seq ($e\_s$) abn then ((($e_1$ and $e_2$) and $e\_s$).

If $v_1$, $v_2$, ..., $v$n are values of $e_1$, $e_2$, ..., $e$n, then the element
($e_1$ or $e_2$ or ... or $e$n) specifying the disjunction of $v_1$, $v_2$, ..., $v$n is defined in the similar way.

If $v$ is a value of $e$, then the element (not $e$) specifies the negation of $v$. It is defined by the
rule (rule (not $e$) var ($e$) abn then (if $e$ then und else true)).

6.4. Conceptual structures

The executable elements handling conceptual structures are defined in this section.

The element ($e$ is atom) specifies that $e$ is an atom. It is defined as follows:

(rule ($e$ is atom) var ($e$) abn then (e is atom)::\{transition\});

(transition (e is atom)::\{transition\} var ($e$) then $f$),

where

$(e$ is atom)::\{transition\} $e^* \# s \leftarrow [s|f] \ {e}^* \# [if [e \in \$sato] then true else und] \# s.$

The element ($e$ is compound) specifies that $e$ is a compound structure. It is defined by the rule

(rule ((e\_s) is compound) seq (e\_s) abn then true).

The element ($e$ is (absolutely typed)) specifies that $e$ is an absolutely typed structure. It is
defined by the rule

(rule $e::\{t\_s\}$ is (absolutely typed)) var ($e$) seq (e\_s) abn then true).

The element ($e$ is (relatively typed)) specifies that $e$ is a relatively typed structure. It is
defined by the rule

(rule $e::\{t\_s\}$ is (relatively typed)) var ($e$) seq (e\_s) abn then true).

The element ($e$ is empty) specifies that $e$ is an empty structure. It is defined by the rule
(rule () is empty) abn then true).

The element ($e$ is nonempty) specifies that $e$ is not an empty structure. It is defined by the rule
(rule (e is empty) var (e) abn then (not (e is empty))).

The empty structure is defined by the rule
(rule () abn then ()::[q]).

The element ($ccs$ is ($t$ *)) specifies that the value of ($e$ is $t$) does not equal und for each element $e$ of $ccs$. It is defined by the rule
(rule ((e e_s) is ($t$ *)) var (e, t) seq (e_s) abn then ((e is $t$) and ((e_s) is ($t$ *)));
(rule ((t *)) var (t) abn then true).

If $c$ is a value of $e$, then the element ($len e$) specifies the length of $c$. It is defined as follows:
(rule (len e) var (e) val (e) abn then (len e :: [*]) :: {transition});
(transition (len cs) :: {transition} var (cs) then $f$),
where (len $c$s) :: {transition}; $e^* # s \leftrightarrow_{[s]} e^* # [len c]$ # $s$.

If $c$s1 and $c$s2 are values of $e$1 and $e$2, then the element ($e$1 = $e$2) specifies the equality of $c$s1 and $c$s2. It is defined as follows:
(rule ($e$1 = $e$2) var ($e$1, $e$2) val ($e$1, $e$2) abn then ($e$1 :: [*] = $e$2 :: [q]) :: {transition});
(transition ($c$s1 = $c$s2) :: {transition} var ($c$s1, $c$s2) then $f$),
where ($c$s1 = $c$s2) :: {transition}; $e^* # s \leftrightarrow_{[s]} e^* # [c$s1 = $c$s2] # $s$.

If $c$s1 and $c$s2 are values of $e$1 and $e$2, then the element ($e$1 ! $e$2) specifies the inequality of the structures $c$s1 and $c$s2. It is defined by the rule
(rule ($e$1 ! = $e$2) var ($e$1, $e$2) val ($e$1, $e$2) abn then (not ($e$1 = $e$2))).

If $c$ is a value of $e$, then the conceptual structure access operation ($e$ . $mt$) returns [$c$s . $mt$]. It is defined as follows:
(rule ($e$ . $mt$) var ($e$, $t$) val ($e$) abn then ($e$ :: [*] . $mt$){transition});
(transition ($c$s . $mt$) :: {transition} var ($c$s, $mt$) then $f$),
where ($c$s . $mt$) :: {transition}; $e^* # s \leftrightarrow_{[s]} e^* # [c$s . $mt$] # $s$.

If $c$s and $n$ are values of $e$1 and $e$2, then the conceptual structure access operation ($e$1 . . $e$2) returns [$c$s . . $n$]. It is defined as follows:
(rule ($e$1 . . $e$2) var ($e$1, $e$2) val ($e$1, $e$2) abn

where (($e$1 :: [*] is compound) and ($e$2 :: [*] is nat) and ($e$2 :: [*] > 0))
then ($e$1 :: [*] . . $e$2 :: [*]){transition});
(transition ($c$s . . $n$) :: {transition} var ($c$s, $n$) then $f$),
where $(cs .. n) :: \{transition\}; \ e^* \# \ s \leftarrow[sf] \ e^* \# [cs .. n] \# s$.

If $cs$ and $v$ are values of $e$ and $e1$, then the conceptual structure update operation $(e . mt := e1)$ returns $[cs . mt := v]$. It is defined as follows:

```
(rule (e . mt := e1) var (e, mt1, e1) val (e, e1) abn)
then (e::[*] . mt1 := e1::[*]):{transition});
```

where $(cs . mt := v)::\{transition\}; \ e^* \# \ s \leftarrow[sf] \ e^* \# [cs . mt := v] \# s$.

The conceptual structure update operation $(e . mt :=)$ is a shortcut for $(e . mt := und)$.

The conceptual structure update operation $(e . mt1 := e1, .. , mt$n := e$n)$, where $n > 1$, is defined as follows:

```
(rule (e . mt1 := e1 cs_s) var (e, mt1, e1) seq (cs_s) abn then ((e . mt1 := e1) . cs_s)).
```

If $ccs$, $s$n and $v$ are values of $e1$, $e2$ and $e3$, then the conceptual structure update operation $(e1 .. e2 := e3) returns [ccs .. n := v]$. It is defined as follows:

```
(rule (e1 .. e2 := e3) var (e1, e2, e3) val (e1, e2, e3) abn
  where ((e1 :: [*] is compound) and (e2 :: [*] is nat) and (e2 :: [*] > 0))
  then (e1 :: [*] . e2 :: [*] := e3 :: [*]):{transition});
```

where $(ccs .. n := v)::\{transition\}; \ e^* \# \ s \leftarrow[sf] \ e^* \# [ccs .. n := v] \# s$.

If $ccs1$ and $ccs2$ are values of $e1$ and $e2$, then the element $(e1 + e2)$ specifies the concatenation of $ccs1$ and $ccs2$. It is defined by the rules

```
(rule (e1 + e2) var (e1, e2) val (e1, e2) abn then (e1 :: [*] +:: [q] e2 :: [*]));
(rule ((ccs_s1) +:: [q] (cs_s2)) seq (cs_1, cs_2) then (cs_s1 cs_s2) :: [q]).
```

If $e$ and $ccs$ are values of $e1$ and $e2$, then the element $(e1 . e2)$ specifies the concatenation of $e$ to the head of $ccs$. It is defined by the rules

```
(rule (e1 . e2) var (e1, e2) val (e1, e2) abn then (e1 :: [*] . +:: [q] e2 :: [*]));
(rule (e . +:: [q] (cs_s)) var(e) seq (cs_s) then (e cs_s) :: [q]).
```

If $e$ and $ccs$ are values of $e2$ and $e1$, then the element $(e1 + . e2)$ specifies the concatenation of $e$ to the tail of $ccs$. It is defined by the rules

```
(rule (e1 + e2) var (e1, e2) val (e1, e2) abn then (e1 :: [*] +:: [q] e2 :: [*]));
(rule ((cs_s) +:: [q] e) var(e) seq (cs_s) then (cs_s e) :: [q]).
```

If $e$ and $n$ are values of $e1$ and $e2$, then the element $(repeat \ e \ n)$ returns $([repeat \ e \ n])$. It is defined by the rule

```
(rule (repeat e n) var (e, n) val (e, n) abn where (n :: [*] is nat)}
```
then (repeat::[q] e::[*] n::[*])).

The element (repeat::[q] $e1 $e2) is defined by the rules

(rule (repeat::[q] e 0) var (e) abn then());

(rule (repeat :: [q] e n) var (e, n) abn

then (let n1 be (n − 1) in ((repeat::[q] e n1) +. e::[q])).

The element (unbracket $ccs) is defined by the rule

(rule (unbracket (cs_s)) seq (cs_s) abn then cs_s).

6.5. Sets

The element ($e is set) specifies that the elements of the compound structure $e are pairwise

distinct is defined as follows:

(rule (e is set) var (e) abn where (e is compound) then (e is set)::[transition]);

(transition (e is set):::[transition] var (e) then $f),

where

(e is set)::[transition] $e* # $s ↦[sf]

$e* # [if [the elements of $e are pairwise distinct then true else und] # $s.

If $e and $ccs are values of $e2 and $e1, then the element ($e1 +.: {set} $e2) specifies the

addition of the element $e to the set $ccs. It is defined by the rule

(rule (e1 +.: {set} e2) var (e1, e2) val (e1, e2) abn

then (if (e2 ::[*] ::[q] in e1 ::[*] ::[q]) then e1 ::[*] ::[q]

else (e2 ::[*] +.: [q] e1 ::[*])).

If $e and $ccs are values of $e2 and $e1, then the element ($e1 −.: {set} $e2) specifies the

deletion of the element $e from the set $ccs. It is defined by the rule

(rule (e1 −.: {set} e2) var (e1, e2) val (e1, e2) abn where (e1 ::[*] is set)

then (e1 ::[*] −.: {set} e2 ::[*]) ::[transition]);

(transition (ccs −.: {set} e) ::[transition] var (ccs, e) then $f),

where ($ccs −.: {set} $e) ::{transition} $e* # $s ↦[sf] $e* # $ccs1 # $s, $ccs1 is a set, and

[ccs1 =_{set} ccs $v] .

If $e and $ccs are the values of $e1 and $e2, then the element ($e1 in $e2) specifies that $e is an

element of $ccs. It is defined as follows:

(rule (e1 in::{set} e2) var (e1, e2) val (e1, e2) abn where (e2 ::[*] is compound)

then (e1 ::[*] in::{set} e2 ::[*]) ::{transition});

(transition (e in::{set} ccs) ::{transition} var (e, ccs) abn then $f).
where

\( (\{ e \text{ in } \{ \text{set} \} \{ \text{ccs} \} \} :: \{ \text{transition} \}; \; \{ e \} \; s \xrightarrow{\{ sf \}} \{ s \} ) \) \\
\( \# \) \[ if \; [ e \in \{ \text{ccs} \} \text{ then } \text{true } \text{ else } \text{und} ] \) \# \( s \).

If \( \{ \text{ccs}1 \} \) and \( \{ \text{ccs}2 \} \) are the values of \( e1 \) and \( e2 \), then the element \( (\{ e1 \} \text{ includes } \{ \text{set} \} \{ \text{ccs} \} \{ e2 \}) \) specifies that \( \{ \text{ccs}1 \} \) includes the elements of \( \{ \text{ccs}1 \} \). It is defined as follows:

\( (\{ \text{rule} \} \{ e1 \} \text{ includes } \{ \text{set} \} \{ \text{ccs}1 \} \{ e2 \} \text{ var } \{ e1 \} \{ e2 \} \text{ val } \{ e1 \} \{ e2 \} \text{ abn} \) \\
\text{where } ((\{ e1 \} \{ \text{set} \} \{ \text{ccs} \} \{ e2 \} \text{ is } \text{compound} \) \text{ and } \{ e2 \} \{ \text{set} \} \{ \text{ccs} \} \{ e2 \} \text{ is } \text{compound} \) \\
\text{then } (\{ e1 \} \{ \text{set} \} \{ \text{ccs} \} \{ e2 \} \{ \text{set} \} \{ \text{ccs} \} \{ e2 \} \text{ transition} \} \text{ var } \{ \text{ccs}1 \} \{ \text{ccs}2 \} \text{ abn then } \{ s \}).

where

\( (\{ \text{rule} \} \{ e1 \} \{ \text{set} \} \{ \text{ccs}1 \} \{ e2 \} \text{ disjunct } \{ e1 \} \{ e2 \} \text{ var } \{ e1 \} \{ e2 \} \text{ abn} \) \\
\text{where } ((\{ e1 \} \{ \text{set} \} \{ \text{ccs} \} \{ e2 \} \text{ is } \text{compound} \) \text{ and } \{ e2 \} \{ \text{set} \} \{ \text{ccs} \} \{ e2 \} \text{ is } \text{compound} \) \\
\text{then } (\{ \text{disjunct} \} \{ \text{set} \} \{ e1 \} \{ e2 \} \{ \text{set} \} \{ \text{ccs} \} \{ e2 \} \text{ transition} \} \text{ var } \{ \text{ccs}1 \} \{ \text{ccs}2 \} \text{ abn then } \{ s \}).

where

\( (\{ \text{disjunct} \} \{ \text{set} \} \{ \text{ccs}1 \} \{ \text{ccs}2 \} \{ \text{set} \} \{ \text{ccs} \} \{ e2 \} \text{ transition} \} \text{ var } \{ \text{ccs}1 \} \{ \text{ccs}2 \} \text{ abn then } \{ s \}).

The elements \( (\{ e1 \} \text{ in } \{ \text{set} \} \{ e2 \}) \), \( (\{ e1 \} \text{ includes } \{ \text{set} \} \{ e2 \}) \) and \( (\{ \text{disjunct} \} \{ \text{set} \} \{ e1 \} \{ e2 \}) \) are shortcuts for \( (\{ e1 \} \text{ in } \{ \text{set} \} \{ e2 \}) \), \( (\{ e1 \} \text{ includes } \{ \text{set} \} \{ e2 \}) \) and \( (\{ \text{disjunct} \} \{ \text{set} \} \{ e1 \} \{ e2 \}) \).

### 6.6. States

The executable elements handling states are defined in this section.

The state access operation (current state) returns the current state is defined by the rule:

\( (\{ \text{rule} \} \{ \text{current state} \} \{ \text{abn} \} \text{ then } \{ \text{cstate} \} \{ \text{q} \}) \).

If \( \{ \text{cs} \} \) is a value of \( e \), then the element \((\{ \text{to state} \} \{ e \})\) replaces the current state to \( \{ \text{cs} \} \). It is defined as follows:

\( (\{ \text{rule} \} \{ \{ \text{to state} \} \{ e \} \} \text{ var } \{ e \} \text{ val } \{ e \} \text{ then } ((\{ \text{to state} \} \{ e \} \{ \text{set} \} \{ e \} \text{ transition} \} \text{ var } \{ \text{cs} \} \text{ then } \{ s \}) \).

where \((\{ \text{to state} \} \{ \text{cs} \} \{ \text{transition} \})\); \( \{ e \} \# \{ s \} \xrightarrow{\{ sf \}} \{ cs \} \).
The state access operation \( (. \$mt) \) returns \([. \$mt]\). It is defined by the rule

\[
\text{(rule \( (. \ mn\) var (\$mt) \text{abn} \ then \ (\text{cstate} \::\ {q}\ , \ mn))\).}
\]

If \$v is a value of \$e, then the state update operation \( (\mathit{mt} := \$e) \) replaces the current state by \([. \mathit{mt} := \$v]\). It is defined by the rule

\[
\text{(rule \( (\mathit{mt} := e) \text{var} (\mathit{mt}, e) \text{abn} \ then \ (\text{to state} \ (\text{cstate} :: \{q\} \ . \ \mathit{mt} \ := \ e)))\).}
\]

The state update operation \( (\mathit{mt} := \$e) \) is an alias for \( (\mathit{mt} := \$e) \). It is defined by the rule

\[
\text{(rule \( (. \ mn\) := \$e) var (\mathit{mt}, e) \text{abn} \ then \ (\mathit{mt} := e))\).
\]

The state update operation \( (\mathit{mt} := \)\) and \( (. \ mn\) := \)\) are shortcuts for \( (\mathit{mt} := \text{und}) \) and \( (. \ mn\) := \text{und})\).

The state update operation \( (\mathit{mt}1 := \$e1, \ldots, \mathit{mt}n := \$en) \), where \(n > 1\), is defined by the rule

\[
\text{(rule \( (\mathit{mt}1 := e1 cs\_s) \text{var} (\mathit{mt}1, e1) \text{seq} (cs\_s) \text{abn} \ then \ (\mathit{mt}1 := e1); (cs\_s))\).}
\]

The state update operation \( (\mathit{mt}1 := \$e1, \ldots, \mathit{mt}n := \$en) \) is an alias for \( (\mathit{mt}1 := \$e1, \ldots, \mathit{mt}n := \$en) \). It is defined by the rule

\[
\text{(rule \( (. \ mn\_1 := e1 cs\_s) \text{var} (\mathit{mt}1, e1) \text{seq} (cs\_s) \text{abn} \ then \ (\mathit{mt}1 := e1 cs\_s))\).}
\]

### 6.7. Statements

The executable elements called statements are defined in this section. They are similar to statements in programming languages.

The element \textit{skip} does nothing. It is defined as follows:

\[
\text{(rule \textit{skip} \text{abn} \ then \ \textit{skip} :: \{transition\});}
\]

\[
\text{(transition \textit{skip} :: \{transition\} \ then \$f),}
\]

where \textit{skip} :: \{transition\}; \$e^* \# \$s \rightsquigarrow [\$f] \ e^* \# \$s.

The element \$e of the form \( (\text{seq} \ e^*) \) is called a sequential composition. It is defined by the rule

\[
\text{(rule \( (\text{seq} \ e\_s) \text{var} (e\_s) \text{seq} (e\_s) \ then \ e\_s)\).}
\]

The elements of \$e^* are called elements in \([\$e]\], and \$e^* is called a body in \([\$e]\]. The element \$e executes its elements sequentially from left to right.

The element \$e of the form \( (if \$co \ then \$e^1 \ else \$e^2) \) is called a conditional element. It is defined as follows:

\[
\text{(rule \( (if \ \text{co} \ then \ e\_s1 \ else \ e\_s2) \text{var} (co) \text{seq} (e\_s1, e\_s2) \text{val} (co) \text{abn}
then (if \ co :: \{*\} \ then \ e\_s1 \ else \ e\_s2) :: \{transition\});}
\]

\[
\text{(transition \( (if \ v \ then \ e\_s1 \ else \ e\_s2) :: \{transition\} \text{var} (v) \text{seq} (e\_s1, e\_s2) \ then \$f),}
\]

where
(if $v$ then $e^* 1$ else $e^* 2$) :: \{transition\}; $e^* \$s \leftrightarrow [5f]

[if [$v \neq \text{und}$] then $e^* 1$ else $e^* 2$] $e^* \# \$s$.

The objects $co$, $e^* 1$ and $e^* 2$ are called a condition, then-branch and else-branch in $[[e]]$. The element (if $con$ then $e^*$) is a shortcut for (if $con$ then $e^*$ else skip).

The conditional element (if :: \{exc\} $con$ then $e^* 1$ else $e^* 2$) is defined by the rule

(rule (if :: \{exc\} con then e_s1 else e_s2) var (con) seq (e_s1, e_s2) val (con)
  exc (con, con :: \{\*\}) abn then (if con :: \{q\} then e_s1 else e_s2)).

The element (if :: \{exc\} $con$ then $e^*$) is a shortcut for (if :: \{exc\} $con$ then $e^*$ else skip).

The conditional element

(if $se1$ $co1$ then $e^* 1$ elseif $se2$ $co2$ then $e^* 2$ ... elseif $se$n $co$n then $e^* n$ else $e^*$),

where $se$n$1 \in \{[es], exc\}$ for each $1 \leq n1 \leq n$, is defined by the rules

(rule (if co then e_s1 elseif e_s2) var (co) seq (e_s1, e_s2) abn
  then (if co then e_s1 else (if e_s2)));

(rule (if co then e_s1 elseif :: \{exc\} e_s2) var (co) seq (e_s1, e_s2) abn
  then (if co then e_s1 else (if :: \{exc\} e_s2)));

(rule (if :: \{exc\} co then e_s1 elseif e_s2) var (co) seq (e_s1, e_s2) abn
  then (if :: \{exc\} co then e_s1 else (if e_s2)));

(rule (if :: \{exc\} co then e_s1 elseif :: \{exc\} e_s2) var (co) seq (e_s1, e_s2) abn
  then (if :: \{exc\} co then e_s1 else (if :: \{exc\} e_s2))).

The element $e$ of the form (let $va$ be $e^* 1$ in $e^* 2$) is defined as follows:

(rule (let va be e_s1 in e_s2) var (va) seq (e_s1, e_s2) abn
  then e_s1; (let va be current value in e_s2) :: \{transition\});

(transitıon (let va be current value in e_s2) :: \{transition\} var (va) seq (e_s2) then \$f$),

where

(let $va$ be current value in $e^* 2$) :: \{transition\}; $e^* \# \$v \# \$s \leftrightarrow [5f]

[sub $\langle va; v \rangle$ $e^* 2$]; $e^* \# \$s$.

The elements $va$, $e^* 1$ and $e^* 2$ are called a variable, value specifier and body in $[[e]]$.

The element $e$ of the form (let :: \{und\} $va$ be $e^* 1$ in $e^* 2$) is defined by the rules

(rule (let :: \{und\} va be e_s1 in e_s2) var (va) seq (e_s1, e_s2) abn
  then e_s1; (let :: \{und\} va be current value in e_s2));

(rule (let :: \{und\} va be current value in e_s2) var (va) seq (e_s2) und
  then (let va be current value in e_s2): \{transition\}).

The element $e$ of the form (let :: \{abn\} $va$ be $e^* 1$ in $e^* 2$) is defined by the rules
(rule (let :: {abn} va be e_s1 in e_s2) var (va) seq (e_s1, e_s2) abn
then e_s1; (let :: {abn} va be current value in e_s2));

(rule (let :: {abn} va be current value in e_s2) var (va) seq (e_s2) abn
then (let va be current value in e_s2) :: {transition}).

The element $e$ of the form (let :: {exc} $va be $e^*1 in $e^*2$) is defined by the rules

(rule (let :: {exc} va be e_s1 in e_s2) var (va, e1) seq (va_s, e_s1, e_s2) abn
then e_s1; (let :: {exc} va be current value in e_s2));

(rule (let :: {exc} va be current value in e_s2) var (va) seq (e_s2) exc
then (let va be current value in e_s2) :: {transition}).

The element $e$ of the form (let :: {seq} $va^* be $e^*1 in $e^*2$), where [len va] = [len e^*1], is
defined by the rules

(rule (let :: {seq} va, va_s be e1, e_s1 in e_s2) var (va, e1) seq (va_s, e_s1, e_s2) abn
then (let va be e1 in (let :: {seq} va_s be e_s1 in e_s2)));

(rule (let :: {seq} be in e_s2) seq (e_s2) abn then e_s2).

The elements $va^*$, $e^*1$ and $e^*2$ are called a variable specification, value specification and body
in [[e]]. The elements of $va^*$ and $e^*1$ are called variables and value specifiers in [[e]].

The element $e$ of the form (let :: {seq, und} $va^* be $e^*1 in $e^*2$), where [len va] =
[len e^*1], is defined by the rules

(rule (let :: {seq, und} va, va_s be e1, e_s1 in e_s2) var (va, e1) seq (va_s, e_s1, e_s2) abn
then (let :: {und} va be e1 in (let :: {seq, und} va_s be e_s1 in e_s2)));

(rule (let :: {seq, und} be in e_s2) seq (e_s2) abn then e_s2).

The element $e$ of the form (let :: {seq, abn} $va^* be $e^*1 in $e^*2$), where [len va] =
[len e^*1], is defined by the rules

(rule (let :: {seq, abn} va, va_s be e1, e_s1 in e_s2) var (va, e1) seq (va_s, e_s1, e_s2) abn
then (let :: {abn} va be e1 in (let :: {seq, abn} va_s be e_s1 in e_s2)));

(rule (let :: {seq, abn} be in e_s2) seq (e_s2) abn then e_s2).

The element $e$ of the form (let: {seq, exc} $va^* be $e^*1 in $e^*2$), where [len va] = [len e^*1],
is defined by the rules

(rule (let :: {seq, exc} va, va_s be e1, e_s1 in e_s2) var (va, e1) seq (va_s, e_s1, e_s2) abn
then (let :: {exc} va be e1 in (let :: {seq, exc} va_s be e_s1 in e_s2)));

(rule (let :: {seq, exc} be in e_s2) seq (e_s2) abn then e_s2).

The element $e$ of the form (while $con do $e^*1) is called a while statement. It is defined by the
rule
(if (while con do e_s) var (con) seq (e_s) abn
    then (if con then e_s; (while con do e_s))).

The objects $con$ and $e^* 1$ are called a condition and body in $[se]$.

The element $e$ of the form ($foreach va in e1 do e^* 1$) is called a foreach statement. It is defined by the rule

$\begin{align*}
(rule (foreach va in e1 do e^* 1) var (va, e1) seq (e_s) val (e1) abn \\
\quad \text{then (foreach::[q} va in e1 :: {*} do e_s)).}
\end{align*}$

The objects $va$, $e1$ and $e^* 1$ are called an iteration variable, iteration structure specifier and body in $[se]$. If $(v^*)$ is a value of $e1$, then the element $e$ executes sequentially $e^* 1$ for values of $va$ from the structure $(v^*)$.

The element ($foreach::[q} va in (v^*) do e^* )$ is defined by the rules

$\begin{align*}
(rule (foreach::[q} va in (v v_s) do e_s) var (va, v) seq (v_s, e_s) abn \\
\quad \text{then (let va be v :: [q} in e_s); (foreach::[q} va in (v v_s) do e_s));}
\end{align*}$

$\begin{align*}
(rule (foreach::[q} va in () do e_s) var (va) seq (e_s) abn then).
\end{align*}$

### 6.8. Countable concepts

The executable elements handling countable concepts are defined in this section.

A normal element $e$ is a countable concept in $[[s]]$ if $[[s . ((countable concept) e)]] \in \mathbb{N} > 0$. Thus, the parametric attribute ($((countable concept) e)$) defines countable concepts. Let $\mathbb{N}$ be a set of countable concepts. A number $n$ is an order in $[[cc, s]]$ if $n = [[s . ((countable concept) cc)]]$. Let $\mathbb{N}$ be a set of orders of countable concepts. An element $n: \{cc\}$ is called an instance in $[[cc]]$. An element $n: \{cc\}$ is an instance in $[[cc, s]]$ if $1 \leq n \leq \mathbb{N}$. The element ($e$ is (countable concept)) specifies that $e$ is a countable concept. It is defined by the rule

$\begin{align*}
(rule (e is (countable concept)) var (e) abn then ((([((countable concept) e)]) > 0)).
\end{align*}$

The element ($e$ is $cc$) specifies that $e$ is an instance of $cc$. It is defined by the rule

$\begin{align*}
(rule (n: \{cc\}) is cc2) var (n, cc1, cc2) abn where (cc2 is (countable concept)) \\
\quad \text{then ((cc1: [q} = cc2: [q}) and (0 < n) and (n <= (.[((countable concept) cc2)]))).}
\end{align*}$

The element ((new instance) $cc$) generates a new instance of the countable concept $cc$ and adds this concept if it was not. It is defined by the rule

$\begin{align*}
(rule ((new instance) cc) var (cc) abn \\
\quad \text{then (let n be (.[((countable concept) cc)])) in}
\end{align*}$
(if (n > 0) then (let n1 be (n + 1) in
 (((countable concept) cc) = n1); n1::{cc}::{q})
else (((countable concept) cc) = 1); n1::{cc}::{q}).

6.9. Rules

The executable elements handling rules are defined in this section.
The element ($e is rule$) specifies that $e$ is a rule. It is defined as follows:
(rule (e is rule) var (e) abn then (e is rule):: {transition});
(transition (e is rule):: {transition} var (e) then $f$),

where

($e is rule):: {transition}$ e* # $s ↦ [sfr] e* # [if [e ∈ $$r$$] then true else und] # $s.$

The element ($e is (extended rule)$) specifies that $e$ is an extended rule. It is defined as follows:
(rule (e is (extended rule)) var (e) abn then (e is (extended rule)):: {transition});
(transition (e is (extended rule)):: {transition} var (e) then $f$),

where

($e is (extended rule)):: {transition}$ e* s $\leftrightarrow [sfr]$

$e^* # [if [e ∈ $$er$$] then true else und] # $s.$

An element $na$ is a name if $na$ is normal. Let $$n$ be a set of names.
The element ($e is name$) specifies that $e$ is a name. It is defined by the rule
(rule (e is name) var (e) abn then (e is normal)).

The element $r$: $na$ adds the rule $r$ with the name $n$ into $\{rules\}$. It is defined by the rule
(rule e:: {na} var (e, na) abn where ((e is rule) and (na is name))
then (\{rules\} := ((\{rules\}). {na} := e::\{q\}))).

The element $er$: $na$ adds the rule $r$ with the name $n$ into $\{rules\}$, where $er$ is a shortcut for $r$. It is defined as follows:

(rule e:: {na} var (e, na) abn where ((e is (extended rule)) and (na is name))
then (\{rules\} := ((\{rules\}). {na} := (rule e)))).

The element (rule $er$) returns $r$, where $r$ is a shortcut for $er$. It is defined as follows:

(transition (rule er):: {transition} var (er) then $f$),

where (rule $er$):: {transition}$ e^* # s \leftrightarrow [sfr] e^* # r # s$, where $r$ is a shortcut for $er$.

6.10. The pattern matching

The executable elements handling the pattern matching are defined in this section.
The conditional pattern matching element $\textit{e}$ of the form

\[(\textit{if } \textit{e1 matches } \textit{p var } (\textit{va'}) \textit{ seq } (\textit{sv'}) \textit{ then } \textit{e1 else } \textit{e2}),\]

where $(\textit{p } (\textit{va'}) (\textit{sv'}))$ is a pattern specification, executes $\textit{e1}$ if $\textit{e1}$ matches $\textit{p}$ and executes $\textit{e2}$, otherwise. It is defined as follows:

\[
\begin{align*}
\text{(rule } & \text{ if } \textit{e1 matches } \textit{p var } (\textit{va_s}) \textit{ seq } (\textit{sv_s}) \textit{ then } \textit{e1 else } \textit{e2} \text{ var } (\textit{e1}, \textit{p}) \\
& \text{ seq } (\textit{va_s}, \textit{sv_s}, \textit{e_s1}, \textit{e_s2}) \textit{ abn} \text{ where } (\text{disjoint } (\textit{va_s}):\{q\} (\textit{sv_s}):\{q\}) \\
& \text{ then } \text{ if } \textit{e1 matches } \textit{p var } (\textit{va_s}) \textit{ seq } (\textit{sv_s}) \textit{ then } \textit{e1 else } \textit{e2}:\{\text{transition}\}; \\
& \text{ transition } \text{ if } \textit{e1 matches } \textit{p var } (\textit{va_s}) \textit{ seq } (\textit{sv_s}) \textit{ then } \textit{e1 else } \textit{e2}:\{\text{transition}\} \\
& \text{ var } (\textit{e1}, \textit{p}) \text{ seq } (\textit{va_s}, \textit{sv_s}, \textit{e_s1}, \textit{e_s2}) \text{ then } \textit{f}),
\end{align*}
\]

where

\[
\begin{align*}
(\text{if } & \textit{e1 matches } \textit{p var } (\textit{va'}) \textit{ seq } (\textit{sv'}) \textit{ then } \textit{e1 else } \textit{e2}:\{\text{transition}\}; \textit{e2} \# \textit{s} \leftarrow [s]\) \\
(\text{if } & \textit{e1 is an instance in } [(\textit{p } (\textit{va'}) (\textit{sv'}))], \textit{smt}, \textit{su}] \text{ for some } \textit{su} \\
\text{ then } & \text{sub } \textit{su} \cup (\text{cstate: } \textit{s}, \textit{cvalue: } \textit{v}[[\textit{s}]] \textit{ e1}) \textit{ e2} \\
\text{ else } & \text{sub } (\text{cstate: } \textit{s}, \textit{cvalue: } \textit{v}[[\textit{s}]] \textit{ e2}); \textit{e2} \# \textit{s}.
\end{align*}
\]

Thus, the semantics of the conditional pattern matching elements combines the semantics of the conditional element with the semantics of transition rules. The elements $\textit{e1}$, $\textit{p}$, $\textit{va'}$, $\textit{sv'}$, $\textit{e1}$ and $\textit{e2}$ are called a matched structure, pattern, state variable specification, sequence variable specification, then-branch and else-branch in $[\textit{e}]$. The elements of $\textit{va'}$ and $\textit{sv'}$ are called state and sequence variables in $[\textit{e}]$.

Let $\{\textit{seva'}\} \subseteq \{\textit{va'}\}$, the elements of the sequence $\textit{seva'}$ are pairwise disjoint, $\textit{set} = \{\textit{seva} : \{\} | \textit{seva} \in \textit{seva'}\}$, $\{\textit{va'}\} \cup \{\textit{va'}\} \cup \{\textit{va'}\} \subseteq \{\textit{va'}\} \cup \textit{set}$, the elements of the sequence $\textit{va'} \textit{va'} \textit{va'} \textit{va'}$ are pairwise disjoint, and $\textit{se} \in \{[\textit{se}], \textit{und}, \textit{exc}, \textit{abn}\}$.

The semantics of the extension

\[
(\textit{if } \textit{e1 matches } \textit{p var } (\textit{va'}) \textit{ seq } (\textit{sv'}) \textit{ val } (\textit{seva'}) \textit{ abn } (\textit{va'} \textit{ und } (\textit{va'} \textit{ exc } (\textit{va'} \textit{ sse w}) \textit{ then } \textit{e1 else } \textit{e2})
\]

of the conditional pattern matching element is similar to the semantics of extended transition rules.

The objects $\textit{var } (\textit{va'})$, $\textit{seq } (\textit{sv'})$, $\textit{val } (\textit{seva'})$, $\textit{abn } (\textit{va'} \textit{ und } (\textit{va'} \textit{ exc } (\textit{va'} \textit{ sse w}) \textit{ then } \textit{e1 else } \textit{e2})$ and $\textit{where } \textit{co}$ in conditional pattern matching elements can be omitted. The omitted objects correspond to $\textit{var } (\textit{, seq } (\textit{, val } (\textit{, abn } (\textit{, und } (\textit{, exc } (\textit{, where true}))}

The pattern matching element

\[
(\textit{e1 matches } \textit{p var } (\textit{va'}) \textit{ seq } (\textit{sv'}) \textit{ val } (\textit{seva'}) \textit{ abn } (\textit{va'} \textit{ und } (\textit{va'} \textit{ exc } (\textit{va'} \textit{ sse w}) \textit{ then } \textit{e1 else } \textit{e2})
\]

is a shortcut for
(if $e_1$ matches $p$ var ($va^*$) seq ($sv^*$) val ($eva^*$) abn ($va^*1$) und ($va^*2$)
   exc ($va^*3$) $se$ where $co$ then true else und).

The selection element $e$ of the form (select $va$ from $e_1$ wrt $p$ var ($va^*$) seq ($sv^*$)),
where ($p$ ($va^*$) ($sv^*$)) is a pattern specification, and $va \in \{va^*$, or $va \in \{sv^*$, selects the
values of the variable $va$ such that an element of $e_1$ matches the pattern $p$. It is defined by the
rule

(rule (select va from of e1 wrt p var (va_s) seq (sv_s))
   var (va,e1,p) seq (va_s, sv_s, t_s) abn
   where ((disjoint (va_s) :: {q} (sv_s) :: {q}) and
   ((va::{q} in (va_s):: {q}) or (va::{q} in (sv_s):: {q})))
   then (select::{check} va from e1 wrt p var (va_s) seq (sv_s))).

The elements $va$, $e_1$, $p$, $va^*$ and $sv^*$ are called a selection variable, matched structure,
pattern, state variable specification, sequence variable specification, then-branch and else-branch in
$[[e]]$. The elements of $va^*$ and $sv^*$ are called state and sequence variables in $[[e]]$.

The element (select::{check} $va$ from $e_1$ wrt $p$ var ($va^*$) seq ($sv^*$)) is defined by the
rules

(rule (select::{check} va from e1::{t_s} wrt p var (va_s) seq (sv_s))
   var (va,e1, p) seq (va_s, sv_s, t_s) abn
   then (select::{check} va from (e1::{t_s}) wrt p var (va_s) seq (sv_s)));

(rule (select::{check} va from e1::{t_s} wrt p var (va_s) seq (sv_s))
   var (va, e1, p) seq (va_s, sv_s, t_s) abn
   then (select::{check} va from (e1::{t_s}) wrt p var (va_s) seq (sv_s)));

(rule (select::{check} va from (e1 e_s) wrt p var (va_s) seq (sv_s))
   var (va, e1, p) seq (e_s, va_s,sv_s) abn then ());

(rule (select::{check} va from (e1 e_s) wrt p var (va_s) seq (sv_s))
   var (va, e1, p) seq (e_s, va_s,sv_s) abn where (va:: {q} in (va_s):: {q})
   then (if e1 matches p var (va_s) seq (sv_s)
   then (va_s:: {q} . + (select::{check} va from (e_s) wrt p var (va_s) seq (sv_s))
   else (select::{check} va from (e_s) wrt p var (va_s) seq (sv_s))));

(rule (select::{check} va from (e1 e_s) wrt p var (va_s) seq (sv_s))
   var (va, e1, p) seq (e_s, va_s,sv_s) abn where (va:: {q} in (sv_s):: {q})
   then (if e1 matches p var (va_s) seq (sv_s)
   then ((va_s):: {q} . + (select::{check} va from (e_s) wrt p var (va_s) seq (sv_s))))
else (select:: {check} va from (e_s) wrt p var (va_s) seq (sv_s)))

The semantics of the extension

(select $va$ from $e1$ wrt p var ($va^+$) seq ($sv^*$) val ($eva^*$)

$abn$ ($va^1$) $und$ ($va^2$) exc ($va^3$) $se$ where $co$

of the selection element is similar to the semantics of the extension of the conditional pattern matching element.

The semantics of the extension

(select: :{seq} $va^1+$ from $e1$ wrt p var ($va^*$) seq ($sv^*$) val ($eva^*$)

$abn$ ($va^1$) $und$ ($va^2$) exc ($va^3$) $se$ where $co$

of the selection element is similar to the semantics of the above selection element extension except that the resulting sequence consists of the compound elements of the length [$va^1+$. Each of these elements contains the values of the selection variables in the order of their occurrences in $va^1+$.

The selection elements

(select $va$ wrt p var ($va^*$) seq ($sv^*$) val ($eva^*$)

$abn$ ($va^1$) $und$ ($va^2$) exc ($va^3$) $se$ where $co$

and

(select: :{seq} $va^1+$ wrt p var ($va^*$) seq ($sv^*$) val ($eva^*$)

$abn$ ($va^1$) $und$ ($va^2$) exc ($va^3$) $se$ where $co$

are shortcuts for

(let $s$ be (current state) in (select $va$ from $s$ wrt p var ($va^*$) seq ($sv^*$) val ($eva^*$)

$abn$ ($va^1$) $und$ ($va^2$) exc ($va^3$) $se$ where $co$)

and

(let $s$ be (current state) in (select:: {seq} $va^1+$ from $s$ wrt p var ($va^*$) seq ($sv^*$)

val ($eva^*$) $abn$ ($va^1$) $und$ ($va^2$) exc ($va^3$) $se$ where $co$).

7. Examples of conceptual operational semantics of programming languages

An operational semantics of executable elements of $c[l]$ in CTSL[o] is called a conceptual operational semantics of $l$. Thus, the conceptual operational semantics of $l$ is defined in terms of the conceptual model of $l$ in CTSL[o].

The conceptual operational semantics for the family of model programming languages (MPLs) is defined in this section. These languages has been described and their conceptual models has been defined in [1].
7.1. MPL1: types, typed variables and basic statements

The MPL1 language [1] is an extension of CTSL that adds types, typed variables, the variable access operation, and the basic statements such as variable declarations, variable assignments, if statements, while statements and block statements.

The element ($c$ is type) specifies types in MPL1. It is defined by the rules

(rule (t is type) var (t) abn then (t::[q] = int::[q]));
(rule (t is type) var (t) abn then (t::[q] = nat::[q])).

The element (subtype $t1$ $t2$) checks that $t1$ is a subtype of $t2$. It is defined by the rule

(rule (subtype $t1$ $t2$) var ($t1$, $t2$) abn
then (($t1$::[q] = nat::[q]) and ($t2$::[q] = int::[q]))).

The element ($na$ is variable) specifies variables. It is defined by the rule

(rule (va is variable) var (va) abn where (va is name) then (. {variable va)}).

The program is defined by the rule

(rule (program n c_s) var (n) seq (c_s) abn where (n is name)
then ((collect body members) c_s) c_s).

Let $m$ be a set of elements called body members.

The element ((collect body members) $c^*$) collects information about members of the body $c^*$. It is defined by the rules

(rule ((collect body members) (var va t) c_s) var (va, t) seq (c_s) abn
where ((va is name) and (not (va is variable)) and (t is type))
then {{{(type va)}} := t :: [q]; {{{(variable va)}} := true};
((collect body members) c_s));

(rule ((collect body members) (var c1 c2) c_s) var (c1, c2) seq (c_s) abn then und);

(rule ((collect body members) c c_s) var (c) seq (c_s) abn
then ((collect body members) c_s));

(rule ((collect variables)) then).

Thus, the body members in MPL1 are variables.

The variable declaration is defined by the rule

(rule (var c_s) seq (c_s) abn then).

The execution of the variable declaration does not collect information about the declared variable, since the corresponding actions are performed by the element ((collect body members) $c^*$).

The variable access is defined by the rule

(rule va var (va) abn where (va is variable) then (. {value va)}).
The element (type $c$) returns the type of the variable or the constant $c$. It is defined by the rules

(rule (type va) var (va) abn (va) abn where (va is variable) then (. {type va}));
(rule (type n) var (n) abn (n) abn where (n is nat) then nat::{q});
(rule (type i) var (i) abn (i) abn where (i is int) then int::{q}).

The variable assignment is defined by the rule

(rule (va := c) var (va, c) val (va) exc (c, c::{*}) abn
where ((va is variable) and

(let::{seq} t1, t2 be (type va), (type c::{*}) in (subtype t2 t1))
then (((value va)) = c::{*}::{q})).

The block statement is defined by the rule

(rule (block c_s) seq (c_s) abn then c_s).

The if statement is defined by the rule

(rule (\if c then c_s1 else c_s2) var (c) seq (c_s1, c_s2) abn (c) abn
then (if::{exc} c then (block c_s1) else (block c_s2)));

The while statement is defined by the rule

(rule (\while c do c_s1) var (c) seq (c_s1) abn (c) abn
then (while::{exc} c do (block c_s1))).

Thus, then- and else- branches of the if statement and the body of the while statement behaves as blocks.

7.2. MPL2: variable scopes

The MPL2 language [1] is an extension of MPL1 that adds the variable scopes feature. The relative scope of the variable $va$ occurring in the element $c$ is the number of blocks surrounding this occurrence of $va$ in $c$. The value and type of $va$ depend on its scope. The variable $va$ can be global (with the scope 0) and local.

The element (scope) returns the current scope. It is defined by the rule

(rule (scope) abn then (. {current scope})).

The same name $na$ can refer to different program objects. For example, $va$ refers to the variables with the name $va$ of the scopes from 0 to [. {current scope}]. To distinguish these program objects, they are versioned. The pair ($na$, $ve$), where $ve$ is a version, refers to the only one program object (with the version $ve$). In the case of variables, the version coincides with the variable scope.
The element \(\text{version $va$}\) returns the correct version of $va$ in the current context of program execution. It is defined by the rule

\[
\text{rule (version va) var (va) abn (va) abn where (va is name)
   then (let sc be (scope) in (version sc w))).}
\]

The element \(\text{version $va$ $sc$}\) is defined by the rule

\[
\text{rule (version va sc) var (va, sc) abn (va, sc) abn
   then (if (. {variable va sc}) then sc::q else (if (sc = 0) then und
      else (let sc1 be (sc − 1) in (version va sc1)))).}
\]

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

\[
\text{rule (va is variable) var (va) abn then (version va)).}
\]

The element \(\text{collect body members} C*\) is defined by the rule

\[
\text{rule ((collect body members) c_s) seq (c_s) abn
   then ((collect body members 1) () c_s).}
\]

The element \(\text{collect body members}::1\) \(\text{($va$) $c^*$}\) is defined by the rules

\[
\text{rule ((collect body members) ::1 ($va$) $c^*$) var (va, t) seq (va_s, c_s) abn (va, t) abn where ((va is name) and (t is type))
   then (let sc be (scope) in
      (if (. {variable va sc}) then und
      else (((type va sc) := t :: {q}); {(variable va sc)} := true);
      ((collect body members)::1 (va_s va) c_s)))));
\]

\[
\text{rule ((collect body members) ::1 (va_s) (var c1 c2) c_s) var (c1, c2) seq (va_s, c_s) abn then und);}
\]

\[
\text{rule ((collect body members)::1 (va_s) c c_s) var (c) seq (va_s, c_s) abn
   then ((collect body members)::1 (va_s) c_s));}
\]

\[
\text{rule ((collect body members)::1 (va_s)) seq (va_s) abn then (va_s)::q).}
\]

Thus, it returns the set of variables declared in the body $c^*$.

The variable access is defined by the rule

\[
\text{rule va var (va) abn (va) abn
   then (let::{und} sc be (version va) in (. {value va sc})}).}
\]

In the case when $c$ is a variable, the rule for the element \(\text{type $c$}\) is replaced by the rule

\[
\text{rule (type va) var (va) abn (va) abn
   then (let::{und} sc be (version va) in (. {type va sc})}).}
\]

The element \(\text{scope + +}\) increases the value of the current scope by 1. It is defined by the rule
The element \((\text{scope} - -)\) decreases the value of the current scope by 1. It is defined by the rule

\[
\text{rule } (\text{scope} - -) \text{ abn then } \{((\text{current scope}) \triangleright ((\text{current scope}) - 1))\}.
\]

The variable assignment is defined by the rule

\[
\text{rule } (\text{va} \triangleright = c) \text{ var } (\text{va}, c) \text{ val } (c) \text{ abn } (\text{va}) \text{ exc } (c, c:\{*\}) \text{ abn}
\]

then \((\text{let}:\{\text{und}\} \text{ sc be } (\text{version va}))\)

in \((\text{if} (\text{let} :\{\text{und}, \text{seq}\} \text{ t1, t2 be } (\text{type va sc}), (\text{type c} :\{*\}) \text{ in } (\text{subtype t2 t1}))\)

then \(((\text{value va sc}) := c:\{*\} :\{q\}) \text{ else und}))\).

The block statement is defined by the rule

\[
\text{rule } (\text{block c_s) seq (c_s) abn}
\]

then \((\text{enter block}); (\text{let} v_s \text{ be } ((\text{collect body members}) c_s)\)

in \(x (\text{catch} :\{\text{und}\} v ((\text{exit block}) v_s); v :\{q\})))\).

The element \((\text{enter block})\) specifies the actions executed when the current state enters the block. It is defined by the rule

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

The element \(((\text{exit block}) (\$va^*)\)) specifies the actions executed when the current state exits the block. It is defined by the rule

\[
\text{rule } (\text{exit block}) (\text{va_s}) \text{ seq (va_s) abn then } ((\text{delete variables}) \text{ va_s}); (\text{scope} - -)).
\]

The element \(((\text{delete variables}) \$va^*)\) deletes the local variables \$va* with the current scope. It is defined by the rules

\[
\text{rule } ((\text{delete variables}) \text{ va_s}) \text{ seq (va_s) abn}
\]

then \((\text{let sc be } (\text{scope}) \text{ in } ((\text{delete variables}) :\{1\} \text{ sc va_s})))\).

The element \(((\text{delete variables}) :\{1\} \text{ sc \$va^*})\) is defined by the rules

\[
\text{rule } ((\text{delete variables}) :\{1\} \text{ sc va_s}) \text{ var } (sc, va) \text{ seq (va_s) abn } (sc, y) \text{ abn}
\]

then \(((\text{variable va sc}) \triangleright); ((\text{type va sc}) \triangleright); ((\text{value va sc}) \triangleright);

((\text{delete variables}) :\{1\} \text{ sc va_s}));

\text{rule } ((\text{delete variables}) :\{1\} \text{ sc} \text{ var } (sc) \text{ abn } (sc) \text{ abn then})).

### 7.3. MPL3: functions

The MPL3 language [1] is an extension of MPL2 that adds the functions feature: declarations and calls of functions, and the return statement. For simplicity, function overloading is prohibited.

The element \((\text{call level})\) returns the current call level. It is defined by the rule

\[
\text{rule } (\text{call level}) \text{ abn then } (. \{\text{current call level}\})�.
\]
The element \((c \text{ is function})\) specifies functions. It is defined by the rule

\[
\text{(rule (f is function) var (f) abn where (f is name) then ( (function f))}.\]

In the case when the first element of the body \(c^*\) is a variable declaration, the rule for the element

\[
((\text{collect body members}):{1} (va^*) c^*)\]

is replaced by the rule

\[
\text{(rule (collect body members) :: 1) (va_s) (var va t) c_s) var (va, t) seq (va_s, c_s) abn (va, t) abn where ((va is name) and (t is type)) then (let::(seq) sc, cl be (scope), (if (sc = 0) then 0 else (call level)) in (if .{(variable va sc cl)) then und else {{(type va sc)): t::(q}); ((variable va sc)): = true); ((collect body members):: {1} (va_s va) c_s))).}\]

The element \((\text{(collect body members)::}{1} (va^*) c^*)\) is also redefined by the extra rules

\[
\text{(rule (collect body members) :: 1) (function f (tna_s) t c_s1) c_s) var (f, t) seq (tna_s, c_s1, c_s) abn (f, t) abn where ((f is name) and (not (f is function)) and (t is type)) then ((collect member arguments) f tna_s); (((return type) f)) := t::(q)); ((body f)) := (c_s1 :: [q]); ((function f)) := true); ((collect body members):: {1} c_s));}

\[(\text{rule (collect body members)::}{1} (function c_s)) seq (c_s) abn then und).\]

Thus, body members in MPL3 are variables and functions.

The element \((\text{(collect member arguments)} m$tna^*$)\) collects information about the typed arguments \(m$tna^*\) of the body member \(m\). It is defined by the rule

\[
\text{(rule (collect member arguments) m tna_s) var (m) seq (tna_s) abn then ((collect member arguments 1) m 0 tna_s)).}\]

The element \((\text{(collect member arguments 1)} m$ n$tna^*)\) is defined by the rules

\[
\text{(rule (collect member arguments 1) m n na t tna_s) var (m, n, na, t) seq (tna_s) abn where ((m is name) and (n is nat) and (na is name) and (t is type)) then (let n1 be (n + 1) in (((argument type) m n1)) := t::(q))); ((argument m n1)) := na::[q]; ((collect member arguments 1) m n1 tna_s))).}\]

\[
\text{(rule (collect member arguments 1) m n) var (m, n) abn where ((m is name) and (n is nat)) then ( ((arity m)) := n)).}\]

Thus, it collects information about function arguments.

The function declaration is defined by the rule:

\[
\text{(rule (function c_s) seq (c_s) abn then).}\]
The execution of the function declaration does not collect information about the declared function, since the corresponding actions are performed by the element (`(collect body members) $c^*$`).

The return statement is defined by the rule

(rule (return c) var (c) val (c) exc (c, c: {*}) abn
then (let: {seq} t1, t2 be (. {((current return type))}, (type c: {*})
in (if (subtype t2 t1) then (return: {type}, c: {*}: {value}): {exc} else und))).

The function call is defined by the rule

(rule (call f a_s) var (f) seq (a_s) abn (f) abn
where ((f is function) and ((len a_s:: {q}) = {. {((arity f)})}))
then (let: {und, seq} av_s, b, cs, crt
be ((argument values) a_s), (body f av_s), (scope), (. {((current return type)})
in (call level + +); ((((current scope)) := 0); b;
(catch: {und} v
(if (v is (not admissible function body value)) then und);
((((current return type)) := crt: {q}); (call level --); (((current scope)) := cs);
(if v matches (return: {type}, v1: {value}): {exc} var (v1) then (((to value) v1:: {q}))
else (((to value) v: {q})))).

The element ($v$ is (not admissible function body value)) specifies values that are not admissible when a function body exits. It is defined by the rule

(rule v is (not admissible function body value) var (v) abn
then (not (v is exception))).

Thus, the values that are not exceptions are not admissible in MPL3 when a function body exits.

The element ((argument values) $a^*$) returns the values of the arguments $a^*$. It is defined by the rules

(rule ((argument values) a, a_s) var (a) seq (a_s) abn (a) abn
then (a + ((argument values) a_s)));
(rule ((argument values)) then ()).

The element (body $f$ ($v^*$)) creates the block with the body of the function $f$ followed the declarations of the local variables corresponding to the arguments of $f$ and the assignment statements assigning the values $v^*$ to these variables.

(rule (body f (v.s)) var (f) seq (v.s) abn (f) abn
then (block: {q}. + (((create local variables) f 0 v.s) + (. {((block f))}))))
The element \((\text{create local variables} \ f \ n \ v^*)\) creates the declarations of the local variables corresponding to the arguments of \(f\) and the assignment statements assigning the values \(v^*\) to these variables. It is defined by the rules

\[
\begin{align*}
\text{(rule (create local variables) f n v_s) var (f, n, v) seq (v_s) abn (f, n, v) abn} \\
\text{then (let :: [seq] n1, a, t} & \\
\text{be (n + 1), (. {((argument f n1))}, (. {((argument type) f n1)}) in} & \\
\text{((var a t) . + ((a := v::[q]) . + ((create local variables) f n1 v_s)))))};
\end{align*}
\]

The element \((\text{call level} + +)\) increases the value of the current call level by 1. It is defined by the rule

\[
\begin{align*}
\text{(rule (call level + +) abn} & \\
\text{then (((current call level)) := (((current call level)) + 1))}. &
\end{align*}
\]

The element \((\text{call level} --)\) decreases the value of the current call level by 1. It is defined by the rule

\[
\begin{align*}
\text{(rule (call level --) abn} & \\
\text{then (((current call level)) := (((current call level)) - 1))}. &
\end{align*}
\]

The element \((\text{version} \ n a)\) is defined by the rules

\[
\begin{align*}
\text{(rule (version va) var (va) abn (va) abn where (va is name} & \\
\text{then (let::[seq] sc, cl be (current scope), (current call level) in (version va sc cl))};
\end{align*}
\]

The element \((\text{version} \ n a \ s c \ $c)\) is defined by the rules

\[
\begin{align*}
\text{(rule (version va y z) var (va, sc, cl) abn (va, sc, cl) abn} & \\
\text{then (if (. {variable va sc cl}) then sc} & \\
\text{else (if (sc = 0) then und} & \\
\text{else (let sc1 be (sc - 1) in (version va sc1 cl))})}. &
\end{align*}
\]

The variable access is defined by the rule

\[
\begin{align*}
\text{(rule va var (va) abn (va) abn} & \\
\text{then (let::[und, seq] sc, cl be (version va), (if (sc::[q] = 0) then 0 else (call level))} & \\
\text{in (. {value va sc cl})})}. &
\end{align*}
\]

In the case when \(\$c\) is a variable, the rule for the element \((\text{type} \ $c)\) is replaced by the rule

\[
\begin{align*}
\text{(rule (type va) var (va) abn (va) abn} & \\
\text{then (let :: [und, seq] sc, cl} & \\
\text{be (version va), (if (sc::[q] = 0) then 0 else (call level))} & \\
\text{in (. {type va sc cl})})}. &
\end{align*}
\]
The variable assignment is defined by the rule

\[(\text{rule (} \text{va} \ \text{=} \ c) \ \text{var (} \text{va}, \text{ e) val (} c) \ \text{abn (} v) \ \text{exc (} c, \ c::\{\}) \ \text{abn}\] then (let : \{\text{und} \} \ sc, \ cl, \ t1, \ t2

\[\text{be (variant va), (if (sc::\{q\} = 0) then 0 else (current call level)),}\]

\[\text{(\. \{type \text{va sc cl}\}\}, \text{(type c::\{\})}}\]

\[\text{in (if (subtype t2 t1) then ((value x sc cl) =: c::\{\})::\{q\} else und))).}\]

The element ((delete variables) \$va\*) is defined by the rules

\[(\text{rule ((delete variables) va_s) seq (} \text{va_s} \ \text{abn}\] then (let : \{seq \} \ sc, \ cl be (scope), (if (sc::\{q\} = 0) then 0 else (call level))

\[\text{in ((delete variables):\{1 sc cl va_s\}))}.\]

The element ((delete variables)::\{1 \$va \$sc \$cl\}) is defined by the rules

\[(\text{rule ((delete variables)::\{1 sc cl va_s\} var (} \text{sc, cl, va) seq (} \text{va_s} \ \text{abn (} \text{sc, cl, va)} \ \text{abn then then ((value va sc cl) ::); ((type va sc cl) ::)}\]

\[\text{)); ((variable va sc cl) ::); ((delete variables)::\{1 sc cl va_s\});}\]

\[(\text{rule ((delete variables)::\{1 sc cl va_s\} var (} \text{sc, cl) abn (} \text{sc, cl) abn then).}\]

7.4. MPL4: procedures

The MPL4 language [1] is an extension of MPL3 that adds the procedures feature: declarations and calls of procedures, and the exit statement. For simplicity, procedure overloading is prohibited.

The sets of function names and procedure names are disjoint.

The element ($c \ is \ procedure$) specifies procedures. It is defined by the rule

\[(\text{rule (} \text{pr is procedure) var (} \text{pr) abn where (pr is name) then (. \{(procedure pr)\}).}\]

The element ((collect body members)::\{1 \ (va\*) \ c\*\}) is redefined by the extra rules

\[(\text{rule ((collect body members)::\{1 \ (va\*) \ c\*\}) is redefined by the extra rules}\]

\[\text{var (} \text{pr) seq (} \text{tna_s, c_s1, c_s) abn (} \text{pr) abn}\]

\[\text{where ((pr is name) and (not (pr is procedure)))}\]

\[\text{then ((collect member arguments) pr tna_s);}\]

\[\text{(((body pr)) :: (c_s1) ::\{q\}); (((procedure pr)) :: true);}\]

\[\text{(((collect body members)::\{1 c_s\});}\]

\[(\text{rule ((collect body members)::\{1 \ (procedure c_s) \ seq (} \text{c_s) abn then und).}\]

Thus, body members in MPL4 are variables, functions and procedures.

The element ((collect member arguments) \$m \$ta\*) is extended to procedures. Its definition is not changed.
The procedure declaration is defined by the rule:

\textit{(rule (procedure c\_s) seq (c\_s) abn then)}.

The execution of the procedure declaration does not collect information about the declared procedure, since the corresponding actions are performed by the element \textit{((collect body members) $c^\ast$)}.

The exit statement is defined by the rule

\textit{(rule exit abn then (exit: \{type\}):: \{exc\}).}

The element \textit{($v$ is (not admissible function body value))} is redefined by the extra rule

\textit{(rule ((exit: \{type\}):: \{exc\} is (not admissible function body value)) var (v) abn then true)}.

Thus, exceptions initiated by \textit{exit} statements are not admissible in MPL4 when a function body exits.

The procedure call is defined by the rule

\textit{(rule (call pr a\_s) var (pr) seq (a\_s) abn (pr) abn where ((pr is procedure) and ((len a\_s: \{q\}) = \{\{arity pr\}\})) then (let:: \{und, seq\} av\_s, b, cs be ((argument values) a\_s), (body pr av\_s), (scope) in (call level + +); \{\{current scope\}\} := 0); b; (catch:: \{und\} v (if (v is (not admissible procedure body value)) then und); (call level – –); \{\{current scope\}\} := cs; (if v matches (exit: \{type\}):: \{exc\} then true else ((to value) v:: \{q\})))}}.

The element \textit{($v$ is (not admissible procedure body value))} specifies exceptions that are not admissible when a procedure call exits. It is defined by the rule

\textit{(rule ((return: \{type\}, v: \{value\}):: \{exc\} is (not admissible procedure body value)) var (v) abn then true)}.

Thus, exceptions initiated by \textit{return} statements are not admissible in MPL4 when a procedure body exits.

The elements \textit{(body $f$ ($v^\ast$))} and \textit{((create local variables) $f$ $n$ $v^\ast$)} are extended to procedures. Their definitions are not changed.

### 7.5. MPL5: pointers

The MPL5 language [1] is an extension of MPL4 that adds the pointers feature: the pointer types, the operations of pointer content access, variable address access and pointer deletion, statements of pointer content assignment and pointer deletion.
The element \( \$c \) is (pointer value) specifies pointers in MPL5. It is defined by the rule
\[
\text{rule (n::\{pointer\} is (pointer value)) var (n) abn then (y is nat)).}
\]
The element (po is pointer) specifies pointers in states. It is defined by the rule
\[
\text{rule (po is pointer) var (po) abn where (po is (pointer value)) then (. \{\{\text{pointer po}\}\}).}
\]
The element \( \$\text{po is (pointer } \$t\)\) specifies pointers with the given content type. It is defined by the rule
\[
\text{rule (po is (pointer t)) var (po, t) abn where ((po is pointer) and (t is type)) then (. \{\{\text{content type po}\}\} = t::q)).}
\]
The element \( \$\text{t is (pointer type)}\) specifies pointer types. It is defined by the rule
\[
\text{rule ((pointer t) is (pointer type)) var (t) abn then (t is type)).}
\]
The element (c is type) is redefined by the extra rule
\[
\text{rule (c is type) abn then (c is (pointer type))).}
\]
The element \( \$\text{po is} \) defined by the rule
\[
\text{rule x::\{pointer\} var (x) abn where (x is nat) then x::\{pointer\}::\{q\}).}
\]
The element ((content type) \$po) returns the content type of \$po. It is defined by the rule
\[
\text{rule ((content type po) var (po) abn where (po is pointer) then (. \{\{\text{content type po}\}\})).}
\]
The element (type \$po) is defined by the rule
\[
\text{rule (type po) var (po) abn where (po is pointer) then (let t be (. \{\{\text{content type po}\}\} in (pointer t)::\{q\}).}
\]
The pointer content access operation is defined by the rule
\[
\text{rule (* c) var (c) val (c) abn (c, c :: \{\} abn where (c::\{\} is pointer) then (. \{\{\text{content c::\{\}\}\})}).}
\]
The pointer content assignment statement is defined by the rule
\[
\text{rule (* c1 := c2) var (c1, c2) val (c1, c2) abn (c1, c2, c1 :: \{\}, c2 :: \{\}) abn where (c1::\{\} is pointer)}
\]
\[
\text{then (let::\{\text{und, seq} t1, t2 be (. \{\{\text{content type c1::\{\}\})), (type c2::\{\})
\text{in (if (subtype t2 t1) then (((content c1::\{\})::\{q\}) = c2::\{\}::\{q\} else und).}}}
\]
The pointer addition operation is defined by the rule
\[
\text{rule (new (pointer t)) var (t) abn (t) abn where (y is type) then (let po be ((new instance) pointer)
\text{in (((content type po)::\{t::q\}); ((pointer po)::\{true\}; po)).}
\]
The pointer deletion operation is defined by the rule
(rule (delete c) var (c) val (c) abn (c, c::*{c} is pointer) 
then (((content c::*{c}) :=); (((content type) c::*{c}) :=); (((pointer c::*{c}) :=)).
The element (version $na $sc $cl) is defined by the rules
(rule (version va y z) var (va, sc, cl) abn (va, sc, cl) abn
then (if (. ((pointer va sc cl))) then sc
else (if (sc = 0) then und
else (let sc1 be (sc − 1) in (version va sc1 cl))))).
The variable address access operation is defined by the rule
(rule (& va) var (va) abn (va) abn
then (let :: {und, seq} sc, cl be (version va),
   (if (sc::*{q} = 0) then 0 else (current call level))
in (if sc::*{q} then (let po be (. ((pointer va sc cl))) in (. ((content po)))))
else und)).
The variable access is defined by the rule
(rule va var (va) abn (va) abn then (let:: {und} po be (& va) in (. ((content po)))).
The element (type va) is defined by the rule
(rule va var (va) abn (va) abn
then (let:: {und} po be (& va) in (. ((content type) po)))).
In the case when the first element of the body $c^*$ is a variable declaration, the rule for the element
((collect body members):: {1} (va::*{c} c*) is replaced by the rule
(rule ((collect body members):: {1} (va_s) var (va t) c_s) var (va, t) seq (va_s, c_s)
abn (va, t) abn where ((va is name) and (t is type))
then (let:: {seq} sc, cl be (scope), (if (sc::*{q} = 0) then 0 else (call level))
in (if (. ((pointer va sc cl))) then und
else (let po be ((new instance) pointer)
in (((content type) po)) := t::*{q}); (((pointer po)) := true);
((collect body members):: {1} (va_s va) c_s)))).
The variable assignment is defined by the rule
(rule (va \:= c) var (va, c) abn (va, c, c::*{c}) abn
then (let:: {und, seq} sc, cl, po, t1, t2
be (version va), (if (sc::*{q} = 0) then 0 else (call level)), (. ((pointer va sc cl))),
(. (((content type) po))),(type c::*{c})
in (if (subtype t2 t1) then (((content po)) := c::*{c}::*{q}) else und)).
The element (\((\text{delete variables})::\{1\} \text{\$var\$ } \text{\$sc\$ } \text{\$cl\$}\)) is defined by the rules
\[
\text{(rule (\((\text{delete variables})::\{1\} \text{\$sc\$ } \text{\$cl\$ } \text{\$va\$ } \text{\$va\$ } \text{\$s\$}) \text{\var\$(\text{sc, cl, va}) \text{\seq\$} \text{\va\$s\$}) \text{\abn\$(\text{sc, cl, va}) \text{\abn then (\{\text{pointer va sc cl}\}) eq\$}; (\((\text{delete variables})::\{1\} \text{\$sc\$ } \text{\$cl\$ } \text{\$va\$s\$})\));
\text{(rule (\((\text{delete variables})::\{1\} \text{\$sc\$ } \text{\$cl\$}) \text{\var\$(\text{sc, cl}) \text{\abn then}.\text{\abn then}}\)).
\]

### 7.6. MPL6: jump statements

The MPL6 language [1] is an extension of MPL5 that adds the jump statements feature: break statement, continue statement, goto statement and labelled statement.

The element (\(e \text{\ is label}\)) specifies labels. It is defined by the rule
\[
\text{(rule (\(e \text{\ is label}\) \text{\var\$(e) \text{\abn then (x is normal)}\).}
\]

The break statement is defined by the rule
\[
\text{(rule break \text{\abn then (break: } \{\text{type}\}\text{:: } \{\text{exc}\}).}
\]

The continue statement is defined by the rule
\[
\text{(rule continue \text{\abn then (continue: } \{\text{type}\}\text{:: } \{\text{exc}\}).}
\]

The goto statement is defined by the rule
\[
\text{(rule (\text{\var\$(l) \text{\abn where (l is label)} \text{\then (goto: } \{\text{type}\}, l: \{\text{label}\}\text{:: } \{\text{exc}\}).}
\]

The element (\(\$v \text{\ is (not admissible function body value)}\)) is redefined by the extra rules
\[
\text{(rule (break: } \{\text{type}\}\text{:: } \{\text{exc}\} \text{\is (not admissible function body value)} \text{\abn then true});
\text{(rule (continue: } \{\text{type}\}\text{\:: } \{\text{exc}\} \text{\is (not admissible function body value)} \text{\abn then true});
\text{(rule (\text{\var\$(l) \text{\abn then (l is label})\).}
\]

The element (\(\$v \text{\ is (not admissible procedure body value)}\)) is redefined by the extra rules
\[
\text{(rule (break: } \{\text{type}\}\text{:: } \{\text{exc}\} \text{\is (not admissible procedure body value)} \text{\abn then true});
\text{(rule (continue: } \{\text{type}\}\text{\:: } \{\text{exc}\} \text{\is (not admissible procedure body value)} \text{\abn then true});
\text{(rule (\text{\var\$(l) \text{\abn then (l is label})\).}
\]

Thus, exceptions initiated by \text{break, continue and goto} statements are not admissible in MPL6 when a function or procedure body exits.

The label statement is defined by the rule
\[
\text{(rule (\text{label l}) \text{\var\$(l) \text{\abn where (l is label)}
\then (catch v (if v matches (goto: } \{\text{type}, l1: \{\text{label}\}\text{:: } \{\text{exc}\} \text{\var\$(l1)\))}
\]

\]}
where \(((l_1\ is\ label)\ and\ \(l_1::\{q\} = l::\{q\}\))\ then\ else\ v::\{q\})).\)

The block statement is defined by the rule

\((\text{rule (block}\ c_s)\ \text{seq} (c_s)\ \text{abn})\)
\n\then\ (\text{enter\ block});
\n\(\text{(let::\{und,\ seq\}\ va_s,\ l_s\ be}\ ((\text{collect\ body\ members})\ c_s),\ ((\text{collect\ labels})\ c_s)\)
\nin\ c_s;\ ((\text{catch\ goto})\ (l_s)\ c_s);\)
\n\(\text{(catch::\{und\}\ v\ ((\text{exit\ block})\ va_s);\ v::\{q\})).}\)

The element \(((\text{collect\ labels})\ \$c^*)\) collects labels from the label statements occurring in \$c^*. It is defined by the rules

\((\text{rule (collect\ labels)}\ (\text{label}\ l)\ c_s\ \text{var}\ (l)\ \text{seq}(c_s)\ \text{abn}\)
\n\where\ (l\ is\ label)\ then\ (l::\{q\}.\ +\ ((\text{collect\ labels})\ c_s)));\)
\n\(\text{(rule (collect\ labels)\ c}_{c_s}\ \text{var}\ (c)\ \text{seq}(c_s)\ \text{then}\ ((\text{collect\ labels})\ c_s));}\)
\n\(\text{(rule (collect\ labels)}\ \text{then}\ ()).}\)

The element \(((\text{catch\ goto})\ (\$l^*)\ \$c^*)\) catches the exceptions initiated by \text{goto} statements when the current block exits. It is defined by the rule

\((\text{rule (catch\ goto)}\ (l_s)\ c_s\ \text{seq}(l_s,\ c_s)\ \text{abn})\)
\n\then\ (catch\ v)
\n\(\text{(if}\ v\ \text{matches}\ (\text{goto:}\{\text{type},\ l:\{\text{label}\}\}:\{\text{exc}\}\ \text{var}(l)\ \text{where}\ (l::\{q\}\ \text{in::}\{\text{set}\}(l_s)::\{q\})\)
\n\then\ v::\{q\};\ c_s;\ ((\text{catch\ goto})\ (l_s)\ c_s\ \text{else}\ v::\{q\})))\).

The while statement is defined by the rules

\((\text{rule (while\ con\ do\ c_s)}\ \text{var}\ (con)\ \text{seq}(c_s)\ \text{exc}(con)\ \text{abn})\)
\n\then\ (\text{while::}\{\text{exc}\}\ \text{con\ do}(\text{block}\ c_s;\ ((\text{delete\ exception})\ \text{continue})));
\n\((\text{((delete\ exception)}\ \text{break))}).\)

\section*{7.7. MPL7: dynamic arrays}

The MPL7 language [1] is an extension of MPL6 that adds the dynamic arrays feature: dynamic array types, the array element access operation and the array element assignment statement.

The element \((\$t\ is\ (\text{dynamic\ array\ type}))\) specifies dynamic array types. It is defined by the rule

\((\text{rule (array} t)\ \text{is (dynamic\ array\ type))\ \text{var}\ (t)\ \text{abn}\ \text{then}\ (t\ is\ type))).\)

The element \((\$t\ is\ (\text{array\ type}))\) specifies array types. It is defined by the rule

\((\text{rule (t\ is\ (array\ type))\ \text{var}\ (t)\ \text{abn}\ \text{then}\ (t\ is\ (dynamic\ array\ type))))).\)

The element \((c\ is\ type)\) is redefined by the extra rule
The array element assignment operation is defined by the rule

\[
\text{(rule (c is type) abn then (c is (array type)))}.
\]

The element \((s e is (dynamic array))\) specifies dynamic arrays. It is defined by the rule

\[
\text{(rule ((v: {content}, t:{type}) :: {{(dynamic array)} is (dynamic array)}) var (v, t) abn where (t is type) then (v is ((array content) t)))}.
\]

The element \((s e is array)\) specifies arrays. It is defined by the rule

\[
\text{(rule (e is array) var (e) abn then (e is (dynamic array)))}.
\]

The element \((s v is ((array content) $t))\) is defined by the rules

\[
\text{(rule ((array content) $t)) var (t) abn then true;}
\]

\[
\text{(rule ((v v_s) is ((array content) t)) var (v, t, v_s) abn where (v is t) then ((v_s) is ((array content) t)))}.
\]

The element \($ar\) is defined by the rule

\[
\text{(rule ar var (ar) abn where (ar is array) then ar: {q}).}
\]

The element \(((element type) $ar)\) returns the element type of \($ar\). It is defined by the rule

\[
\text{(rule ((element type) ar) var (ar) abn where (ar is array) then (ar . {type})).}
\]

The element \((s dar is (array $t))\) specifies dynamic arrays with the given element type. It is defined by the rule

\[
\text{(rule (dar is (array t)) var (dar, t) abn}
\]

\[
\text{where ((dar is (dynamic array)) and (t is type))}
\]

\[
\text{then ((element type) dar) = t:: {q})).}
\]

The element \((type $dar)\) is defined by the rule

\[
\text{(rule (type dar) var (dar) abn where (ar is (dynamic array))}
\]

\[
\text{then (let t be ((element type) dar) in (array t):: {q}))}.}
\]

The array content access operation is defined by the rule

\[
\text{(rule (content c) var (c) val (c) abn (c, c :: {*}) abn where (c :: {*} is array)}
\]

\[
\text{then (c::{*} . {content}))).}
\]

The \(len\) operation for arrays is defined by the rule

\[
\text{(rule (len c) var (c) val (c) abn (c, c :: {*}) abn where (c :: {*} is array)}
\]

\[
\text{then (content c::{*}:: {q})}.}
\]

The array element access operation is defined by the rule

\[
\text{(rule (c1 [ c2 ]) var (c1, c2) val (c1, c2) abn (c1, c1::{*}, c2, c2::{*}) abn}
\]

\[
\text{where ((c1::{*} is array) and (c2::{*} is nat)) then ((content c1::{*}) .. c2::{*})).}
\]

The array element assignment statement is defined by the rule

\[
\text{(rule (c1 [ c2 ] := c3) var (c1,c2,c3) val (c1,c2,c3)}
\]
\[ \text{abn}(c_1, c_2, c_3, c_1::\{\}, c_2::\{\}, c_3::\{\}) \]

where \((c_1::\{\} \text{ is (dynamic array)}) \text{ and } (c_2::\{\} \text{ is nat})\)

then \((\text{let}::\{\text{und}, \text{seq}\} t_1, t_2 \text{ be } ((\text{element type}) c_1::\{\}). (\text{type} c_3::\{\}))\)

in \((\text{if} (\text{subtype} t_2 t_1)\)

then \((c_1::\{\} \cdot \{\text{content}\} := ((\text{content} c_1::\{\}) \cdot c_2::\{\} := c_3::\{\}::\{q\})\)

else und))).

### 7.8. MPL8: static arrays

The MPL8 language [1] is an extension of MPL7 that adds the static arrays feature: static array types, the array element access operation and the array element assignment statement.

The element \((t \text{ is (static array type)})\) specifies static array types. It is defined by the rule

(rule \(((\text{array} t n) \text{ is (static array type)}) \text{ var } (t) \text{ abn then } ((t \text{ is type}) \text{ and } (n \text{ is nat})))\).

The element \((t \text{ is (array type)})\) is redefined by the extra rule

(rule \((t \text{ is (array type)}) \text{ var } (t) \text{ abn then } (t \text{ is (static array type))})\).

The element \((e \text{ is (static array)})\) specifies dynamic arrays. It is defined by the rule

(rule \(((v::\{\text{content}\}, t::\{\text{type}\}) :: ((\text{static array}) \text{ is (static array)}) \text{ var } (v, t) \text{ abn}\)

where \((t \text{ is type}) \text{ then } (v \text{ is ((array content) } t)))\).

The element \((e \text{ is array})\) is redefined by the extra rule

(rule \((e \text{ is array}) \text{ var } (e) \text{ abn then } (e \text{ is (static array))})\).

The element \((\text{ar} \text{ is (array } t s n))\) specifies static arrays with the given element type and length. It is defined by the rule

(rule \((\text{ar} \text{ is (array } t n) \text{ var } (\text{ar}, t) \text{ abn}\)

where \(((\text{ar} \text{ is (dynamic array)}) \text{ and } (t \text{ is type}))\)

then \(((((\text{element type } \text{ ar}) = t::\{q\}) \text{ and } ((\text{len } \text{ ar} \cdot \{\text{content}\}) = n)))\).

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

(rule \((\text{type } \text{ ar}) \text{ var } (\text{ar}) \text{ abn} \text{ where } (\text{ar} \text{ is (static array)))\)

then \((\text{let}::\{\text{seq}\} t, n \text{ be } ((\text{element type } \text{ ar}), (\text{len } \text{ ar}) \text{ in (array } t n)::\{q\})))\).

The array element assignment statement is redefined by the extra rule

(rule \((c_1 [c_2] := c_3) \text{ var } (c_1, c_2, c_3) \text{ val } (c_1, c_2, c_3)\)

\[ \text{abn}(c_1, c_2, c_3, c_1 :: \{\}, c_2 :: \{\}, c_3 :: \{\}) \text{ abn} \]

where \(((c_1 :: \{\} \text{ is (static array)}) \text{ and } (c_2 :: \{\} \text{ is nat}) \text{ and} \]

\[ (c_2 :: \{\} \leq (\text{len } c_1 :: \{\})) \]
then (let:: {und, seq} t1, t2 be ((element type) c1:: {*}), (type c3:: {*})
in (if (subtype t2 t1)
  then (c1:: {*} . {content} := ((content c1:: {*}).. c2:: {*} := c3:: {*}:: {q}))
  else und))).

7.9. MPL9: structures

The MPL9 language [1] is an extension of MPL8 that adds the structures feature: the structure types, the structure field access operation, structure declarations, and the structure field assignment statement.

The element ((collect body members 1) (va*) c*) is redefined by the extra rules

(rule ((collect body members 1) (structure st (tna_s)) var (st) seq (tna_s, c_s) abn (st)
  abn where ((st is name) and (not (st is (structure type))))
then (((declare fields) st tfi_s); (((structure type) st) := true));
  ((collect body members 1) c_s));

(rule ((collect body members 1) (structure st (tna_s)) var (st) seq (tna_s, c_s)
  abn (st) abn then und).

Thus, body members in MPL8 are variables, functions, procedures and structure types.

The element ((declare fields) $st $tna*) declares the fields of $st. It is defined by the rules

(rule ((declare fields) $st $tna*)
  where ((na is name) and (t is type))
then (((type na st) := t:: {q}); (((fields na st)) := true); ((declare fields) $st tna_s);
(rule ((declare fields) st) var (st) abn then).

The structure declaration is defined by the rule

(rule (structure st (tna_s)) var (st) seq (tna_s) abn then).

The execution of the structure declaration does not collect information about the declared structure type, since the corresponding actions are performed by the element ((collect body members) $c*).

The element ($na is (structure type)) specifies structure types. It is defined by the rule

(rule (na is (structure type)) var (na) abn where (na is normal)
  then (, (((structure type) na)))).

The element (c is type) is redefined by the extra rule

(rule (c is type) abn then (c is (structure type))).

The element (fields $st) returns the sequence of fields of $st. It is defined by the rule

(rule (fields st) var (st) abn where (st is (structure type)))
The structure field access operation is defined by the rule
\[
\text{(select } fi \text{ wrt } (v. \{\text{field } fi \text{ st}\}) \text{ var } (v, fi) \text{ abn}).
\]
The element \((\text{type } \$fi \text{ st})\) checks that \(\$fi\) is a field of \(\$st\). It is defined by the rule
\[
\text{(rule } (fi \text{ is } (field st)) \text{ var } (fi, st) \text{ abn where } (st \text{ is } (structure type))
\text{ then } (. \{\text{field } fi \text{ st}\})).
\]
The element \((\text{type } \$fi \text{ st})\) returns the type of the field \(\$fi\) of \(\$st\). It is defined by the rule
\[
\text{(rule } (\text{type } fi \text{ st}) \text{ var } (fi, st) \text{ abn where } (st \text{ is } (structure type)) \text{ then } (. \{\text{type } fi \text{ st}\})).
\]
The element \((\text{str is structure})\) specifies structures. It is defined by the rule
\[
\text{(rule } ((\text{co:} \{\text{content}\}, \text{st:} \{\text{type}\}) :: \{\text{structure} \text{ is } \text{ structure}\} \text{ var } (co, st) \text{ abn}
\text{ where } (st \text{ is } (structure type)) \text{ then } (co \text{ is } (structure content) st))).
\]
The element \((\text{se is } (structure content) \text{ st})\) is defined by the rules
\[
\text{(rule } () \text{ is } (structure content) st) \text{ var } (st) \text{ abn then true};
\text{(rule } ((v: \{fi\} e_s) \text{ is } (structure content) st) \text{ var } (v, fi, st, e_s) \text{ abn}
\text{ where } ((fi \text{ is } (field st)) \text{ and } (let t be } (\text{type } fi \text{ st}) \text{ in } (v \text{ is } t))
\text{ then } ((e_s) \text{ is } (structure content st))).
\]
The element \((\text{str is st})\) specifies structures with the given type. It is defined by the rule
\[
\text{(rule } (\text{str is st}) \text{ var } (str, st) \text{ abn}
\text{ where } ((\text{str is structure}) \text{ and } (st \text{ is } (structure type)))
\text{ then } ((\text{str } . \{\text{type}\}) = st : : \{q\})).
\]
The element \(\text{str is defined by the rule}
\[
\text{(rule } str \text{ var } (str) \text{ abn where } (str \text{ is structure}) \text{ then } str : : \{q\}).
\]
The element \((\text{type } \text{ str})\) is defined by the rule
\[
\text{(rule } (\text{type } str) \text{ var } (str) \text{ abn where } (str \text{ is structure}) \text{ then } (str . \{\text{type}\})).
\]
The element \((\text{fields } \text{ str})\) returns the sequence of fields of \(\text{str}\). It is defined by the rule
\[
\text{(rule } (\text{fields } str) \text{ var } (str) \text{ abn where } (c : : \{\ast\} \text{ is structure})
\text{ then } (let st be (type } str \text{) in } (\text{fields } st))).
\]
The element \((\text{fi is } (field } str))\) checks that \(\text{fi}\) is a field of \(\text{str}\). It is defined by the rule
\[
\text{(rule } (\text{fi is } (field } str)) \text{ var } (fi, str) \text{ abn where } (str \text{ is structure})
\text{ then } (let st be (type } str \text{) in } (fi \text{ is } (field } st))).
\]
The element \((\text{type } \text{ fi } \text{ str})\) returns the type of the field \(\text{fi}\) of \(\text{str}\). It is defined by the rule
\[
\text{(rule } (\text{type } fi \text{ str}) \text{ var } (fi, str) \text{ abn where } (str \text{ is structure})
\text{ then } (let st be (type } \text{ str in } (type } fi \text{ str))).
\]
The structure field access operation is defined by the rule
\[
\text{(rule } (c . fi) \text{ var } (c, fi) \text{ val } (c) \text{ abn } (c, c : : \{\ast\}) \text{ abn}
\text{ }}
where \((c :: {*} \text{ is structure}) \text{ and } (fi is (field c :: {*}))\)
then \(((\text{str} . \{\text{content}\}) . \{fi\}))\).

The structure field assignment statement is defined by the rule
\((\text{rule } (c1 \setminus \text{. } fi := c2) \text{ var } (c1, fi, c2) \text{ val } (c1, c2) \text{ abn } (c1, c2, c1 :: {*} , c2 :: {*} )) \text{ abn}
\text{ where } ((c1 :: {*} \text{ is structure}) \text{ and } (fi is (field c1 :: {*} )))\)
then \((\text{let} : \{\text{und}, \text{ seq}\} t1, t2 \text{ be } (\text{type } fi c1 :: {*} ), (\text{type } c2 :: {*} ))
in \text{ if } (\text{subtype } t2 t1)
\text{ then } (c1 :: {*} . \{\text{content}\} := ((c1 :: {*} . \{\text{content}\}).fi := c2 :: {*} :: \{q\}))
\text{ else und})).\)

8. Conclusion

In the paper the notion of the conceptual operational semantics of a programming language has been proposed. The conceptual operational semantics of a programming language is an operational semantics of the programming language in terms of its conceptual model [3]. The special kind of CTSs, operational CTSs, oriented to specification of conceptual operational semantics of programming languages has been proposed, the language CTSL has been extended to this kind of CTSs, and the technique of the use of the extended CTSL as a domain-specific language for specification of conceptual operational semantics has been presented. We have conducted the incremental development of the conceptual operational semantics for the family of sample programming languages to illustrate this technique.

There is only one more approach which, like our approach, can specify both the structural and dynamic parts of the operational semantics of a programming language in quite general unified way. This approach is based on abstract state machines (ASMs) [4]. ASMs are the special kind of transition systems in which states are algebraic systems.

The key features of our approach in comparison with the approach based on ASMs are as follows.

The instantiation semantics and, in particular, states are directly described in CTSs in ontological terms whereas its conceptual structure can be only modelled by the appropriate choice of symbols of the signature of an algebraic system.

The transition relation in ASMs is built with the finite set of algebraic operations [5]. The transition relation in operational CTSs is based on the pattern matching on the conceptual structure of states.
The set of predefined executable elements of the CTSL language have analogues for the algebraic operations used in sequential ASMs, and also includes the elements for parsing the conceptual state structure.

The languages of executable specifications of abstract state machines AsmL [6] and XasM [7] are general-purpose languages of specification of discrete dynamic systems. They are not domain-specific languages oriented to development of operational semantics of programming languages in contrast to the CTSL language.

At present, our technique is applied to only the sequential fragments of programming languages. We plan to extend it to the concurrent fragments of programming languages.

References