A Proposal for the Data Type Part of E-LOTOS Applicable to the Formal Description of OSI and ODP Standards*

Draft of 1995/12/14

Mihaela Sighireanu Hubert Garavel
INRIA Rhône-Alpes
VERIMAG — Miniparc-ZIRST
rue Lavoisier
38330 MONTBONNOT ST MARTIN
FRANCE

December 1995

Abstract

This paper proposes an abstract syntax, a static and a dynamic semantics for the data type language of ELOTOS. This language is monomorphic, explicitly typed and allows overloading of operations. Furthermore, we introduce several features (in/out parameters, exceptions) which make this language compatible with interface language IDL. The static semantics of this language is given both informally and formally, using attribute grammars. The dynamic semantics is a denotational one.

*This work has been supported in part by the European Commission, under project ISC-CAN-65 "EUCALYPTUS-2: A European/Canadian LOTOS Protocol Tool Set".

Contents

1 Introduction

This papers presents static and dynamic semantics for the core of data type language proposed for E-LOTOS. This language is based on the first approach presented in document [JGL+95] and we add new features as extended selectors, updaters. Also, in the aim of compatibility with anothers ODP languages, principally IDL, we have introduced in our proposition function with in/out parameters, and an implicit "void" type (IDL, C, C++ — void type, Pascal, Ada — procedure).

So, we propose here a functional (not algebraic) data type language with extensions to support others ODL languages.

This paper is not concerned with the integration of data language in behavioral part.

- In Section 4 we propose an abstract syntax of language.
- In Section 5 we explain how some constructions can be seen like shorthands of "basic" constructions.
- In Section 6 we give an informal static semantics for proposed language.
- In Section 7 we give a denotational dynamic semantics for proposed language.

2 Motivation

The language we consider here is monomorphic, explicitly typed, and allows overloaded operations (functions and constructors).

A type expression is nothing but a type identifiers (possibly an identifier of a predefined type, such as **bool**, etc.).

TE ::= T *type expression*

This approach is backward compatible with LOTOS, and differs from second proposition made in document [JGL+95] which considers also anonymous records as type expressions.

We have introduced attributes **in** and **out** for parameters of functions in the aim to support ODP languages like IDL. However, we left for further studies the integration of **inout** attribute.

In the same aim, the proposed syntax for function definition allows specification of exceptions which could be raised by this function.

3 Vocabulary

We present here some notions used in this paper.

- type: correspond of sort of LOTOS, it is a name for values domain;
- constructor: operations which are need to build up data objects;
- function: operations which maps data objects to another data object;
- exception: a named error raised by a program.

4 Abstract syntax

We present here the full abstract grammar, including derived constructions. Non-primitive constructions are indicated by $\star$ in grammar presentation and their derivation is described in section 5.

4.1 Notations

The classes of *terminals identifiers* of the data type language are:

identifier domain	*meaning*	*abbreviations*
SCon	special constants	K
Var	variable identifiers	V
Typ	type identifiers	T
Con	constructor identifiers	C
Fun	function identifiers	F
Exc	exception identifiers	X

We will also define the following classes of non-terminal symbols:

symbol domain	*meaning*	*abbreviations*
Decl	declaration	D
Pat	patterns	P
Match	match expression	M
Exp	expression	E
Instr	instructions	I

4.2 Syntax

4.2.1 Declarations

$(D1)$ D ::= **type** T **is** T' **endtype** *type synonym*

$(D2)$ | **type** T **is** *type declaration*
$C_1(V_1^1 : T_1^1, ..., V_{n_1}^1 : T_{n_1}^1)$
...
$C_p(V_1^p : T_1^p, ..., V_{n_p}^p : T_{n_p}^p)$
endtype$[T]$

$(D3)$ | **function** $F([\textbf{in} \mid \textbf{out}]V_1 : T_1, ..., [\textbf{in} \mid \textbf{out}]V_n : T_n)[\ \textbf{:}\ T]$ *function declaration*
$[\textbf{raises}\ X_1, ..., X_p]$ **is**
I
endfunc$[F]$

$(D4)$ $\star$ | **function** $F(V_1 : T_1, ..., V_n : T_n)\ \textbf{:}\ T$ *simple function declaration*
$[\textbf{raises}\ X_1, ..., X_p]$ **is**
E
endfunc$[F]$

$(D5)$	|	**exception** X	*exception declaration*
$(D6)$	|	$D_1 D_2$	*sequence of declaration*

Syntactic restrictions:

1. In $(D2)$, $p \geq 0$; if $p = 0$ then T could be considered as an external type.
2. In $(D2)$, as in LOTOS, constructors and functions can be declared to be infixed.
3. In $(D2)$, each constructor C_i, field names must be pairwise distinct
$$\forall k, l \in \{1, ...n_i\} \quad V_k^i \neq V_l^i$$
4. In $(D2)$, all constructors of a type T, fields (V_i^j) having the same name must have the same type:
$$\forall i, j \in \{1, ...p\} \quad \forall k \in \{1, ...n_i\} \quad \forall l \in \{1, ...n_j\} X_k^i = X_l^j \Rightarrow T_k^i = T_l^j$$
5. In $(D3)$ and $(D4)$, formal parameter names (V_i) must be pairwise distinct:
$$\forall i, j \in \{1, ...n\} \quad V_i \neq V_j$$
6. In $(D3)$, if the sort of formal parameter is not specified (i.e. "**in**" or "**out**"), the default value is "**in**".
7. In $(D3)$, if the type of result is not specified (i.e. it is "**void**") a least a formal parameter must be an "**out**" parameter:
$$\{\ i \mid \textbf{out}\ V_i : T_i\} \neq \emptyset$$
8. In $(D3)$ and $(D4)$, names of raised exceptions (X_i) must be pairwise different:
$$\forall i, j \in \{1, ...p\} \quad X_i \neq X_j$$
9. In $(D3)$ and $(D4)$, the instruction I or the expressions E may contain F calls.

4.2.2 Patterns

$(P1)$	P ::=	K		*special constant*
$(P2)$	|	$V : T$		*variable*
$(P3)$	|	**any** T		
$(P4)$	|	$C(V_1 := P_1, ..., V_n := P_n[, \bullet\bullet\bullet])$		*value construction*
$(P5)$	$\star$ |	$C(P_1, ..., P_n[, \bullet\bullet\bullet])$		*simple value construction*
$(P6)$	|	P_0 **of** T		*typed pattern*

Syntactic restrictions:

1. In $(P4)$, no pattern may bind the same variable V twice.
2. In $(P4)$, if formal parameters are specified, parameters not explicitely specified are denoted by "$\bullet\bullet\bullet$".
3. In $(P5)$, if formal parameters aren't specified, "$\bullet\bullet\bullet$" denotes the tail of (ordered) list of parameters (i.e. parameters $> n$).

4.2.3 Match expression

$(M1)$ M ::= E **::** P

$(M2)$ | M_1 **when** E

$(M3)$ | M_1 **and** M_2

$(M4)$ | M_1 **or** M_2

4.2.4 Expressions

$(E1)$ E ::= K *constant denotation*

$(E2)$ | V *value variable*

$(E3)$ | $C(V_1{:=}E_1, ..., V_n{:=}E_n)$ *constructor application*

$(E4)$ $\star$ | $C(E_1, ..., E_n)$ *simple constructor application*

$(E5)$ | **eval** *evaluation of functions*
$[P_1{:=}]\ F_1(V_1^1{:=}E_1^1 | P_1^1, ..., V_{n_1}^1{:=}E_{n_1}^1 | P_{n_1}^1)$
...
$[P_p{:=}]\ F_p(V_1^p{:=}E_1^p | P_1^p, ..., V_{n_p}^p{:=}E_{n_p}^p | P_{n_p}^p)$
in E_0

$(E6)$ $\star$ | $F(V_1{:=}E_1, ..., V_n{:=}E_n)$ *function application*

$(E7)$ $\star$ | $F(E_1, ..., E_n)$ *function application*

$(E8)$ | M *match expression*

$(E9)$ | **case** *general case expression*
$M_0 \rightarrow E_0$
...
$M_n \rightarrow E_n$
$\star$ [**otherwise** E_{n+1}]
endcase

$(E10)$ $\star$ | **case** E' **in** *usual case expression*
$P_1^0, ..., P_{n_0}^0$[**when** E_0] $\rightarrow E_0'$
...
$P_1^p, ..., P_{n_p}^p$[**when** E_p] $\rightarrow E_p'$
[**otherwise** E_{p+1}']
endcase

$(E11)$ $\star$ | **if** E_0 **then** E_0' *conditional expression*
[**elsif** E_1 **then** E_1']
...
[**elsif** E_n **then** E_n']
[**else** E_{n+1}']
endif

$(E12)$	$\star$	|	E_0 **andthen** E_1	*logical expression*
$(E13)$	$\star$	|	E_0 **orelse** E_1	*logical expression*
$(E14)$	$\star$	|	**let** P_0**:=**$E_0, ..., P_n$**:=**E_n **in** E	*local binding*
$(E15)$		|	$E.V_0$	*select expression*
$(E16)$		|	$E_0.\{V_1$**:=**$E_1, ..., V_n$**:=**$E_n\}$	*update expression*
$(E17)$		|	E_0 **=** E_1	*equality expression*
$(E18)$	$\star$	|	E_0 <> E_1	*non-equality expression*
$(E19)$		|	**raise** X	*raise exception*
$(E20)$		|	**trap** E_0 **handle** $X_1 \rightarrow E_1$ ... $X_n \rightarrow E_n$ **endtrap**	*handle exception*
$(E21)$		|	E_0 **of** T	*layered value*

Syntax restrictions:

1. In $(E3) - -(E7)$, constructors and functions without arguments, or constructors and functions infixed, are included by these rules.
2. In $(E3)$, $(E6)$, and $(E15)$, no variable V_i can be bound twice.
3. In $(E5)$, actual parameters corresponding to "**in**" formal parameters must be expressions values. Actual parameters corresponding to "**out**" formal parameters must be pattern expressions.

Note: The proposed grammar differs from this used in document [SG95] by the following points:

- the syntax used for selector expression becomes "$E.X_0$" in place of "**select** X_0 **in** E";
- the syntax of tester expression becomes "E **::** P" in place of "E **match** P".

4.2.5 Instructions

$(I1)$	I ::=		**return** $[E]$ [**with** $V_1 := E_1, ..., V_n := E_n$]	
$(I2)$		|	**eval** $[P_0$**:=**$]\ F_0(V_1^0$**:=**$E_1^0 \mid P_1^0, ..., V_{n_0}^0$**:=**$E_{n_0}^0 \mid P_{n_0}^0)$ ... $[P_p$**:=**$]\ F_p(V_1^p$**:=**$E_1^p \mid P_1^p, ..., V_{n_p}^p$**:=**$E_{n_p}^p \mid P_{n_p}^p)$ **in** I_0	*evaluation of function*
$(I3)$		|	**case** $M_0 \rightarrow I_0$ ... $M_n \rightarrow I_n$ $\star$ [**otherwise** I_{n+1}] **endcase**	*general case instruction*

$(I4)$ $\star$ | **case** E' **in** *usual case instruction*
$P_1^0, ..., P_{n_0}^0$[**when** E_0] $\rightarrow I_0$
...
$P_1^p, ..., P_{n_p}^p$[**when** E_p] $\rightarrow I_p$
[**otherwise** I_{p+1}]
endcase

$(I5)$ $\star$ | **if** E_0 **then** I_0 *conditional instruction*
[**elsif** E_1 **then** I_1]
...
[**elsif** E_n **then** I_n]
[**else** I_{n+1}]
endif

$(I6)$ $\star$ | **let** P_0**:=**$E_0, ..., P_n$**:=**E_n **in** I_0 *local binding instruction*

$(I7)$ | **trap** I_0 **handle** *handle exception instruction*
$X_1 \rightarrow I_1$
...
$X_n \rightarrow I_n$
endtrap

$(I8)$ | $F(V_1$**:=**$E_1 | P_1, ..., V_n$**:=**$E_n | P_n)$ *function application instruction*

Syntax restrictions:

1. In $(I1)$, $(I2)$, and $(I8)$, no variable V_i or V_i^j can be bound twice.
2. In $(I2)$ and $(I8)$, actual parameters of functions to "**in**" formal parameters must be expressions values. Actual parameters corresponding to "**out**" formal parameters must be pattern expressions.
3. In $(I2)$, if the type of result of function is "**void**", no pattern P_i must be specified. Otherwise, the pattern P_i must be specified for the result binding.

5 Shorthands

5.1 Simple function declaration

$(D4)$ **function** $F(V_1 : T_1, ..., V_n : T_n)$
: T **is** E
endfunc

$\equiv$ **function** $F($**in**$V_1 : T_1, ...,$ **in**$V_n : T_n)$
: T **is**
return E
endfunc

5.2 Simple value construction

$(P5)$ $C(P_1, ..., P_n) \equiv C(V_1$**:=**$P_1, ..., V_n$**:=**$P_n)$

where V_i is the ith formal parameter of constructor C.

5.3 Simple constructor application

$(E4)\quad C(E_1,...,E_n) \quad\equiv\quad C(V_1{:=}E_1,...,V_n{:=}E_n)$

where V_i is the ith formal parameter of constructor C.

5.4 Function application

$(E6-7)\quad F([V_1{:=}]E_1,...,[V_n{:=}]E_n) \quad\equiv\quad \textbf{eval } V:T = F([V_1{:=}]E_1,...,[V_n{:=}]E_n) \textbf{ in } V$

where T is the type of result of function F.

5.5 Otherwise clause

$(E9),(E10),$
$(I3),(I4)$

$$\textbf{otherwise } E \mid I \quad\equiv\quad 0 :: 0 \rightarrow E \mid I$$

5.6 Usual case expression

$(E10)$ **case** E' **in**
$P_1^0,...,P_{n_0}^0[\textbf{when } E_0] \rightarrow E_0'$
...
$P_1^p,...,P_{n_p}^p[\textbf{when } E_p] \rightarrow E_p'$
$[\textbf{otherwise } E_{p+1}']$
endcase

$\equiv$ **case**
$E' :: P_1^0$ **or**...**or** $E' :: P_{n_0}^0[\textbf{when } E_0] \rightarrow E_0'$
...
$E' :: P_1^p$ **or**...**or** $E' :: P_{n_p}^p[\textbf{when } E_p] \rightarrow E_p'$
$[\textbf{otherwise } E_{p+1}']$
endcase

5.7 Conditional expression

$(E11)$ **if** E_0 **then** E_0'
elsif E_1 **then** E_1'
...
elsif E_n **then** E_n'
else E_{n+1}'
endif

$\equiv$ **case**
$E_0 ::$ **true** $\rightarrow E_0'$
$E_1 ::$ **true** $\rightarrow E_1'$
...
$E_n ::$ **true** $\rightarrow E_n'$
otherwise $\rightarrow E_{n+1}'$
endcase

5.8 Logical expressions

$(E12)$ E_0 **andthen** E_1 $\equiv$ **if** E_0 **then** E_1 **else false endif**

$(E13)$ E_0 **orelse** E_1 $\equiv$ **if** E_0 **then true else** E_1 **endif**

5.9 Local binding expression

$(E14)$ **let** P_0**:=**$E_0, ..., P_n$**:=**E_n **in** E_0 $\equiv$ **case** E_0 **::** P_0 **and** ... **and** E_n **::** $P_n \rightarrow E_0$ **endcase**

5.10 Non-equality

$(E18)$ $E_0 <> E_1 \equiv$ **not** $(E_0 = E_1)$

where **not** is the boolean function of (pervasive) booleans.

5.11 Tester function

E_0 **match** $P_0 \equiv E_0$ **::** P_0

This function was used in the document [SG95] for testers.

5.12 Usual case instruction

$(I4)$ **case** E' **in**
$P_1^0, ..., P_{n_0}^0$[**when** E_0] $\rightarrow I_0$
...
$P_1^p, ..., P_{n_p}^p$[**when** E_p] $\rightarrow I_p$
[**otherwise** E'_{p+1}]
endcase $\equiv$ **case**
E' **::** P_1^0 **or**...**or** E' **::** $P_{n_0}^0$[**when** E_0] $\rightarrow I_0$
...
E' **::** P_1^p **or**...**or** E' **::** $P_{n_p}^p$[**when** E_p] $\rightarrow I_p$
[**otherwise** I_{p+1}]
endcase

5.13 Conditional instruction

(*I*5) **if** E_0 **then** I_0
elsif E_1 **then** I_1
...
elsif E_n **then** I_n
else I_{n+1}
endif $\equiv$ **case**
E_0 **::** **true** $\rightarrow I_0$
E_1 **::** **true** $\rightarrow I_1$
...
E_n **::** **true** $\rightarrow I_n$
otherwise $\rightarrow I_{n+1}$
endcase

5.14 Local binding instruction

(*I*6) **let** P_0**:=**$E_0, ..., P_n$**:=**E_n **in** I_0 $\equiv$ **case**
E_0 **::** P_0 **and** ... **and** E_n **::** $P_n \rightarrow I_0$
endcase

6 Static semantics

We give in this section a (informal) static semantics for proposed syntax. A formal definition is presented in Annex A

6.1 Static semantics (1): binding

- All occurrences of type identifiers in pattern or expressions are *use*-occurrences[1] that shall be bound to the definition of the corresponding type.
- All occurrences of constructor identifiers in pattern or expressions are *use*-occurrences that shall be bound to the definition of the corresponding constructor (possibly with some overloading resolution).
- All occurrences of function identifiers in pattern, in expressions, or in instructions are *use*-occurrences that shall be bound to the definition of the corresponding function (possibly with some overloading resolution).
- Occurrences of variable identifier in pattern, in expressions, or in **instructions** can be either *use*-occurrences or *def*-occurrences[2], depending on the context.

  More precisely, occurrence of V in left of "**:=**" sign is a *bind*-occurrence, but occurrence of V in right of "**:=**" is a *use*-occurrence.
- Occurrences of exception identifiers in expressions or commands are *use*-occurrences that shall be bound to the definition of the corresponding exception.

[1] also called *place-marking occurrences* in [ISO88]
[2] also called *binding occurrences* in [ISO88]

More precisely, we define the following mapping:

$$vars : \mathrm{Pat} \cup \mathrm{Match} \rightarrow \mathrm{Var}$$

which give the set of variables declared in each syntactic expression, as follows:

$$\begin{array}{rcl}
vars\{K\} & = & \emptyset \\
vars\{V : T\} & = & \{V\} \\
vars\{\textbf{any } T\} & = & \emptyset \\
vars\{C([V_1\textbf{:=}]P_1, ..., [V_n\textbf{:=}]P_n[, \textbf{...}])\} & = & vars\{P_1\} \uplus ... \uplus vars\{P_n\} \\
vars\{P_0 \textbf{ of } T\} & = & vars\{P_0\}
\end{array}$$

$$\begin{array}{rcl}
vars\{E \textbf{ :: } P\} & = & vars\{P\} \\
vars\{M_1 \textbf{when } E\} & = & vars\{M_1\} \\
vars\{M_1 \textbf{ and } M_2\} & = & vars\{M_1\} \uplus vars\{M_2\} \\
vars\{M_1 \textbf{ or } M_2\} & = & vars\{M_1\} \textbf{ iff } vars\{M_1\} = vars\{M_2\}
\end{array}$$

where the operator "$\uplus$" denotes the union of sets the intersection of which is empty. If the intersection is not empty (which could occur in a pattern expressions such as "$C(X : T, X : T)$" for instance), this is a static semantics error. Note that if "**...**" is used the union can be empty.

For a given pattern expression P_i, the variables of $vars\{P_i\}$ have the following scope:

- These variables are not visible in P_i. For instance, in the following pattern expression "$C(X : T, Y : T, X+Y)$", variables X and Y occurring in "$X+Y$" shall not be bound to the definitions "$X : T$" and "$Y : T$" contained in P_i.
- If P_i occurs in "**eval**" expression (rule ($E5$)), these variables are only visible in E_0. They mask all variables with the same names, possibly defined in enclosing scopes. The same scope for an occurrence of P_i in an "**eval**" instruction.
- If P_i occurs in a "**let**" expression (rule ($E14$)), these variables are only visible in E_0. They mask all variables with the same names, possibly defined in enclosing scopes. The same scope for an occurrence of P_i in a "**let**" instruction.

For a given match expression M_i, the variables of $vars\{M_i\}$ have the following scope:

- If M_i occurs in a match expression M of form M_i**when** E (rule ($M2$)), these variables are visibles in E and in the shell scope.They mask all variables with the same names, possibly defined in enclosing scopes.
- If M_i occurs in a match expression M of form M_i **and** M_j, these variables are not visibles in M_j. The same for a "**or**" match expression.
- If M_i occurs in a expression E of the form "M_i" (rule ($E8$), all variables of M_i are not visible.
- If M_i occurs in a "general" case expression (rule ($E9$)), these variables are only visibles in the correspondent expression E_i. The same scope for an occurrence of M_i in a general case instruction.

6.2 Static semantics (2): typing

6.2.1 Declarations

There are additional type-checking constraints for declarations:

$(D1)$: Type T' must be already declared.

$(D2)$: Types $T_1^1, ..., T_{n_p}^p$ must be already declared or can be T.

$(D3) - (D4)$:

- In $(D3)$, $type\{I\} = T$; if not, error is raised.
- In $(D4)$, $type\{E\} = T$; if not, error is raised.
- In $(D3) - (D4)$, instruction I and expression E, must raised only exceptions belonging to list $X_1, ..., X_p$.
- In $(D3)$, all "**out**" formal parameters of F must be bound by instruction I.

6.2.2 Patterns

The result type of a pattern expression P, noted $type\{P\}$, is defined as follows:

$$
\begin{aligned}
type\{K\} &= type\ of\ constant\ K \\
type\{X : T\} &= T \\
type\{\textbf{any}\ T\} &= T \\
type\{C([V_1{:=}]P_1, ..., [V_n{:=}]P_n[, \bullet\bullet\bullet])\} &= type\ of\ the\ result\ of\ constructor\ C \\
type\{P_0\ \textbf{of}\ T\} &= T
\end{aligned}
$$

There are additional type-checking constraints for pattern expressions:

$(P4)$: In $(P4)$, each $type\{P_i\}$ must be equal to the type declared for V_i formal parameter of constructor C.

$(P5)$: In $(P4)$, each $type\{P_i\}$ must be equal to the type of the i-th argument of constructor C.

$(P5)$: In $(P5)$, one must have $type\{P_0\} = T$.

6.2.3 Match expression

The result type of a match expression M, noted $type\{M\}$, is defined as follows:

$$
\begin{aligned}
type\{E\ ::\ P\} &= bool \\
type\{M_1\textbf{when}\ E\} &= bool \\
type\{M_1\ \textbf{and}\ M_2\} &= bool \\
type\{M_1\ \textbf{or}\ M_2\} &= bool
\end{aligned}
$$

There are one additional type-checking constraint for match expressions:

$(M1)$: In $(M1)$, one must have $type\{E\} = type\{P\}$;

$(M2)$: In $(M2)$, one must have $TypeE = bool$.

6.2.4 Expressions

The result type of a expression E, noted $type\{E\}$, is defined as follows:

$$
\begin{array}{rcl}
type\{K\} & = & type\ of\ constant\ K \\
type\{X\} & = & type\ of\ variable\ X \\
type\{C([V_1{:=}]E_1, ..., [V_n{:=}]E_n)\} & = & type\ of\ the\ result\ of\ constructor\ C \\
type\{\textbf{eval}\ ...\ \textbf{in}\ E_0\} & = & type\{E_0\} \\
type\{F([V_1{:=}]E_1, ..., [V_n{:=}]E_n)\} & = & type\ of\ the\ result\ of function\ F \\
type\{M\} & = & bool \\
type\{\textbf{case}\ M_0 \rightarrow E_0 ... \textbf{endcase}\} & = & type\{E_0\} \\
type\{\textbf{case}\ E'\ \textbf{in} ... \rightarrow E'_0 ... \textbf{endcase}\} & = & type\{E'_0\} \\
type\{\textbf{if}\ E_0\ \textbf{then}\ E'_0 ...\} & = & type\{E'_0\} \\
type\{E_0\ \textbf{andthen}\ E_1\} & = & bool \\
type\{E_0\ \textbf{orelse}\ E_1\} & = & bool \\
type\{\textbf{let}\ P_0 = E_0, ..., P_n = E_n\ \textbf{in}\ E\} & = & type\{E\} \\
type\{E_0.V_0\} & = & type\ of\ label\ V_0\ \ of\ type\ of\ E_0 \\
type\{E_0.\{V_1{:=}E_1, ...\}\} & = & type\{E_0\} \\
type\{E_0 = E_1\} & = & bool \\
type\{E_0 <> E_1\} & = & bool \\
type\{\textbf{raise}\ X\} & = & exn \\
type\{\textbf{trap}\ E_0\ \textbf{handle}\ X_1 \rightarrow E_1 ...\ \textbf{endtrap}\} & = & type\{E_0\} \\
type\{E_0\ \textbf{of}\ T\} & = & T
\end{array}
$$

There are additional type-checking constraints for expressions:

$(E3)$: In $(E3)$, one must have:

- n must be equal with the number of parameters of C (i.e. no currification is allowed).
- Each $type\{E_i\}$ must be equal to the type declared for V_i formal parameter of constructor C.

$(E4)$: In $(E4)$, one must have:

- n must be equal with the number of parameters of C (i.e. no currification is allowed).
- Each $type\{E_i\}$ must be equal to the type of the i-th argument of constructor C.

$(E5)$: In $(E5)$, one must have:

- n_i must be equal with the number of parameters of F_i (i.e. no currification is allowed).
- Each $type\{E_j^i\}$ or $type\{P_j^i\}$ must be equal to the type declared for V_j formal parameter of function F_i.
- Each $type\{P_i\}$ must be equal to the result type of function F_i.

$(E6)$-$(E7)$: The types checking rules are derived from type-checking rules of "**eval**" expression.

$(E9)$: In a general "**case**" expression, one must have $type\{E_0\} = ... = type\{E_n\} = type\{E_{n+1}\}$.

$(E10)$: In a usual "**case**" expression, one must have:

- $type\{E'\} = type\{P_i^j\}$ for all $0 \leq i \leq p$ and $1 \leq j \leq n_i$;
- $type\{E_0\} = ... = type\{E_p\} = type\{E_{p+1}\}$.

$(E11)$: In an "**if**" expression, one must have:

- $type\{E_0\} = type\{E_1\} = ... = type\{E_{n-1}\} = bool$ and
- $type\{E'_0\} = type\{E'_1\} = ... = type\{E'_n\}$.

$(E12)$-$(E13)$: In an expression of the form "E_0 **andthen** E_1" or "E_0 **orelse** E_1", one must have $type\{E_0\} = type\{E_1\} = bool$.

$(E14)$: In a "**let**" expression, for all i, one must have $type\{P_i\} = type\{E_i\}$.

$(E15)$: In a select expression of the form "$E_0.V_0$" V_0 must be a formal parameter for a constructor of $type\{E_0\}$.

$(E16)$: In an update expression, for each V_i the type corresponding to this formal parameter must be equal to $type\{E_i\}$.

$(E17)$-$(E18)$: In an expression of the form "$E_0 = E_1$" or "$E_0 <> E_1$", one must have $type\{E_0\} = type\{E_1\}$.

$(E20)$: In a "**handle**" expression, one must have $type\{E_0\} = type\{E_1\} = ... = type\{E_n\}$.

$(E21)$: In a "layered" expression, one must have $type\{E_0\} = T$.

6.2.5 Instructions

The result type of a instruction I, noted $type\{I\}$, is defined as follows:

$$
\begin{array}{rcl}
type\{\textbf{return } E \ [\textbf{with } ...]\} & = & type\{E\} \\
type\{\textbf{eval } ... \textbf{ in } I_0\} & = & type\{I_0\} \\
type\{\textbf{case } M_1 \rightarrow I_1 ... \textbf{ endcase}\} & = & type\{I_1\} \\
type\{\textbf{if } E_0 \textbf{ then } I'_0 ... \textbf{ endif}\} & = & type\{I'_0\} \\
type\{\textbf{let } ... \textbf{ in } I_0\} & = & type\{I_0\} \\
type\{I_0 \textbf{ handle } X_1 \rightarrow I_1 ...\} & = & type\{I_0\} \\
type\{F(V_1{:=}E_1|P_1, ..., V_n{:=}E_n|P_n)\} & = & type \textit{ of the result of function } F
\end{array}
$$

There are additional type-checking constraints for instructions:

($I1$): In a "**return**" instruction, one must have for each i $type\{V_i\} = type\{E_i\}$.

($I2$): In "**eval**" instruction, one must have:

- n_i must be equal with the number of parameters of F_i (i.e. no currification is allowed).
- Each $type\{E_j^i\}$ or $type\{P_j^i\}$ must be equal to the type declared for V_j formal parameter of function F_i.
- Each $type\{P_i\}$ must be equal to the result type of function F_i.

($I3$)-($I4$): In a "**case**" instruction, one must have $type\{I_1\} = ... = type\{I_n\} = type\{I_{n+1}\}$.

($I5$): In an "**if**" instruction, one must have:

- $type\{E_0\} = type\{E_1\} = ... = type\{E_{n-1}\} = bool$ and
- $type\{I'_0\} = type\{I'_1\} = ... = type\{I'_n\}$.

($I6$): In a "**let**" instruction, for all i, one must have $type\{P_i\} = type\{E_i\}$.

($I7$): In a "**handle**" instruction, one must have $type\{I_0\} = type\{I_1\} = ... = type\{I_n\}$.

($I8$): In function application instruction, one must have:

- n must be equal with the number of parameters of F (i.e. no currification is allowed).
- Each $type\{E_j\}$ or $type\{P_j\}$ must be equal to the type declared for V_j formal parameter of function F.

7 Dynamic semantics

This section provides a formal denotational semantics for the dynamic semantics of proposed language.

The notation is summarized below:

*	multiplication
$\rho[x/i]$	substitution "ρ with x for i"
$< ... >$	sequence formation
$s\downarrow k$	k-th member of the sequence s (1-based)
$s\dagger k$	drop the first k members of the sequence s
$s\S t$	concatenation of sequences s and t
$\#s$	the length of sequence s

7.1 Reduced syntax

Since types are fully dealt with in the static semantics, the dynamic semantics ignore them. The syntax presented is therefore reduced by the following transformations, for the purpose of dynamic semantics:

- All explicit type ascriptions "$: T$" are omitted, and qualifications "**of** T" are also omitted from expressions and patterns.
- Any declaration of the form "**type** T **is** ... **endtype**" is replaced by the list of declaration of corresponding constructors.
- The class of type identifiers "Typ" is omitted.

7.1.1 Declarations

$(D3)$ D ::= **function** $F(\{\textbf{in}|\textbf{out}V\}^*)$ **is**
I
$(D5)$ | **exception** X
$(D6)$ | $D_1 D_2$

7.1.2 Patterns

$(P1)$ P ::= K
$(P2)$ | V
$(P3)$ | **any**
$(P4)$ | $C(V_1{:=}P_1, ..., V_n{:=}P_n[, \bullet\bullet\bullet])$

7.1.3 Match expression

$(M1)$ M ::= E **::** P
$(M2)$ | M_1 **when** E
$(M3)$ | M_1 **and** M_2
$(M4)$ | M_1 **or** M_2

7.1.4 Expressions

$(E1)$ E ::= K
$(E2)$ | V
$(E3)$ | $C(V_1{:=}E_1, ..., V_n{:=}E_n)$
$(E5)$ | **eval**
$[P_1{:=}]\ F_1(V_1^1{:=}E_1^1|P_1^1, ..., V_{n_1}^1{:=}E_{n_1}^1|P_{n_1}^1)$
...
$[P_p{:=}]\ F_p(V_1^p{:=}E_1^p|P_1^p, ..., V_{n_p}^p{:=}E_{n_p}^p|P_{n_p}^p)$
in E_0
$(E8)$ | M
$(E9)$ | **case** $M_0 \rightarrow E_0 ... M_n \rightarrow E_n$ **endcase**
$(E15)$ | $E.V_0$
$(E16)$ | $E_0.\{V_1{:=}E_1, ..., V_n{:=}E_n\}$
$(E17)$ | $E_0 = E_1$
$(E19)$ | **raise** X
$(E20)$ | **trap** E_0 **handle**
$X_1 \rightarrow E_1$
...
$X_n \rightarrow E_n$
endtrap

7.1.5 Instructions

$(I1)$ I ::= **return** $[E]$ [**with** $V_1 := E_1, ..., V_n := E_n$]

$(I2)$ | **eval**
$[P_1=]\ F_1\,(V_1^1{:=}E_1^1|P_1^1, ..., V_{n_1}^1{:=}E_{n_1}^1|P_{n_1}^1)$
...
$[P_p=]\ F_p(V_1^p{:=}E_1^p|P_1^p, ..., V_{n_p}^p{:=}E_{n_p}^p|P_{n_p}^p)$
in I_0

$(I3)$ | **case** $M_0 \rightarrow I_0 ... M_n \rightarrow I_n$ **endcase**

$(I7)$ | **trap** I_0 **handle**
$X_1 \rightarrow I_1$
...
$X_n \rightarrow I_n$
endtrap

$(I8)$ | $F(V_1{:=}E_1|P_1, ..., V_n{:=}E_n|P_n)$

7.2 Semantics Domains

7.2.1 Simple Objects

All objects in the dynamic semantics are built from identifier domains together with the followings simple object domains:

		SVal	special values
n	$\in$	Nat	natural numbers
b	$\in$	Bool	booleans
χ	$\in$	ExcNames	exception values
		Ans= $\{$ *void*$\}$	answers of "procedures"

ExcNamesis a infinite set. A subset BasicExcNames $\subset$ ExcNamesof the exception names are bound to predefined exceptions in the initial dynamic semantics basis. A subset of this predefined exception would be raised by corresponding "pervasive" functions. For instance, arithmetic operations like `/, mod, ...` can rise exceptions like `Div, Mod, ....` Another predefined exception is `Match`, and is raised upon failure of pattern-matching. We consider also the predefined exception `Bind` which is raised if a variable used are not bound in the corresponding environment. The exception `Field` is raised when an impossible access at a field of a value is demanded.

SVal is the domains of values denoted by the special constants SCon. Each integer or real constant denotes a value according to normal mathematical conventions.

7.3 Compound Objects

The compound objects for the dynamic semantics are shown below:

		$\mathrm{ConVal} = \mathrm{Con} \times (\mathrm{Var} \xrightarrow{\text{fin}} \mathrm{Val})$	constructed values
v	$\in$	$\mathrm{Val} = \mathrm{SVal} \cup \mathrm{Nat} \cup \mathrm{Bool} \cup \mathrm{ConVal}$	base values
f	$\in$	$\mathrm{FunVal} = \mathrm{Vexp}^* \rightarrow \mathrm{I_{cont}} \rightarrow \mathrm{I_{cont}}$	function values
p	$\in$	$\mathrm{Param} = (\mathrm{Var} \rightarrow (\mathrm{Exp} \cup \mathrm{Pat}))^*$	actual parameter
vd	$\in$	$\mathrm{Vden} = \mathrm{Val} + \mathrm{FunVal}$	de-notated values
ve	$\in$	$\mathrm{Vexp} = \mathrm{Val} \cup \{\mathit{void}\}$	expressed values
ρ	$\in$	$\mathrm{Env} = (\mathrm{Var} \cup \mathrm{Fun} \cup \mathrm{Exc}) \rightarrow (\mathrm{Vexp} + \{\mathit{no\_bound}\})$	environment
δ	$\in$	$\mathrm{D_{cont}} = \mathrm{Env} \rightarrow \mathrm{I_{cont}}$	declaration continuations
π	$\in$	$\mathrm{P_{cont}} = \mathrm{Env} \rightarrow (\mathrm{I_{cont}} + \{\mathit{FAIL}\})$	pattern continuation
μ	$\in$	$\mathrm{M_{cont}} = \mathrm{Env} \rightarrow (\mathrm{I_{cont}} + \{\mathit{FAIL}\})$	match continuation
ε	$\in$	$\mathrm{E_{cont}} = \mathrm{Vexp} \rightarrow \mathrm{I_{cont}}$	expression continuations
γ	$\in$	$\mathrm{I_{cont}} = \mathrm{Env} \rightarrow \mathrm{Vexp} \rightarrow \mathrm{Env}$	instruction continuations

The order of evaluation within call is left-to-right.

7.4 Semantic functions

The semantic functions used to expressed denotational semantics are shown below:

$$
\begin{array}{rcl}
\mathcal{K} & : & \mathrm{SCon} \rightarrow \mathrm{SVal} \\
\mathcal{D} & : & \mathrm{Decl} \rightarrow \mathrm{Env} \rightarrow \mathrm{D_{cont}} \rightarrow \mathrm{I_{cont}} \\
\mathcal{P} & : & \mathrm{Pat} \rightarrow \mathrm{Env} \rightarrow \mathrm{Vexp} \rightarrow \mathrm{P_{cont}} \rightarrow \mathrm{I_{cont}} \\
\mathcal{M} & : & \mathrm{Match} \rightarrow \mathrm{Env} \rightarrow \mathrm{M_{cont}} \rightarrow \mathrm{I_{cont}} \\
\mathcal{E} & : & \mathrm{Exp} \rightarrow \mathrm{Env} \rightarrow \mathrm{E_{cont}} \rightarrow \mathrm{I_{cont}} \\
\mathcal{E}^* & : & ((\mathbf{in} \times \mathrm{Exp}) \cup (\mathbf{out} \times \mathrm{Pat}))^* \rightarrow \mathrm{Env} \rightarrow \mathrm{I_{cont}} \rightarrow \mathrm{I_{cont}} \\
\mathcal{I} & : & \mathrm{Instr} \rightarrow \mathrm{Env} \rightarrow \mathrm{I_{cont}} \rightarrow \mathrm{I_{cont}}
\end{array}
$$

The definition of $\mathcal{K}$ is omitted because an accurate definition of $\mathcal{K}$ is not in scope of this paper. It should be included in the semantics of modular language because it manipulates built-in constants.

7.4.1 Notations and functions

- $\mathit{wrong} : \mathrm{ExcNames} \rightarrow \mathrm{I_{cont}}$ implementation dependent.
- $D? : \mathrm{E_{cont}} \rightarrow \mathrm{E_{cont}}$ for D in Nat, Bool, SVal, ConVal, ExcNames (all domain or sub-domain which composed Vexp). These are function testers of belonging the D domain.

$$D? \quad = \quad \lambda\varepsilon.\lambda\mathrm{ve}.\ \texttt{if}\ \mathit{isD}(\mathrm{ve})\ \texttt{then}\ \varepsilon(\mathrm{ve})\ \texttt{else}\ \mathit{wrong}(\text{“bad value”})$$

- $\_=\_ : D \times D \rightarrow \mathrm{Bool}$ for D in Nat, Bool, SVal, ConVal, ExcNames (all domain or sub-domain which composed Vexp). These are (overloaded) functions of syntactical equality for domains of Val. For domains such Nat, Bool, and ConVal, the equality function is predefined. For ConVal,

"=" is syntactical equality, defined as:

$$
\begin{array}{lcl}
\mathrm{v}_1 = \mathrm{v}_2 & = & \texttt{if } (\mathrm{v}_1\downarrow 1) = (\mathrm{v}_2\downarrow 1) \texttt{ then } ; \text{equality of constructor identifiers} \\
 & & \quad \texttt{if } (\mathrm{v}_1\downarrow 2) = (\mathrm{v}_2\downarrow 2) \texttt{ then } \mathit{true} \\
 & & \quad \texttt{else } \mathit{false} \\
 & & \texttt{else } \mathit{false}
\end{array}
$$

where we have used also the equality of maps:

$$f = g \quad \equiv \quad \mathrm{Dom}(f) = \mathrm{Dom}(g) \texttt{ and } \forall s \in \mathrm{Dom}(f) f(a) = g(a)$$

- $\mathit{fail?} : \mathrm{I}_{\mathrm{cont}} \rightarrow \mathrm{I}_{\mathrm{cont}}$

$$\mathit{fail?} \quad = \quad \lambda\gamma.\lambda\rho.\texttt{if } \rho = \mathit{FAIL} \texttt{ then } \mathit{FAIL} \texttt{ else } \gamma(\rho)$$

- $\mathit{select} : \mathrm{Var} \rightarrow \mathrm{Vexp} \rightarrow \mathrm{P}_{\mathrm{cont}}$; $\mathit{select}\ (V)(\mathrm{ve})(\gamma)$ extracts the value of field V of value ve. This field exists (it is verified at the compile time). The result value is passed to the rest of program π.
- $\_.\_ : \mathrm{Vexp} \rightarrow \mathrm{Var} \rightarrow \mathrm{E}_{\mathrm{cont}} \rightarrow \mathrm{E}_{\mathrm{cont}}$;

$$
\begin{array}{lcl}
\mathrm{ve}.V(\varepsilon) & = & \texttt{if } \mathit{isConVal}(\mathrm{ve}) \texttt{ then} \\
 & & \quad \texttt{if } V \notin \mathrm{Dom}(()\mathrm{ve}\downarrow 2) \texttt{ then} \\
 & & \quad\quad \mathit{wrong}(\text{“no such field”}) \\
 & & \quad \texttt{else } \varepsilon((\mathrm{ve}\downarrow 2)(V)) \\
 & & \texttt{else } \mathit{wrong}(\text{“no constrcuted value expression”})
\end{array}
$$

First, the function test if the given value is a constructed value: if not, exception `Field` is raised; otherwise, it tests if the variable V is a name for a field of ve: if not, exception `Field` is raised; otherwise, the value of field is returned.

- $\mathit{extends} : \mathrm{Env} \rightarrow (\{\mathbf{in}, \mathbf{out}\} \times \mathrm{Var})^* \rightarrow \mathrm{Vexp}^* \rightarrow \mathrm{I}_{\mathrm{cont}}$

$$
\begin{array}{lcl}
\mathit{extends}(\rho)(fp^*)(\mathrm{ve}^*)(\gamma) & = & \texttt{if } \#fp^* = 0 \texttt{ then } \gamma(\rho) \\
 & & \texttt{else} \\
 & & \quad \texttt{if } fp^*\downarrow 1 = \mathbf{in}\ V \texttt{ then} \\
 & & \quad\quad \mathit{extends}(\rho[(\mathrm{ve}^*\downarrow 1)/(fp^*\downarrow 1)])(fp^*\dagger 1)(\mathrm{ve}^*\dagger 1)(\gamma) \\
 & & \quad \texttt{else } ; fp^*\downarrow 1 = \mathbf{out}\ V \\
 & & \quad\quad \mathit{extends}(\rho[\mathit{no\_bound}/(fp^*\downarrow 1)])(fp^*\dagger 1)(\mathrm{ve}^*\dagger 1)(\gamma)
\end{array}
$$

This function add to environment $\rho$the bindings of formal parameters at actual parameters. "**in**" formal parameters are bound to corresponding actual parameter, "**out**" formal parameters are not bound, corresponding actual parameter is ignored.

- *permute* : Fun $\rightarrow$ Param $\rightarrow$ (**in** $\times$ Exp) $\cup$ (**out** $\times$ Pat))* is generated at compile time for each function.
- *out* : Fun $\rightarrow$ Param $\rightarrow$ (Var $\times$ Pat)* gives for each function F its actual "**out**" parameters.
- *match* : (Var $\times$ Pat)* $\rightarrow$ Env $\rightarrow$ $\mathrm{P}_{\mathrm{cont}}$ $\rightarrow$ $\mathrm{P}_{\mathrm{cont}}$

$$
\begin{array}{rcl}
match(<\ >)(\rho)(\pi) & = & \pi(\rho) \\
match((V_0, P_0)\ VP^*)(\rho)(\pi) & = & \texttt{if } \rho(V_0) = no\_bound \texttt{ then } wrong(\text{“bad match expresion”}) \\
 & & \texttt{else } \mathcal{P}[[P_0]](\rho(V_0))(\lambda\rho_0. \\
 & & \qquad \texttt{if } \rho_0 = FAIL \texttt{ then } wrong(\text{“bad match expresion”}) \\
 & & \qquad \texttt{else } match(VP^*)(\rho \oplus \rho_0)(\pi)
\end{array}
$$

7.4.2 Definition of $\mathcal{D}$

$$(D3) \quad \mathcal{D}[[\textbf{function } F(\{\textbf{in}|\textbf{out } V\}^*) \textbf{ is } I]](\rho)(\delta) \quad = \quad \delta(\rho[pr/F])$$

where

$$
\begin{array}{rcl}
pr & = & \lambda \mathrm{ve}^*.\lambda\gamma.\lambda\rho. \\
 & & \quad \texttt{if } \#\mathrm{ve}^* = \#(\{\textbf{in}|\textbf{out } V\}^*) \texttt{ then} \\
 & & \quad\quad extends(\rho)(\{\textbf{in}|\textbf{out } V\}^*)(\mathrm{ve}^*)(\lambda\rho'.\mathcal{I}[[I]](\rho')(\gamma)) \\
 & & \quad \texttt{else} \\
 & & \quad\quad wrong(\text{“wrong number of arguments”})
\end{array}
$$

The declaration continuation δ is applied to environment which maps F identifier to a function value. This function value maps a command continuation γ (the rest of program after call of function) and a list of expressed values ve* to the result of γ applied to the result of instruction I executed in the static environment ρ enriched with the binding of actual parameters. The "**in**" formal parameters are bound to actual parameters, the "**out**" parameters are declared in environment, but no bound.

$$(D5) \quad \mathcal{D}[[\textbf{exception } X]](\rho)(\delta) \quad = \quad \delta(\rho[no\_bound/X])$$

An exception declaration introduces exception name in environment, but the binding of this exception is made only in a "**trap**" expression or instruction.

$$(D6) \quad \mathcal{D}[[D_1\, D_2]](\rho)(\delta) \quad = \quad \mathcal{D}[[D_1]](\rho)(\lambda\rho_1.\mathcal{D}[[D_2]](\rho_1)(\lambda\rho_2.\delta(\rho_2)))$$

The declaration D_1 is evaluated in the environment ρ and the result is the environment ρ_1. Then, D_2 is evaluated in the environment ρ_1 (composition of ρ with the result of declaration D_1) and the result is the environment ρ_2. Finally, the environment ρ_2 is passed to the continuation of program δ.

7.4.3 Definition of $\mathcal{P}$

$$
\begin{array}{lrcl}
(P1) & \mathcal{P}[[K]](\rho)(\mathrm{ve})(\pi) & = & \texttt{if } \mathrm{ve} = \mathcal{K}[[K]] \texttt{ then } \pi(\rho) \\
 & & & \texttt{else } FAIL
\end{array}
$$

If the pattern is a constant K, the value to match must be equal to his denotation (given by the semantic function $\mathcal{K}$). Otherwise, a "*FAIL*" result is given.

$$
\begin{array}{lrcl}
(P2) & \mathcal{P}[[V]](\rho)(\mathrm{ve})(\pi) & = & \texttt{if } \mathrm{ve} = void \texttt{ then} \\
 & & & \quad wrong\ (\text{“bad result of function”}) \\
 & & & \texttt{else } \pi(\rho[\mathrm{ve}/V])
\end{array}
$$

If the pattern is a variable, the continuation of pattern π will be evaluated in the environment $\rho[\mathrm{v}/V]$, where the variable V is bound to value v.

$$(P3)\quad \mathcal{P}[[\textbf{any}]](\rho)(\mathrm{v})(\pi) \;=\; \texttt{if } \mathrm{ve} = \mathit{void} \texttt{ then } \mathit{wrong}\,(\text{“bad result of function”}) \texttt{ else } \pi(\rho)$$

If the pattern is “**any**” (wildcard pattern in SML), the initial environment is not modified, and the result is the continuation of calculus by π.

$$(P4)\quad \mathcal{P}[[C\,(V_1 \texttt{:=} P_1, ..., V_n \texttt{:=} P_n[, \bullet\bullet\bullet])]](\rho)(\mathrm{ve})(\pi) \;=\; \texttt{if } \mathit{isConVal}(\mathrm{ve}) \texttt{ and } \mathrm{ve}\downarrow 1 = C \texttt{ then } \mathit{select}(\mathrm{ve})(V_1)(\lambda \mathrm{ve}_1.\mathcal{P}[[P_1]](\rho)(\mathrm{ve}_1)\ (\mathit{fail?} \circ \lambda\rho_1.\mathit{select}(\mathrm{ve})(V_2)(\lambda \mathrm{ve}_2.\mathcal{P}[[P_2]](\rho_1)(\mathrm{ve}_2)(\ \ldots\ (\mathit{fail?} \circ \lambda\rho_{n-1}.\mathit{select}(\mathrm{ve})(V_n)(\lambda \mathrm{ve}_n.\mathcal{P}[[P_n]](\rho_{n-1})(\mathrm{ve}_n)(\pi)))\ \ldots)))) \texttt{ else } \mathit{FAIL}$$

For a structured pattern, one first verification is that of type of value. If the value is not of corresponding constructor, the pattern fail. Otherwise, the program continues by evaluation the patterns of parameters, one by one, in the environments build by the previously evaluated pattern. If “$\bullet\bullet\bullet$” is encountered, the program is continuated (π) in the result environment (ρ_n).

For instance, the denotation of pattern $C(\bullet\bullet\bullet)$ is:

$$(P4')\quad \mathcal{P}[[C(\bullet\bullet\bullet)]](\rho)(\mathrm{ve})(\pi) \;=\; \texttt{if } \mathit{isConVal}(\mathrm{ve}) \texttt{ and } \mathrm{ve}\downarrow 1 = C \texttt{ then } \mathit{FAIL} \texttt{ else } \pi(\rho)$$

7.4.4 Definition of $\mathcal{M}$

$$(M1)\quad \mathcal{M}[[E :: P]](\rho)(\mu) \;=\; \mathcal{E}(E)(\rho)(\lambda \mathrm{ve}.\mathcal{P}[[P]](\rho)(\mathrm{ve})(\mathit{fail?} \circ \lambda\rho'.\mu(\rho')))$$

The expression E is evaluated in the environment ρ, and the content of result value is passed to evaluate the the pattern P. The environment result of this evaluation is passed to the continuation of match expression.

$$(M2)\quad \mathcal{M}[[M_1 \textbf{ when } E]](\rho)(\mu) \;=\; \mathcal{M}[[M_1]](\rho)(\lambda\rho'.\mathit{fail?} \circ \mathcal{E}[[E]](\rho')(\lambda \mathrm{ve}.\ \texttt{if } \mathit{isBool}(\mathrm{ve}) \texttt{ then } \texttt{if } b \texttt{ then } \mu(\rho') \texttt{ else } \mathit{FAIL} \texttt{ else } \mathit{wrong}(\text{“non-boolean argument of guard”})$$

The match expression M_1 is evaluated in the environment ρ and then, if not “*FAIL*” result, the expression E is evaluated in the result environment ρ'. If the right value of result of E evaluation is a boolean value and is “*true*”, then the rest of program continues in the environment ρ'. Otherwise, the result is “*FAIL*”.

$$(M3)\quad \mathcal{M}[[M_1 \textbf{ and } M_2]](\rho)(\mu) \;=\; \mathcal{M}[[M_1]](\rho)(\lambda\rho'.\mathit{fail?} \circ \mathcal{M}[[M_2]](\rho)(\lambda\rho''.\mathit{fail?} \circ \texttt{if } \rho' \cap \rho'' \neq \emptyset \texttt{ then } \mathit{wrong}(\text{“non-disjunct matches”}) \texttt{ else } \mu(\rho[\rho' \oplus \rho''])))$$

The match expression M_1 is evaluated in the environment ρ and the result is the environment ρ' or “*FAIL*”. If “*FAIL*” does not result, the match expression M_2 is evaluated in the same environment ρ, and gives environment ρ'' or “*FAIL*” as result. If “*FAIL*” does not result, we verify that environments ρ' and ρ'' are not common elements, and the program is continued in the environment $\rho[\rho' \oplus \rho'']$ (“$\oplus$” is the disjunct union)

$$(M4)\quad \mathcal{M}[[M_1 \textbf{ or } M_2]](\rho)(\mu) \;=\; \mathcal{M}[[M_1]](\rho)(\lambda\rho'.\ \texttt{if } \rho' \neq \mathit{FAIL} \texttt{ then } \mu(\rho[\rho']) \texttt{ else } \mathcal{M}[[M_2]](\rho)(\mu)$$

The match expression M_1 is evaluated in the environment ρ and the result is the environment ρ' or "*FAIL*". If "*FAIL*" does not result, the program is continuated in the composed environment $\rho[\rho']$. Otherwise, the result program is the evaluation of match expression M_2 in the initial environment ρ.

7.4.5 Definition of $\mathcal{E}$

$$(E1)\quad \mathcal{E}[[K]](\rho)(\varepsilon) \;=\; \varepsilon(\mathcal{K}[[K]])$$

A constant expression is evaluated by $\mathcal{K}$ at its denotation and the result value passed to the continuation of program. The constant exceptions which can be raised by predefined ("pervasive") operations are also treat by $\mathcal{K}$.

$$(E2)\quad \mathcal{E}[[V]](\rho)(\varepsilon) \;=\; \texttt{if } \rho(V) = \mathit{no\_bound} \texttt{ then } \mathit{wrong}(\text{"unbound variable"}) \texttt{ else } \varepsilon(\rho(V))$$

A variable has as denotation its binding value (an expressed value) in the environment ρ.

$$(E3)\quad \mathcal{E}[[(C(V_1{:=}E_1, ..., V_n{:=}E_n))]](\rho)(\varepsilon) \;=\; \mathcal{E}[[E_1]](\rho)(\lambda \mathrm{ve}_1. \ldots \mathcal{E}[[E_n]](\rho)(\lambda \mathrm{ve}_n. \varepsilon((C, \{V_1 \mapsto \mathrm{ve}_1, ..., V_n \mapsto \mathrm{ve}_n\}))) \ldots)$$

The value is build by the constructor and the evaluation of all expressions given as parameters. The evaluation of parameters is left-to-right.

For instance, a constant constructor is evaluated as follows:

$$(E3')\quad \mathcal{E}[[(C())]](\rho)(\varepsilon) \;=\; \varepsilon((C,\ \emptyset))$$

$$\begin{aligned}(E5)\quad & \mathcal{E}[[\textbf{eval} \\ & \quad [P_1{:=}]\ F_1(V_1^1{:=}E_1^1|P_1^1, ..., V_{n_1}^1{:=}E_{n_1}^1|P_{n_1}^1) \\ & \quad \ldots \\ & \quad [P_p{:=}]\ F_p(V_1^p{:=}E_1^p|P_1^p, ..., V_{n_p}^p{:=}E_{n_p}^p|P_{n_p}^p) \\ & \textbf{in } E_0]](\rho)(\varepsilon) = \\ & \quad \mathcal{I}[[F_1(V_1^1{:=}E_1^1|P_1^1, ..., V_{n_1}^1{:=}E_{n_1}^1|P_{n_1}^1)]](\rho)(\lambda\rho_1\,\mathrm{ve}_1. \\ & \qquad [\mathcal{P}[[P_1]](\rho)(\lambda\rho_1'.\texttt{if } \rho_1' = \mathit{FAIL} \texttt{ then } \mathit{wrong}\ (\text{"wrong match expression"})\ \texttt{else}] \\ & \ldots \\ & \quad \mathcal{I}[[F_p(V_1^p{:=}E_1^p|P_1^p, ..., V_{n_p}^p{:=}E_{n_p}^p|P_{n_p}^p)]](\rho)(\lambda\rho_n\mathrm{ve}_n. \\ & \qquad [\mathcal{P}[[P_n]](\rho)(\lambda\rho_n'.\texttt{if } \rho_n' = \mathit{FAIL} \texttt{ then } \mathit{wrong}\ (\text{"wrong match expression"})\ \texttt{else }] \\ & \mathcal{E}[[E_0]](\rho[\rho_1[\oplus\rho_1'] \oplus \ldots \oplus \rho_n[\oplus\rho_n']])(\varepsilon)[])\ldots[])\end{aligned}$$

Each function application is evaluated (as a instruction) an the result (a value and an environment) is passes to a pattern evaluation, if the pattern P_i is indicated, or it simple pass to evaluate another function application. The expression E_0 is evaluated in the environment which is composition of initial environment and the disjunct union of each environment resulted by evaluation of function application (and pattern P_i if exists).

$$(E8)\quad \mathcal{E}[[M]](\rho)(\varepsilon) \;=\; \mathcal{M}[[M]](\rho)(\lambda\rho'.\texttt{if } \rho' = \mathit{FAIL} \texttt{ then } \varepsilon(\mathit{false}\texttt{else } \varepsilon(\mathit{true})$$

The match expression M is evaluated in the context ρ and if it fails, the result is the continuation of program with boolean value *false*, otherwise, the result is the continuation of program with boolean value *true*.

$$(E9)\quad \mathcal{E}[[\textbf{case } M_0 \to E_0 ... M_n \to E_n]](\rho)(\varepsilon) \;=\; \mathcal{M}[[M_0]](\rho)(\lambda\rho_0.\texttt{if } \rho_0 \neq \mathit{FAIL} \texttt{ then } \mathcal{E}[[E_0]](\rho[\rho_0])(\varepsilon) \texttt{ else } \ldots \mathcal{M}[[M_n]](\rho)(\lambda\rho_n.\texttt{if } \rho_0 \neq \mathit{FAIL} \texttt{ then } \mathcal{E}[[E_n]](\rho[\rho_n])(\varepsilon) \texttt{ else } \mathit{wrong}(\text{"wrong match expression"}))\ldots)$$

The match expression M_i is evaluated in the initial environment ρ. If it does not fails, expression E_i is evaluated in the composed environment $\rho[\rho_i]$. Otherwise, the match expression M_{i+1} (if exists) is evaluated. Finally, if all match expressions fail, `Match` exception is raised.

$$(E15)\quad \mathcal{E}[[E.V_0]](\rho)(\varepsilon) = \mathcal{E}[[E]](\rho)(\lambda \mathrm{ve}.\texttt{if } isConVal(\mathrm{ve}) \texttt{ then } \mathrm{ve}.V_0(\varepsilon) \texttt{ else } wrong(\text{“non constructed value expression”}))$$

The expression E is evaluated in the initial environment and the result expressed value is tested for belonging to domain of constructed values, ConVal. If the test is true, operator "$\_.\_$" is applied, otherwise, exception `Field` is raised.

$$\begin{aligned}(E16)\quad \mathcal{E}[[E_0.\{V_1 := E_1, ..., V_n := E_n\}]](\rho)(\varepsilon) = {} & \mathcal{E}[[E_0]](\rho)(\lambda \mathrm{ve}_0.\ \ \mathrm{ve}_0.V_1 \circ \lambda \mathrm{ve}_1. \\ & \qquad\qquad ... \\ & \qquad\qquad \mathrm{ve}_0.V_n \circ \lambda \mathrm{ve}_n. \\ & \mathcal{E}[[E_1]](\rho[V_1 \mapsto \mathrm{ve}_1]...[V_n \mapsto \mathrm{ve}_n])(\lambda \mathrm{ve}'_1. \\ & ... \\ & \mathcal{E}[[E_n]](\rho[V_1 \mapsto \mathrm{ve}_1]...[V_n \mapsto \mathrm{ve}_n])(\lambda \mathrm{ve}'_n. \\ & \varepsilon(((\mathrm{ve}_0 \downarrow 1), (\mathrm{ve}_0 \downarrow 2) + \{V_1 \mapsto \mathrm{ve}'_1, ..., V_n \mapsto \mathrm{ve}'_n\}))) \\ & ...)...)\end{aligned}$$

First, expression E_0 is evaluated and after this their components $V_1, ..., V_n$ are selected (exceptions may be raised here). Expressions $E_1, ...E_n$ are evaluated sequentially in the environment obtained by definition of variables $V_1, ..., V_n$ having as bindings the correspondent expression in value ve_0. If all evaluations are finish successfully, a new value, obtained from ve_0 and replacing old values of field V_i with correspondent expression ve'_i, is passed to continuation of expression ε.

$$\begin{aligned}(E17)\quad \mathcal{E}[[E_0 = E_1]](\rho)(\varepsilon) = {} & \mathcal{E}[[E_0]](\rho)(\lambda \mathrm{ve}_0. \\ & \mathcal{E}[[E_1]](\rho)(\lambda \mathrm{ve}_1.\texttt{if } \mathrm{ve}_0 = \mathrm{ve}_1 \texttt{ then } \varepsilon(true) \texttt{ else } \varepsilon(false))\end{aligned}$$

Both values E_0 and E_1 are evaluated (sequentially) in the initial environment ρ, and their values are compared (syntactically, the type comparison was made at compile time); the continuation of evaluation ε receives the boolean value of result.

$$\begin{aligned}(E19)\quad \mathcal{E}[[\textbf{raise } X]](\rho)(\varepsilon) = {} & \texttt{if } \rho(X) = no\_bound \texttt{ then } wrong(\text{“undeclared exception } X\text{”}) \\ & \texttt{else } \rho(X)(<>)(\varepsilon(void))(\rho)\end{aligned}$$

If exception X was not declared, an exception of type "undeclared exception" is raised, otherwise the program returns with the continuation of exception X given by function $\rho(X)$ which has insignificant parameters. Predefined expressions must be initialized in the initial environment.

$$(E20)\quad \mathcal{E}[[\textbf{trap } E_0 \textbf{ handle } X_1 \to E_1 ... X_n \to E_n]](\rho)(\varepsilon) = \mathcal{E}[[E_0]](\rho[ex_1/X_1]...[ex_n/X_n])(\lambda \mathrm{ve}_0.\varepsilon(\mathrm{ve}_0))$$

where

$$\begin{aligned}ex_i = {} & \lambda \mathrm{ve}^*.\lambda\gamma'.\lambda\rho'. \\ & \texttt{if } \#\mathrm{ve}^* = 0 \texttt{ then } \mathcal{E}[[E_i]](\rho)(\varepsilon) \\ & \texttt{else } wrong(\text{“wrong number of arguments”})\end{aligned}$$

A "**trap**" expression binds handled exceptions with functions which treat them, and after this evaluates expression E_0. Function ex_i bound to exception X_i has no parameters, evaluates expression E_i in the initial environment ρ, and continues initial expression evaluation ε.

7.4.6 Definition of $\mathcal{I}$

$$\begin{aligned}(I1)\quad \mathcal{I}[[\textbf{return } [E] \textbf{ with } V_1 := E_1, ..., V_n := E_n]](\rho)(\gamma) = {} & [\mathcal{E}[[E]](\rho)(\lambda \mathrm{ve}.] \\ & [\mathcal{E}[[E_1]](\rho)(\lambda \mathrm{ve}_1. \\ & \quad ... \\ & \mathcal{E}[[E_n]](\rho)(\lambda \mathrm{ve}_n.] \\ & \gamma(\ \emptyset | [\{V_1 \mapsto \mathrm{ve}_1, ..., V_n \mapsto \mathrm{ve}_n\}])(void | \mathrm{ve})\end{aligned}$$

If E is specified it is evaluated in the initial environment at ve, otherwise, the value returned is *void*. If "**with**" clause exists, expressions E_i are evaluated sequentially, and result values are bound to corresponding V_i variable; otherwise, the result environment is empty. Finally, program γ continues with parameters the result environment and the result value.

$$
\begin{array}{ll}
(I2) & \mathcal{I}[[\textbf{eval} \\
& \quad [P_1\textbf{:=}]\ F_1(V_1^1\textbf{:=}E_1^1|P_1^1,...,V_{n_1}^1\textbf{:=}E_{n_1}^1|P_{n_1}^1) \\
& \quad ... \\
& \quad [P_p\textbf{:=}]\ F_p(V_1^p\textbf{:=}E_1^p|P_1^p,...,V_{n_p}^p\textbf{:=}E_{n_p}^p|P_{n_p}^p) \\
& \textbf{in}\ I_0]](\rho)(\gamma) = \\
& \quad \mathcal{I}[[F_1(V_1^1\textbf{:=}E_1^1|P_1^1,...,V_{n_1}^1\textbf{:=}E_{n_1}^1|P_{n_1}^1)]](\rho)(\lambda\rho_1\mathrm{ve}_1. \\
& \qquad [\mathcal{P}[[P_1]](\rho)(\lambda\rho_1'.\texttt{if}\ \rho_1' = \mathit{FAIL}\ \texttt{then}\ \mathit{wrong}\ (\text{“wrong match expression”})\ \texttt{else}] \\
& ... \\
& \quad \mathcal{I}[[F_p(V_1^p\textbf{:=}E_1^p|P_1^p,...,V_{n_p}^p\textbf{:=}E_{n_p}^p|P_{n_p}^p)]](\rho)(\lambda\rho_n\mathrm{ve}_n. \\
& \qquad [\mathcal{P}[[P_n]](\rho)(\lambda\rho_n'.\texttt{if}\ \rho_n' = \mathit{FAIL}\ \texttt{then}\ \mathit{wrong}\ (\text{“wrong match expression”})\ \texttt{else}\ ] \\
& \mathcal{I}[[I_0]](\rho[\rho_1[\oplus\rho_1'] \oplus ... \oplus \rho_n[\oplus\rho_n']])(\gamma)([])...[])])
\end{array}
$$

Each function application is evaluated (as a instruction) and the result (a value and an environment) is passes to a pattern evaluation (if P_i is indicated) and then it pass to evaluate another function application. The instruction I_0 is evaluated in the environment which is composition of initial environment and the disjunct union of each environment resulted by evaluation of function application (and pattern P_i if exists).

$$
\begin{array}{lll}
(I3) & \mathcal{I}[[\textbf{case}\ M_0 \to I_0 ... M_n \to I_n]](\rho)(\gamma) & = \mathcal{M}[[M_0]](\rho)(\lambda\rho_0.\texttt{if}\ \rho_0 \neq \mathit{FAIL}\ \texttt{then}\ \mathcal{I}[[I_0]](\rho[\rho_0])(\gamma)\ \texttt{else} \\
& & \quad ... \\
& & \quad \mathcal{M}[[M_n]](\rho)(\lambda\rho_n.\texttt{if}\ \rho_0 \neq \mathit{FAIL}\ \texttt{then}\ \mathcal{I}[[I_n]](\rho[\rho_n])(\gamma) \\
& & \quad \texttt{else}\ \mathit{wrong}(\text{“wrong match instruction”}))...)
\end{array}
$$

The match expression M_i is evaluated in the initial environment ρ. If it does not fails, instruction I_i is evaluated in the composed environment $\rho[\rho_i]$. Otherwise, the match expression M_{i+1} (if exists) is evaluated. Finally, if all match expressions fail, `Match` exception is raised.

$$
(I7)\quad \mathcal{E}[[\textbf{trap}\ I_0\ \textbf{handle}\ X_1 \to I_1 ... X_n \to I_n]](\rho)(\gamma) \quad = \quad \mathcal{I}[[I_0]](\rho[ex_1/X_1]...[ex_n/X_n])(\lambda\mathrm{ve}_0.\varepsilon(\mathrm{ve}_0))
$$

where

$$
\begin{array}{lll}
ex_i & = & \lambda\mathrm{ve}^*.\lambda\gamma'.\lambda\rho'. \\
& & \quad \texttt{if}\ \#\mathrm{ve}^* = 0\ \texttt{then}\ \mathcal{I}[[I_i]](\rho)(\gamma) \\
& & \quad \texttt{else}\ \mathit{wrong}(\text{“wrong number of arguments”})
\end{array}
$$

A "**trap**" instruction binds handled exceptions with functions which treat them, and after this evaluates instruction I_0. Function ex_i bound to exception X_i has no parameters, evaluates instruction I_i in the initial environment ρ, and continues initial instruction evaluation γ.

$$
\begin{array}{lll}
(I8) & \mathcal{I}[[F(V_1\textbf{:=}E_1|P_1,...,V_n\textbf{:=}E_n|P_n)]](\rho)(\gamma) & = \mathcal{E}^*(\mathit{permute}(F)((V_1,E_1|P_1),...,(V_n,E_n|P_n))) \\
& & \quad (\rho) \\
& & \quad (\lambda\mathrm{ve}^*.\rho(F)(\mathrm{ve}^*) \\
& & \qquad (\lambda\rho'.\lambda\mathrm{ve}.\mathit{match}(\rho') \\
& & \qquad\quad (\mathit{out}(F)(V_1\textbf{:=}E_1|P_1,...,V_n\textbf{:=}E_n|P_n)) \\
& & \qquad\quad (\lambda\rho''.\gamma(\rho'')(\mathrm{ve}))))
\end{array}
$$

Actual parameters of F are permuted such as the order correspond to these of declaration of function, and evaluated by $\mathcal{E}^*$ in the initial environment ρ. The result is a sequence of expressed values (values corresponding to "**out**" parameters are *void*), which is passed to function bind in the environment ρ to symbol F (this function exists, it was checked at compile time). The continuation of function evaluation receive as result an environment ρ' (where all "**out**" parameters are bound to expressed

values), and a value ve (which could be *void*). Finally, all patterns corresponding to "**out**" parameters are evaluated by function *match* and the result of this evaluation is the environment ρ'', which is passed with value ve to the remainder program γ.

7.4.7 Definition of $\mathcal{E}^*$

$$
\begin{array}{lcl}
\mathcal{E}^*[[]](\rho)(\gamma) & = & \gamma(<\ >) \\
\mathcal{E}^*[[\textbf{in}\ E_0\ ap^*]](\rho)(\gamma) & = & \mathcal{E}[[E_0]](\rho)(\lambda \mathrm{ve}_0.\mathcal{E}^*[[ap^*]](\rho)(\lambda \mathrm{ve}^*.\gamma(<\mathrm{ve}_0>\ \S\ \mathrm{ve}^*))) \\
\mathcal{E}^*[[\textbf{out}\ P_0\ ap^*]](\rho)(\gamma) & = & \mathcal{E}^*[[ap^*]](\rho)(\lambda \mathrm{ve}^*.\gamma(<\mathit{void}>\ \S\ \mathrm{ve}^*)))
\end{array}
$$

An "**in**" actual parameter determines the evaluation of corresponding expression, and after evaluation of the remainder list of parameters ap^*, it is added in the front of list. An "**out**" parameter implies the evaluation of the remainder list of parameters ap^*, and the concatenation of value *void*.

A Static Semantics — Formal Definition

Note: The attribute grammar presented here is incomplete: we have compressed the passes to solve overloading of operators. This passes are largely presented in [BH88] for LOTOS, and they are the same for our attribute grammar.

A.1 Notations

When A and B denotes sets, $A \xrightarrow{\text{fin}} B$ denotes the set of *finite maps* (partial functions with finite domain) from A to B.

The domain and range of a finite map, f, are denoted $\text{Dom}(f)$ (for instance, A) and $\text{Ran}(f)$ (for instance, B). A finite map will often be written explicitly in the form $\{a_1 \mapsto b_1, \cdots, a_n \mapsto b_n\}$, $k \geq 0$; in particular, the empty map is $\{\}$. The *undefined* value for partial application is denoted by $\perp$.

When the range of a partial map is the set of sub set of a domain, $A \xrightarrow{\text{fin}} \mathcal{P}(B)$ denotes *set finite maps* from A to B.

When f and g are finite maps (or set-finite maps) we define:

- the map (or set-finite map) $f \oslash g$ with domain $\text{Dom}(f) \cup \text{Dom}(g)$ and values:

$$(f \oslash g)(x) = \begin{cases} f(x) & \texttt{if} \quad g(x) = \perp \ (or\ g(x) = \emptyset) \\ g(x) & \texttt{otherwise} \end{cases}$$

  This operator is called usually *f modified by g*.

- the map (or set-finite map) $f \oplus g$ with domain $\text{Dom}(f) \cup \text{Dom}(g)$ and values:

$$(f \oplus g)(x) = \begin{cases} f(x) & \texttt{if} \quad g(x) = \perp \ (or\ g(x) = \emptyset) \\ g(x) & \texttt{if} \quad f(x) = \perp \ (or\ f(x) = \emptyset) \\ (f(x) \cup g(x)) & \texttt{if} \quad g(x) \cap f(x) = \emptyset) \\ error & \texttt{otherwise} \end{cases}$$

  This operator is called usually disjunct composition of f and g.

- the map (or set-finite map) $f \odot g$ with domain $\text{Dom}(f) \cup \text{Dom}(g)$ and values:

$$(f \odot g)(x) = \begin{cases} f(x) & \texttt{if} \quad g(x) = \perp \ (or\ g(x) = \emptyset) \\ g(x) & \texttt{if} \quad f(x) = \perp \ (or\ f(x) = \emptyset) \\ f(x) & \texttt{if} \quad g(x) = f(x) \\ error & \texttt{otherwise} \end{cases}$$

  This operator is therefore called *match* composition of f and g.

- the map (or set-finite map) $f \ominus \{a_1, ..., a_n\}$ where $\{a_1, ..., a_n\} \subset \text{Dom}(f)$, with domain $\text{Dom}(f) \setminus \{a_1, ..., a_n\}$ and values:

$$(f \ominus \{a_1, ..., a_n\})(x) = \begin{cases} f(x) & \texttt{if} \quad x \notin \{a_1, ..., a_n\} \\ \perp & \texttt{otherwise} \end{cases}$$

  this operator is usually called restriction of domain of f.

A.2 Attributes, environments and semantic actions

A.2.1 Attribute of types

In the abstract grammar, T represents the identifier associated with this type. We give for each type a unique name and informations concerning its renaming. In this aim, for each type T occuring in the program we associate an attribute *binding*, defined as $binding[\mathrm{T}] = (uid,\ ren)$ where:

- *uid* is a unique identifier;
- *ren* may have two values:
  - $ren = T'$ if type T renames type T';
  - $ren = no\_ren$ if type T don't renames any type.

In the aim to introduce built-in type (and also allows special constants use), we define the object $\mathbf{Type} = type(\mathrm{SCon}) \cup \mathrm{Typ}$ which contains all possibles names of types.

A.2.2 Environment $\mathcal{T}$

We define the type environment $\mathcal{T}$ as a finite map from the set of type occurring to a set of "bindings":

$$\mathcal{T} \in \mathbf{TyEnv} = \mathbf{Type} \xrightarrow{\text{fin}} (\mathrm{Nat} \times (\mathbf{Type} \cup no\_ren)) \cup \bot$$

$\mathcal{T}[T]$ have as value $binding[\mathrm{T}]$ if the type T is defined in environment $\mathcal{T}$, and $\bot$ if type T is not defined in $\mathcal{T}$.

The $\oplus$ operator over finite maps is used to verify that type environments are nor overlapped.

A.2.3 Semantic actions for types

Semantic actions used are:

- $use_T(T, \mathcal{T})$: this action arises error if the type was not declared; otherwise, it defines the binding of T.

$$use_T(T, \mathcal{T}) = \begin{cases} \texttt{if } \mathcal{T}[T] = \bot \texttt{ then} \\ \quad error:\ undeclared\ type\ T \\ \quad binding[T] = (\bot, \bot) \\ \texttt{else} \\ \quad binding[T] = \mathcal{T}[T] \end{cases}$$

  Note: if an error is arisen, we bind the type at $\bot$ binding. This allows the continuation of calculus besides this erroneous binding.

  A notation is used:

$$use_T(T_1, ..., T_n, \mathcal{T}) = \begin{cases} use_T(T_1, \mathcal{T}) \\ ... \\ use_T(T_n, \mathcal{T}) \end{cases}$$

- $def_T(T, ren)$: this action returns an environment $\mathcal{T}$ containing only the type T.

$$def_T(T, ren) = \mathcal{T}\ \ where \begin{cases} \mathcal{T}[T] = new\_binding_T(ren) \\ \texttt{and } \forall T' \neq T \quad \mathcal{T}[T'] = \bot \end{cases}$$

  where $new\_binding_T(ren)$ returns the type binding having a new number for *uid* field, and *ren* for *ren* field.

A.2.4 Environment $\mathcal{V}$

We define variables environment $\mathcal{V}$ as a finite (partial) application from domain of variable identifiers to **Type**:

$$\mathcal{V} \in \textbf{VarEnv} = \text{Var} \xrightarrow{\text{fin}} \textbf{Type}$$

$\mathcal{V}[V]$ has as value the type identifier of type of variable V.

Operators $\oslash$, $\oplus$, and $\ominus$ are used for composition of variable environments.

A.2.5 Semantic action for variables

Semantic action used is $use_V(V, \mathcal{V})$ this action arises error if the variable was not declared; otherwise, it return the type of V.

$$use_V(V, \mathcal{V}) = \begin{cases} \texttt{if } \mathcal{V}[V] = \perp \texttt{ then} \\ \quad \textit{error: undeclared variable } V \\ \texttt{else } \mathcal{V}[V] \end{cases}$$

A notation is used:

$$use_V(V_1, ..., V_n, \mathcal{V}) = \begin{cases} use_V(V_1, \mathcal{V}) \\ ... \\ use_V(V_n, \mathcal{V}) \end{cases}$$

A.2.6 Environment $\mathcal{X}$

We define exceptions environment $\mathcal{X}$ as a finite (partial) application from domain of exception identifiers to **Type**:

$$\mathcal{X} \in \textbf{ExcEnv} = \text{Exc} \xrightarrow{\text{fin}} \textbf{Type}$$

For all X, $\mathcal{X}[X]$ has as value the `exn` type identifier.

A.2.7 Semantic action for exceptions

Semantic action used is $use_E(X, \mathcal{X})$ this action arises error if the exception E not declared; otherwise, it return the type of X.

$$use_E(X, \mathcal{X}) = \begin{cases} \texttt{if } \mathcal{X}[X] = \perp \texttt{ then} \\ \quad \textit{error: undeclared exception } X \\ \texttt{else exn} \end{cases}$$

A notation is used:

$$use_E(X_1, ..., X_n, \mathcal{X}) = \begin{cases} use_E(X_1, \mathcal{X}) \\ ... \\ use_E(X_n, \mathcal{X}) \end{cases}$$

A.2.8 Attribute for functions and constructors

In the following, we use the term *operation* to denote functions or constructors.

An operation occurrence has a correlated tuple ($FormalP$, Res, Exc, Pos, IsC) where:

- *FormalP* is the (ordered) list of field names and field types; it is an element of $\mathbf{Nat} \xrightarrow{\text{fin}} \mathbf{VarEnv}$;
- *OutP* is the set of "**out**" formal parameters; it is an element of **VarEnv**;
- *Res* is the type of result; it is an element of **Type**;
- *Exc* is the list of exceptions raised; it is an element of **ExcEnv**.
- *Pos* may have two values *infixed* or *prefixed*; this values is given by syntactical analyze.
- *IsC* is a boolean value; if $IsC = \mathbf{true}$ then the operation is a constructor, otherwise is a function.

In conclusion, the type of attribute associated with an operation is:

$$\mathbf{OpType} = (\mathbf{Nat} \xrightarrow{\text{fin}} \mathbf{VarEnv}) \times \mathbf{VarEnv} \times (\mathbf{Type} \cup \mathbf{void}) \times \mathbf{ExcEnv} \times \{\mathit{infixed}, \mathit{prefixed}\} \times \mathbf{bool}$$

Each *use*-occurrence of an operation may have a lot of attribute tuples because overloading of operations is allowed. The unique tuple corresponding at an operation O is noted $binding$[O].

A.2.9 Environment $\mathcal{F}$

We define the environment of operations (functions and constructors) $\mathcal{F}$ as a finite set map from the set of occurences of operations to the multi set of attributes of operations (type of operations):

$$\mathcal{F} \in \mathbf{OpEnv} = (\text{Con} \cup \text{Fun}) \xrightarrow{\text{fin}} \mathcal{P}(\mathbf{OpType})$$

$\mathcal{F}[O]$ has as value the set of tuples corresponding to O in the environment $\mathcal{F}$. $\mathcal{F}[O] = \perp$ if operation O is not defined in $\mathcal{F}$.

The operator $\oplus$ over set-finite maps is used to verify that operation environments are not overlapped.

The operations used over the enviroinment $\mathcal{F}$ are:

- The operation `constructor` extracts for a given type T, from an environment **OpEnv**, the environment of constructors of type T:

  $$\texttt{constructors} : \mathbf{Type} \times \mathbf{OpEnv} \rightarrow \mathbf{OpEnv}$$

  defined as:

  $$\texttt{constructors}(T, \mathcal{F}) = \mathcal{F}' \ \texttt{where} \ \left\{ \begin{array}{ll} \mathcal{F}'[C] = \{bind | bind \in \mathcal{F}[C] \\ \qquad \wedge \quad bind.R = T \\ \qquad \wedge \quad bind.IsC = \mathbf{true}\} \end{array} \right.$$

- The renaming of profile of operations, noted $\widehat{\rho}$, is the morphism extension of the renaming map:

  $$\rho : \text{Typ} \rightarrow \mathbf{Type}$$

  The definition of $\widehat{\rho}$ is:

  $$\widehat{\rho} : \mathbf{Nat} \xrightarrow{\text{fin}} \mathbf{VarEnv} \rightarrow \mathbf{Nat} \xrightarrow{\text{fin}} \mathbf{VarEnv}$$

  $$\widehat{\rho}(\{i \mapsto \{V_1 \mapsto T_1, ..., V_n \mapsto T_n\}\}) = \{i \mapsto \{V_1 \mapsto \rho(T_1), ..., V_n \mapsto \rho(T_n)\}\}$$

- The operator `select_binding` returns the type environment of fields having a given name:

$$\texttt{select\_binding} : \mathcal{P}(\mathbf{OpType}) \times \mathrm{Var} \times \mathbf{TyEnv} \times \mathbf{bool} \rightarrow \mathbf{TyEnv}$$

$$\texttt{select\_binding}(bindings, V, \mathcal{T}, C) = \mathcal{T}' \; where \left\{ \begin{array}{ll} T \in \mathcal{T} \text{ iff} & \exists bind \in bindings. \quad bind.IsC = C \\ & \wedge((V,T) \in \mathrm{Ran}(bind.FormalP) \\ & \vee bind.V = T \\ & \vee \mathcal{T} = \emptyset) \end{array} \right.$$

- The operator `select_result` returns the type environment of results:

$$\texttt{select\_result} : \mathcal{P}(\mathbf{OpType}) \rightarrow \mathbf{TyEnv}$$

$$\texttt{select\_result}(bindings) = \mathcal{T} \; where \left\{ \; T \in \mathcal{T} \text{ iff } \exists bind \in bindings. \quad bind.Res = T \right.$$

A.2.10 Semantic actions for operations

Semantic actions use are:

- $use_F(O, \mathcal{F})$: this action arises error if the operation was not declared; otherwise, it binds the operation O with the set of its overloading $\mathcal{F}[O]$.

$$use_F(O, \mathcal{F}) = \left\{ \begin{array}{l} \texttt{if } \mathcal{F}[O] = \perp \texttt{ then} \\ \quad error: \; undeclared \; operation \; O \\ \texttt{else} \\ \quad bind \; operation \; O \; with \; \mathcal{F}[O] \end{array} \right.$$

  Note: if an error is arisen, we bind the type at "$\perp$" binding. This allows the continuation of calculus besides this erroneous binding.

  A notation is used:

$$use_F(O_1, ..., O_n, \mathcal{F}) = \left\{ \begin{array}{l} use_F(O_1, \mathcal{F}) \\ ... \\ use_F(O_n, \mathcal{F}) \end{array} \right.$$

- $def_F(O, FP, OP, E, R, Pos, IsC)$: this action returns an environment $\mathcal{F}$ containing only the operation O and his binding.

$$def_F(O, FP, OP, E, R, P, C) = \mathcal{F} \; where \left\{ \begin{array}{l} \texttt{if number of arguments} \neq 2 \texttt{ and } Pos = infixed \texttt{ then} \\ \quad error: \; bad \; position \\ \quad \mathcal{F}[O] = \perp \\ \texttt{else } \mathcal{F}[O] = \{new\_binding_F(FP, OP, E, R, Pos, IsC)\} \\ \forall O' \neq T \quad \mathcal{F}[O'] = \perp \end{array} \right.$$

  where $new\_binding_F(FP, OP, E, R, P, C)$ returns a 6-uple of type **OpType** which fields are initialized respectively with FP, OP, E, R, P, C.

A.2.11 Environment $\mathcal{E}$

We define the global environment $\mathcal{E}$ as a tuple of types environment (field noted TE), variables environment (field noted VE), exceptions environment (filed noted EE), operations environment (filed noted FE):

$$\mathcal{E} \in Env = \mathbf{TyEnv} \times \mathbf{VarEnv} \times \mathbf{ExcEnv} \times \mathbf{OpnEnv}$$

The selector of a filed of this tuple is noted, for instance $\mathcal{E}.VE$, and a injection function is used, for example the global environment formed by a variable environment $\mathcal{V}$ is ($\emptyset, \mathcal{V}, \emptyset, \emptyset$).

The operation $\oplus_E$ is defined by:

$$\mathcal{E} \oplus_E \mathcal{E}' = (\mathcal{E}.\mathcal{T} \oplus \mathcal{E}'.\mathcal{T}, \mathcal{E}.\mathcal{V} \oplus \mathcal{E}'.\mathcal{V}, \mathcal{E}.\mathcal{X} \oplus \mathcal{E}'.\mathcal{X}, \mathcal{E}.\mathcal{F} \oplus \mathcal{E}'.\mathcal{F})$$

A.3 Attributed grammar (1) type binding

A.3.1 Declarations

$(D1)$ $D \downarrow \mathcal{E} \uparrow \mathcal{E}'$ ::= **type** T **is** T' **endtype**

$$\left\{ \begin{array}{l} use_T(T', \mathcal{E}.TE) \\ \rho(T') = T \\ \mathcal{T}' = \mathcal{E}.TE \oplus def_T(T, T') \\ \mathcal{E}' = \mathcal{E}.\{TE := \mathcal{T}'\} \end{array} \right.$$

$(D2)$ | **type** T **is**

$$\left\{ \begin{array}{l} \mathcal{T}' = def_T(T, no\_ren) \\ \mathcal{T}'' = \mathcal{T}' +_T \mathcal{E}.TE \end{array} \right.$$

...
$C_i(V_1^i : T_1^i, ..., V_{n_i}^i : T_{n_i}^i)$
$\{$ $use_T(T_1^i, ..., T_{n_i}^i, \mathcal{T}'')$
...
endtype
$\{$ $\mathcal{E}' = \mathcal{E}.\{TE := \mathcal{T}'\}$

$(D3)$ | **function** F([**in** | **out**]$V_1 : T_1, ...,$[**in** | **out**]$V_n : T_n$)[**:** T]
[**raises** $X_1, ..., X_p$] **is**
$\{$ $use_T(T_1, ..., T_n[, T], \mathcal{E}.TE)$
$I \downarrow \mathcal{E}$
endfunc

$(D6)$ | $D_1 \downarrow \mathcal{E} \uparrow \mathcal{E}'' D_2 \downarrow \mathcal{E}'' \uparrow \mathcal{E}'$

A.3.2 Patterns

$(P1)$ $P \downarrow \mathcal{E}$ ::= K
$\{$ $use_T(type\{K\}, \mathcal{E}.TE)$

$(P2)$ | $V : T$
$\{$ $use_T(T, \mathcal{E}.TE)$

$(P3)$ | **any** T
$\{$ $use_T(T, \mathcal{E}.TE)$

$(P4)$ | $C(V_1$**:=**$P_1 \downarrow \mathcal{E}, ..., V_n$**:=**$P_n \downarrow \mathcal{E}[, \bullet\bullet\bullet])$

$(P6)$ $P \downarrow \mathcal{E}$ ::= $P_0 \downarrow \mathcal{E}$ **of** T
$\{$ $use_T(T, \mathcal{E})$

A.3.3 Expressions

$(E1)$ $E \downarrow \mathcal{E}$ ::= K
$\{$ $use_T(type\{K\}, \mathcal{E}.TE)$

$(E21)$ | E_0 **of** T
$\{$ $use_T(T, \mathcal{E}.TE)$

A.4 Attributed grammar (2) variable and exception binding

A.4.1 Declarations

$(D2)$ $D \downarrow \mathcal{E} \uparrow \mathcal{E}'$ ::= **type** T **is**
  ...
  $C_i(V_1^1 : T_1^i, ..., V_{n_i}^i : T_{n_i}^i)$
    $\{\ \mathcal{V}_i = \{V_1^1 \mapsto T_1^1\} \oplus ... \oplus \{V_{n_i}^i \mapsto T_{n_i}^i\}$
  ...
  endtype
    $\{\ \mathcal{V} = \mathcal{V}_1 \odot ... \odot \mathcal{V}_p$

$(D3)$ | **function** $F([\textbf{in} \mid \textbf{out}]V_1 : T_1, ..., [\textbf{in} \mid \textbf{out}]V_n : T_n)[\ \textbf{:}\ T]$
  [**raises** $X_1, ..., X_p$] **is**
    $\begin{cases} \forall i \in 1..n \quad \textbf{in}V_i : T_i \quad \mathcal{V}_{\text{in}} \oplus = \{V_i \mapsto T_i\} \\ \forall i \in 1..n \quad \textbf{out}V_i : T_i \quad \mathcal{V}_{\text{out}} \oplus = \{V_i \mapsto T_i\} \\ \mathcal{V}'' = \mathcal{V}_{\text{in}} \oplus \mathcal{V}_{\text{out}} \\ use_E(X_1, ..., X_p, \mathcal{E}.EE) \\ \mathcal{X}'' = \{X_1 \mapsto \texttt{exn}\} \oplus ... \oplus \{X_p \mapsto \texttt{exn}\} \\ \mathcal{E}'' = \mathcal{E}.\{VE := \mathcal{V}_{\text{in}}, EE := \mathcal{X}''\} \end{cases}$
  $I\mathcal{E}''\mathcal{V}_{\text{out}}$
  endfunc

$(D5)$ | **exception** X
    $\begin{cases} \mathcal{X}' = \mathcal{E}.\mathcal{X} \oplus \{X \mapsto \texttt{exn}\} \\ \mathcal{E}' = \mathcal{E}.\{\mathcal{X} := \mathcal{X}'\} \end{cases}$

A.4.2 Patterns

$(P1)$ $P \downarrow \mathcal{E} \uparrow \mathcal{V}$ ::= K
    $\{\ \mathcal{V} = \emptyset$

$(P2)$ | $V : T$
    $\{\ \mathcal{V} = \{V \mapsto T\}$

$(P3)$ | **any** T
    $\{\ \mathcal{V} = \emptyset$

$(P4)$ | $C(V_1 \textbf{:=} P_1 \downarrow \mathcal{E} \uparrow \mathcal{V}_1, ..., V_n \textbf{:=} P_n \downarrow \mathcal{E} \uparrow \mathcal{V}_n[, \textbf{...}])$
    $\{\ \mathcal{V} = \mathcal{V}_1 \oplus ... \oplus \mathcal{V}_n$

$(P6)$ | $P_0 \downarrow \mathcal{E} \uparrow \mathcal{V}'$ **of** T
    $\{\ \mathcal{V} = \mathcal{V}'$

A.4.3 Expressions

$$
\begin{array}{lll}
(E2) & E \downarrow \mathcal{E} \quad ::= & V \\
 & & \quad \{ \ use_V(V, \mathcal{E}.VE) \\
(E5) & & \textbf{eval} \\
 & & \quad [P_1 \downarrow \mathcal{E} \uparrow \mathcal{V}_1 \textbf{:=}] \ F_1(V_1^1 \textbf{:=} E_1^1 \downarrow \mathcal{E} | P_1^1 \downarrow \mathcal{E} \uparrow \mathcal{V}_1^1, ..., V_{n_1}^1 \textbf{:=} E_{n_1}^1 \mathcal{E} | P_{n_1}^1 \downarrow \mathcal{E} \uparrow \mathcal{V}_{n_1}^1) \\
 & & \quad\quad \{ \ \mathcal{V}^1 = [\mathcal{V}_1 \oplus] \mathcal{V}_1^1 \oplus ... \oplus \mathcal{V}_{n_1}^1 \\
 & & \quad ... \\
 & & \quad [P_p \downarrow \mathcal{E} \uparrow \mathcal{V}_1 \textbf{:=}] \ F_p(V_1^p \textbf{:=} E_1^p \downarrow \mathcal{E} | P_1^p \downarrow \mathcal{E} \uparrow \mathcal{V}_1^p, ..., V_{n_p}^p \textbf{:=} E_{n_p}^p \downarrow \mathcal{E} | P_{n_p}^p \downarrow \mathcal{E} \uparrow \mathcal{V}_{n_p}^p) \\
 & & \quad\quad \left\{ \begin{array}{l} \mathcal{V}^p = [\mathcal{V}_p \oplus] \mathcal{V}_1^p \oplus ... \oplus \mathcal{V}_{n_p}^p \\ \mathcal{E}' = \mathcal{E}.\{VE = VE \oslash \mathcal{V}^1 ... \oplus \mathcal{V}^p\} \end{array} \right. \\
 & & \quad \textbf{in} \ E_0 \downarrow \mathcal{E}' \\
(E8) & | & \quad M \downarrow \mathcal{E} \uparrow \mathcal{V}' \\
(E9) & | & \quad \textbf{case} \\
 & & \quad\quad M_0 \downarrow \mathcal{E} \uparrow \mathcal{V}_0 \rightarrow E_0 \downarrow \mathcal{E}_0 \\
 & & \quad\quad\quad \{ \ \mathcal{E}_0 = \mathcal{E}.\{VE = VE \oslash \mathcal{V}_0\} \\
 & & \quad\quad ... \\
 & & \quad\quad M_n \downarrow \mathcal{E} \uparrow \mathcal{V}_n \rightarrow E_n \downarrow \mathcal{E}_n \\
 & & \quad\quad\quad \{ \ \mathcal{E}_n = \mathcal{E}.\{VE = VE \oslash \mathcal{V}_n\} \\
 & & \quad \textbf{endcase} \\
(E15) & | & \quad E \downarrow \mathcal{E}.V_0 \\
(E16) & | & \quad E_0 \downarrow \mathcal{E}.\{V_1 \textbf{:=} E_1 \downarrow \mathcal{E}', ..., V_n \textbf{:=} E_n \downarrow \mathcal{E}'\} \\
 & & \quad\quad \{ \ \mathcal{E}' = \mathcal{E} \oslash (\{V_1 \mapsto type\{V_1\}\} \oplus ... \{V_n \oplus type\{V_n\}\}) \\
(E17) & | & \quad E_0 \mathcal{E} = E_1 \mathcal{E} \\
(E19) & | & \quad \textbf{raise} \ X \\
 & & \quad\quad \{ \ use_E(X, \mathcal{E}.EE) \\
(E20) & | & \quad \textbf{trap} \ E_0 \downarrow \mathcal{E} \ \textbf{handle} \\
 & & \quad\quad X_1 \rightarrow E_1 \downarrow \mathcal{E} \\
 & & \quad\quad\quad \{ \ use_E(X_1, \mathcal{E}.EE) \\
 & & \quad\quad ... \\
 & & \quad\quad X_n \rightarrow E_n \downarrow \mathcal{E} \\
 & & \quad\quad\quad \{ \ use_E(X_n, \mathcal{E}.EE) \\
 & & \quad \textbf{endtrap} \\
(E21) & | & \quad E_0 \downarrow \mathcal{E} \ \textbf{of} \ T
\end{array}
$$

A.4.4 Instructions

$$
\begin{array}{lll}
(I1) & I \downarrow \mathcal{E} \uparrow \mathcal{V} \quad ::= & \textbf{return } [E \downarrow \mathcal{E}] \, [\textbf{with } V_1 := E_1 \downarrow \mathcal{E}, ..., V_n := E_n \downarrow \mathcal{E}] \\
& & \quad \{ \ \mathcal{V} = \{V_1 \mapsto type\{E_1\}\} \oplus ... \oplus \{V_n \mapsto type\{E_n\}\} \\
(I2) & \quad | & \textbf{eval} \\
& & \quad [P_1 \downarrow \mathcal{E} \uparrow \mathcal{V}_1 :=] \ F_1(V_1^1 := E_1^1 \downarrow \mathcal{E} | P_1^1 \downarrow \mathcal{E} \uparrow \mathcal{V}_1^1, ..., V_{n_1}^1 := E_{n_1}^1 \mathcal{E} | P_{n_1}^1 \downarrow \mathcal{E} \uparrow \mathcal{V}_{n_1}^1) \\
& & \quad \{ \ \mathcal{V}^1 = [\mathcal{V}_1 \oplus] \mathcal{V}_1^1 \oplus ... \oplus \mathcal{V}_{n_1}^1 \\
& & \quad ... \\
& & \quad [P_p \downarrow \mathcal{E} \uparrow \mathcal{V}_1 :=] \ F_p(V_1^p := E_1^p \downarrow \mathcal{E} | P_1^p \downarrow \mathcal{E} \uparrow \mathcal{V}_1^p, ..., V_{n_p}^p := E_{n_p}^p \downarrow \mathcal{E} | P_{n_p}^p \downarrow \mathcal{E} \uparrow \mathcal{V}_{n_p}^p) \\
& & \quad \left\{ \begin{array}{l} \mathcal{V}^p = [\mathcal{V}_p \oplus] \mathcal{V}_1^p \oplus ... \oplus \mathcal{V}_{n_p}^p \\ \mathcal{E}' = \mathcal{E}.\{ VE = VE \oslash \mathcal{V}^1 ... \oplus \mathcal{V}^p \} \end{array} \right. \\
& & \textbf{in } I_0 \mathcal{E}' \mathcal{V}' \\
& & \quad \{ \ \mathcal{V} = \mathcal{V}' \\
(I3) & \quad | & \textbf{case} \\
& & \quad M_0 \downarrow \mathcal{E} \uparrow \mathcal{V}_0 \rightarrow I_0 \downarrow \mathcal{E}_0 \uparrow \mathcal{V}_0' \\
& & \quad \{ \ \mathcal{E}_0 = \mathcal{E}.\{ VE = VE \oslash \mathcal{V}_0 \} \\
& & \quad ... \\
& & \quad M_n \downarrow \mathcal{E} \uparrow \mathcal{V}_n \rightarrow I_n \downarrow \mathcal{E}_n \\
& & \quad \{ \ \mathcal{E}_n = \mathcal{E}.\{ VE = VE \oslash \mathcal{V}_n \} \\
& & \textbf{endcase} \\
& & \quad \{ \ \mathcal{V} = \mathcal{V}_0 \oslash ... \mathcal{V}_n \\
& & \textbf{endcase} \\
(I7) & \quad | & \textbf{trap } I_0 \mathcal{E} \mathcal{V}_0 \textbf{ handle} \\
& & \quad X_1 \rightarrow I_1 \mathcal{E} \mathcal{V}_1 \\
& & \quad \{ \ use_E(X_1, \mathcal{E}.EE) \\
& & \quad ... \\
& & \quad X_n \rightarrow I_n \mathcal{E} \mathcal{V}_n \\
& & \quad \{ \ use_E(X_n, \mathcal{E}.EE) \\
& & \textbf{endtrap} \\
& & \quad \{ \ \mathcal{V} = \mathcal{V}_1 \oslash ... \oslash \mathcal{V}_n \\
(I8) & \quad | & F(V_1 := E_1 \downarrow \mathcal{E} | P_1 \downarrow \mathcal{E} \uparrow \mathcal{V}_1, ..., V_n := E_n \downarrow \mathcal{E} | P_n \downarrow \mathcal{E} \uparrow \mathcal{V}_n) \\
& & \quad \{ \ \mathcal{V} = \mathcal{V}_1 \oplus ... \oplus \mathcal{V}_n
\end{array}
$$

A.5 Attributed grammar (3) typing

For typing an function binding, two traverse are needed:

- an descendant one to bind *use*-occurrences of functions;
- a ascendant one to bind occurrences of functions and to calculate the set of possible type for expressions.

A.5.1 Declarations

$(D1)$ $D \downarrow \mathcal{E} \uparrow \mathcal{E}'$ ::= **type** T **is** T' **endtype**
$$\left\{ \begin{array}{l} \mathcal{F}' = \mathcal{E}.FE \oplus \widehat{\rho}(\texttt{constructor}(T', \mathcal{E}.FEq)) \\ \mathcal{E}' = \mathcal{E}.\{FE = \mathcal{F}'\} \end{array} \right.$$

$(D2)$ | **type** T **is**
...
$C_i(V_1^1 : T_1^i, ..., V_{n_i}^i : T_{n_i}^i)$
$$\left\{ \begin{array}{l} \mathcal{F}_i = def_F(C_i, \{j \mapsto (V_j^i \mapsto T_j^i) \mid j \in 1..n_i\},\ \emptyset, T, pos_i, \textbf{true}) \end{array} \right.$$
...
endtype
$$\left\{ \begin{array}{l} \mathcal{F}' = \mathcal{F}_1 \oplus ... \oplus \mathcal{F}_p \mathcal{E}.\mathcal{F} \\ \mathcal{E}' = \mathcal{E}.\{FE = \mathcal{F}'\} \end{array} \right.$$

$(D3)$ | **function** F([**in** | **out**]$V_1 : T_1, ...,$ [**in** | **out**]$V_n : T_n$)[**:** T]
[**raises** $X_1, ..., X_p$] **is**
$$\left\{ \begin{array}{l} \mathcal{F}' = \mathcal{E}.\mathcal{F} \oplus def_F(F, \{i \mapsto (V_i \mapsto T_i) \mid i \in 1..n\}, \mathcal{V}_{\text{out}}, \mathcal{X}'', T, pos, \textbf{false}) \\ \mathcal{E}'' = \mathcal{E}.\{FE := \mathcal{F}'\} \end{array} \right.$$
$I \downarrow \mathcal{E}'' \downarrow T \uparrow \mathcal{T}$
$$\left\{ \begin{array}{l} [\texttt{if } T \notin \mathcal{T} \texttt{ then } \textit{type error}] \\ \texttt{if } \textit{void} \notin \mathcal{T} \texttt{ then } \textit{type error} \end{array} \right.$$
endfunc

A.5.2 Patterns

$(P1)$ $P \downarrow \mathcal{E} \uparrow \mathcal{T}$ ::= K
$$\left\{ \begin{array}{l} \mathcal{T} = \{type\{K\}\} \end{array} \right.$$

$(P2)$ | $V : T'$
$$\left\{ \begin{array}{l} \mathcal{T} = \{T\} \end{array} \right.$$

$(P3)$ | **any** T
$$\left\{ \begin{array}{l} \mathcal{T} = \{T\} \end{array} \right.$$

$(P4)$ | $C(V_1 \textbf{:=} P_1 \downarrow \mathcal{E} \uparrow \mathcal{T}_1, ..., V_n \textbf{:=} P_n \downarrow \mathcal{E} \uparrow \mathcal{T}_n[, \bullet\bullet\bullet])$
$$\left\{ \begin{array}{l} binding[C] = use_F(C, \mathcal{E}.FE) \\ bind = \texttt{select\_binding}(binding[C], V_1, \mathcal{T}_1, \textbf{true}) \cap \\ \qquad ... \\ \qquad \texttt{select\_binding}(binding[C], V_n, \mathcal{T}_n, \textbf{true}) \\ \mathcal{T} = \texttt{select\_result}(bind) \end{array} \right.$$

$(P6)$ $P_0 \downarrow \mathcal{E} \uparrow \mathcal{T}_0$ **of** T
$$\left\{ \begin{array}{l} \texttt{if } T \notin \mathcal{T}_0 \texttt{ then } \textit{type error} \\ \mathcal{T} = \{T\} \end{array} \right.$$

A.5.3 Match expressions

$(M1)$ $M \downarrow \mathcal{E}$ ::= $E \downarrow \mathcal{E} \downarrow T \uparrow \mathcal{T}$ **::** $P \downarrow \mathcal{E} \downarrow T' \uparrow \mathcal{T}'$
$$\left\{ \begin{array}{l} \texttt{if } \mathcal{T} \cap \mathcal{T}' = \emptyset \texttt{ then } \textit{type error} \\ \texttt{if } |\mathcal{T} \cap \mathcal{T}'| > 1 \texttt{ then } \textit{ambiguous expression} \\ T = \mathcal{T} \cap \mathcal{T}' \end{array} \right.$$

$(M2)$ | $M_1 \downarrow \mathcal{E}$ **when** $E \downarrow \mathcal{E} \downarrow$ **bool** $\uparrow \mathcal{T}$
$$\left\{ \begin{array}{l} \texttt{if } \textbf{bool} \notin \mathcal{T} \texttt{ then } \textit{type error} \end{array} \right.$$

$(M3)$ | $M_1 \downarrow \mathcal{E}$ **and** $M_2 \downarrow \mathcal{E}$

$(M4)$ | $M_1 \downarrow \mathcal{E}$ **or** $M_2 \downarrow \mathcal{E}$

A.5.4 Expressions

$(E1)$ $E \downarrow \mathcal{E} \downarrow T \uparrow \mathcal{T}$ ::= K

$$\begin{cases} \mathcal{T} = \{type\{K\}\} \\ \texttt{if } type\{K\} \neq T \texttt{ then } type\ error \end{cases}$$

$(E2)$ | V

$$\begin{cases} \mathcal{T} = use_V(V, \mathcal{E}.VE) \\ \texttt{if } T \notin \mathcal{T} \texttt{ then } type\ error \end{cases}$$

$(E3)$ | $C(V_1 := E_1 \downarrow \mathcal{E} \downarrow T_1' \uparrow \mathcal{T}_1, ..., V_n := E_n \downarrow \mathcal{E} \downarrow T_n' \mathcal{T}_n \uparrow)$

$$\begin{cases} binding[C] = use_F(C, \mathcal{E}.FE) \\ bind = \texttt{select\_binding}(binding[C], V_1, \mathcal{T}_1, \textbf{true}) \cap \\ \quad ... \\ \qquad \texttt{select\_binding}(bindC, V_n, \mathcal{T}_n, \textbf{true}) \\ \mathcal{T} = \texttt{select\_result}(bind) \\ bind = \texttt{select\_binding}(binding[C], Res, T) \\ \texttt{if } bind = \emptyset \texttt{ then } type\ error \\ T_1' = bind.FormalP(V_1) \\ ... \\ T_n' = bind.FormalP(V_n) \end{cases}$$

$(E5)$ | **eval**

...

$[P_i \downarrow \mathcal{E} \uparrow \mathcal{T}_i' =] \ F_1(V_1^i := E_1^i \downarrow \mathcal{E} \downarrow T_1^i | P_1^i \downarrow \mathcal{E} \uparrow \mathcal{T}_1^i, ..., V_{n_i}^i := E_{n_i}^i \downarrow \mathcal{E} \downarrow T_{n_i}^i | P_{n_i}^i \downarrow \mathcal{E} \uparrow \mathcal{T}_{n_i}^i)$

$$\begin{cases} binding[F_i] = use_F(F_i, \mathcal{E}.FE) \\ bind = \texttt{select\_binding}(binding[F_i], V_1^i, \mathcal{T}_1^i, \textbf{false}) \cap \\ \quad ... \\ \qquad \texttt{select\_binding}(binding[F_i], V_{n_i}^i, \mathcal{T}_{n_i}^i, \textbf{false}) \\ \mathcal{T}_i = \texttt{select\_result}(bind) \\ \texttt{if } \mathcal{T}_i' \cap \mathcal{T}_i = \emptyset \texttt{ then } type\ error \\ \texttt{if } |\mathcal{T}_i \cap \mathcal{T}_i'| > 1 \texttt{ then } ambiguous\ expression \end{cases}$$

...

in $E_0 \downarrow \mathcal{E} \downarrow T_0 \uparrow \mathcal{T}_0$

$$\begin{cases} \mathcal{T} = \mathcal{T}_0 \\ T_0 = T \end{cases}$$

$(E8)$ | $M \downarrow \mathcal{E}$
$\{\ \mathcal{T} = \{\mathbf{bool}\}$

$(E9)$ | **case**
$M_0 \downarrow \mathcal{E} \rightarrow E_0 \downarrow \mathcal{E} \downarrow T' \uparrow \mathcal{T}_0$
...
$M_n \downarrow \mathcal{E} \rightarrow E_n \downarrow \mathcal{E} \downarrow T' \uparrow \mathcal{T}_n$
endcase
$\begin{cases} \mathcal{T} = \mathcal{T}_0 \cap ... \cap \mathcal{T}_n \\ \texttt{if } \mathcal{T} = \emptyset \texttt{ then } \textit{type error} \\ T' = T \end{cases}$

$(E15)$ | $E\mathcal{E} \downarrow T' \uparrow \mathcal{T}_0.V_0$
$\begin{cases} bind = \texttt{select\_binding}(\mathrm{Ran}(\mathcal{E}.FE), V_0,\ \emptyset, \mathbf{true}) \\ \texttt{if } bind = \emptyset \texttt{ then } \textit{type error} \\ T' = \texttt{select\_result}(\texttt{select\_binding}(bind, V_0, T, \mathbf{true})) \end{cases}$

$(E16)$ | $E_0\mathcal{E} \downarrow T' \uparrow \mathcal{T}_0.\{V_1 \texttt{:=} E_1\mathcal{E} \downarrow T'_1 \uparrow \mathcal{T}_1, ..., V_n \texttt{:=} E_n\mathcal{E} \downarrow T'_n \uparrow \mathcal{T}_n\}$
$\begin{cases} bind = \texttt{select\_binding}(\mathrm{Ran}(\mathcal{E}.FE), V_1, \mathcal{T}_1, \mathbf{true}) \cap \\ \quad ... \\ \quad \texttt{select\_binding}(\mathrm{Ran}(\mathcal{E}.FE), V_n, \mathcal{T}_n, \mathbf{true}) \\ \texttt{if } bind = \emptyset \texttt{ then } \textit{type error} \\ \mathcal{T} = \texttt{select\_result}(bind) \\ bind = \texttt{select\_binding}(bind, Res, T) \\ T' = T \\ T'_1 = \texttt{select\_binding}(bind, V_1,\ \emptyset, \mathbf{true}) \\ ... \\ T'_n = \texttt{select\_type}(bind, V_n,\ \emptyset, \mathbf{true}) \end{cases}$

$(E17)$ | $E_0\mathcal{E} \downarrow T' \uparrow \mathcal{T}_0 = E_1\mathcal{E} \downarrow T' \uparrow \mathcal{T}_1$
$\begin{cases} \mathcal{T} = \{\mathbf{bool}\} \\ \texttt{if } \mathcal{T}_0 \cap \mathcal{T}_1 = \emptyset \texttt{ then } \textit{error type} \\ \texttt{if } |\mathcal{T}_0 \cap \mathcal{T}_1| > 1 \texttt{ then } \textit{ambiguous type} \\ T' = \mathcal{T}_0 \cap \mathcal{T}_1 \end{cases}$

$(E20)$ | **trap** $E_0\mathcal{E} \downarrow T' \uparrow \mathcal{T}_0$ **handle**
$X_1 \rightarrow E_1\mathcal{E} \downarrow T' \uparrow \mathcal{T}_1$
...
$X_n \rightarrow E_n\mathcal{E} \downarrow T' \uparrow \mathcal{T}_n$
endtrap
$\begin{cases} \mathcal{T} = \mathcal{T}_0 \cap \mathcal{T}_1 \cup ... \cup \mathcal{T}_n \\ \texttt{if } \mathcal{T} = \emptyset \texttt{ then } \textit{error type} \\ T' = T \end{cases}$

$(E21)$ | $E_0\mathcal{E} \downarrow T' \uparrow \mathcal{T}_0$ **of** T'
$\begin{cases} \texttt{if } T' \notin \mathcal{T}_0 \texttt{ then } \textit{error type} \\ \texttt{if } T' \neq T \texttt{ then } \textit{error type} \\ T' = T \end{cases}$

A.5.5 Instructions

$(I1)$ $I \downarrow \mathcal{E} \downarrow T \uparrow \mathcal{T}$ ::= **return** $[E \downarrow \mathcal{E} \downarrow T' \uparrow \mathcal{T}']$ [**with** $V_1 := E_1 \downarrow \mathcal{E} \downarrow T_1 \uparrow \mathcal{T}_1, ..., V_n := E_n \downarrow \mathcal{E} \downarrow T_n \uparrow \mathcal{T}_n$]

$$\begin{cases} \mathcal{T} = [\mathcal{T}']\{void\} \\ T' = T \\ T_1 = \mathcal{E}.VE(V_1) \\ ... \\ T_n = \mathcal{E}.VE(V_n) \end{cases}$$

$(I2)$ | **eval**

...

$[P_i \downarrow \mathcal{E} \uparrow \mathcal{T}'_i =]\ F_1(V_1^i := E_1^i \downarrow \mathcal{E} \downarrow T_1^i | P_1^i \downarrow \mathcal{E} \uparrow \mathcal{T}_1^i, ..., V_{n_i}^i := E_{n_i}^i \downarrow \mathcal{E} \downarrow T_{n_i}^i | P_{n_i}^i \downarrow \mathcal{E} \uparrow \mathcal{T}_{n_i}^i)$

$$\begin{cases} binding[F_i] = use_F(F_i, \mathcal{E}.FE) \\ bind = \texttt{select\_binding}(binding[F_i], V_1^i, \mathcal{T}_1^i, \textbf{false}) \cap \\ \qquad ... \\ \qquad \texttt{select\_binding}(binding[F_i], V_{n_i}^i, \mathcal{T}_{n_i}^i, \textbf{false}) \\ \mathcal{T}_i = \texttt{select\_result}(bind) \\ \texttt{if } \mathcal{T}'_i \cap \mathcal{T}_i = \emptyset \texttt{ then } type\ error \\ \texttt{if } |\mathcal{T}_i \cap \mathcal{T}'_i| > 1 \texttt{ then } ambiguous\ expression \end{cases}$$

...

in $I_0 \downarrow \mathcal{E} \downarrow T_0 \uparrow \mathcal{T}_0$

$$\begin{cases} \mathcal{T} = \mathcal{T}_0 \\ T_0 = T \end{cases}$$

$(I3)$ | **case**

$M_0 \downarrow \mathcal{E} \rightarrow I_0 \downarrow \mathcal{E} \downarrow T' \uparrow \mathcal{T}_0$

...

$M_n \downarrow \mathcal{E} \rightarrow I_n \downarrow \mathcal{E} \downarrow T' \uparrow \mathcal{T}_n$

endcase

$$\begin{cases} \mathcal{T} = \mathcal{T}_0 \cap ... \cap \mathcal{T}_n \\ \texttt{if } \mathcal{T} = \emptyset \texttt{ then } type\ error \\ T' = T \end{cases}$$

$(I7)$ | **trap** I_0 **handle**

$X_1 \rightarrow I_1 \mathcal{E} \downarrow T' \uparrow \mathcal{T}_1$

...

$X_n \rightarrow I_n \mathcal{E} \downarrow T' \uparrow \mathcal{T}_n$

endtrap

$$\begin{cases} \mathcal{T} = \mathcal{T}_0 \cap \mathcal{T}_1 \cup ... \cup \mathcal{T}_n \\ \texttt{if } \mathcal{T} = \emptyset \texttt{ then } error\ type \\ T' = T \end{cases}$$

$(I8)$ | $F(V_1 := E_1 \downarrow \mathcal{E} \downarrow T_1 | P_1 \downarrow \mathcal{E} \uparrow \mathcal{T}_1, ..., V_n := E_n \downarrow \mathcal{E} \downarrow T_n | P_n \downarrow \mathcal{E} \uparrow \mathcal{T}_n)$

$$\begin{cases} binding[F] = use_F(F, \mathcal{E}.FE) \\ bind = \texttt{select\_binding}(binding[F_i], V_1, \mathcal{T}_1, \textbf{false}) \cap \\ \qquad ... \\ \qquad \texttt{select\_binding}(binding[F], V_n, \mathcal{T}_n, \textbf{false}) \\ \mathcal{T} = \texttt{select\_result}(bind) \\ bind = \texttt{select\_binding}(bind, Res, T) \\ T_1 = \texttt{select\_binding}(bind, V_1, \emptyset, \textbf{false}) \\ ... \\ T_n = \texttt{select\_binding}(bind, V_n, \emptyset, \textbf{false}) \end{cases}$$

References

[BH88] Pascal Bouchon and Jean-Michel Houdouin. Analyseur LOTOS pour CÆSAR. Rapport de fin d'e'tudes ENSIMAG, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, June 1988.

[ISO88] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[JGL+95] Alan Jeffrey, Hubert Garavel, Guy Leduc, Charles Pecheur, and Mihaela Sighireanu. Towards a proposal for datatypes in E-LOTOS. Annex A of ISO/IEC JTC1/SC21/WG7 Second Working Draft on Enhancements to LOTOS. Output document of the edition meeting, Ottawa (Canada), July, 20–26, 1995, October 1995.

[SG95] Mihaela Sighireanu and Hubert Garavel. Application of the Proposed E-LOTOS Datatype Language to the Description of OSI and ODP Standards. Rapport SPECTRE 95-17, VERIMAG, Grenoble, December 1995. Input document [xxx] of the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Liège (Belgium), December, 18–21, 1995.