E-LOTOS user language

Source: France[*], Romania[†]

Output document of ISO/IEC JTC1/SC21/WG7/1.21.20.2.3
'Enhancements to LOTOS'

Kansas City meeting, May 1996

DRAFT of 1996/09/30

Contents

1 Introduction **4**

2 Overview **4**
2.1 Syntax 4
2.2 Static semantics 5
2.3 Translation to the core language 7

3 Declarations **7**
3.1 Overview 7
3.2 Type synonym 9
3.3 Type declaration 9
3.4 Channel declaration 10
3.5 Exception declaration 10
3.6 Process declaration 10
3.7 Process declaration with in/out parameters 11
3.8 Function declaration 11
3.9 Procedure declaration 12

4 Record type **12**
4.1 Overview 12
4.2 Empty record 13
4.3 In record type 13
4.4 In/out record type 14

5 Record of typed gates/exceptions **14**
5.1 Empty record 15
5.2 Union 15

6 Patterns **15**
6.1 Wildcard 16
6.2 Variable pattern 16

[*]Represented by Hubert Garavel (Inria Rhône-Alpes, Verimag).
[†]Represented by Mihaela Sighireanu (RSI).

6.3 Expression pattern . . . 17
6.4 Constructor application . . . 17
6.5 Explicit typing . . . 17

7 Record pattern **18**
7.1 Empty record . . . 18
7.2 Attributed pattern list . . . 18
7.3 Tuple pattern . . . 19

8 Match expressions **19**
8.1 Simple match . . . 20
8.2 Guarded match . . . 20
8.3 And match . . . 20
8.4 Or match . . . 21

9 Value expressions **21**
9.1 Overview . . . 21
9.2 Primitive constants . . . 23
9.3 Variable . . . 24
9.4 Nondeterministic termination . . . 24
9.5 Constructor application . . . 24
9.6 Raising exception . . . 25
9.7 Assignment . . . 25
9.8 Sequential composition . . . 25
9.9 Trap . . . 26
9.10 General case . . . 26
9.11 Usual case . . . 28
9.12 Match expression . . . 28
9.13 If-then-else . . . 29
9.14 Conjunction . . . 29
9.15 Disjunction . . . 29
9.16 Equality . . . 30
9.17 Inequality . . . 30
9.18 Select field . . . 30
9.19 Field updating . . . 31
9.20 Variable declaration . . . 31
9.21 Rename . . . 32
9.22 Function call . . . 33
9.23 Procedure call . . . 33
9.24 Iteration . . . 33
9.25 Breakable iteration . . . 34
9.26 Break loop . . . 35
9.27 Explicit typing . . . 35

10 Record expression **35**
10.1 Overview . . . 35
10.2 Empty record . . . 36
10.3 Union . . . 36
10.4 Tuple . . . 37

11 In/out parameters **37**
11.1 Overview . . . 37
11.2 Actual parameters . . . 38

11.3 Empty . . . 38
11.4 Record . . . 38
11.5 Tuple . . . 39

12 Actual gate/exception parameters 39
12.1 Overview . . . 39
12.2 Empty record . . . 40
12.3 Record . . . 40
12.4 Tuple . . . 41

13 Behaviour expression 41
13.1 Overview . . . 41
13.2 Action . . . 44
13.3 Internal action . . . 45
13.4 Termination . . . 45
13.5 Exception raising . . . 46
13.6 Inaction . . . 46
13.7 Time block . . . 46
13.8 Delay . . . 47
13.9 Assignment . . . 47
13.10 Sequential composition . . . 47
13.11 Disabling . . . 48
13.12 Synchronization . . . 48
13.13 Concurrency . . . 48
13.14 Choice . . . 49
13.15 Choice over values . . . 49
13.16 Trap . . . 49
13.17 General case . . . 50
13.18 Variable declaration . . . 51
13.19 Gate hiding . . . 51
13.20 Renaming . . . 52
13.21 Process instantiation . . . 52
13.22 Iteration . . . 53
13.23 Interleaving . . . 53
13.24 Parallel over values . . . 54
13.25 Parallel composition . . . 54
13.26 Usual case . . . 55
13.27 If-then-else . . . 55
13.28 Breakable iteration . . . 55
13.29 Breaking iteration . . . 56

A Upward compatibility with LOTOS 57
A.1 General structure . . . 57
A.2 Data part . . . 57
A.3 Behaviour part . . . 58
A.3.1 Declarations . . . 58
A.3.2 Behaviour expressions . . . 58

1 Introduction

2 Overview

2.1 Syntax

Remark: For the user language we proposed two approaches:

1. A syntax in which exceptions and gates are unified. However, to avoid some confusions, and to preserve the user sense about exception, we give them to different notation for their identifiers in syntax: X for exceptions and G for gates. We mention that this identifiers belong over the same domain. The same remark is true for exceptions and gate type identifiers.

2. A syntax in which exceptions and gates are differentiated. This syntax is boxed.

A discussion about each of these approaches should be provided.

The terminals of the abstract syntax are:

identifier domain	*meaning*	*abbreviations*
SCon	special constants	K
Var	variable identifiers	V
Typ	type identifiers	S
Con	constructor identifiers	C
Proc	process identifiers	Π
Fun	function identifiers	F
Gat	gate and/or exception identifiers	G, X
$\boxed{\text{Exc}}$	$\boxed{\text{exception identifiers}}$	$\boxed{X}$
ChName	channels (gate type) identifiers	Γ, Ξ
$\boxed{\text{ExcName}}$	$\boxed{\text{exception type identifiers}}$	$\boxed{\Xi}$

The non-terminals are:

symbol domain	*meaning*	*abbreviations*
Decl	declaration	D
RTyp	record type	RT
RGat	record typed-gate	RG
$\boxed{\text{RExc}}$	record typed-exceptions	RX
Pat	pattern	P
RPat	record pattern	RP
EMatch	match expression	EM
Exp	expression	E
RExp	record expression	RE
IOPar	in/out parameters	IOP
GPar	gate parameters	GP
$\boxed{\text{XPar}}$	$\boxed{\text{exception parameters}}$	$\boxed{XP}$
Behav	behavior expression	B

In the grammars, non-primitive constructs (which are defined by translation into terms of primitive constructs) are marked with a "$\star$".

2.2 Static semantics

The static semantics is given by a series of judgments, such as "$\mathcal{C} \vdash D \Rightarrow \mathcal{C}'$" meaning "in the context $\mathcal{C}$, the declaration d gives the context $\mathcal{C}'$". The context gives the bindings for any free identifiers, and is given by the grammar:

$$
\begin{array}{rcll}
\mathcal{C} & ::= & S \mapsto S' & \textit{type} \\
 & \mid & V \mapsto (S, d) & \textit{variable} \\
 & \mid & C \mapsto \{\mathit{profile}_1, \ldots, \mathit{profile}_n\} & \textit{constructor} \\
 & \mid & F \mapsto \{\mathit{profile}_1, \ldots, \mathit{profile}_n\} & \textit{function} \\
 & \mid & \Pi \mapsto \mathit{profile} & \textit{process} \\
 & \mid & \Gamma \mapsto t & \textit{channel} \\
 & \mid & \boxed{\Xi \mapsto t} & \\
 & \mid & G \mapsto \mathbf{gate}\ t & \textit{gate} \\
 & \mid & \boxed{X \mapsto \mathbf{exn}\ t} & \\
 & \mid & \{\} & \textit{empty context} \\
 & \mid & \mathcal{C} + \mathcal{C} & \textit{disjoint union} \\
\mathit{profile} & ::= & rg, rt, \boxed{rx} \ \rightarrow\ \mathbf{exit}\ t\ \rightarrow\ \mathit{uid} & \textit{profiles} \\
d & ::= & \mathit{def} \ \mid\ \mathit{undef} & \textit{variable initialization} \\
t & ::= & S & \textit{type expression} \\
 & \mid & \mathbf{none} & \\
 & \mid & (rt) & \\
rt & ::= & () & \textit{empty list of typed variable} \\
 & \mid & V \mapsto (S, d), rt & \textit{list construction} \\
rg & ::= & () & \textit{empty list of typed gates} \\
 & \mid & G \mapsto \mathbf{gate}\ t, rg & \textit{list construction} \\
rx & ::= & () & \textit{empty list of typed exceptions} \\
 & \mid & X \mapsto \mathbf{exn}\ t, rx & \textit{list construction}
\end{array}
$$

Contexts for types, variables, channels, gates, and process are finite maps. Contexts for constructors and functions are set finite maps.

Note that the grammar of record types overlaps with that of contexts. Whenever "rt" belongs over contexts, ',' (composition of records) is '+' (composition of contexts); $\mathbf{in}(rt) \overset{\mathtt{def}}{=} \{V \mapsto (S, d) \in rt \mid d = \mathit{def}\}$, and $\mathbf{out}(rt) \overset{\mathtt{def}}{=} \{V \mapsto (S, d) \in rt \mid d = \mathit{undef}\}$.

Similar for the grammar of list of gate and exception parameters. Whenever "rg" (resp. "rx") belongs over contexts, ',' (composition of records) is '+' (composition of contexts).

We give below the definitions of the operations we will used over contexts. When A and B are sets, $\mathrm{Fin}(A)$ denotes the set of finite subsets of A, and $A \overset{\mathrm{fin}}{\rightarrow} B$ 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 $\mathrm{Dom}(f)$ and $\mathrm{Ran}(f)$. A finite map will often be written explicitly in the form $\{a_1 \mapsto b_1, \cdots, a_k \mapsto b_k\}$, $k \geq 0$; in particular, the empty map is $\{\}$.

When f and g are finite maps we define:

- $f + g$ with domain $\mathrm{Dom}(f) \cup \mathrm{Dom}(g)$ and values:

$$(f+g)(a) = \begin{cases} f(a) & \textbf{if } a \notin \mathrm{Dom}(g) \\ g(a) & \textbf{if } a \notin \mathrm{Dom}(f) \\ \mathit{error} & \textbf{otherwise} \end{cases}$$

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

- $f \oplus g$ with domain $\mathrm{Dom}(f) \cup \mathrm{Dom}(g)$ and values:

$$(f \oplus g)(a) = \begin{cases} f(a) & \textbf{if } a \notin \mathrm{Dom}(g) \\ g(a) & \textbf{otherwise} \end{cases}$$

  This operator is called f *modified by* (or *overridden*) g.

- $f \odot g$ with domain $\mathrm{Dom}(f) \cup \mathrm{Dom}(g)$ and values:

$$(f \odot g)(a) = \begin{cases} f(a) & \textbf{if } a \notin \mathrm{Dom}(g) \\ g(a) & \textbf{if } a \notin \mathrm{Dom}(f) \\ f(a) & \textbf{if } g(a) = f(a) \\ \mathit{error} & \textbf{otherwise} \end{cases}$$

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

- $f \ominus \{a_1, ..., a_n\}$ where $\{a_1, ..., a_n\} \subset \mathrm{Dom}(f)$, is a map with domain $\mathrm{Dom}(f) \setminus \{a_1, ..., a_n\}$ and values:

$$(f \ominus \{a_1, ..., a_n\})(a) = f(a) \textbf{ if } a \notin \{a_1, ..., a_n\}$$

  This operator is usually called restriction of domain of f.

When the range of a partial map is a finite set of subsets, $A \overset{\mathrm{sfin}}{\rightarrow} \mathrm{Fin}(B)$ denotes *finite set-maps* from A to B. If f and g are set finite maps, the set finite map $f + g$, called *disjoint* composition, has domain $\mathrm{Dom}(f) \cup \mathrm{Dom}(g)$ and values:

$$(f \oplus g)(a) = \begin{cases} f(a) & \textbf{if } a \notin \mathrm{Dom}(g) \\ g(a) & \textbf{if } a \notin \mathrm{Dom}(f) \\ f(a) \ \cup \ g(a) & \textbf{if } g(a) \ \cap \ f(a) = \emptyset \\ \mathit{error} & \textbf{otherwise} \end{cases}$$

Notes:

- All semantics objects in the static semantics are built from identifiers. With each special constant K we associate a type name *type*(K) which is either **integer**, **bool**, **real**, **time**, iff we suppose that this will be the built-in types.

- To solve overloading of constructors and functions, we attach at each profile definition an integer number, *uid*∈UIDs (unique identifier). We use the function *newid* to generate a new (unused) identifier at each call.

- In the abstract grammar used for semantics we use some simplifications. For instance, we have not given the "**end**" keyword for each construct. Also, phrases within single brackets ⟨⟩ are called *first options*. To reduce the number of rules, we have adopted the following convention:

  In each instance of a rule, the options must be either all present or all absent.

- The relation between static semantics of user language and static semantics of core language is summarized below:

user language	*core language*
$\mathcal{C} \vdash D \Rightarrow \mathcal{C}'$	$\mathcal{C} \vdash D \Rightarrow \mathcal{C}'$
$\mathcal{C} \vdash S \Rightarrow S'$	$\mathcal{C} \vdash T \Rightarrow$ **type** + *subtyping*
$\mathcal{C} \vdash RT \Rightarrow rt$	$\mathcal{C} \vdash RT \Rightarrow$ **record** + *subtyping*
$\mathcal{C} \vdash RG \Rightarrow rg$	
$\mathcal{C} \vdash RX \Rightarrow rx$	
$\mathcal{C} \vdash (P \Rightarrow t) \Rightarrow rt$	$\mathcal{C} \vdash (P \Rightarrow T) \Rightarrow (RT)$
$\mathcal{C} \vdash (RP \Rightarrow rt) \Rightarrow rt'$	$\mathcal{C} \vdash (RP \Rightarrow RT) \Rightarrow (RT')$
$\mathcal{C} \vdash EM \Rightarrow rt$	
	$\mathcal{C} \vdash (RV \Rightarrow RT) \Rightarrow (RT')$
$\mathcal{C} \vdash E \Rightarrow \textbf{exit}\ t$	$\mathcal{C} \vdash E \Rightarrow \textbf{exit}(RT)$
$\mathcal{C} \vdash RE \Rightarrow \textbf{exit}\ (rt)$	$\mathcal{C} \vdash RE \Rightarrow \textbf{exit}(RT)$
$\mathcal{C} \vdash (IOP \Rightarrow rt) \Rightarrow \mathcal{C}'$	$\mathcal{C} \vdash (RE, RP \Rightarrow RT) \Rightarrow \textbf{exit}(RT'), RT''$
$\mathcal{C} \vdash GP \Rightarrow rg$	
$\mathcal{C} \vdash XP \Rightarrow rx$	
$\mathcal{C} \vdash B \Rightarrow \textbf{exit}\ t$	$\mathcal{C} \vdash B \Rightarrow \textbf{exit}(RT)$

2.3 Translation to the core language

The translation of the user language in the core language follows the following steps:

Step 1 : makes a *syntactical* translation of all operators marked with $\star$. These operators are syntactic sugar of a set of reduced operators.

Step 2 : does static semantics checks and solves *overloading*.

Step 3 : does a *contextual* translation. "**...**" notation and user convenient notations are expanded into the core language.

The translation is given in terms of morphisms for each non-terminal of the abstract grammar. We synthesize below the profiles of this functions, where index u denotes a user language non-terminal domain, and index c denotes a core language non-terminal domain.

$[[.,.]] : \text{Context} \times \text{Decl}_u \rightarrow \text{Decl}_c$
$[[.,.]] : \text{Context} \times \text{RTyExp}_u \rightarrow \text{RTyExp}_c \times \text{RTyExp}_c$
$[[.,.]] : \text{Context} \times \text{Pat}_u \rightarrow \text{Pat}_c$
$[[.,.]] : \text{Context} \times \text{RPat}_u \times \text{RTypExp}_u \rightarrow \text{RPat}_c$
$[[.,.]] : \text{Context} \times \text{Exp}_u \rightarrow \text{Behav}_c$
$[[.,.]] : \text{Context} \times \text{RExp}_u \times \text{RTypExp}_u \rightarrow \text{Behav}_c$
$[[.,.]] : \text{Context} \times \text{IOPar}_u \times \text{RTypExp}_u \rightarrow \text{RExp}_c \times \text{RPat}_c$
$[[.,.]] : \text{Context} \times \text{Behav}_u \rightarrow \text{Behav}_c$

3 Declarations

3.1 Overview

Syntax

D ::= **type** S **is** S' **endtype** *type synonym* (D$_u$1)

| **type** S **is** *type* (D$_u$2)
$C_1\ [RT_1] \mid \ldots \mid C_p\ [RT_p]$
endtype $[S]$

$\star$ | **channel** Γ **is** S **endch** *channel* (D$_u$3)

$\star$ | **channel** Γ **is** RT **endch** *channel* (D$_u$4)

$\star$ | **exception** Ξ **is** S **endch** $\boxed{\textit{type of exception}}$ (D$_u$5)

$\star$ | **exception** Ξ **is** RT **endexc** $\boxed{\textit{type of exception}}$ (D$_u$6)

| **process** Π $[RG]$ $[(\textbf{[in]}\ V_1\!:\!S_1, ..., \textbf{[in]}\ V_n\!:\!S_n)]$ **:noexit** *(no-exiting) process* (D$_u$7)
$\boxed{\textbf{[raises}\ RX]}$ **is**
B
endproc $[\Pi]$

| **process** Π $[RG]$ $(\textbf{in} \mid \textbf{out}\ V_1\!:\!S_1, ..., \textbf{in} \mid \textbf{out}\ V_n\!:\!S_n)$ *(exiting) process* (D$_u$8)
$\boxed{\textbf{[raises}\ RX]}$ **is**
B
endproc $[\Pi]$

$\star$ | **function** F $[(\textbf{[in]}\ V_1\!:\!S_1, ..., \textbf{[in]}\ V_n\!:\!S_n)]$: S *function declaration* (D$_u$9)
$[\boxed{\textbf{raises}}\ RX]$ **is**
E
endfunc $[F]$

$\star$ | **function** F $(\textbf{in} \mid \textbf{out}\ V_1\!:\!S_1, ..., \textbf{in} \mid \textbf{out}\ V_n\!:\!S_n)$ *procedure declaration* (D$_u$10)
$[\boxed{\textbf{raises}}\ [RX]$ **is**
E
endfunc $[F]$

Remarks:

1. In (D$_u$2), $p \geq 0$; if $p = 0$ then S could be considered as an external type.
2. In (D$_u$2), (D$_u$9) and (D$_u$10), as in LOTOS, constructors and functions can be declared to be infixed.
3. In (D$_u$7) and (D$_u$9), the attribute of the formal parameter is by default "**in**".

Static semantics

The static semantics is given by assertions of form:

$\mathcal{C} \vdash D \Rightarrow \mathcal{C}'$

meaning "In the context $\mathcal{C}$, the declaration D is well formed and gives context $\mathcal{C}'$."

Translation to the core language

The translation function is defined by:

$$[[.,.]] \; : \; \text{Context} \times \text{Decl}_u \; \rightarrow \; \text{Decl}_c$$

Each declaration of the user language is translated in a declaration of the core language (see [JL96]).

3.2 Type synonym

Syntax

type S **is** S' **endtype**

Static semantics

$$\frac{\mathcal{C} \;\vdash\; S' \Rightarrow S''}{\mathcal{C} \;\vdash\; (\textbf{type}\ S\ \textbf{is}\ S'\ ) \Rightarrow (S \mapsto S'')}$$

Translation to the core language

Identity.

3.3 Type declaration

Syntax

type S **is** C_1 $[RT_1]$ | ... | C_p $[RT_p]$ **endtype** $[S]$

The default parameter for a constructor is "()", i.e. the empty record.

Static semantics

$$\frac{\begin{array}{lll} \mathcal{C}+(S \mapsto S) & \vdash & RT_1 \Rightarrow rt_1 \quad \cdots \quad \mathcal{C}+(S \mapsto S) \vdash RT_p \Rightarrow rt_p \\ & & (rt_1 \odot \ldots \odot rt_p) \;\neq\; error \\ & & \mathcal{C}_c = (+_{i=1}^{i=p}\ C_i \mapsto (), rt_i, () \;\rightarrow\; \textbf{exit}\ S \;\rightarrow\; newuid_i) \\ & & \mathcal{C}(C_1)((), rt_1, ())(S) = \cdots = \mathcal{C}(C_p)((), rt_p, ())(S) = \emptyset \end{array}}{\mathcal{C} \;\vdash\; (\textbf{type}\ S\ \textbf{is}\ C_1\ RT_1\ \ldots\ C_p\ RT_p\ ) \Rightarrow (S \mapsto S) + \mathcal{C}_c}$$

Translation to the core language

The constructor declaration C RT is translated into C_uid RT_i, i.e.,

$$[[\mathcal{C},\ \textbf{type}\ S\ \textbf{is}\ C\ [RT]\ \mid\ \ldots\ \textbf{endtype}]] \overset{\texttt{def}}{=} \ \textbf{type}\ S\ \textbf{is}\ C\_uid\ [RT]\ \mid\ \ldots\ \textbf{endtype}$$

where $uid = \mathcal{C}(C)((), rt, ())(S)$.

3.4 Channel declaration

Syntax

channel Γ **is** S **endch**

channel Γ **is** RT **endchan**

Static semantics

$$\frac{\mathcal{C} \vdash S \Rightarrow S'}{\mathcal{C} \vdash (\textbf{channel } \Gamma \textbf{ is } S\ ) \Rightarrow \Gamma \mapsto S'}$$

$$\frac{\mathcal{C} \vdash RT \Rightarrow rt}{\mathcal{C} \vdash (\textbf{channel } \Gamma \textbf{ is } RT\ ) \Rightarrow (\Gamma \mapsto rt)}$$

Translation to the core language

$$[[\mathcal{C},\ \textbf{channel } \Gamma \textbf{ is } S \textbf{ endchan}]] \overset{\texttt{def}}{=} \textbf{type } \Gamma \textbf{ is } S \textbf{ endtype}$$

$$[[\mathcal{C},\ \textbf{channel } \Gamma \textbf{ is } RT \textbf{ endchan}]] \overset{\texttt{def}}{=} \textbf{type } \Gamma \textbf{ is } RT \textbf{ endtype}$$

3.5 Exception declaration

Similar to channel declaration.

3.6 Process declaration

Syntax

process Π $[RG]$ $[$`(`$[\textbf{in}]\ V_1 : S_1, ..., [\textbf{in}]\ V_n : S_n$`)`$]$ **:noexit** $[\boxed{\textbf{raises } RX}]$ **is** B **endproc** $[\Pi]$

The default list of gates is `[]`, the default list of parameters is `()`, the default attribute for parameters is "**in**", and the default list of exceptions is *[]*.

Static semantics

$$\frac{\begin{array}{rcl} \mathcal{C} & \vdash & RG \Rightarrow rg \quad \mathcal{C} \vdash RT \Rightarrow rt \quad \mathcal{C} \vdash RX \Rightarrow rx \\ (\mathcal{C} \oplus rg \oplus rt \oplus rx)+ & & \\ (\Pi \mapsto rg, rt, rx \ \rightarrow \ \textbf{exit none} \ \rightarrow \ \mathit{newid}) & \vdash & B \Rightarrow \textbf{exit none} \end{array}}{\begin{array}{rcl} \mathcal{C} & \vdash & (\textbf{process } \Pi\ RG\ RT\ \textbf{:noexit} \boxed{\textbf{raises } RX} \textbf{ is } B\ ) \\ & & \Rightarrow (P \mapsto rg, rt, rx \ \rightarrow \ \textbf{exit none} \ \rightarrow \ \mathit{newid}) \end{array}}$$

Translation to the core language

The "**noexit**" keyword is translated into "**exit (none)**".

$$\left[\left[\mathcal{C}, \begin{array}{l} \textbf{process } \Pi\ RG\ RT\ \textbf{:noexit} \\ \qquad \boxed{\textbf{raises } RX} \textbf{ is} \\ \quad B \\ \textbf{endproc } [\Pi] \end{array} \right]\right] \overset{\texttt{def}}{=} \left(\begin{array}{l} \textbf{process } \Pi RG\ RT\ \textbf{:exit none} \\ \qquad \boxed{\textbf{raises } RX} \textbf{ is} \\ \quad [[\mathcal{C},\ B]] \\ \textbf{endproc} \end{array} \right)$$

3.7 Process declaration with in/out parameters

Syntax

$$\textbf{process}\ \Pi\ [RG]\ [(\textbf{in} \mid \textbf{out}\ V_1 : S_1, ..., \textbf{in} \mid \textbf{out}\ V_n : S_n)]\ [\boxed{\textbf{raises}\ RX}]\ \textbf{is}\ B\ \textbf{endproc}\ [\Pi]$$

The default list of gates is `[]`, the default list of parameters is `()`, and the default list of exceptions is `[]`.

Static semantics

$$\frac{\begin{array}{rcl} \mathcal{C} & \vdash & RG \Rightarrow rg \quad \mathcal{C} \vdash RT \Rightarrow rt \quad \mathcal{C} \vdash RX \Rightarrow rx \\ & & rt' = \mathit{def}(\textbf{out}(rt)) \\ (\mathcal{C} \oplus rg \oplus rt \oplus rx)+ & & \\ (\Pi \mapsto rg, rt, rx \ \rightarrow\ \textbf{exit}\ (rt')\ \rightarrow\ \mathit{newid}) & \vdash & B \Rightarrow \textbf{exit}\ (rt') \end{array}}{\begin{array}{rcl} \mathcal{C} & \vdash & (\textbf{process}\ \Pi\ RG\ RT\ \boxed{\textbf{raises}\ RX}\ \textbf{is}\ B\ ) \\ & & \Rightarrow (\Pi \mapsto rg, rt, rx\ \rightarrow\ \textbf{exit}\ (rt)'\ \rightarrow\ \mathit{newid}) \end{array}}$$

Translation to the core language

$$\left[\left[\ \mathcal{C},\ \begin{array}{l} \textbf{process}\ \Pi\ RG\ RT \\ \quad \textbf{raises}\ XL\ \textbf{is} \\ \ B \\ \textbf{endproc}\ [\Pi] \end{array}\ \right]\right] \overset{\mathtt{def}}{=} \left(\begin{array}{l} \textbf{process}\ \Pi\ RG\ \textbf{in}(RT){:}\textbf{exit}\ \textbf{out}(RT) \\ \quad \textbf{raises}\ XL\ \textbf{is} \\ \ \ \textbf{local var out}(RT) \\ \ \ \textbf{init}\ [[\ \mathcal{C},\ B]] \\ \ \ \textbf{in exit}\ \mathit{vars}(\textbf{out}(RT)) \\ \ \ \textbf{endloc} \\ \textbf{endproc} \end{array} \right)$$

where $\mathit{vars}(V : S : d\ \ldots) = (V\texttt{=>}V, \ldots)$.

3.8 Function declaration

Syntax

$$\textbf{function}\ F\ [([\textbf{in}]\ V_1 : S_1, ..., [\textbf{in}]\ V_n : S_n)]\ :\ T\ [\boxed{\textbf{raises}}\ RX]\ \textbf{is}\ E\ \textbf{endfunc}\ [F]$$

Static semantics

$$\frac{\begin{array}{rcl} \mathcal{C} & \vdash & RX \Rightarrow rx \quad \mathcal{C} \vdash RT \Rightarrow rt \quad \mathcal{C} \vdash S \Rightarrow S' \\ & & \mathcal{C}(F)((), rt, rx)(S') = \emptyset \\ & & \mathcal{C}_F = F \mapsto (), rt, rx\ \rightarrow\ \textbf{exit}\ S'\ \rightarrow\ \mathit{newuid} \\ (\mathcal{C} \oplus rt \oplus rx) + \mathcal{C}_F & \vdash & E \Rightarrow \textbf{exit}\ S' \end{array}}{\begin{array}{rcl} \mathcal{C} & \vdash & (\textbf{function}\ F\ RT\ {:}S\ \boxed{\textbf{raises}}\ RX\ \textbf{is}\ E\ ) \Rightarrow \mathcal{C}_F \end{array}}$$

Translation to the core language

The function declaration with "**in**" parameters is syntactic sugar of process declaration (below, RT has only "**in**" attributes):

$$\left[\!\!\left[\ ,\ \begin{array}{l} \mathit{function}\ F\ [RT]\ :\ S \\ \qquad [\boxed{\textbf{raises}}\ RX]\ \textbf{is} \\ \quad E \\ \textbf{endfunc}\ [F] \end{array}\right]\!\!\right] \stackrel{\mathrm{def}}{=} \left(\begin{array}{l} \textbf{process}\ F\_uid\ [RT]\ :\ \textbf{exit}(S) \\ \qquad [\textbf{raises}\ RX]\ \textbf{is} \\ \quad [[\mathcal{C},\ E]] \\ \textbf{endproc} \end{array}\right)$$

where *uid* is the unique identifier which allows overloading solving.

3.9 Procedure declaration

Syntax

$$\textbf{function}\ F\ (\textbf{in}\mid\textbf{out}\ V_1 : S_1, ..., \textbf{in}\mid\textbf{out}\ V_n : S_n)\ [\boxed{\textbf{raises}}\ RX]\ \textbf{is}\ E\ \textbf{endfunc}\ [F]$$

Static semantics

$$\begin{array}{rcl} \mathcal{C} & \vdash & RX \Rightarrow rx \quad \mathcal{C} \vdash RT \Rightarrow rt \\ & & \mathcal{C}_F = F \mapsto (), rt, rx\ \rightarrow\ \textbf{exit}\ rt'\ \rightarrow\ newuid \\ & & rt' = def(\textbf{out}(rt)) \\ & & \mathcal{C}(F)((), rt, rx)(rt') = \emptyset \\ (\mathcal{C} \oplus rx \oplus rt) + \mathcal{C}_F & \vdash & E \Rightarrow \textbf{exit}\ (rt') \\ \hline \mathcal{C} & \vdash & (\textbf{function}\ F\ RT\ \boxed{\textbf{raises}}\ RX\ \textbf{is}\ E\ ) \Rightarrow \mathcal{C}_F \end{array}$$

Translation to the core language

Procedure declaration is syntactic sugar for process declaration.

$$\left[\!\!\left[\ \mathcal{C},\ \begin{array}{l} \textbf{function}\ F\ RT \\ \qquad \boxed{\textbf{raises}}\ RX\ \textbf{is} \\ \quad E \\ \textbf{endfunc}\ [F] \end{array}\right]\!\!\right] \stackrel{\mathrm{def}}{=} \left(\begin{array}{l} \textbf{process}\ F\_uid\ \textbf{in}(RT) : \textbf{exit}\ \textbf{out}(RT) \\ \qquad \textbf{raises}\ RX\ \textbf{is} \\ \quad \textbf{local var out}(RT) \\ \quad \textbf{init}\ [[\mathcal{C},\ E]] \\ \quad \textbf{in exit}\ vars(\textbf{out}(RT)) \\ \quad \textbf{endloc} \\ \textbf{endproc} \end{array}\right)$$

where *uid* is the unique identifier which allows overloading solving, and $vars(V\ :\ S\ :\ d\ \ldots) = (V\text{=>}V, \ldots)$.

4 Record type

4.1 Overview

Syntax

$$\begin{array}{rcll} RT & ::= & () & \textit{empty record}\ (\mathrm{RT}_u 1) \\ & \mid & ([\textbf{in}]V_1 : S_1, ..., [\textbf{in}]V_n : S_n) & \textit{in record type}\ (\mathrm{RT}_u 2) \\ & \mid & (\textbf{in}\mid\textbf{out}\ V_1 : S_1, ..., \textbf{in}\mid\textbf{out}\ V_n : S_n) & \textit{in/out record type}\ (\mathrm{RT}_u 3) \end{array}$$

These lists appear in the declaration of types, constructors, channels, gates, functions and processes, and in local variable declarations.

Static semantics

The static semantics is given by assertions of form:

$$\mathcal{C} \vdash RT \Rightarrow rt$$

meaning "In the context $\mathcal{C}$, the variable list RT is well formed and has the type rt".

Translation to the core language

This syntactic domain corresponds to the product of two syntactic domain of the core language, i.e.,

$$[[.,.]] : \text{Context} \times \text{RTyExp}_u \;\rightarrow\; \text{RTyExp}_c \times \text{RTyExp}_c$$

where the first result domain correspond to the "**in**" parameters, and the second to the "**out**" parameters.

4.2 Empty record

Syntax

()

Static semantics

$$\frac{}{\mathcal{C} \;\vdash\; () \Rightarrow ()}$$

Translation to the core language

$$[[\mathcal{C},\ ()]] \overset{\texttt{def}}{=} (),\ ()$$

4.3 In record type

Syntax

$([\mathbf{in}]V_1 : S_1, ..., [\mathbf{in}]V_n : S_n)$

The default attribute is "**in**".

Static semantics

$$\frac{\begin{array}{ll}\mathcal{C} \;\vdash\; S_1 \Rightarrow S_1' \quad \cdots \quad \mathcal{C} \vdash S_n \Rightarrow S_n' \\ \qquad (\mathcal{C}' = (+_{i=1}^{i=n}\ V_i \mapsto (S_i',\ def)) \;\neq\; error)\end{array}}{\mathcal{C} \;\vdash\; (\mathbf{in}\ V_1 : S_1, ..., \mathbf{in}\ V_n : S_n) \Rightarrow (V_1 : T_1' : def, \ldots, V_n : T_n' : def)}$$

Translation to the core language

$$[[\mathcal{C},\ ([\mathbf{in}]V_1 : S_1, ..., [\mathbf{in}]V_n : S_n)]] \overset{\texttt{def}}{=} (V_1 : S_1, ..., V_n : S_n),\ ()$$

4.4 In/out record type

Syntax

$(\mathbf{in} \mid \mathbf{out}\ V_1 : S_1, \ldots, \mathbf{in} \mid \mathbf{out}\ V_n : S_n)$

Static semantics

$$\frac{\begin{array}{ll} \mathcal{C} \vdash & S_1 \Rightarrow S_1' \quad \cdots \quad \mathcal{C} \vdash S_n \Rightarrow S_n' \\ & \mathcal{C}' = (+_{i=1}^{i=n}\ V_i \mapsto (S_i',\ d_i)) \neq \ error \end{array}}{\mathcal{C} \vdash (\mathbf{in} \mid \mathbf{out}\ V_1 : S_1, \ldots, \mathbf{in} \mid \mathbf{out}\ V_n : S_n) \Rightarrow (V_1 : S_1' : d_1, \ldots, V_n : S_n' : d_n)}$$

where

$$d_i = \begin{cases} undef & \textbf{if} \quad \textbf{out}\ V_i : S_i \\ def & \textbf{otherwise} \end{cases}$$

Translation to the core language

$$[[\mathcal{C},\ (\mathbf{in} \mid \mathbf{out}\ V_1 : S_1, \ldots, \mathbf{in} \mid \mathbf{out}\ V_n : S_n)]] \overset{\mathtt{def}}{=} (V_i \Rightarrow S_i \mid \mathbf{in}\ V_i : S_i),\ (V_i \Rightarrow S_i \mid \mathbf{out}\ V_i : S_i)$$

5 Record of typed gates/exceptions

Syntax

RG	$::=$	`[]`	*empty record*
	$\mid$	$[G_1 : \Gamma_1, \ldots, G_n : \Gamma_n]$	*union*

These lists appear in the declaration of (functions and) processes, and in '**hide**" operator.

Similar for the record of typed exception, RX.

RX	$::=$	`[]`	*empty record*
	$\mid$	$[X_1 : \Xi_1, \ldots, X_n : \Xi_n]$	*union*

Static semantics

The static semantics is given by assertions of form:

$\mathcal{C} \vdash RG \Rightarrow rg$
$\mathcal{C} \vdash RX \Rightarrow rx$

Translation to the core language

There is no direct correspondent in the core language.

Each record of typed gates is translated into a list of type Gat $\times$ RTyExp.

Each record of typed exceptions is translated into a list of type Exc $\times$ RTyExp.

5.1 Empty record

Syntax

`[]`

Static semantics

$$\frac{}{\mathcal{C} \ \vdash \ \texttt{[]} \Rightarrow ()}$$

Translation to the core language

$$[[\mathcal{C},\ \texttt{[]}]] \overset{\texttt{def}}{=} \texttt{[]}$$

5.2 Union

Syntax

$\texttt{[}G_1\!:\!\Gamma_1, ..., G_n\!:\!\Gamma_n\texttt{]}$

$\texttt{[}X_1\!:\!\Xi_1, ..., X_n\!:\!\Xi_n\texttt{]}$

Static semantics

$$\frac{\mathcal{C} \ \vdash \ \Gamma_1 \Rightarrow t_1 \quad \cdots \quad \mathcal{C} \vdash \Gamma_n \Rightarrow t_n \qquad \mathcal{C}' = (+_{i=1}^{i=n}\ G_i \mapsto \mathbf{gate}\ t_i) \ \neq \ \mathit{error}}{\mathcal{C} \ \vdash \ \texttt{[}G_1\!:\!\Gamma_1, ..., G_n\!:\!\Gamma_n\texttt{]} \Rightarrow (G_1 : \mathbf{gate}\ t_1, \ldots,\ G_n : \mathbf{gate}\ t_n)}$$

$$\frac{\mathcal{C} \ \vdash \ \Xi_1 \Rightarrow t_1 \quad \cdots \quad \mathcal{C} \vdash \Xi_n \Rightarrow t_n \qquad \mathcal{C}' = (+_{i=1}^{i=n}\ X_i \mapsto \mathbf{exn}\ t_i) \ \neq \ \mathit{error}}{\mathcal{C} \ \vdash \ \texttt{[}X_1\!:\!\Xi_1, ..., X_n\!:\!\Xi_n\texttt{]} \Rightarrow (X_1 : \mathbf{exn}\ t_1, \ldots,\ X_n : \mathbf{exn}\ t_n)}$$

Translation to the core language

$$[[\mathcal{C}, G\!:\!\Gamma]] \overset{\texttt{def}}{=} \begin{cases} G\!:(\$1 \Rightarrow S) & \textbf{iff}\ \mathcal{C}(\Gamma) = S \\ G\!:(rt) & \textbf{iff}\ \mathcal{C}(\Gamma) = (rt) \end{cases}$$

6 Patterns

Syntax

P	$::=$	**any** S	*wildcard*	$(\mathrm{P}_u 1)$
	$\mid$	? V [**as** P]	*variable*	$(\mathrm{P}_u 2)$
	$\mid$	! E	*expression*	$(\mathrm{P}_u 3)$

| $C\ [RP]$ *constructor application* (P_u4)

| P **of** T *typed pattern* (P_u5)

Static semantics

The static semantics is given by assertions of form:

$\mathcal{C} \vdash (P \Rightarrow t) \Rightarrow rt$

meaning "In the context $\mathcal{C}$, matching pattern P to type t gives the context rt."

Translation to the core language

The translation function has the following profile $[[.,\ .]]\ :\ \mathrm{Context} \times \mathrm{Pat}_u \rightarrow \mathrm{Pat}_c$

6.1 Wildcard

Syntax

any S

Static semantics

$$\frac{}{\mathcal{C} \quad \vdash \quad (\textbf{any}\ S \Rightarrow t) \Rightarrow \{\}}\ [t \equiv S]$$

Translation to the core language

$[[\mathcal{C},\ \textbf{any}\ S]] \stackrel{\texttt{def}}{=} \textbf{any} : S$

6.2 Variable pattern

Syntax

$?\ V\ [\textbf{as}\ P]$

Static semantics

$$\frac{\begin{array}{lll} \mathcal{C} & \vdash & V \Rightarrow (S, d) \\ \langle\mathcal{C} & \vdash & (P \Rightarrow S) \Rightarrow rt'\rangle \end{array}}{\mathcal{C} \quad \vdash \quad (V\ \ \langle\textbf{as}\ P\rangle\ \Rightarrow S) \Rightarrow (V \mapsto (S,\ \mathit{def}))\ \langle + rt'\rangle}$$

Translation to the core language

$[[\mathcal{C}, ?\ V]] \stackrel{\texttt{def}}{=}\ ?\ V$

6.3 Expression pattern

Syntax

$! \; E$

Static semantics

$$\frac{\mathcal{C} \;\vdash\; E \Rightarrow \textbf{exit } t}{\mathcal{C} \;\vdash\; (! \; E \Rightarrow t) \Rightarrow \{\}}$$

Translation to the core language

$$[[\mathcal{C}, \; ! \; E]] \overset{\texttt{def}}{=} \; ! \; [[\mathcal{C}, \; E]]$$

6.4 Constructor application

Syntax

$C \; [RP]$

The default record pattern is T().

Static semantics

$$\frac{\begin{array}{lll} & & \mathcal{C}(C)((), rt, ())(\textbf{exit } t) = \{uid\} \\ \mathcal{C} & \vdash & (RP \Rightarrow rt) \Rightarrow rt' \end{array}}{\mathcal{C} \;\vdash\; (C \; RP \Rightarrow t) \Rightarrow rt'}$$

Note: The first clause imposes that only one constructor profile match.

Translation to the core language

$$[[\mathcal{C}, \; C \; RP]] \overset{\texttt{def}}{=} C\_uid \; [[rt, \; RP]]$$

where $\mathcal{C}(C)((), rt, ())(\textbf{exit } t) = \{uid\}$.

6.5 Explicit typing

Syntax

$P \; \textbf{of} \; S$

Static semantics

$$\frac{\begin{array}{lll} \mathcal{C} & \vdash & S \Rightarrow S' \\ \mathcal{C} & \vdash & (P \Rightarrow S) \Rightarrow rt' \end{array}}{\mathcal{C} \;\vdash\; (P \; \textbf{of} \; T \Rightarrow S') \Rightarrow rt'}$$

Translation to the core language

$[[\mathcal{C},\ P\ \mathbf{of}\ S]] \overset{\texttt{def}}{=} [[\mathcal{C},\ P]]\ :\ S$

7 Record pattern

Syntax

RP	$::=$	$()$	*empty record*(RP$_u$1)
	$\mid$	$(V_1 \texttt{=>} P_1, ..., V_n \texttt{=>} P_n[, \texttt{...}])$	*union*(RP$_u$2)
	$\mid$	$(P_1, ..., P_n)$	*tuple*(RP$_u$3)

Static semantics

$\mathcal{C} \vdash (RP \Rightarrow rt) \Rightarrow rt'$

Translation to the core language

The profile of the function is

$[[.,\ .,\ ]]\ :\ \text{Context} \times \text{RPat}_u \times \text{RTyExp} \rightarrow \text{Pat}_c$

7.1 Empty record

Syntax

$()$

Static semantics

$$\frac{}{\mathcal{C}\ \vdash\ (\ () \Rightarrow ()\ ) \Rightarrow \{\}}$$

Translation to the core language

Identity.

7.2 Attributed pattern list

Syntax

$(V_1 \texttt{=>} P_1, ..., V_n \texttt{=>} P_n[, \texttt{...}])$

Static semantics

$$\begin{array}{lll} & & (\exists \rho \in P(n))\ (\forall i \in 1..n) \quad V_i = V'_{\rho(i)} \\ \mathcal{C} & \vdash & (P_1 \Rightarrow S_{\rho(1)}) \Rightarrow \mathcal{C}_1 \quad \cdots \quad \mathcal{C} \vdash (P_n \Rightarrow S_{\rho(n)}) \Rightarrow \mathcal{C}_n \\ & & \mathcal{C}' = (+_{i=1}^{i=n}\ \mathcal{C}_i)\ \neq\ \mathit{error} \\ \hline \mathcal{C} & \vdash & (\ (V_1\texttt{=>}P_1, ..., V_n\texttt{=>}P_n) \Rightarrow (V'_1 : S_1 : d_1, \ldots, V'_n : S_n : d_n)\ ) \Rightarrow \mathcal{C}' \end{array}$$

where $P(n)$ is the set of permutations of $1..n$.

$$\begin{array}{lll} & & (\exists \rho \in P(m))\ (\forall i \in 1..n) \quad V_i = V'_{\rho(i)}) \\ \mathcal{C} & \vdash & (P_1 \Rightarrow S_{\rho(1)}) \Rightarrow \mathcal{C}_1 \quad \cdots \quad \mathcal{C} \vdash (P_n \Rightarrow S_{\rho(n)}) \Rightarrow \mathcal{C}_n \\ \mathcal{C} & \vdash & V'_{\rho(n+1)} \Rightarrow (S_{\rho(n+1)},\ d_i) \quad \cdots \quad \mathcal{C} \vdash V'_{\rho(m)} \Rightarrow (S_{\rho(m)},\ d_m)) \\ & & \mathcal{C}' = (+_{i=1}^{i=n}\ \mathcal{C}_i) + (+_{i=n+1}^{i=m}\ V'_{\rho(i)} \mapsto (S_{\rho(i)}, d_{\rho(i)})))\ \neq\ \mathit{error} \\ \hline \mathcal{C} & \vdash & (\ (V_1\texttt{=>}P_1, ..., V_n\texttt{=>}P_n, \boldsymbol{\ldots}) \Rightarrow (V'_1 : S_1 : d_1, \ldots, V'_m : S_m : d_m)\ ) \Rightarrow \mathcal{C}' \end{array}$$

Translation to the core language

$$\begin{array}{l} [[\mathcal{C},\ (V_1\texttt{=>}P_1, ..., V_n\texttt{=>}P_n, \boldsymbol{\ldots}) \Rightarrow (V'_1 : S_1 : d_1, \ldots, V'_m : S_m : d_m)]] \overset{\texttt{def}}{=} \\ \quad (V'_1\texttt{=>}[[\mathcal{C},\ P_{\rho(1)}]], ..., V'_n\texttt{=>}[[\mathcal{C},\ P_{\rho(n)}]], V'_{n+1}\texttt{=>?}V'_{n+1}, \ldots, V'_m\texttt{=>?}V'_m) \end{array}$$

7.3 Tuple pattern

Syntax

$(P_1, ..., P_n)$

Static semantics

$$\begin{array}{lll} \mathcal{C} & \vdash & (P_1 \Rightarrow S_1) \Rightarrow \mathcal{C}_1 \quad \cdots \quad \mathcal{C} \vdash (P_n \Rightarrow S_n) \Rightarrow \mathcal{C}_n \\ & & \mathcal{C}' = (+_{i=1}^{i=n}\ \mathcal{C}_i)\ \neq\ \mathit{error} \\ \hline \mathcal{C} & \vdash & (\ (P_1, ..., P_n) \Rightarrow (V_1 : S_1 : d_1, \ldots, V_n : S_n : d_n)\ ) \Rightarrow \mathcal{C}' \end{array}$$

Translation to the core language

$$[[\mathcal{C},\ (P_1, ..., P_n) \Rightarrow (V_1 : S_1 : d_1, \ldots, V_n : S_n : d_n)]] \overset{\texttt{def}}{=} (V_1\texttt{=>}[[\mathcal{C},\ P_1]], ..., V_n\texttt{=>}[[\mathcal{C},\ P_n]])$$

8 Match expressions

Syntax

$$\begin{array}{lll} EM & ::= & E\ \texttt{::}\ P \qquad \textit{simple match} \ (\mathrm{EM}_u1) \\ & \mid & EM_1\ \textbf{when}\ E \qquad \textit{guarded match} \ (\mathrm{EM}_u2) \\ & \mid & EM_1\ \textbf{and}\ EM_2 \qquad \textit{conjunction} \ (\mathrm{EM}_u3) \\ & \mid & EM_1\ \textbf{or}\ EM_2 \qquad \textit{disjunction} \ (\mathrm{EM}_u4) \end{array}$$

Static semantics

The static semantics is given by assertions of form:

$\mathcal{C} \vdash EM \Rightarrow rt'$

meaning "In the context $\mathcal{C}$, the match expression EM gives the context rt'."

Translation to the core language

There is no direct translation to the core language. The translation will be given at the "**case**" behaviour/value expressions.

8.1 Simple match

Syntax

$E :: P$

Static semantics

$$\frac{\begin{array}{lll}\mathcal{C} & \vdash & E \Rightarrow S \\ \mathcal{C} & \vdash & (P \Rightarrow S) \Rightarrow rt'\end{array}}{\begin{array}{lll}\mathcal{C} & \vdash & (E::P) \Rightarrow rt'\end{array}}$$

8.2 Guarded match

Syntax

EM_1 **when** E

Static semantics

$$\frac{\begin{array}{rll}\mathcal{C} & \vdash & EM_1 \Rightarrow rt' \\ \mathcal{C}+\mathcal{C}' & \vdash & E \Rightarrow \textbf{exit bool}\end{array}}{\begin{array}{rll}\mathcal{C} & \vdash & (EM_1 \textbf{ when } E) \Rightarrow rt'\end{array}}$$

8.3 And match

Syntax

EM_1 **and** EM_2

Static semantics

$$\frac{\begin{array}{lll}\mathcal{C} & \vdash & EM_1 \Rightarrow rt_1 \\ \mathcal{C} & \vdash & EM_2 \Rightarrow rt_2\end{array}}{\begin{array}{lll}\mathcal{C} & \vdash & (EM_1 \textbf{ and } EM_2) \Rightarrow rt_1 + rt_2\end{array}}$$

8.4 Or match

Syntax

EM_1 **or** EM_2

Static semantics

$$\frac{\begin{array}{rcl}\mathcal{C} & \vdash & EM_1 \Rightarrow rt' \\ \mathcal{C} & \vdash & EM_2 \Rightarrow rt'\end{array}}{\mathcal{C} \vdash (EM_1 \ \mathbf{or}\ EM_2) \Rightarrow rt'}$$

9 Value expressions

9.1 Overview

Syntax

$E \ ::= \ K$ — *constant denotation* ($\mathrm{E}_u 1$)

$\mid \ V$ — *value variable* ($\mathrm{E}_u 2$)

$\mid$ **any** S — *nondeterministic termination* ($\mathrm{E}_u 3$)

$\mid \ C\ [([V_1 \texttt{=>}]E_1, ..., [V_n \texttt{=>}]E_n\ [, \bullet\bullet\bullet])]$ — *constructor application* ($\mathrm{E}_u 4$)

$\mid \ \boxed{\textbf{raise}}\ X\ E$ — *raising exception* ($\mathrm{E}_u 5$)

$\mid \ \boxed{\textbf{raise}}\ X\ ([V_1 \texttt{=>}]E_1, ..., [V_n \texttt{=>}]E_n\ [, \bullet\bullet\bullet])$ — *raising exception* ($\mathrm{E}_u 6$)

$\mid \ P\ \texttt{:=}\ E$ — *variable assignment* ($\mathrm{E}_u 7$)

$\mid \ E_1\ ;\ E_2$ — *expression continuation* ($\mathrm{E}_u 8$)

$\mid$ **trap** — *trap exceptions* ($\mathrm{E}_u 9$)
  $X_1\ [(V_1^1 : S_1^1, ..., V_{p_1}^1 : S_{p_1}^1)] \ \rightarrow \ E_1$
  ...
  $X_p\ [(V_1^p : S_1^p, ..., V_{n_p}^p : S_{n_p}^p)] \ \rightarrow \ E_p$
  [**exit** $[(V_1^p : S_1^p, ..., V_{n_p}^n : S_{n_p}^p)] \ \rightarrow \ E_{p+1}$]
in E_0
endtrap

$\mid$ **case** — *general case expression* ($\mathrm{E}_u 10$)
  $EM_0 \ \rightarrow \ E_0$
  ...
  $EM_p \ \rightarrow \ E_p$

$\star$ [**otherwise** E_{p+1}]
endcase

$\star$ | **case** E' **is** *usual case expression* (E_u11)
$P_1^0, ..., P_{n_0}^0$ [**when** E_0] $\rightarrow$ E'_0
...
$P_1^p, ..., P_{n_p}^p$ [**when** E_p] $\rightarrow$ E'_p
[**otherwise** E'_{p+1}]
endcase

$\star$ | E **match** P *match expression* (E_u12)

$\star$ | **if** E_0 **then** E'_0 *conditional expression* (E_u13)
elsif E_1 **then** E'_1
...
elsif E_n **then** E'_n
[**else** E'_{n+1}]
endif

$\star$ | E_0 **andthen** E_1 *logical expression* (E_u14)

$\star$ | E_0 **orelse** E_1 *logical expression* (E_u15)

| E_0 = E_1 *equality expression* (E_u16)

$\star$ | E_0 <> E_1 *non-equality expression* (E_u17)

$\star$ | E.V *select expression* (E_u18)

$\star$ | E.{V_1=>E_1, ..., V_n=>E_n} *update expression* (E_u19)

| **local var** $V_1 : S_1, \ldots, V_n : S_n$ *variable declaration* (E_u20)
[**init** E_0]
in E
endloc

| **rename** *renaming* (E_u21)
X_0 [($V_1^0 : S_1^0, \ldots, V_{n_0}^0 : S_{n_0}^0$)] **is** X'_0[([$V_1'^0$=>]$E_1'^0, \ldots,$ [$V_{m_0}'^0$=>]$E_{m_0}'^0$)]
...
X_p [($V_1^p : S_1^p, \ldots, V_{n_p}^p : S_{n_p}^p$)] **is** X'_p[([$V_1'^p$=>]$E_1'^p, \ldots,$ [$V_{m_p}'^p$=>]$E_{m_p}'^p$)]
in E
endren

$\star$ | F [([V_1=>]E_1, ..., [V_n=>]E_n [, **...**])] *function call* (E_u22)

$[[[X_1\texttt{=>}]X'_1, \ldots, [X_p\texttt{=>}]X'_p[, \textbf{...}]]]$

$\star \mid$ $F\ [([V_1\texttt{=>}]E_1 \mid P_1, ..., [V_n\texttt{=>}]E_n \mid P_n\ [, \textbf{...}])]$ *procedure call* (E$_u$23)
$[[[X_1\texttt{=>}]X'_1, \ldots, [X_p\texttt{=>}]X'_p[, \textbf{...}]]]$

$\star \mid$ **loop forever** *iteration* (E$_u$24)
var $V_1 : S_1, \ldots, V_n : S_n$ **in**
init E_0
in E
endloop

$\star \mid$ **loop** $X[(V_1 : S_1, ..., V_n : S_n)]$ *breakable iteration* (E$_u$25)
init E_0
in E
endloop

$\star \mid$ **break** $[X]\ [(V_1\texttt{=>}E_1, ..., V_n\texttt{=>}E_n)]$ *break loop* (E$_u$26)

$\mid$ E **of** S *explicit typing* (E$_u$27)

Static semantics

The static semantics is given by assertions of form:

$\mathcal{C} \vdash E \Rightarrow \textbf{exit}\ t$

meaning "In the context $\mathcal{C}$, the expression E has the type t."

Translation to the core language

$[[.,\ .]]\ :\ \text{Context},\ \text{Exp}_u \rightarrow \text{Behav}_c$

9.2 Primitive constants

Syntax

K

Static semantics

$$\frac{}{\mathcal{C} \ \vdash \ K \Rightarrow \textbf{exit}\ S}[K : S]$$

Translation to the core language

$[[\mathcal{C},\ K]] \overset{\texttt{def}}{=} \textbf{exit}\ (K)$

9.3 Variable

Syntax

V

Static semantics

$$\frac{\mathcal{C} \;\vdash\; V \Rightarrow (S,\ def)}{\mathcal{C} \;\vdash\; V \Rightarrow \textbf{exit } S}$$

Translation to the core language

$$[[\mathcal{C},\ V]] \overset{\text{def}}{=} \textbf{exit } (V)$$

9.4 Nondeterministic termination

Syntax

any S

Static semantics

$$\frac{\mathcal{C} \;\vdash\; S \Rightarrow S'}{\mathcal{C} \;\vdash\; \textbf{any } S \Rightarrow \textbf{exit } S'}$$

Translation to the core language

$$[[\mathcal{C},\ \textbf{any } S]] \overset{\text{def}}{=} \textbf{exit } (\textbf{any } S)$$

9.5 Constructor application

Syntax

$C\ [RE]$

Static semantics

$$\frac{\begin{array}{lll} & & \mathcal{C}(C)((), rt, ())(\textbf{exit } S) = \{uid\} \\ \mathcal{C} & \vdash & RE \Rightarrow \textbf{exit } (rt) \end{array}}{\mathcal{C} \;\vdash\; C\ RE \Rightarrow S}$$

Translation to the core language

$$[[\mathcal{C},\ C\ RE]] \overset{\text{def}}{=} \left(\begin{array}{l} \textbf{case } [[\mathcal{C},\ RE \Rightarrow rt]] \textbf{ is} \\ \quad ?x \;\rightarrow\; \textbf{exit } C\_uid\ x \\ \textbf{endcase} \end{array} \right)$$

where $(), rt, () \;\rightarrow\; \textbf{exit } S \;\rightarrow\; uid \in \mathcal{C}(C)$.

9.6 Raising exception

Syntax

raise X E

raise X RE

Static semantics

$$\frac{\begin{array}{lll}\mathcal{C} & \vdash & X \Rightarrow \textbf{exn}\ S \\ \mathcal{C} & \vdash & E \Rightarrow \textbf{exit}\ S\end{array}}{\begin{array}{lll}\mathcal{C} & \vdash & \textbf{raise}\ X\ E \Rightarrow \textbf{exit}\ ()\end{array}}$$

$$\frac{\begin{array}{lll}\mathcal{C} & \vdash & X \Rightarrow \textbf{exn}\ (rt) \\ \mathcal{C} & \vdash & RE \Rightarrow \textbf{exit}\ (rt)\end{array}}{\begin{array}{lll}\mathcal{C} & \vdash & \textbf{raise}\ X\ RE \Rightarrow \textbf{exit}\ ()\end{array}}$$

Translation to the core language

$$[[\mathcal{C},\ \textbf{raise}\ X\ E]] \overset{\texttt{def}}{=} \textbf{signal}\ X\ [[\mathcal{C},\ E]]\ \ ;\ \ \textbf{block}$$

$$[[\mathcal{C},\ \textbf{raise}\ X\ RE]] \overset{\texttt{def}}{=} \textbf{signal}\ X\ [[\mathcal{C},\ RE \Rightarrow rt]]\ \ ;\ \ \textbf{block}$$

In the last case, $\mathcal{C} \vdash X \Rightarrow \textbf{exn}\ (rt)$.

9.7 Assignment

Syntax

P `:=` E

Static semantics

$$\frac{\begin{array}{lll}\mathcal{C} & \vdash & E \Rightarrow \textbf{exit}\ S \\ \mathcal{C} & \vdash & (P \Rightarrow S) \Rightarrow rt\end{array}}{\begin{array}{lll}\mathcal{C} & \vdash & (P\ \texttt{:=}\ E) \Rightarrow \textbf{exit}\ (rt)\end{array}}$$

Translation to the core language

$$[[\mathcal{C},\ P\ \texttt{:=}\ E]] \overset{\texttt{def}}{=} [[\mathcal{C},\ P]]\texttt{:=}[[\mathcal{C},\ E]]$$

9.8 Sequential composition

Syntax

E_1 ; E_2

Static semantics

$$\frac{\begin{array}{rcl} \mathcal{C} & \vdash & E_1 \Rightarrow \mathbf{exit}\ (rt_1) \\ \mathcal{C} \oplus rt_1 & \vdash & E_2 \Rightarrow \mathbf{exit}\ (rt_2) \end{array}}{\begin{array}{rcl} \mathcal{C} & \vdash & (E_1\ ;\ E_2) \Rightarrow \mathbf{exit}\ (rt_1 \oplus rt_2) \end{array}}$$

Translation to the core language

$$[[\mathcal{C},\ E_1 \quad ; \quad E_2]] \overset{\mathtt{def}}{=} [[\mathcal{C},\ E_1]]\,;[[\mathcal{C},\ E_2]]$$

9.9 Trap

Syntax

$$\begin{array}{l} \mathbf{trap} \\ \quad X_1\ RT_1\ \rightarrow\ E_1 \\ \quad \ldots \\ \quad X_p\ RT_p\ \rightarrow\ E_p \\ \quad [\mathbf{exit}\ RT_{p+1}\ \rightarrow\ E_{p+1}] \\ \ \mathbf{in}\ E_0\ \mathbf{endtrap} \end{array}$$

Static semantics

$$\frac{\begin{array}{rcl} \mathcal{C} & \vdash & RT_1 \Rightarrow rt_1 \quad \cdots \quad \mathcal{C} \vdash RT_p \Rightarrow rt_p \\ \mathcal{C} \oplus rt_1 & \vdash & E_1 \Rightarrow \mathbf{exit}\ t \quad \cdots \quad \mathcal{C} \oplus rt_p \vdash E_p \Rightarrow \mathbf{exit}\ t \\ \mathcal{C} \oplus (+_{i=1}^{i=p}\ X_i \mapsto \mathbf{exn}\ (rt_i)) & \vdash & E_0 \Rightarrow \mathbf{exit}\ t \end{array}}{\mathcal{C} \ \vdash \ \left(\begin{array}{l} \mathbf{trap} \\ \quad X_1\ RT_1\ \rightarrow\ E_1 \\ \quad \ldots \\ \quad X_p\ RT_p\ \rightarrow\ E_p \\ \ \mathbf{in}\ E_0 \end{array} \right) \Rightarrow \mathbf{exit}\ t}$$

$$\frac{\begin{array}{rcl} \mathcal{C} & \vdash & RT_1 \Rightarrow rt_1 \quad \cdots \quad \mathcal{C} \vdash RT_p \Rightarrow rt_p \\ \mathcal{C} \oplus rt_1 & \vdash & E_1 \Rightarrow \mathbf{exit}\ t \quad \cdots \quad \mathcal{C} \oplus rt_p \vdash E_p \Rightarrow \mathbf{exit}\ t \\ \mathcal{C} & \vdash & RT_{p+1} \Rightarrow rt_{p+1} \\ \mathcal{C} \oplus rt_{p+1} & \vdash & E_{p+1} \Rightarrow \mathbf{exit}\ t \\ \mathcal{C} \oplus (+_{i=1}^{i=p}\ X_i \mapsto \mathbf{exn}\ (rt_i)) & \vdash & E_0 \Rightarrow \mathbf{exit}\ (rt_{p+1}) \end{array}}{\mathcal{C} \ \vdash \ \left(\begin{array}{l} \mathbf{trap} \\ \quad X_1\ RT_1\ \rightarrow\ E_1 \\ \quad \ldots \\ \quad X_n\ RT_p\ \rightarrow\ E_p \\ \quad \mathbf{exit}\ RT_{p+1}\ \rightarrow\ E_{p+1} \\ \mathbf{in}\ E_0 \end{array} \right) \Rightarrow \mathbf{exit}\ t}$$

9.10 General case

Syntax

$$\mathbf{case}\ EM_0\ \rightarrow\ E_0\ \ldots\ EM_p\ \rightarrow\ E_p\ \mathbf{endcase}$$

Static semantics

$$\frac{\begin{array}{rcl} \mathcal{C} & \vdash & EM_0 \Rightarrow \mathcal{C}_0 \quad \cdots \quad \mathcal{C} \vdash EM_p \Rightarrow rt_p \\ \mathcal{C} \oplus \mathcal{C}_0 & \vdash & E_0 \Rightarrow \textbf{exit}\ t \quad \cdots \quad \mathcal{C} \oplus \mathcal{C}_p \vdash E_p \Rightarrow \textbf{exit}\ t \end{array}}{\begin{array}{rcl} \mathcal{C} & \vdash & (\textbf{case}\ EM_0 \ \rightarrow\ E_0 \ \ldots\ EM_p \ \rightarrow\ E_p \ ) \Rightarrow \textbf{exit}\ t \end{array}}$$

Translation to the core language

The general case is a syntactic sugar for the simple case of the core language. Below we should show that the translation process terminates. The complexity of the translation process shows that the general case is more convenient for specification purposes.

$$\left[\!\!\left[\mathcal{C}, \begin{array}{l} \textbf{case} \\ \quad EM_0 \;\rightarrow\; E_0 \\ \quad \ldots \\ \quad EM_p \;\rightarrow\; E_p \\ \quad [\textbf{otherwise}\ E_{p+1}] \\ \textbf{endcase} \end{array} \right]\!\!\right] \stackrel{\text{def}}{=} \left[\!\!\left[\mathcal{C}, \begin{array}{l} \textbf{case} \\ \quad EM_0 \;\rightarrow\; E_0 \\ \quad \textbf{otherwise}\ (\textbf{case} \\ \qquad\qquad\quad EM_1 \;\rightarrow\; E_1 \\ \qquad\qquad\quad \textbf{otherwise}\ \ldots \\ \qquad\qquad \textbf{endcase}) \\ \textbf{endcase} \end{array} \right]\!\!\right]$$

$$\left[\!\!\left[\mathcal{C}, \begin{array}{l} \textbf{case} \\ \quad E::P \;\rightarrow\; E_0 \\ \quad [\textbf{otherwise}\ E_1] \\ \textbf{endcase} \end{array} \right]\!\!\right] \stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{case}\ [[\mathcal{C},\ E]]\ \textbf{in} \\ \quad [[\mathcal{C},\ P]] \;\rightarrow\; [[\mathcal{C},\ E_0]] \\ \quad [\ \textbf{otherwise} \;\rightarrow\; [[\mathcal{C},\ E_1]]\ ] \\ \textbf{endcase} \end{array} \right)$$

$$\left[\!\!\left[\mathcal{C}, \begin{array}{l} \textbf{case} \\ \quad EM_0\ \textbf{when}\ E \;\rightarrow\; E_0 \\ \quad [\textbf{otherwise}\ E_1] \\ \textbf{endcase} \end{array} \right]\!\!\right] \stackrel{\text{def}}{=} \left[\!\!\left[\mathcal{C}, \begin{array}{l} \textbf{case} \\ \quad EM \;\rightarrow\; \textbf{case} \\ \qquad\qquad\quad E::\textbf{true} \;\rightarrow\; E_0 \\ \qquad\qquad\quad [\textbf{otherwise}\ E_1] \\ \qquad\qquad \textbf{endcase} \\ \quad [\textbf{otherwise}\ E_1] \\ \textbf{endcase} \end{array} \right]\!\!\right]$$

$$\left[\!\!\left[\mathcal{C}, \begin{array}{l} \textbf{case} \\ \quad EM_1\ \textbf{or}\ EM_2 \;\rightarrow\; E_0 \\ \quad [\textbf{otherwise}\ E_1] \\ \textbf{endcase} \end{array} \right]\!\!\right] \stackrel{\text{def}}{=} \left[\!\!\left[\mathcal{C}, \begin{array}{l} \textbf{case} \\ \quad EM_1 \;\rightarrow\; E_0 \\ \quad EM_2 \;\rightarrow\; E_0 \\ \quad [\textbf{otherwise}\ E_1] \\ \textbf{endcase} \end{array} \right]\!\!\right]$$

$$\left[\!\!\left[\mathcal{C}, \begin{array}{l} \textbf{case} \\ \quad EM_1\ \textbf{and}\ EM_2 \;\rightarrow\; E_0 \\ \quad [\textbf{otherwise}\ E_1] \\ \textbf{endcase} \end{array} \right]\!\!\right] \stackrel{\text{def}}{=} \left[\!\!\left[\mathcal{C}, \begin{array}{l} \textbf{case} \\ \quad EM_1 \;\rightarrow\; (\textbf{case} \\ \qquad\qquad\quad EM_2 \;\rightarrow\; E_0 \\ \qquad\qquad\quad \textbf{otherwise}\ E_1 \\ \qquad\qquad \textbf{endcase}) \\ \quad [\textbf{otherwise}\ E_1] \\ \textbf{endcase} \end{array} \right]\!\!\right]$$

9.11 Usual case

Syntax

$$\begin{array}{l}
\textbf{case } E \textbf{ in} \\
\quad P_1^0, ..., P_{n_0}^0[\textbf{when } E_0] \quad \rightarrow \quad E'_0 \\
\quad \ldots \\
\quad P_1^p, ..., P_{n_p}^p[\textbf{when } E_p] \quad \rightarrow \quad E'_p \\
\quad [\textbf{otherwise } E'_{p+1}] \\
\textbf{endcase}
\end{array}$$

Translation to the core language

The usual case is syntactic sugar of the general case.

$$\left[\left[\mathcal{C}, \begin{array}{l}
\textbf{case } E \textbf{ in} \\
\quad P_1^0, ..., P_{n_0}^0[\textbf{when } E_0] \quad \rightarrow \quad E'_0 \\
\quad \ldots \\
\quad P_1^p, ..., P_{n_p}^p[\textbf{when } E_p] \quad \rightarrow \quad E'_p \\
\quad [\textbf{otherwise } E'_{p+1}] \\
\textbf{endcase}
\end{array} \right]\right] \stackrel{\texttt{def}}{=} \left[\left[\mathcal{C}, \begin{array}{l}
\textbf{case} \\
\quad E::P_1^0 \textbf{ and } \ldots \textbf{ and } E::P_{n_0}^0 \\
\qquad [\textbf{when } E_0] \quad \rightarrow \quad E'_0 \\
\quad \ldots \\
\quad E::P_{n_p}^p \textbf{ and } \ldots \textbf{ and } E::P_{n_p}^p \\
\qquad [\textbf{when } E_p] \quad \rightarrow \quad E'_p \\
\quad \textbf{otherwise} \quad \rightarrow \quad E'_{p+1} \\
\textbf{endcase}
\end{array} \right]\right]$$

9.12 Match expression

Syntax

E **match** P

Translation to the core language

The match expression is syntactic sugar of general case.

$$[[\mathcal{C},\ E \textbf{ match } P]] \stackrel{\texttt{def}}{=} \left[\left[\mathcal{C}, \begin{array}{l}
\textbf{case} \\
\quad E::P \quad \rightarrow \quad \textbf{true} \\
\quad \textbf{otherwise} \quad \rightarrow \quad \textbf{false} \\
\textbf{endcase}
\end{array} \right]\right]$$

We suppose that the declaration of type "**bool**" is:

type bool is true | **false endtype**

9.13 If-then-else

Syntax

if E_0 **then** E'_0
elsif E_1 **then** E'_1
$\ldots$
elsif E_n **then** E'_n
else E'_{n+1}
endif

Translation to the core language

$$\left[\!\!\left[\; \mathcal{C},\; \begin{array}{l} \textbf{if } E_0 \textbf{ then } E'_0 \\ \textbf{elsif } E_1 \textbf{ then } E'_1 \\ \ldots \\ \textbf{elsif } E_n \textbf{ then } E'_n \\ \textbf{else } E'_{n+1} \\ \textbf{endif} \end{array} \;\right]\!\!\right] \overset{\texttt{def}}{=} \left(\begin{array}{l} \textbf{case } [[\mathcal{C},\ E_0]] \textbf{ in} \\ \quad \textbf{true} \;\rightarrow\; [[\mathcal{C},\ E'_0]] \\ \quad \textbf{false} \;\rightarrow \\ \qquad\qquad \textbf{case } [[\mathcal{C},\ E_1]] \textbf{in} \\ \qquad\qquad\quad \textbf{true} \;\rightarrow\; [[\mathcal{C},\ E'_1]] \\ \qquad\qquad\quad \ldots \\ \qquad\qquad \textbf{endcase} \\ \textbf{endcase} \end{array} \right)$$

9.14 Conjunction

Syntax

E_0 **andthen** E_1

Translation to the core language

$$[[\mathcal{C},\ E_0 \textbf{ andthen } E_1]] \overset{\texttt{def}}{=} \left(\begin{array}{l} \textbf{case } [[E_0]] \textbf{ in} \\ \quad \textbf{true} \;\rightarrow\; [[\mathcal{C},\ E_1]] \\ \quad \textbf{false} \;\rightarrow\; \textbf{false} \\ \textbf{endcase} \end{array} \right)$$

9.15 Disjunction

Syntax

E_0 **orelse** E_1

Translation to the core language

$$[[\mathcal{C},\ E_0 \textbf{ orelse } E_1]] \overset{\texttt{def}}{=} \left(\begin{array}{l} \textbf{case } [[\mathcal{C},\ E_0]] \textbf{ in} \\ \quad \textbf{true} \;\rightarrow\; \textbf{true} \\ \quad \textbf{false} \;\rightarrow\; [[\mathcal{C},\ E_1]] \\ \textbf{endcase} \end{array} \right)$$

9.16 Equality

Syntax

E_1 = E_2

Translation to the core language

$$[[\mathcal{C},\ E_1 \texttt{ = } E_2]] \overset{\texttt{def}}{=} \left(\begin{array}{l} \textbf{case}\ \texttt{(}E_1\texttt{,}\ E_2\texttt{)}\ \textbf{is} \\ \quad \texttt{(?}V_1\texttt{, ?}V_2\texttt{)} \rightarrow \\ \qquad \textbf{case}\ V_1\ \textbf{is} \\ \qquad\quad \texttt{!}V_2 \rightarrow \textbf{true} \\ \qquad\quad \textbf{otherwise} \rightarrow \textbf{false} \\ \qquad \textbf{endcase} \\ \textbf{endcase} \end{array} \right)$$

9.17 Inequality

Syntax

E_1 <> E_2

Translation to the core language

$$[[\mathcal{C},\ E_1 \texttt{ <> } E_2]] \overset{\texttt{def}}{=} \left[\left[\mathcal{C},\ \begin{array}{l} \textbf{case}\ E_1\texttt{=}E_2\ \textbf{is} \\ \quad \textbf{true} \rightarrow \textbf{false} \\ \quad \textbf{false} \rightarrow \textbf{true} \\ \textbf{endcase} \end{array} \right]\right]$$

9.18 Select field

Syntax

$E.V$

Static semantics

$$\begin{array}{lll} \mathcal{C} & \vdash & E \Rightarrow \textbf{exit}\ S \\ \mathcal{C} & \vdash & C \Rightarrow (), rt, () \rightarrow \textbf{exit}\ S \rightarrow uid \\ rt & \vdash & V \Rightarrow (S : def) \\ \hline \mathcal{C} & \vdash & (E.V) \Rightarrow \textbf{exit}\ S' \end{array}$$

Translation to the core language

$$[[\mathcal{C},\ E_0.V_0]] \overset{\texttt{def}}{=} \left[\left[\ \mathcal{C},\ \begin{array}{l} \textbf{local var}\ V{:}S_0 \\ \textbf{init}\ V{:=}E_0 \\ \textbf{in} \\ \quad \textbf{case}\ V{:}S_0\ \textbf{in} \\ \qquad C_i\ (V_0{:=}?V_0, \ldots)\ \rightarrow\ V_0 \\ \ldots \\ \quad \textbf{endcase} \\ \textbf{endlet} \end{array} \right]\right]$$

where $\mathcal{C} \vdash E_0 \Rightarrow \textbf{exit}\ S_0$, and C_i are constructors of type S_0 such that they have as field V_0 (they exist from the static semantics checking).

9.19 Field updating

Syntax

$E.\{V_1{=>}E_1, ..., V_n{=>}E_n\}$

Static semantics

$$\begin{array}{ccl} \mathcal{C} & \vdash & E \Rightarrow \textbf{exit}\ S \\ \mathcal{C} & \vdash & C \Rightarrow (), rt, ()\ \rightarrow\ \textbf{exit}\ S\ \rightarrow\ uid \\ rt & \vdash & V_1 \Rightarrow (S_1 : def) \quad \cdots \quad rt \vdash V_n \Rightarrow (S_n : def) \\ \mathcal{C} & \vdash & E_1 \Rightarrow \textbf{exit}\ S_1 \quad \cdot \quad \mathcal{C} \vdash E_n \Rightarrow \textbf{exit}\ S_n \\ \hline \mathcal{C} & \vdash & (E.\{V_1{=>}E_1, \ldots, V_n{=>}E_n\}) \Rightarrow \textbf{exit}\ S \end{array}$$

Translation to the core language

$$[[\mathcal{C},\ E_0.\{V_1{=>}E_1, ..., V_n{=>}E_n\}]] \overset{\texttt{def}}{=} \left[\left[\ \mathcal{C},\ \begin{array}{l} \textbf{local var}\ V_0 : S_0 \\ \textbf{init}\ ?V_0{:=}E_0 \\ \textbf{in case}\ V_0\ \textbf{in} \\ \qquad\quad C_i\_uid_i\ (\ldots)\ \rightarrow \\ \qquad\qquad C_i\ (V_1{=>}E_1, ..., V_n{=>}E_n, \ldots) \\ \qquad\quad \ldots \\ \qquad \textbf{endcase} \\ \textbf{endloc} \end{array} \right]\right]$$

where C_i are the constructors of the sort S_0 such that $V_1, \ldots, V_n$ are fields of their record types (they exists by the static semantic).

9.20 Variable declaration

Syntax

local var $V_1{:}S_1, \ldots, V_n{:}S_n$ [**init** E_0] **in** E **endloc**

Static semantics

$$\frac{\begin{array}{rcl} \mathcal{C} & \vdash & S_1 \Rightarrow S_1' \quad \cdots \quad \mathcal{C} \vdash S_n \Rightarrow S_n' \\ \langle \mathcal{C} \oplus (+_{i=1}^{i=n}\ V_i \mapsto (S_i',\ undef)) & \vdash & E_0 \Rightarrow \textbf{exit}\ (rt_0)\rangle \\ \mathcal{C} \oplus (+_{i=1}^{i=n}\ V_i \mapsto (S_i',\ undef))\langle \oplus rt_0 \rangle & \vdash & E \Rightarrow \textbf{exit}\ rt \end{array}}{\mathcal{C} \vdash \left(\begin{array}{l} \textbf{local}\ V_1 : S_1, \ldots, V_n : S_n \\ \textbf{init}\ E_0 \\ \textbf{in}\ E \end{array}\right) \Rightarrow \textbf{exit}\ (rt \ominus \{V_1, \ldots, V_n\})}$$

Translation to the core language

$$\left[\left[\mathcal{C}, \begin{array}{l} \textbf{local var}\ V_1 : S_1, \ldots, V_n : S_n \\ \textbf{init}\ E_0 \\ \textbf{in}\ E \end{array} \right]\right] \stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{local var}\ V_1 \Rightarrow V_1 : V_1 \Rightarrow S_1, \ldots \\ \textbf{init}\ [[\mathcal{C},\ E_0]] \\ \textbf{in}\ [[\mathcal{C},\ E]] \end{array}\right)$$

9.21 Rename

Syntax

rename
  $X_0\ [RT_0]$ **is** $X_0'[RE_0]$
  $\ldots$
  $X_p\ [RT_p]$ **is** $X_p'[RE_p]$
in E
endren

Static semantics

$$\frac{\begin{array}{rcl} \mathcal{C} & \vdash & RT_0 \Rightarrow rt_0 \quad \cdots \quad \mathcal{C} \vdash RT_p \Rightarrow rt_p \\ \mathcal{C} \oplus rt_0 & \vdash & \textbf{raise}\ X_0'\ RE_0 \Rightarrow \textbf{exit}\ () \quad \cdots \quad \mathcal{C} \oplus rt_p \vdash \textbf{raise}\ X_p'\ RE_p \Rightarrow \textbf{exit}\ () \\ \mathcal{C} \oplus (+_{i=0}^{i=p}\ X_i \mapsto \textbf{exn}\ (rt_i)) & \vdash & E \Rightarrow \textbf{exit}\ t \end{array}}{\mathcal{C} \vdash \left(\begin{array}{l} \textbf{rename} \\ \quad X_0\ [RT_0]\ \textbf{is}\ X_0'[RE_0] \\ \quad \ldots \\ \quad X_p\ [RT_p]\ \textbf{is}\ X_p'[RE_p] \\ \textbf{in}\ E \end{array}\right) \Rightarrow \textbf{exit}\ t}$$

Translation to the core language

$$\left[\left[\mathcal{C}, \begin{array}{l} \textbf{rename} \\ \quad X_0\ [RT_0]\ \textbf{is}\ X_0'[RE_0] \\ \quad \ldots \\ \quad X_p\ [RT_p]\ \textbf{is}\ X_p'[RE_p] \\ \textbf{in}\ E \\ \textbf{endren} \end{array} \right]\right] \stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{rename} \\ \quad X_0\ [[\mathcal{C},\ RT_0]]\ \textbf{is}\ X_0'\ [[\mathcal{C},\ RE_0 \Rightarrow rt_0]] \\ \quad \ldots \\ \quad X_p\ [[\mathcal{C},\ RT_p]]\ \textbf{is}\ X_p'\ [[\mathcal{C},\ RE_p \Rightarrow rt_p]] \\ \textbf{in}\ [[\mathcal{C},\ E]] \\ \textbf{endren} \end{array}\right)$$

where for all $i \in 1..p\ \mathcal{C}(X_i') = \textbf{exn}\ (rt_i)$.

9.22 Function call

Syntax

$F\ RE\ XP$

Static semantics

$$\begin{array}{ccl} & & \mathcal{C}(F)((), rt, rx) = \{\mathbf{exit}\ t\ \rightarrow\ uid\} \\ \mathcal{C} & \vdash & RE \Rightarrow (rt) \\ \mathcal{C} & \vdash & XP \Rightarrow rx \\ \hline \mathcal{C} & \vdash & F\ RE\ XP\ \Rightarrow \mathbf{exit}\ t \end{array}$$

Translation to the core language

$[[\mathcal{C},\ F\ RE\ XP]] \overset{\mathtt{def}}{=} F\_uid\ \Box\ [[\mathcal{C},\ RE \Rightarrow rt]]\ [[\mathcal{C},\ XP \Rightarrow rx]]$
where $\mathcal{C}(F)((), rt, rx) = \{\mathbf{exit}\ t\ \rightarrow\ uid\}$ (a single element set!).

9.23 Procedure call

Syntax

$F\ IOP\ XP$

Static semantics

$$\begin{array}{ccl} & & \mathcal{C}(F)((), rt, rx) = \{\mathbf{exit}\ t\ \rightarrow\ uid\} \\ \mathcal{C} & \vdash & IOP \Rightarrow rt \\ \mathcal{C} & \vdash & XP \Rightarrow rx \\ \hline \mathcal{C} & \vdash & F\ IOP\ XP\ \Rightarrow \mathbf{exit}\ t \end{array}$$

Translation to the core language

$$[[\mathcal{C},\ F\ IOP\ XP]] \overset{\mathtt{def}}{=} \left(\begin{array}{l} \mathbf{trap} \\ \quad \mathbf{exit}(\mathtt{?}x)\ \rightarrow\ (\mathbf{out}(IOP, rt))\mathtt{:=}x \\ \mathbf{in}\ F\_uid\ \Box\ [[\mathcal{C},\ \mathbf{in}(IOP, rt)]]\ [[\mathcal{C},\ XP \Rightarrow rx]] \end{array} \right)$$

where $\mathcal{C}(F)((), rt, rx) = \{\mathbf{exit}\ t\ \rightarrow\ uid\}$ (a single element set!).

9.24 Iteration

Syntax

loop forever
[**var out** $V_1 : S_1, \ldots,$ **out** $V_n : S_n$]
[**init** E_0]
in E
endloop

The default values are "**var** ()" and "**init exit**".

Static semantics

$$\frac{\begin{array}{rcll} \mathcal{C} & \vdash & RT \Rightarrow rt & \\ \mathcal{C} \oplus rt & \vdash & E_0 \Rightarrow \textbf{exit}\ (rt_0) & \langle \mathrm{Dom}(rt_0) \subset \mathrm{Dom}(rt)\rangle \\ \mathcal{C} \oplus rt \oplus rt_0 & \vdash & E \Rightarrow \textbf{exit}\ (rt') & \mathrm{Dom}(rt') = \mathrm{Dom}(rt)\rangle \end{array}}{\mathcal{C} \ \vdash \left(\begin{array}{l} \textbf{loop forever} \\ \textbf{var}\ RT \\ \textbf{init}\ E_0 \\ \textbf{in}\ E \end{array}\right) \Rightarrow \textbf{exit none}}$$

Translation to the core language

$$\left[\left[\ \mathcal{C},\ \begin{array}{l} \textbf{loop forever} \\ \textbf{var}\ V_1 : S_1, \ldots, V_n : S_n \\ \textbf{init}\ E_0 \\ \textbf{in}\ E \end{array}\right]\right] \overset{\mathtt{def}}{=} \left(\begin{array}{l} \textbf{loop forever} \\ \textbf{var}\ V_1 \Rightarrow V_1 : V_1 \Rightarrow S_1, \ldots \\ \textbf{init}\ [[\mathcal{C},\ E_0]] \\ \textbf{in}\ [[\mathcal{C},\ E]] \end{array}\right)$$

9.25 Breakable iteration

Syntax

loop $X[RT]$
 [**init** E_0]
in E
endloop

The default type for exception is () (the empty record); the default initialization is "**init exit**".

Static semantics

$$\frac{\begin{array}{rcll} \mathcal{C} & \vdash & \textbf{out}\ RT \Rightarrow rt & \\ \mathcal{C} \oplus rt & \vdash & E_0 \Rightarrow \textbf{exit}\ (rt_0) & \mathrm{Dom}(rt_0) \subset \mathrm{Dom}(rt) \\ \mathcal{C} \oplus rt \oplus rt_0 \oplus & & & \\ (X \mapsto \textbf{exn}\ (rt)) & \vdash & E \Rightarrow \textbf{exit}\ (rt') & \mathrm{Dom}(rt') = \mathrm{Dom}(rt) \end{array}}{\mathcal{C} \ \vdash \left(\begin{array}{l} \textbf{loop}\ X\ RT \\ \textbf{init}\ E_0 \\ \textbf{in}\ E \end{array}\right) \Rightarrow \textbf{exit none}}$$

Translation to the core language

$$\left[\left[\ \mathcal{C},\ \begin{array}{l} \textbf{loop}\ X[(\\ \\ \\ \end{array}\right.\right.$$