A core data and behaviour language for E-LOTOS

Alan Jeffrey

Based on discussions at the COST 247 short term scientific mission
attended by Hubert Garavel,. Guy Leduc, Charles Pecheur,
Ricardo Peña and Mihaela Sighireanu
University of Liège April 1996

Abstract

This paper presents an integrated core data and behaviour language for LOTOS. This core language is not directly usable for specifications, but we can define some syntax sugar to make it more usable and compatible with existing specifications.

1 Introduction

This paper presents static and dynamic semantics for a fragment of LOTOS with a functional (rather than algebraic) data language.

The fragment considered is based on the core languages discussed in [1], and extends it by considering *exception handling* and *subtyping*.

Exception handling has been suggested as a useful addition to LOTOS, allowing termination to be generalized from just the δ-gate to other gates.

Subtyping appears in existing LOTOS in two places:

- When the functionality of a process is calculated, **noexit** functionality is treated specially, for example:

Process	*Functionality*
$\mathbf{stop}$	$\mathbf{noexit}$
$\mathbf{exit}(1)$	$\mathbf{exit}\,\mathit{int}$
$\mathbf{stop} \mathbin{\Box} \mathbf{exit}(1)$	$\mathbf{exit}\,\mathit{int}$
$\mathbf{stop} \mathbin{\|\|} \mathbf{exit}(1)$	$\mathbf{noexit}$

- When untyped gates are used, data of more than type can be sent on a single gate, for example:

$$G!1; G!\mathit{true}; \mathbf{stop}$$

In existing LOTOS, these two phenomena are treated by different *ad hoc* mechanisms. In this paper we propose unifying both into a form of *record subtyping*. This includes record types **none** (a type with no inhabitants) and $\langle\,\_\rangle$ a completely unspecified record. This subtyping relation allows record types to be *intersected* and *unioned*, for example:

$$\begin{array}{rcl}
\langle a : int\, b : bool\, \_\rangle \sqcap \langle a : int\, c : float\, \_\rangle & = & \langle a : int\, b : bool\, c : float\, \_\rangle \\
\langle a : int\, b : bool\, \_\rangle \sqcup \langle a : int\, c : float\, \_\rangle & = & \langle a : int\, \_\rangle \\
T \sqcup \mathbf{none} & = & T \\
T \sqcap \mathbf{none} & = & \mathbf{none}
\end{array}$$

This then provides simple rules for calculating the functionality of a behaviour, such as:

$$\frac{\begin{array}{l}\mathcal{C} \vdash B_1 \Rightarrow \mathbf{exit}\, T_1 \\ \mathcal{C} \vdash B_2 \Rightarrow \mathbf{exit}\, T_2\end{array}}{\mathcal{C} \vdash B_1 \,\Box\, B_2 \Rightarrow \mathbf{exit}\, T_1 \sqcup T_2}$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash B_1 \Rightarrow \mathbf{exit}\, T_1 \\ \mathcal{C} \vdash B_2 \Rightarrow \mathbf{exit}\, T_2\end{array}}{\mathcal{C} \vdash B_1 \,||\, B_2 \Rightarrow \mathbf{exit}\, T_1 \sqcap T_2}$$

This paper is concerned with the integration of a functional data language with the LOTOS behavioural language. In particular:

- In Section 2 we present a 'core' behaviour language based on [1]. This is given a static and dynamic semantics in the style of the SML formal language definition [3].
- In Section 3 we provide syntax sugar to make the core language more usable, for example providing 'if' statements as syntax sugar for 'case'. The most important addition is the use of immediately exiting LOTOS behaviours to perform data computations.

This paper uses syntax and definitions from [2, 1], to which the reader is referred for an introduction to the language, examples of its use, and motivation for the design choices made.

2 Core language

The core language we consider here is monomorphic, explicitly typed, and allows record subtyping. In this paper we do not consider implicit typing or overloading.

The terminals of the abstract syntax are:

syntactic category	*symbol*
type identifier	S
variable identifier	V
gate identifier	G
process identifier	Q

The non-terminals are:

syntactic category	*symbol*
type	T
constant	K
primitive constant	R
pattern	P
behaviour	B
pattern-match	M

In this paper we will not discuss the primitive constants, but we assume they contain standard constants such as integers, floats, and strings.

In this abstract grammar we have not given **end** keywords for each of the constructs, for example there is no **endproc** keyword. In the concrete grammar these should be included where appropriate. We have also used $\langle\cdots\rangle$ as the syntax for records rather than $(\cdots)$, and $\_$ as the syntax for record wildcard rather than ..., in order to clarify the difference between the concrete and meta languages (for example $\langle E_1,\ldots,E_n,\_\rangle$ rather than $(E_1,\ldots,E_n,\ldots)$).

The static semantics is given by judgements such as $\mathcal{C} \vdash B \Rightarrow \mathbf{exit}\,T$ where $\mathcal{C}$ is a *context* given by the grammar:

$$\begin{array}{lll}
\mathcal{C} & ::= & V \Rightarrow T \\
 & \mid & S \Rightarrow \mathbf{type} \\
 & \mid & C \Rightarrow (T \rightarrow S) \\
 & \mid & Q \Rightarrow ([(\mathbf{gate}\,T)^*] \rightarrow T \rightarrow T) \\
 & \mid & G \Rightarrow \mathbf{gate}\,T \\
 & \mid & () \\
 & \mid & \mathcal{C},\mathcal{C}
\end{array}$$

where all identifiers in a context must be unique. We view contexts up to ‘,’ being a commutative monoid with unit $()$. We write $\mathcal{C}_1;\mathcal{C}_2$ for the context given by over-riding $\mathcal{C}_1$ by $\mathcal{C}_2$, and $\mathcal{C} \vdash V \Rightarrow T$ for $\mathcal{C} = \mathcal{C}', V \Rightarrow T$ (and similarly for the other judgements).

The dynamic semantics is given by judgements such as $\mathcal{E} \vdash B \xrightarrow{\alpha(K)} B'$ where $\mathcal{E}$ is an *environment* given by the grammar:

$$\begin{array}{lll}
\mathcal{E} & ::= & C \Rightarrow (T \rightarrow S) \\
 & \mid & Q \Rightarrow \lambda[G^*]M \\
 & \mid & () \\
 & \mid & \mathcal{E},\mathcal{E}
\end{array}$$

where all identifiers in an environment must be unique. We use the same notation for environments as we do for contexts. Note that since LOTOS allows gates to be untyped, we have to perform run-time type-checking, so we have to carry the types of constructors in environments.

2.1 Declarations

Declarations come in two flavours: *datatype declarations* such as:

$$\begin{array}{l} \textbf{type}\, \mathit{intlist}\, \textbf{is} \\ \quad \mathit{nil}\langle\rangle \\ \quad \mathit{cons}\langle \mathit{int} * \mathit{intlist}\rangle \end{array}$$

and *process declarations* such as:

$$\begin{array}{l} \textbf{process}\, \mathit{Stack}\,[i : \textbf{gate}\, \mathit{int}, o : \textbf{gate}\, \mathit{int}]\,\langle l : \mathit{intlist}\rangle : \textbf{exit}\, \textbf{none}\, \textbf{is} \\ \textbf{case}\, l\, \textbf{of} \\ \quad \mathit{nil}\langle\rangle \rightarrow \\ \qquad i?(x : \mathit{int}); \mathit{Stack}[i,o]\langle \mathit{cons}\langle x, l\rangle\rangle \\ \quad \mathit{cons}\langle y : \mathit{int}, \mathit{ys} : \mathit{intlist}\rangle \rightarrow \\ \qquad i?x : \mathit{int}; \mathit{Stack}[i,o]\langle \mathit{cons}\langle x, l\rangle\rangle \,\Box\, o!y; \mathit{Stack}[i,o]\langle \mathit{ys}\rangle \end{array}$$

The syntax of declarations is:

$$\begin{array}{lcl} D & ::= & \textbf{type}\, S\, \textbf{is}\, (C(T))^{*} \\ & \mid & \textbf{process}\, Q\,[(G : \textbf{gate}\, T)^{*}]\, T : \textbf{exit}\, T\, \textbf{is}\, M \end{array}$$

The static semantics of declarations is given with judgements of the form:

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

The dynamic semantics of declarations is given with judgements of the form:

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

2.1.1 Type declarations

Syntax:

$$\textbf{type}\, S\, \textbf{is}\, (C(T))^{*}$$

Static semantics:

$$\frac{\mathcal{C} \vdash \vec{T} \Rightarrow \textbf{type}}{\mathcal{C} \vdash \textbf{type}\, S\, \textbf{is}\, \vec{C}(\vec{T}) \Rightarrow (S \Rightarrow \textbf{type}, \vec{C} \Rightarrow (\vec{T} \rightarrow S))}$$

Dynamic semantics:

$$\frac{}{\mathcal{E} \vdash \textbf{type}\, S\, \textbf{is}\, \vec{C}(\vec{T}) \Rightarrow (\vec{C} \Rightarrow \vec{T} \rightarrow S)}$$

2.1.2 Process declarations

Syntax:

$$\mathbf{process}\,Q\,[(G : \mathbf{gate}\,T)^*]\,T : \mathbf{exit}\,T\ \mathbf{is}\,M$$

Static semantics:

$$\frac{\begin{array}{c}\mathcal{C} \vdash \vec{T} \Rightarrow \mathbf{type} \\ \mathcal{C}; \vec{G} \Rightarrow \mathbf{gate}\,\vec{T} \vdash M \Rightarrow T \rightarrow \mathbf{exit}\,T'\end{array}}{\begin{array}{c}\mathcal{C} \vdash \mathbf{process}\,Q\,[\vec{G} : \mathbf{gate}\,\vec{T}]\,T : \mathbf{exit}\,T'\ \mathbf{is}\,M \Rightarrow \\ (Q \Rightarrow [\mathbf{gate}\,\vec{T}] \rightarrow T \rightarrow \mathbf{exit}\,T')\end{array}}$$

Dynamic semantics:

$$\frac{}{\begin{array}{c}\mathcal{E} \vdash \mathbf{process}\,Q\,[\vec{G} : \mathbf{gate}\,\vec{T}]\,T : \mathbf{exit}\,T'\ \mathbf{is}\,M \Rightarrow \\ (Q \Rightarrow \lambda[\vec{G}]M)\end{array}}$$

2.2 Types

A type is either:

- a type identifier S,
- a fixed record of types $\langle V_1 : T_1 \cdots V_n : T_n \rangle$,
- an extensible record of types $\langle V_1 : T_1 \cdots V_n : T_n \,\_\rangle$, or
- the empty type **none**.

We can define a subtyping relation

Process	*Functionality*
$\mathbf{exit}\,\langle a := 1\ b := \mathit{true}\,\_\rangle$	$\mathbf{exit}\,\langle a : \mathit{int}\ b : \mathit{bool}\,\_\rangle$
$\mathbf{exit}\,\langle a := 1\ c := 1.0\,\_\rangle$	$\mathbf{exit}\,\langle a : \mathit{int}\ c : \mathit{float}\,\_\rangle$
$\mathbf{exit}\,\langle a := 1\ b := \mathit{true}\,\_\rangle \Box \mathbf{exit}\,\langle a := 1\ c := 1.0\,\_\rangle$	$\mathbf{exit}\,\langle a : \mathit{int}\,\_\rangle$
$\mathbf{exit}\,\langle a := 1\ b := \mathit{true}\,\_\rangle \,\|\, \mathbf{exit}\,\langle a := 1\ c := 1.0\,\_\rangle$	$\mathbf{exit}\,\langle a : \mathit{int}\ b : \mathit{bool}\ c : \mathit{float}\,\_\rangle$

The syntax of types is:

$$\begin{array}{lcl} T & ::= & S \\ & \mid & \langle (V : T)^* [\,\_\,] \rangle \\ & \mid & \mathbf{none} \end{array}$$

where we require record field names to be disjoint. A *record type* is any type other than a type identifier S.

The static semantics is given by judgements of the form:

$$\mathcal{C} \vdash T \Rightarrow \mathbf{type}$$

Types have no dynamic semantics.

We define *record subtyping* as a preorder $\sqsubseteq$ on record types, generated by:

$$\begin{array}{rcl}
\mathbf{none} & \sqsubseteq & \langle \vec{V} : \vec{T}\,[\_] \rangle \\
\langle \vec{V}_1 : \vec{T}_1\, \vec{V}_2 : \vec{T}_2\,[\_] \rangle & \sqsubseteq & \langle \vec{V}_1 : \vec{T}_1\, \_ \rangle \\
\langle \vec{V}_1 : \vec{T}_1\, \vec{V}_2 : \vec{T}_2 \rangle & \sqsubseteq & \langle \vec{V}_2 : \vec{T}_2\, \vec{V}_1 : \vec{T}_1 \rangle \\
\langle \vec{V}_1 : \vec{T}_1\, \vec{V}_2 : \vec{T}_2\, \_ \rangle & \sqsubseteq & \langle \vec{V}_2 : \vec{T}_2\, \vec{V}_1 : \vec{T}_1\, \_ \rangle
\end{array}$$

This is a preorder with bottom **none** and top $\langle \_ \rangle$. We shall write $T \equiv T'$ for the resulting equivalence on types (given by commutativity of record fields).

If:

$$T' \equiv \langle \vec{V} : \vec{T}\, \vec{V}' : \vec{T}'\,[\_] \rangle \quad T'' \equiv \langle \vec{V} : \vec{T}\, \vec{V}'' : \vec{T}''\,[\_] \rangle \quad \text{where } V'_i = V''_j \text{ implies } T'_i \neq T''_j$$

then we can define type union as:

$$\begin{array}{rcl}
\mathbf{none} \sqcup T & \equiv & T \\
T \sqcup \mathbf{none} & \equiv & T \\
T' \sqcup T'' & \equiv & \begin{cases} T' & \text{if } T' \equiv T'' \\ \langle \vec{V} : \vec{T}\, \_ \rangle & \text{otherwise} \end{cases}
\end{array}$$

If:

$$T' \equiv \langle \vec{V} : \vec{T}\, \vec{V}' : \vec{T}'\,[\_] \rangle \quad T'' \equiv \langle \vec{V} : \vec{T}\, \vec{V}'' : \vec{T}''\,[\_] \rangle \quad \text{where } V'_i = V''_j \text{ implies } T'_i \neq T''_j$$

then we can define type intersection as:

$$\begin{array}{rcl}
\mathbf{none} \sqcap T & \equiv & \mathbf{none} \\
T \sqcap \mathbf{none} & \equiv & \mathbf{none} \\
T' \sqcap T'' & \equiv & \begin{cases} T' & \text{if } T' \equiv T'' \\ T' & \text{if } \vec{V}'' \text{ is empty and } T'' \text{ is extensible} \\ T'' & \text{if } \vec{V}' \text{ is empty and } T' \text{ is extensible} \\ \langle \vec{V} : \vec{T}\, \vec{V}' : \vec{T}'\, \vec{V}'' : \vec{T}''\, \_ \rangle & \text{if } \vec{V}' \text{ and } \vec{V}'' \text{ are disjoint} \\ & \quad \text{and } T' \text{ and } T'' \text{ are extensible} \\ \mathbf{none} & \text{otherwise} \end{cases}
\end{array}$$

Type union and type intersection are join (or least upper bound) and meet (or greatest lower bound) for subtyping.

2.3 Type identifiers

Syntax:

$$S$$

Static semantics:

$$\frac{}{\mathcal{C}, S \Rightarrow \mathbf{type} \vdash S \Rightarrow \mathbf{type}}$$

2.3.1 Records

Syntax:

$$\langle (V : T)^{*} [\,\_\,] \rangle$$

Static semantics:

$$\frac{\mathcal{C} \vdash \vec{T} \Rightarrow \mathbf{type}}{\mathcal{C} \vdash \langle \vec{V} : \vec{T} [\,\_\,] \rangle \Rightarrow \mathbf{type}}$$

2.3.2 Empty type

Syntax:

none

Static semantics:

$$\frac{}{\mathcal{C} \vdash \mathbf{none} \Rightarrow \mathbf{type}}$$

2.4 Patterns

Patterns are used in defining processes, case statements, and communication offers, for example:

$$G\langle a := !1\, b := ?x : bool\,\_\rangle; \mathbf{exit}\,\langle a := !0\, b := !x\,\_\rangle$$

The syntax of patterns is:

$$\begin{array}{lcl} P & ::= & R \\ & \mid & \langle (V := P)^{*} [\,\_\,] \rangle \\ & \mid & C(P) \\ & \mid & \mathbf{any} : S \\ & \mid & ?V\,\mathbf{as}\,P \\ & \mid & !K \end{array}$$

where we require all record field names to be unique.

The static semantics is given by judgements of the form:

$$\mathcal{C} \vdash P \Rightarrow (T \rightarrow \mathcal{C}') \qquad \mathcal{C} \vdash P \Rightarrow (S \rightarrow \mathcal{C}')$$

The dynamic semantics is given by judgements of the form:

$$\mathcal{C} \vdash (P \Rightarrow K) \Rightarrow \sigma \qquad \mathcal{C} \vdash (P \Rightarrow K) \Rightarrow \mathbf{fail}$$

where σ is a *substitution*.

2.4.1 Primitive constants

Syntax:

$$R$$

Static semantics:

$$\frac{}{\mathcal{C} \vdash R \Rightarrow (S \rightarrow ())}\,[R : S]$$

Dynamic semantics:

$$\frac{}{\mathcal{E} \vdash (R \Rightarrow R) \Rightarrow ()}$$

$$\frac{}{\mathcal{E} \vdash (R \Rightarrow K) \Rightarrow \textbf{fail}}\,[R \neq K]$$

2.4.2 Records

Syntax:

$$\langle (V := P)^{*}\,[\,\_\,]\rangle$$

Static semantics:

$$\frac{\mathcal{C} \vdash \vec{P} \Rightarrow (\vec{T} \rightarrow \vec{C})}{\mathcal{C} \vdash \langle \vec{V} := \vec{P}\,[\,\_\,]\rangle \Rightarrow (\langle \vec{V} : \vec{T}\,[\,\_\,]\rangle \rightarrow (\vec{\mathcal{C}}\,))}$$

Dynamic semantics:

$$\frac{\mathcal{E} \vdash (P \Rightarrow K) \Rightarrow \textbf{fail}}{\mathcal{E} \vdash (\langle \cdots V := P \cdots \rangle \Rightarrow \langle \cdots V := K \cdots \rangle) \Rightarrow \textbf{fail}}$$

$$\frac{\mathcal{E} \vdash (\vec{P} \Rightarrow \vec{K}) \Rightarrow \vec{\sigma}}{\mathcal{E} \vdash (\langle \vec{V} := \vec{P}\rangle \Rightarrow \langle \vec{V} := \vec{K}\rangle) \Rightarrow (\vec{\sigma})}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash (\vec{P} \Rightarrow \vec{K}) \Rightarrow \vec{\sigma} \\ \mathcal{E} \vdash \vec{K}' \Rightarrow \vec{T}'\end{array}}{\mathcal{E} \vdash (\langle \vec{V} := \vec{P}\,\_\,\rangle \Rightarrow \langle \vec{V} := \vec{K}\,\vec{V}' := \vec{K}'\,[\,\_\,]\rangle) \Rightarrow (\vec{\sigma})}$$

2.4.3 Constructor application

Syntax:

$$C(P)$$

Static semantics:

$$\frac{\begin{array}{l}\mathcal{C} \vdash C \Rightarrow (T \rightarrow S)\\ \mathcal{C} \vdash P \Rightarrow (T \rightarrow \mathcal{C}')\end{array}}{\mathcal{C} \vdash C(P) \Rightarrow (S \rightarrow \mathcal{C}')}$$

Dynamic semantics:

$$\frac{\mathcal{E} \vdash (P \Rightarrow K) \Rightarrow (\sigma \mid \mathbf{fail})}{\mathcal{E} \vdash (C(P) \Rightarrow C(K)) \Rightarrow (\sigma \mid \mathbf{fail})}$$

$$\frac{}{\mathcal{E} \vdash (C(P) \Rightarrow K) \Rightarrow \mathbf{fail}}[K \neq C(\cdots)]$$

2.4.4 Wildcard

Syntax:

$$\mathbf{any} : T$$

Static semantics:

$$\frac{}{\mathcal{C} \vdash \mathbf{any} : T \Rightarrow (T \rightarrow ())}$$

Dynamic semantics:

$$\frac{\mathcal{E} \vdash K \Rightarrow T}{\mathcal{E} \vdash (\mathbf{any} : T \Rightarrow K) \Rightarrow ()}$$

2.4.5 Bound variables

Syntax:

$$?V\ \mathbf{as}\ P$$

Static semantics:

$$\frac{\mathcal{C} \vdash P \Rightarrow (T \Rightarrow \mathcal{C}')}{\mathcal{C} \vdash ?V\ \mathbf{as}\ P \Rightarrow (T \rightarrow (\mathcal{C}', V \Rightarrow T))}$$

Dynamic semantics:

$$\frac{\mathcal{E} \vdash (P \Rightarrow K) \Rightarrow \sigma}{\mathcal{E} \vdash (?V\ \mathbf{as}\ P \Rightarrow K) \Rightarrow (\sigma, K/V)}$$

2.4.6 Constants

Syntax:

$$!K$$

Static semantics:

$$\frac{\mathcal{C} \vdash K \Rightarrow T}{\mathcal{C} \vdash\ !K \Rightarrow (T \rightarrow ())}$$

Dynamic semantics:

$$\frac{}{\mathcal{E} \vdash (!K \Rightarrow K) \Rightarrow ()}$$

$$\frac{}{\mathcal{E} \vdash (!K \Rightarrow K') \Rightarrow \textbf{fail}}[K \neq K']$$

2.5 Pattern-matching

The syntax of pattern-matching is:

$$M \quad ::= \quad P[B] \rightarrow B(\ |\ P[B] \rightarrow B)^*$$

The static semantics is given by judgements of the form:

$$\mathcal{C} \vdash M \Rightarrow (T \rightarrow \textbf{exit}\, T')$$

The dynamic semantics is given by judgements of the form:

$$\mathcal{C} \vdash (M \Rightarrow K) \xrightarrow{\alpha(K')} B$$

2.5.1 Pattern-match

Syntax:

$$P[B] \rightarrow B(\ |\ P[B] \rightarrow B)^*$$

Static semantics:

$$\frac{\begin{array}{c} \mathcal{C} \vdash P \Rightarrow (T \rightarrow \mathcal{C}') \\ \mathcal{C};\mathcal{C}' \vdash B_1 \Rightarrow \textbf{exit}\, bool \\ \mathcal{C};\mathcal{C}' \vdash B_2 \Rightarrow \textbf{exit}\, T' \end{array}}{\mathcal{C} \vdash (P[B_1] \rightarrow B_2) \Rightarrow (T \rightarrow \textbf{exit}\, T')}$$

$$\frac{\begin{array}{c} \mathcal{C} \vdash P \Rightarrow (T_1 \rightarrow \mathcal{C}') \\ \mathcal{C};\mathcal{C}' \vdash B_1 \Rightarrow \textbf{exit}\, bool \\ \mathcal{C};\mathcal{C}' \vdash B_2 \Rightarrow \textbf{exit}\, T_1' \\ \mathcal{C} \vdash M \Rightarrow (T_2 \rightarrow \textbf{exit}\, T_2') \end{array}}{\mathcal{C} \vdash (P[B_1] \rightarrow B_2) \mid M \Rightarrow (T_1 \sqcap T_2 \rightarrow \textbf{exit}\, T_1' \sqcup T_2')}$$

Dynamic semantics:

$$\frac{\begin{array}{l}\mathcal{E} \vdash (P \Rightarrow K) \Rightarrow \sigma \\ \mathcal{E} \vdash B_1[\sigma] \xrightarrow{\delta true} B'_1 \\ \mathcal{E} \vdash B_2[\sigma] \xrightarrow{\alpha(K)} B'_2\end{array}}{\mathcal{E} \vdash ((P[B_1] \rightarrow B_2[\,|\,M] \Rightarrow K) \xrightarrow{\alpha(K)} B'_2}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash (P \Rightarrow K) \Rightarrow \sigma \\ \mathcal{E} \vdash B_1[\sigma] \xrightarrow{\delta false} B'_1 \\ \mathcal{E} \vdash (M \Rightarrow K) \xrightarrow{\alpha(K)} B'\end{array}}{\mathcal{E} \vdash ((P[B_1] \rightarrow B_2 \mid M) \Rightarrow K) \xrightarrow{\alpha(K)} B'}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash (P \Rightarrow K) \Rightarrow \mathbf{fail} \\ \mathcal{E} \vdash (M \Rightarrow K) \xrightarrow{\alpha(K)} B'\end{array}}{\mathcal{E} \vdash ((P[B_1] \rightarrow B_2 \mid M) \Rightarrow K) \xrightarrow{\alpha(K)} B'}$$

2.6 Constants

The syntax of constants is:

$$\begin{array}{lcl} K & ::= & R \\ & \mid & V \\ & \mid & \langle (V := K)^* \, [\_] \rangle \\ & \mid & C(K) \end{array}$$

where we require all record field names to be unique.

The static semantics is given by judgements of the form:

$$\mathcal{C} \vdash K \Rightarrow T$$

The dynamic semantics is given by judgements of the form:

$$\mathcal{E} \vdash K \Rightarrow T$$

Note that the dynamic semantics has to type-check constants: this is because LOTOS allows processes such as:

$$\mathbf{hide}\, G : \mathbf{gate}\, bool\, \mathbf{in}\, G?V : bool; H!V; \mathbf{stop}$$

which can ‘randomly generate’ any boolean—in order to ensure type safety we therefore have to carry type information at run time.

Note also that the dynamic rules for type checking are just the same as the static rules, so we omit them.

2.6.1 Primitive constants

Syntax:

$$R$$

Static semantics:

$$\frac{}{\mathcal{C} \vdash R \Rightarrow S}[R : S]$$

2.6.2 Variables

Syntax:

$$V$$

Static semantics:

$$\frac{}{\mathcal{C}, V \Rightarrow T \vdash V \Rightarrow T}$$

2.6.3 Records

Syntax:

$$\langle (V := K)^{*} \, [\,\_\,] \rangle$$

Static semantics:

$$\frac{\mathcal{C} \vdash \vec{K} \Rightarrow \vec{T}}{\mathcal{C} \vdash \langle \vec{V} := \vec{T} \, [\,\_\,] \rangle \Rightarrow \langle \vec{V} : \vec{T} \, [\,\_\,] \rangle}$$

2.6.4 Constructor application

Syntax:

$$C(K)$$

Static semantics:

$$\frac{\begin{array}{l} \mathcal{C} \vdash K \Rightarrow T \\ \mathcal{C} \vdash C \Rightarrow (T \rightarrow S) \end{array}}{\mathcal{C} \vdash C(K) \Rightarrow S}$$

2.7 Behaviours

The syntax for behaviours given here is simple, and consists of the following changes to existing LOTOS:

- Patterns and pattern-matching are used uniformly throughout the language.
- Enabling is generalized to include exception handling as well as normal termination.
- Gate renaming is added as an explicit operator, and includes the ability to perform simple data transformations as well.
- We use behaviours of functionality $\mathbf{exit}\,\mathit{bool}$ as selection predicates.

Exception handling is based on generalized termination, and allows a behaviour to terminate either with the δ gate, or by any other gate, for example a process which traps a division-by-zero exception is:

$$\begin{array}{l}
G?X:\mathit{int}; G?Y:\mathit{int}; \mathbf{exit}\langle X/Y\rangle \ggg \\
\quad \mathbf{accept}\langle Z:\mathit{int}\rangle \rightarrow H!Z; \mathbf{stop} \\
\quad \mathbf{trap}\,\mathit{Div}\langle\rangle \rightarrow H!0; \mathbf{stop}
\end{array}$$

Gate renaming has always been available in LOTOS, but only through the ‘back door’ of process definition. Here, we make it an explicit operator, and also allow simple data transformations to be made. For example a field of a gate can be hidden with:

$$\mathbf{rename}\,G\langle a:\mathit{int}, b:\mathit{float}\rangle := G\langle a\rangle\,\mathbf{in}\,B$$

or two fields can be swapped with:

$$\mathbf{rename}\,G\langle a:\mathit{int}, b:\mathit{int}\rangle := G\langle b, a\rangle\,\mathbf{in}\,B$$

The syntax for behaviours is:

$$\begin{array}{lcl}
B & ::= & \mathbf{exit}\,P \\
 & | & \mathbf{i}; B \\
 & | & GP[B]; B \\
 & | & Q[G^*](K) \\
 & | & B\,|[G^*]|\,B \\
 & | & B \,\|\, B \\
 & | & B \,\Box\, B \\
 & | & \mathbf{stop} \\
 & | & \mathbf{hide}\,G : \mathbf{gate}\,T\,\mathbf{in}\,B \\
 & | & \mathbf{case}\,K\,\mathbf{of}\,M \\
 & | & B \ggg \mathbf{accept}\,M\,(\mathbf{trap}\,G\,M)^* \\
 & | & \mathbf{rename}\,(G(P) := G(K))^*\,\mathbf{in}\,B
\end{array}$$

The static semantics is given by judgements of the form:

$$\mathcal{C} \vdash B \Rightarrow \mathbf{exit}\, T$$

The dynamics semantics is given by judgements of the form:

$$\mathcal{E} \vdash B \xrightarrow{\alpha(K)} B'$$

where α ranges over actions:

$$a ::= G \mid \mathbf{i} \qquad \alpha ::= a \mid \delta$$

2.7.1 Termination

Syntax:

$$\mathbf{exit}\, P$$

Static semantics:

$$\frac{\mathcal{C} \vdash P \Rightarrow (T \rightarrow ())}{\mathcal{C} \vdash \mathbf{exit}\, P \Rightarrow \mathbf{exit}\, T}$$

Dynamic semantics:

$$\frac{\mathcal{E} \vdash (P \Rightarrow K) \Rightarrow ()}{\mathcal{E} \vdash \mathbf{exit}\, P \xrightarrow{\delta(K)} \mathbf{stop}}$$

2.7.2 Internal action prefix

Syntax:

$$\mathbf{i}; B$$

Static semantics:

$$\frac{\mathcal{C} \vdash B \Rightarrow \mathbf{exit}\, T}{\mathcal{C} \vdash \mathbf{i}; B \Rightarrow \mathbf{exit}\, T}$$

Dynamic semantics:

$$\frac{}{\mathcal{E} \vdash \mathbf{i}; B \xrightarrow{\mathbf{i}\langle\rangle} B}$$

2.7.3 Action prefix

Syntax:

$$GP[B];B$$

Static semantics:

$$\frac{\begin{array}{r}\mathcal{C} \vdash G \Rightarrow \mathbf{gate}\, T'' \\ \mathcal{C} \vdash P \Rightarrow (T' \rightarrow \mathcal{C}') \\ \mathcal{C};\mathcal{C}' \vdash B_1 \Rightarrow \mathbf{exit}\, bool \\ \mathcal{C};\mathcal{C}' \vdash B_2 \Rightarrow \mathbf{exit}\, T\end{array}}{\mathcal{C} \vdash GP[B_1];B_2 \Rightarrow \mathbf{exit}\, T}\,[T' \sqsubseteq T'']$$

Dynamic semantics:

$$\frac{\begin{array}{l}\mathcal{E} \vdash (P \Rightarrow K) \Rightarrow \sigma \\ \mathcal{E} \vdash B_1[\sigma] \xrightarrow{\delta true} B_1'\end{array}}{\mathcal{E} \vdash GP[B_1];B_2 \xrightarrow{G(K)} B_2[\sigma]}$$

2.7.4 Process instantiation

Syntax:

$$Q[G^*](K)$$

Static semantics:

$$\frac{\begin{array}{l}\mathcal{C} \vdash Q \Rightarrow [\mathbf{gate}\, \vec{T}] \rightarrow T \rightarrow \mathbf{exit}\, T' \\ \mathcal{C} \vdash \vec{G} \Rightarrow \mathbf{gate}\, \vec{T} \\ \mathcal{C} \vdash K \Rightarrow T\end{array}}{\mathcal{C} \vdash Q[\vec{G}](K) \Rightarrow \mathbf{exit}\, T'}$$

Dynamic semantics:

$$\frac{\begin{array}{l}\mathcal{E} \vdash Q \Rightarrow \lambda[\vec{G}]M \\ \mathcal{E} \vdash \mathbf{rename}\, \vec{G} := \vec{G}'\, \mathbf{in}\, \mathbf{case}\, K\, \mathbf{of}\, M \xrightarrow{\alpha(K')} B'\end{array}}{\mathcal{E} \vdash Q[\vec{G}'](K) \xrightarrow{\alpha(K')} B'}$$

2.7.5 Parameterized concurrency

Syntax:

$$B\,|[G^*]|\,B$$

Static semantics:

$$\frac{\begin{array}{l}\mathcal{C} \vdash B_1 \Rightarrow \mathbf{exit}\, T_1 \\ \mathcal{C} \vdash B_2 \Rightarrow \mathbf{exit}\, T_2 \\ \mathcal{C} \vdash \vec{G} \Rightarrow \mathbf{gate}\, \vec{T}\end{array}}{\mathcal{C} \vdash \Rightarrow B_1 \,|[\vec{G}]|\, B_2 \Rightarrow \mathbf{exit}\, T_1 \sqcap T_2}$$

Dynamic semantics:

$$\frac{\begin{array}{l}\mathcal{E} \vdash B_1 \xrightarrow{G(K)} B_1' \\ \mathcal{E} \vdash B_2 \xrightarrow{G(K)} B_2'\end{array}}{\mathcal{E} \vdash B_1 \,|[\vec{G}]|\, B_2 \xrightarrow{G(K)} B_1' \,|[\vec{G}]|\, B_2'}\,[G \in \vec{G}]$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{a(K)} B_1'}{\mathcal{E} \vdash B_1 \,|[\vec{G}]|\, B_2 \xrightarrow{a(K)} B_1' \,|[\vec{G}]|\, B_2}\,[a \notin \vec{G}]$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{a(K)} B_2'}{\mathcal{E} \vdash B_1 \,|[\vec{G}]|\, B_2 \xrightarrow{a(K)} B_1 \,|[\vec{G}]|\, B_2'}\,[a \notin \vec{G}]$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash B_1 \xrightarrow{\delta(K)} B_1' \\ \mathcal{E} \vdash B_2 \xrightarrow{\delta(K)} B_2'\end{array}}{\mathcal{E} \vdash B_1 \,|[\vec{G}]|\, B_2 \xrightarrow{\delta(K)} B_1' \,|[\vec{G}]|\, B_2'}$$

2.7.6 Synchronized concurrency

Syntax:

$$B \,||\, B$$

Static semantics:

$$\frac{\begin{array}{l}\mathcal{C} \vdash B_1 \Rightarrow \mathbf{exit}\, T_1 \\ \mathcal{C} \vdash B_2 \Rightarrow \mathbf{exit}\, T_2\end{array}}{\mathcal{C} \vdash B_1 \,||\, B_2 \Rightarrow \mathbf{exit}\, T_1 \sqcap T_2}$$

Dynamic semantics:

$$\frac{\begin{array}{l}\mathcal{E} \vdash B_1 \xrightarrow{G(K)} B_1' \\ \mathcal{E} \vdash B_2 \xrightarrow{G(K)} B_2'\end{array}}{\mathcal{E} \vdash B_1 \,||\, B_2 \xrightarrow{G(K)} B_1' \,||\, B_2'}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\mathbf{i}\langle\rangle} B_1'}{\mathcal{E} \vdash B_1 \parallel B_2 \xrightarrow{\mathbf{i}\langle\rangle} B_1' \parallel B_2}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{\mathbf{i}\langle\rangle} B_2'}{\mathcal{E} \vdash B_1 \parallel B_2 \xrightarrow{\mathbf{i}\langle\rangle} B_1 \parallel B_2'}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash B_1 \xrightarrow{\delta(K)} B_1' \\ \mathcal{E} \vdash B_2 \xrightarrow{\delta(K)} B_2'\end{array}}{\mathcal{E} \vdash B_1 \parallel B_2 \xrightarrow{\delta(K)} B_1' \parallel B_2'}$$

2.7.7 Choice

Syntax:

$$B \Box B$$

Static semantics:

$$\frac{\begin{array}{l}\mathcal{C} \vdash B_1 \Rightarrow \mathbf{exit}\, T_1 \\ \mathcal{C} \vdash B_2 \Rightarrow \mathbf{exit}\, T_2\end{array}}{\mathcal{C} \vdash B_1 \Box B_2 \Rightarrow \mathbf{exit}\, T_1 \sqcup T_2}$$

Dynamic semantics:

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\alpha(K)} B_1'}{\mathcal{E} \vdash B_1 \Box B_2 \xrightarrow{\alpha(K)} B_1'}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{\alpha(K)} B_2'}{\mathcal{E} \vdash B_1 \Box B_2 \xrightarrow{\alpha(K)} B_2'}$$

2.7.8 Deadlock

Syntax:

stop

Static semantics:

$$\frac{}{\mathcal{C} \vdash \mathbf{stop} \Rightarrow \mathbf{exit}\, \mathbf{none}}$$

No dynamic semantics rules are needed.

2.7.9 Gate hiding

Syntax:

$$\mathbf{hide}\,G : \mathbf{gate}\,T\,\mathbf{in}\,B$$

Static semantics:

$$\frac{\begin{array}{c}\mathcal{C} \vdash T \Rightarrow \mathbf{type}\\ \mathcal{C};G \Rightarrow \mathbf{gate}\,T \vdash B \Rightarrow \mathbf{exit}\,T'\end{array}}{\mathcal{C} \vdash \mathbf{hide}\,G : \mathbf{gate}\,T\,\mathbf{in}\,B \Rightarrow \mathbf{exit}\,T'}$$

Dynamic semantics:

$$\frac{\mathcal{E} \vdash B \xrightarrow{G(K)} B'}{\mathcal{E} \vdash \mathbf{hide}\,G : \mathbf{gate}\,T\,\mathbf{in}\,B \xrightarrow{\mathbf{i}()} \mathbf{hide}\,G : \mathbf{gate}\,T\,\mathbf{in}\,B'}$$

$$\frac{\mathcal{E} \vdash B \xrightarrow{\alpha(K)} B'}{\mathcal{E} \vdash \mathbf{hide}\,G : \mathbf{gate}\,T\,\mathbf{in}\,B \xrightarrow{\alpha(K)} \mathbf{hide}\,G : \mathbf{gate}\,T\,\mathbf{in}\,B'}\,[\alpha \neq G]$$

2.7.10 Case

Syntax:

$$\mathbf{case}\,K\,\mathbf{of}\,M$$

Static semantics:

$$\frac{\begin{array}{l}\mathcal{C} \vdash K \Rightarrow T''\\ \mathcal{C} \vdash M \Rightarrow (T \rightarrow \mathbf{exit}\,T')\end{array}}{\mathcal{C} \vdash \mathbf{case}\,K\,\mathbf{of}\,M \Rightarrow \mathbf{exit}\,T'}\,[T'' \sqsubseteq T]$$

Dynamic semantics:

$$\frac{\mathcal{E} \vdash (M \Rightarrow K) \xrightarrow{\alpha(K')} B}{\mathcal{E} \vdash \mathbf{case}\,K\,\mathbf{of}\,M \xrightarrow{\alpha(K')} B}$$

2.7.11 Generalized enabling

Syntax:

$$B >\!\!>\!\!> \mathbf{accept}\,M\,(\mathbf{trap}\,G\,M)^*$$

Static semantics:

$$\frac{\begin{array}{c}\mathcal{C} \vdash M \Rightarrow (T \rightarrow \mathbf{exit}\,T')\\ \mathcal{C} \vdash \vec{M} \Rightarrow (\vec{T} \rightarrow \mathbf{exit}\,T')\\ \mathcal{C};\vec{G} \Rightarrow \mathbf{gate}\,\vec{T} \vdash B \Rightarrow \mathbf{exit}\,T''\end{array}}{\mathcal{C} \vdash B >\!\!>\!\!> \mathbf{accept}\,M\,\mathbf{trap}\,\vec{G}\,\vec{M} \Rightarrow \mathbf{exit}\,T'}\,[T'' \sqsubseteq T]$$

Dynamic semantics:

$$\frac{\mathcal{E} \vdash B \xrightarrow{a(K)} B'}{\mathcal{E} \vdash B \ggg \mathbf{accept}\, M\, \mathbf{trap}\, \vec{G}\, \vec{M} \xrightarrow{a(K)} B \ggg \mathbf{accept}\, M\, \mathbf{trap}\, \vec{G}\, \vec{M}}\,[a \notin \vec{G}]$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash B \xrightarrow{\delta(K)} B' \\ \mathcal{E} \vdash (M \Rightarrow K) \xrightarrow{\alpha(K')} B''\end{array}}{\mathcal{E} \vdash B \ggg \mathbf{accept}\, M\, \mathbf{trap}\, \vec{G}\, \vec{M} \xrightarrow{\alpha(K')} B''}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash B \xrightarrow{G_i(K)} B' \\ \mathcal{E} \vdash (M_i \Rightarrow K) \xrightarrow{\alpha(K')} B''\end{array}}{\mathcal{E} \vdash B \ggg \mathbf{accept}\, M\, \mathbf{trap}\, \vec{G}\, \vec{M} \xrightarrow{\alpha(K')} B''}$$

2.7.12 Gate renaming

Syntax:

$$\mathbf{rename}\,(G(P) := G(K))^*\ \mathbf{in}\, B$$

Static semantics:

$$\frac{\begin{array}{r}\mathcal{C} \vdash \vec{G}' \Rightarrow \mathbf{gate}\, \vec{T} \\ \mathcal{C} \vdash (\vec{P} \Rightarrow \vec{T}) \Rightarrow \vec{\mathcal{C}} \\ \mathcal{C}; \vec{\mathcal{C}} \vdash \vec{K} \Rightarrow \vec{T}' \\ \mathcal{C}; \vec{G} \Rightarrow \mathbf{gate}\, \vec{T}' \vdash B \Rightarrow \mathbf{exit}\, T'\end{array}}{\mathcal{C} \vdash \mathbf{rename}\, \vec{G}(\vec{P}) := \vec{G}'(\vec{K})\, \mathbf{in}\, B \Rightarrow \mathbf{exit}\, T'}$$

Dynamic semantics:

$$\frac{\begin{array}{l}\mathcal{E} \vdash B \xrightarrow{G_i(K)} B' \\ \mathcal{E} \vdash (P_i \Rightarrow K) \Rightarrow \sigma\end{array}}{\mathcal{E} \vdash \mathbf{rename}\, \vec{G}(\vec{P}) := \vec{G}'(\vec{K})\, \mathbf{in}\, B \xrightarrow{G'_i(K_i[\sigma])} \mathbf{rename}\, \vec{G}(\vec{P}) := \vec{G}'(\vec{K})\, \mathbf{in}\, B'}$$

$$\frac{\mathcal{E} \vdash B \xrightarrow{\alpha(K)} B'}{\mathcal{E} \vdash \mathbf{rename}\, \vec{G}(\vec{P}) := \vec{G}'(\vec{K})\, \mathbf{in}\, B \xrightarrow{\alpha(K)} \mathbf{rename}\, \vec{G}(\vec{P}) := \vec{G}'(\vec{K})\, \mathbf{in}\, B'}\,[\alpha \notin \vec{G}]$$

3 Syntactic sugar

The core language has a fairly simple syntax and semantics, but needs more features before it is usable as a specification language. In this section we provide some syntactic sugar to make the core language more usable, and to bring it closer to the user level language.

3.1 Omitting type information

In the core language, all bound variables must be explicitly typed, for example:

$$
\begin{array}{l}
\mathbf{process}\,\mathit{Stack}\,[i : \mathbf{gate}\,\mathit{int}, o : \mathbf{gate}\,\mathit{int}]\,\langle l : \mathit{intlist}\rangle : \mathbf{exit}\,\mathbf{none}\,\mathbf{is}\\
\mathbf{case}\,l\,\mathbf{of}\\
\quad \mathit{nil}\langle\rangle \rightarrow\\
\qquad i?(x : \mathit{int}); \mathit{Stack}[i,o]\langle \mathit{cons}\langle x, l\rangle\rangle\\
\quad \mathit{cons}\langle y : \mathit{int}, \mathit{ys} : \mathit{intlist}\rangle \rightarrow\\
\qquad i?x : \mathit{int}; \mathit{Stack}[i,o]\langle \mathit{cons}\langle x, l\rangle\rangle \mathbin{\Box} o!y; \mathit{Stack}[i,o]\langle \mathit{ys}\rangle
\end{array}
$$

Many of these annotations are unnecessary, for example there is no need to annotate y and ys since their type can be deduced from the type of l. Type annotations can be omitted from pattern P when:

- P occurs in a communication $G(P)[B_1];B_2$ where the type of G determines the type of P, or when
- P is used in a pattern match $P \rightarrow B$, since the surrounding context will give the type of P.

For example we could give the above process as:

$$
\begin{array}{l}
\mathbf{process}\,\mathit{Stack}\,[i : \mathbf{gate}\,\mathit{int}, o : \mathbf{gate}\,\mathit{int}]\,\langle l : \mathit{intlist}\rangle : \mathbf{exit}\,\mathbf{none}\,\mathbf{is}\\
\mathbf{case}\,l\,\mathbf{of}\\
\quad \mathit{nil}\langle\rangle \rightarrow\\
\qquad i?x; \mathit{Stack}[i,o]\langle \mathit{cons}\langle x, l\rangle\rangle\\
\quad \mathit{cons}\langle y, \mathit{ys}\rangle \rightarrow\\
\qquad i?x; \mathit{Stack}[i,o]\langle \mathit{cons}\langle x, l\rangle\rangle \mathbin{\Box} o!y; \mathit{Stack}[i,o]\langle \mathit{ys}\rangle
\end{array}
$$

By default gates have type $\mathbf{gate}\,\langle\,\_\,\rangle$.

3.2 Pattern shorthand

We can use variables as patterns:

$$P ::= \cdots \mid ?V : T$$

by defining the shorthand:

$$?V : T \;\stackrel{\mathsf{def}}{=}\; ?V\,\mathbf{as}\,\mathbf{any} : T$$

For example:

$$i?x : \mathit{int}; o!x; \mathbf{stop}$$

is shorthand for:

$$i(?x\,\mathbf{as}\,\mathbf{any} : \mathit{int}); o!x; \mathbf{stop}$$

3.3 Process declaration

Not all process declarations are case-statements, so we allow process declarations to have behaviours as bodies (not just pattern-matches):

$$D ::= \cdots \mid \textbf{process}\, Q\, [(G : \textbf{gate}\, T)^*]\, \langle (V : T)^*\, [\_] \rangle : \textbf{exit}\, T\, \textbf{is}\, B$$

This can be expanded as:

$$\begin{array}{l} \textbf{process}\, Q\, [\vec{G} : \textbf{gate}\, \vec{T}]\, \langle \vec{V} : \vec{T}\, [\_] \rangle : \textbf{exit}\, T\, \textbf{is}\, B \\ \quad \overset{\text{def}}{=} \quad \textbf{process}\, Q\, [\vec{G} : \textbf{gate}\, \vec{T}]\, \langle \vec{V} : \vec{T}\, [\_] \rangle : \textbf{exit}\, T\, \textbf{is}\, \langle \vec{V} := ?\vec{V} : \vec{T}\, [\_] \rangle \rightarrow B \end{array}$$

For example:

$$\textbf{process}\, Q\, [G]\, \langle x : int\, y : int \rangle : \textbf{exit}\, int\, \textbf{is}\, G!x; \textbf{exit}(!y)$$

is shorthand for:

$$\textbf{process}\, Q\, [G]\, \langle x : int\, y : int \rangle : \textbf{exit}\, int\, \textbf{is}\, \langle x := ?x\, y := ?y \rangle \rightarrow G!x; \textbf{exit}(!y)$$

3.4 Expressions

In the core language, there is no non-trivial computation of expressions, for example only constants are allowed in process instantiation, output and **exit**. This is obviously impractical, and we need to extend the language with a syntax for expressions, which may have non-trivial behaviour (for example failing to terminate or raising an exception).

Introduce new syntactic terminals F (function identifiers) and X (exceptions) and non-terminals E (expressions) and EM (expression matches):

$$\begin{array}{rcl} E & ::= & R \\ & \mid & V \\ & \mid & \langle (V := E)^*\, [\_] \rangle \\ & \mid & C(E) \\ & \mid & F[X^*](E) \\ & \mid & \textbf{raise} X(E) \\ & \mid & \textbf{case}\, E\, \textbf{of}\, EM\, (\textbf{trap}\, X\, EM)^* \\ EM & ::= & P[E] \rightarrow E(\mid P[E] \rightarrow E)^* \\ D & ::= & \cdots \mid \textbf{function}\, F\, [(X : \textbf{exception}\, T)^*]\, T : T\, \textbf{is}\, EM \end{array}$$

We translate an expression of type T to a behaviour of type **exit** T, and an exception of type T to a gate of type T (using fresh variables Y_i). This coding is based on Moggi's translation of the call-by-value λ-calculus into the monadic metalanguage [4]:

$$\begin{array}{rcl} R & \overset{\text{def}}{=} & \textbf{exit}(R) \\ V & \overset{\text{def}}{=} & \textbf{exit}(!V) \end{array}$$

$$
\begin{aligned}
\langle \vec{V} := \vec{E}\,[\,\_\,]\rangle &\stackrel{\text{def}}{=} E_1 \ggg \mathbf{accept}\,?Y_1 \rightarrow \cdots E_n \ggg \mathbf{accept}\,?Y_n \rightarrow \\
&\qquad\quad \mathbf{exit}\,!\langle V_1 := Y_1 \ldots V_n := Y_n\,[\,\_\,]\rangle \\
C(E) &\stackrel{\text{def}}{=} E \ggg \mathbf{accept}\,?Y \rightarrow \mathbf{exit}\,(!C(Y)) \\
F[\vec{X}](E) &\stackrel{\text{def}}{=} E \ggg \mathbf{accept}\,?Y \rightarrow F[\vec{X}](Y) \\
\mathbf{raise}\,X(E) &\stackrel{\text{def}}{=} E \ggg \mathbf{accept}\,?Y \rightarrow X!Y; \mathbf{stop} \\
\mathbf{case}\,E\;\mathbf{of}\,EM\,\mathbf{trap}\,\vec{X}\,\vec{EM} &\stackrel{\text{def}}{=} E \ggg \mathbf{accept}\,EM\,\mathbf{trap}\,\vec{X}\,\vec{EM}
\end{aligned}
$$

An expression match of type $T \rightarrow T'$ is translated to a behaviour match of type $T \rightarrow \mathbf{exit}\,T'$:

$$
P_1[E_1] \rightarrow E_1' \mid \cdots \mid P_n[E_n] \rightarrow E_n' \stackrel{\text{def}}{=} P_1[E_1] \rightarrow E_1' \mid \cdots \mid P_n[E_n] \rightarrow E_n'
$$

A function declaration of type $[\mathbf{exception}\,\vec{T}] \rightarrow T \rightarrow T'$ is translated to a process declaration of type $[\mathbf{gate}\,\vec{T}] \rightarrow T \rightarrow \mathbf{exit}\,T'$:

$$
\mathbf{function}\,F\,[\vec{X} : \mathbf{exception}\,\vec{T}]\,T : T'\,\mathbf{is}\,EM \stackrel{\text{def}}{=} \mathbf{process}\,F\,[\vec{X} : \mathbf{gate}\,\vec{T}]\,T : \mathbf{exit}\,T'\,\mathbf{is}\,EM
$$

3.5 Expressions in behaviours

Once the data language has been extended with expressions, the behaviour language should be similarly extended:

$$
\begin{array}{rcl}
B & ::= & \cdots \\
 & \mid & \mathbf{exit}\,(O) \\
 & \mid & GO[E];B \\
 & \mid & Q[G^*](E) \\
 & \mid & \mathbf{case}\,E\;\mathbf{of}\,M\,(\mathbf{trap}\,G\,M)^* \\
 & \mid & \mathbf{raise}\,X(O) \\
O & ::= & \langle (V := (!E \mid P))^*\,[\,\_\,]\rangle
\end{array}
$$

These all use similar translations as for expressions, for example:

$$
\begin{aligned}
G\langle \cdots V_1 := P\,V_2 := !E \cdots\rangle[E];B &\stackrel{\text{def}}{=} G\langle \cdots V_2 := !E\,V_1 := P \cdots\rangle[E];B \\
G\langle \vec{V} := !\vec{E}\,\vec{V}' := \vec{P}\,[\,\_\,]\rangle[E];B &\stackrel{\text{def}}{=} \langle \vec{V} := \vec{E}\rangle \ggg \mathbf{accept}\,\langle \vec{V} := ?\vec{V}\rangle \rightarrow \\
&\qquad\quad G\langle \vec{V} := !\vec{V}\,\vec{V}' := \vec{P}\,[\,\_\,]\rangle[E];B
\end{aligned}
$$

The keyword ‘**exception**’ can be treated as synonymous with ‘**gate**’.

3.6 Tuples

The core syntax only supports records with named fields, not nameless tuples where fields are identified positionally. We can extend the syntax of types with tuples:

$$
T \;::=\; \cdots \mid \langle T(*\,T)^*[\,*\,\_\,]\rangle
$$

Following the style of ML, tuples can be made synonymous with records with numbered fields:

$$\langle T_1 * \cdots * T_n [* \_] \rangle \quad \overset{\mathsf{def}}{=} \quad \langle \$1 : T_1 \cdots \$n : T_n [, \_] \rangle$$

We can also provide syntactic sugar for tuple patterns and constants:

$$\begin{array}{lcl} P & ::= & \cdots \mid \langle P (,P)^* [, \_] \rangle \\ K & ::= & \cdots \mid \langle K (,K)^* \rangle \\ E & ::= & \cdots \mid \langle E (,E)^* \rangle \end{array}$$

This is translated into the core language by providing the appropriate field names, for example the syntactic sugar:

$$\mathbf{hide}\, G : \mathbf{gate}\, \langle \mathit{int} * \_ \rangle\, \mathbf{in}\, G \langle 1, \mathit{true} \rangle ; \mathbf{exit}\, \langle \rangle$$

is expanded to:

$$\mathbf{hide}\, G : \mathbf{gate}\, \langle \$1 : \mathit{int}\, \_ \rangle\, \mathbf{in}\, G \langle \$1 := 1\, \$2 := \mathit{true} \rangle ; \mathbf{exit}\, \langle \rangle$$

This expansion is given by the rule:

$$\begin{array}{l} \mathcal{C} \vdash G \langle P_1, \ldots, P_m, P_{m+1}, \ldots, P_n [, \_] \rangle ; B \\ \quad \overset{\mathsf{def}}{=} \quad G \langle V_1 := P_1 \cdots V_m := P_m \, \$(m+1) := P_{m+1} \cdots \$n := P_n [\_] \rangle ; B \\ \qquad\qquad \text{where } \mathcal{C} \vdash G \Rightarrow \mathbf{gate}\, \langle V_1 : T_1 \cdots V_m : T_m [\_] \rangle \end{array}$$

and similarly for constructor application, case expressions, enabling, process definitions, process instantiation and gate renaming. In each case we have a unique translation, *except* in the case of **exit**, since the context does not provide a unique type for the δ gate, so we will use the translation:

$$\mathbf{exit}\, \langle P_1, \ldots, P_n [, \_] \rangle \quad \overset{\mathsf{def}}{=} \quad \mathbf{exit}\, \langle \$1 := P_1 \cdots \$n := P_n [\_] \rangle$$

Note that these translations require some static semantic information to determine the appropriate field names.

This expansion can be extended to any occurrence of record syntax in expressions or behaviours.

3.7 Infix operators

Once we have a syntax for tuples, we can define infix operators as syntax sugar for function application. We extend the data language with:

$$\begin{array}{lcl} E & ::= & \cdots \mid E\, C\, E \mid E\, F\, E \\ K & ::= & \cdots \mid K\, C\, K \\ P & ::= & \cdots \mid P\, C\, P \end{array}$$

and define translations:

$$\begin{aligned} E_1\, C\, E_2 &\overset{\text{def}}{=} C\langle E_1, E_2\rangle \\ E_1\, F\, E_2 &\overset{\text{def}}{=} F\langle E_1, E_2\rangle \\ K_1\, C\, K_2 &\overset{\text{def}}{=} C\langle K_1, K_2\rangle \\ P_1\, C\, P_2 &\overset{\text{def}}{=} C\langle P_1, P_2\rangle \end{aligned}$$

We leave issues of parsing and priority of infix operators for further work.

3.8 Boolean expressions

We can define a language of boolean expressions from case statements:

$$\begin{array}{lll} E & ::= & \cdots \\ & | & \mathbf{if}\, E_1\, \mathbf{then}\, E_2\, \mathbf{else}\, E_3 \\ & | & \mathbf{if}\, E_1\, \mathbf{then}\, B_2\, \mathbf{else}\, B_3 \\ & | & E_1\ \mathbf{andalso}\, E_2 \\ & | & E_1\ \mathbf{orelse}\, E_2 \\ & | & \mathbf{not}\, E \\ & | & E_1 = E_2 \\ & | & E_1 \neq E_2 \end{array}$$

with the expansions:

$$\begin{aligned} \mathbf{if}\, E_1\, \mathbf{then}\, E_2\, \mathbf{else}\, E_3 &\overset{\text{def}}{=} \mathbf{case}\, E_1\, \mathbf{of}\, \mathit{true} \to E_2 \mid \mathit{false} \to E_3 \\ \mathbf{if}\, E_1\, \mathbf{then}\, B_2\, \mathbf{else}\, B_3 &\overset{\text{def}}{=} \mathbf{case}\, E_1\, \mathbf{of}\, \mathit{true} \to B_2 \mid \mathit{false} \to B_3 \\ E_1\ \mathbf{andalso}\, E_2 &\overset{\text{def}}{=} \mathbf{if}\, E_1\, \mathbf{then}\, E_2\, \mathbf{else}\, \mathit{false} \\ E_1\ \mathbf{orelse}\, E_2 &\overset{\text{def}}{=} \mathbf{if}\, E_1\, \mathbf{then}\, \mathit{true}\, \mathbf{else}\, E_2 \\ \mathbf{not}\, E &\overset{\text{def}}{=} \mathbf{if}\, E\, \mathbf{then}\, \mathit{false}\, \mathbf{else}\, \mathit{true} \\ E_1 = E_2 &\overset{\text{def}}{=} \mathbf{case}\, E_1\, \mathbf{of}\, ?Y \to \mathbf{case}\, E_2\, \mathbf{of}\, !Y \to \mathit{true} \mid \mathbf{any} \to \mathit{false} \\ E_1 \neq E_2 &\overset{\text{def}}{=} \mathbf{not}(E_1 = E_2) \end{aligned}$$

3.9 Gate renaming

If we do not specify any change of data representation in a gate renaming, then by default none happens. We extend the syntax of renaming to make the pattern-matching optional:

$$\begin{array}{lll} B & ::= & \cdots \\ & | & \mathbf{rename}\, (G(P) := G(K) \mid G := G)^*\, \mathbf{in}\, B \end{array}$$

then make the default behaviour to do no change of representation:

$$\mathbf{rename} \cdots G := G' \cdots \mathbf{in}\, B \overset{\text{def}}{=} \mathbf{rename} \cdots G(?X) := G'(!X) \cdots \mathbf{in}\, B$$

3.10 Compatibility with existing specifications

For compatibility with existing specifications, we make the following syntactic sugar:

$$\begin{aligned}
\mathbf{noexit} &\stackrel{\text{def}}{=} \mathbf{exit}\,\mathbf{none} \\
GP;B &\stackrel{\text{def}}{=} GP[\mathit{true}];B \\
GO_1\cdots O_n[E];B &\stackrel{\text{def}}{=} G\langle O_1,\ldots,O_n\rangle[E];B \\
B_1\,|||\,B_2 &\stackrel{\text{def}}{=} B_1\,|[]|\,B_2 \\
B_1\,[>B_2 &\stackrel{\text{def}}{=} (B_1\,|||\,(\mathbf{raise}X\langle\rangle \,\Box\, \mathbf{exit}\,\mathbf{any}))\ggg \\
&\qquad\quad \mathbf{accept}\,?V \to \mathbf{exit}\,(!V) \\
&\qquad\quad \mathbf{trap}\,X\,\langle\rangle \to B_2 \\
\mathbf{let}\,\vec{V}=\vec{E}\,\mathbf{in}\,B &\stackrel{\text{def}}{=} \mathbf{case}\,\langle\vec{E}\rangle\,\mathbf{of}\,\langle ?\vec{V}\rangle \to B \\
\mathbf{choice}\,P \,\Box\, B &\stackrel{\text{def}}{=} \mathbf{exit}\,\mathbf{any} \ggg \mathbf{accept}\,P \to B \\
B \gg \mathbf{accept}\,M &\stackrel{\text{def}}{=} B \ggg \mathbf{accept}\,?V \to \mathbf{i};\mathbf{exit}\,!V \ggg \mathbf{accept}\,M
\end{aligned}$$

For compatibility with existing specifications, ! is optional from patterns in **exit**, ! and ? are mandatory in communications, and ? is optional from patterns in all other contexts.

3.11 Record calculation

We can provide syntactic sugar for record evaluation which gives some of the flavour of imperative programming to LOTOS. For example the behaviour:

$$x := 0;;y := x+1$$

can be defined to be bisimilar to:

$$\mathbf{exit}\,\langle x := 0\, y := 1\,\_\rangle$$

We extend the behaviour language with:

$$\begin{aligned}
B \quad ::= \quad & \cdots \\
\mid \quad & V := E \\
\mid \quad & B;;B \\
\mid \quad & G(P)[E]
\end{aligned}$$

and translate them into the core language as:

$$\begin{aligned}
\mathcal{C} \vdash V := E &\stackrel{\text{def}}{=} \mathbf{exit}\,\langle V := E\,\_\rangle \\
\mathcal{C} \vdash B_1;;B_2 &\stackrel{\text{def}}{=} B_1 \ggg \mathbf{accept}\,\langle \vec{V}_1 := ?\vec{V}_1\,\_\rangle \to \\
&\qquad B_2 \ggg \mathbf{accept}\,\langle \vec{V}_2 := ?\vec{V}_2\,\_\rangle \to \\
&\qquad \mathbf{exit}\,\langle \vec{V}_1 := !\vec{V}_1 \vec{V}_2 := !\vec{V}_2\,\_\rangle \\
\mathcal{C} \vdash G(P)[E] &\stackrel{\text{def}}{=} G(P)[E];\mathbf{exit}\,\langle \vec{V} := !\vec{V}\,\_\rangle
\end{aligned}$$

where $\mathcal{C} \vdash B_i \Rightarrow \mathbf{exit}\langle \vec{V}_i : T_i\,\_\rangle$ and $\mathcal{C} \vdash P \Rightarrow (T \rightarrow (\vec{V} \Rightarrow \vec{T}))$. Note that this requires static semantic information.

We can extend the data language similarly.

These extensions allow for an imperative flavour of specification, for example:

$$(G?x \,|||\, G?y);;z := (x+y)$$

is bisimilar to:

$$\begin{array}{l} (G?x;\mathbf{exit}\langle x :=!x\,\_\rangle \,|||\, G?y;\mathbf{exit}\langle y :=!y\,\_\rangle) >\!\!>\!\!> \\ \quad \mathbf{accept}\langle x :=?x\, y :=?y\,\_\rangle \rightarrow \mathbf{exit}\langle x :=!x\, y :=!y\, z :=!(x+y)\,\_\rangle \end{array}$$

Using Moggi's equations for monadic let, we can show that $;;$ is associative with $\mathbf{exit}\langle\,\_\rangle$ as a unit.

3.12 Out parameters

To make it simpler to interface with other languages, it would be useful to provide **out** parameters as well as **in** parameters to processes:

$$\begin{array}{rcl} D & ::= & \mathbf{process}\, Q\,[(G[: \mathbf{gate}\, T])^*]\,\langle((\mathbf{in} \mid \mathbf{out})V : T)^*\rangle\, \mathbf{is}\, B \\ B & ::= & Q[G^*]\langle(?V \mid !E)^*\rangle \end{array}$$

where declarations are expanded:

$$\begin{array}{l} \mathbf{process}\, Q\,[\vec{G}]\,\langle\cdots \mathbf{out}\, V_1 : T_1\, \mathbf{in}\, V_2 : T_2 \cdots\rangle\, \mathbf{is}\, B \\ \quad \overset{\text{def}}{=} \quad \mathbf{process}\, Q\,[\vec{G}]\,\langle\cdots \mathbf{in}\, V_2 : T_2\, \mathbf{out}\, V_1 : T_1 \cdots\rangle\, \mathbf{is}\, B \\ \mathbf{process}\, Q\,[\vec{G}]\,\langle \mathbf{in}\, \vec{V}_1 : \vec{T}_1\, \mathbf{out}\, \vec{V}_2 : \vec{T}_2\rangle\, \mathbf{is}\, B \\ \quad \overset{\text{def}}{=} \quad \mathbf{process}\, Q\,[\vec{G}]\,\langle \vec{V}_1 : \vec{T}_1\rangle : \mathbf{exit}\langle \vec{V}_2 : \vec{T}_2\,\_\rangle\, \mathbf{is}\, B \end{array}$$

and process use is expanded:

$$\begin{array}{l} Q[\vec{G}]\langle\cdots ?V\, !E \cdots\rangle \\ \quad \overset{\text{def}}{=} \quad Q[\vec{G}]\langle\cdots !E\, ?P \cdots\rangle \\ Q[\vec{G}]\langle !\vec{E}\, ?\vec{V}\rangle \\ \quad \overset{\text{def}}{=} \quad Q[\vec{G}]\langle \vec{E}\rangle >\!\!>\!\!> \mathbf{accept}\langle ?\vec{V}\rangle \rightarrow \mathbf{exit}\langle \vec{V} :=!\vec{V}\,\_\rangle \end{array}$$

The default parameter style is **in**.

We can define similar sugar for expressions.

References

[1] Alan Jeffrey. Semantics for a fragment of LOTOS with functional data and abstract datatypes. In *Revised Working Draft on Enhancements to LOTOS (v3)*, ISO/IEC JTC1/SC21/WG7, chapter Annexe A. 1995.

[2] Alan Jeffrey, Hubert Garavel, Guy Leduc, Charles Pecheur, and Mihaela Sighireanu. Towards a proposal for datatypes in E-LOTOS. In *Revised Working Draft on Enhancements to LOTOS (v2)*, ISO/IEC JTC1/SC21/WG7 N1053, chapter Annexe A. 1995.

[3] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[4] Eugenio Moggi. Notions of computation and monad. *Inform. and Comput.*, 93:55–92, 1991.