Semantics for a fragment of LOTOS with functional data and abstract datatypes

Alan Jeffrey, University of Sussex

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

Liège meeting, December 1995

Abstract

This paper presents static and dynamic semantics for a fragment of LOTOS with a functional (rather than algebraic) data language. We present a 'core' functional data language (which is explicitly and monomorphically typed), and show how it can be integrated into the LOTOS behavioural model. We then introduce data abstraction, and discover that data abstraction fits badly with the LOTOS communication model, which assumes that the data space is 'flat'. We propose three possible solutions: 1) accepting the loss of data abstraction, 2) banning abstract types from communication, or 3) allowing the specifier to determine the behaviour of each type in output synchronization and input hiding. We then show that both of the last two options allow gates to be treated as first-class values (ending the distinction between gates and data), and the 3rd has the expressive power of the π-calculus.

Contents

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 [2], but does *not* consider many of the 'difficult' features: polymorphism, overloading, exceptions, modularity, and subtyping are *not* discussed in this document. They are left for future development.

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' functional language based on [2]. This is given a static and dynamic semantics in the style of the SML formal language definition [5].
- In Section 3 we show how this core language can be incorporated into the LOTOS behavioral model. We give a static and dynamic semantics to a fragment of LOTOS with ACT ONE expressions replaced by expressions from the core functional language. The resulting semantics is much simpler than that of the ISO standard.
- In Section 4 we introduce a simple form of data abstraction, and discuss the problems that this poses. In particular, the LOTOS output synchronization and input hiding models make it difficult to work with abstract datatypes, since they assume a 'flat' data model. We present three possible solutions to this problem:
  1. Keep the existing semantics, and accept the loss of data abstraction.
  2. Ban abstract datatypes from communications (and the other contexts such as **choice** which cause problems).
  3. Allow specifiers to define how each abstract datatype behaves in an output synchronization or an input hiding.
- In Section 5 we show how LOTOS gates can be viewed as an abstract datatype. This simplifies the behavioural model considerably, since we do not need separate gate- and data-parameterization and -instantiation for processes. We show how two of the possible solutions to abstract datatypes produce different models of gates:
  1. One model has gates as 'static' entities which cannot be communicated. The resulting language is more powerful than existing LOTOS, but does not have the expressive power of Milner's π-calculus.

2. The other model has gates as ‘mobile’ entities which can be communicated between processes. The resulting language is as powerful as the π-calculus.

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

2 Data language

2.1 Syntax

The fragment of the functional language we consider here is monomorphic, explicitly typed, and allows anonymous records. Future extensions of this language should investigate implicit typing, modularization, and polymorphism or overloading.

The terminals of the abstract syntax are:

symbol domain	*meaning*	*abbreviations*
Var	variable identifiers	x
Fun	operation identifiers	f
Con	constructor identifiers	c
Srt	sort identifiers	S
Lab	field identifiers	l

The *type expressions* are:

$$\begin{array}{rcl} T & ::= & S \\ & \mid & \{l_1 : T_1, \ldots, l_n : T_n\} \end{array}$$

The *declarations* are:

$$\begin{array}{rcl} D & ::= & \textbf{datatype}\ S := c_1\ \textbf{of}\ T_1 \mid \cdots \mid c_n\ \textbf{of}\ T_n\ \textbf{endtype} \\ & \mid & \textbf{function}\ f\, p : T := e\ \textbf{endfun} \\ & \mid & \\ & \mid & DD \end{array}$$

The *expressions* are:

$$\begin{array}{rcl} e & ::= & x \\ & \mid & \textbf{error}\ T \\ & \mid & \textbf{case}\ e\ \textbf{of}\ p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n\ \textbf{endcase} \\ & \mid & ce \\ & \mid & fe \\ & \mid & \{l_1 = e_1, \ldots, l_n = e_n\} \\ & \mid & e.\{l = e\} \end{array}$$

The *patterns* are:

$$\begin{array}{rcl} p & ::= & x : T \\ & \mid & \textbf{any}\ T \\ & \mid & cp \\ & \mid & \{l_1 = p_1, \ldots, l_n = p_n\} \end{array}$$

2.2 Static semantics

The static semantics is given by judgements:

- $C \vdash e \Rightarrow T$ 'In the context C, expression e has type T.'
- $C \vdash D \Rightarrow C'$ 'In the context C, declaration D gives context C'.'
- $C \vdash (p \Rightarrow T) \Rightarrow C'$ 'In the context C, binding pattern p to type T gives context C'.'
- $C \vdash T \Rightarrow \textbf{type}$ 'In the context C, T is a type.'

A *context* is a multiset of bindings:

$$\begin{array}{rcl} C & ::= & x \Rightarrow T \\ & \mid & f \Rightarrow T \rightarrow T' \\ & \mid & c \Rightarrow T \rightarrow S \\ & \mid & S \Rightarrow \textbf{type} \\ & \mid & \\ & \mid & C, C \end{array}$$

with the restriction that each binding must be unique, that is:

$$\textbf{if}\ C = C_1, x \Rightarrow T, C_2, x' \Rightarrow T', C_3\ \textbf{then}\ x \neq x'$$

and similarly for the other bindings. We view contexts up to ',' being a commutative monoid.

Let $C \oplus C'$ be context overriding, that is:

$$C \oplus () = C \qquad (C[, x \Rightarrow T]) \oplus (C', x \Rightarrow T') = (C, x \Rightarrow T') \oplus C'$$

and similarly for the other bindings.

Write $C \vdash x \Rightarrow T$ for $C = (C', x \Rightarrow T)$, and similarly for the other bindings.

2.2.1 Static semantics of expressions

The static semantics of expressions is given by inference rules:

$$\frac{}{C, x \Rightarrow T \vdash x \Rightarrow T} \qquad \frac{C \vdash T \Rightarrow \textbf{type}}{C \vdash \textbf{error}\ T \Rightarrow T}$$

$$\frac{\begin{array}{l} C \vdash e \Rightarrow T \\ C \vdash (p_0 \Rightarrow T) \Rightarrow C_0 \quad \cdots \quad C \vdash (p_n \Rightarrow T) \Rightarrow C_n \\ C \oplus C_0 \vdash e_0 \Rightarrow T' \quad \cdots \quad C \oplus C_n \vdash e_n \Rightarrow T' \end{array}}{C \vdash \textbf{case}\ e\ \textbf{of}\ p_0 \rightarrow e_0 \mid \cdots \mid p_n \rightarrow e_n\ \textbf{endcase} \Rightarrow T'}$$

$$\frac{C \vdash c \Rightarrow T \to S \quad C \vdash e \Rightarrow T}{C \vdash ce \Rightarrow S} \qquad \frac{C \vdash f \Rightarrow T \to T' \quad C \vdash e \Rightarrow T}{C \vdash fe \Rightarrow T'}$$

$$\frac{C \vdash e_1 \Rightarrow T_1 \quad \cdots \quad C \vdash e_n \Rightarrow T_n}{C \vdash \{l_1 = e_1, \ldots, l_n = e_n\} \Rightarrow \{l_1 : T_1, \ldots, l_n : T_n\}}$$

$$\frac{C \vdash e \Rightarrow \{l_1 : T_1, \ldots, l_n : T_n\} \quad C \vdash e' \Rightarrow T_i}{C \vdash e.\{l_i = e'\} \Rightarrow \{l_1 : T_1, \ldots, l_n : T_n\}}$$

Note that this type system does not allow empty **case** statements: this is to ensure that every expression is uniquely typed. If we allowed empty type expressions, then ‘**case** {} **of endcase**’ would be ambiguously typed. The expression ‘**error** T’ is equivalent to an empty **case** statement with type T.

2.2.2 Static semantics of declarations

The static semantics of declarations is given by inference rules:

$$\frac{C \vdash T_i \Rightarrow \textbf{type} \quad \cdots \quad C \vdash T_i \Rightarrow \textbf{type}}{\begin{array}{l} C \vdash (\textbf{datatype}\, S := c_1 \textbf{ of } T_1 \mid \cdots \mid c_n \textbf{ of } T_n \textbf{ endtype}) \\ \qquad \Rightarrow (S \Rightarrow \textbf{type}, c_1 \Rightarrow T_1 \to S, \ldots, c_n \Rightarrow T_n \to S) \end{array}}$$

$$\frac{C \vdash ((p \Rightarrow T') \Rightarrow C') \quad C \oplus C' \vdash e \Rightarrow T}{C \vdash (\textbf{function}\ f\,p : T := e \textbf{ endfun}) \Rightarrow (f \Rightarrow T' \to T)}$$

$$\frac{}{C \vdash () \Rightarrow ()} \qquad \frac{C \vdash D_1 \Rightarrow C_1 \quad C \vdash D_2 \Rightarrow C_2}{C \vdash D_1 D_2 \Rightarrow C_1, C_2}$$

These rules do not explicitly support recursive declarations: these are handled by rules such as those for specifications in Section 3.2.1 which close declarations by checking for judgements of the form $C \vdash D \Rightarrow C$.

2.2.3 Static semantics of patterns

The static semantics of patterns is given by inference rules:

$$\frac{C \vdash T \Rightarrow \textbf{type}}{C \vdash ((x : T) \Rightarrow T) \Rightarrow (x \Rightarrow T)} \qquad \frac{C \vdash T \Rightarrow \textbf{type}}{C \vdash (\textbf{any}\ T \Rightarrow T) \Rightarrow ()} \qquad \frac{C \vdash c \Rightarrow T \to S \quad C \vdash (p \Rightarrow T) \Rightarrow C'}{C \vdash (cp \Rightarrow S) \Rightarrow C'}$$

$$\frac{C \vdash (p_1 \Rightarrow T_1) \Rightarrow C_1 \quad \cdots \quad C \vdash (p_n \Rightarrow T_n) \Rightarrow C_n}{C \vdash (\{l_1 = p_1, \ldots, l_n = p_n\} \Rightarrow \{l_1 : T_1, \ldots, l_n : T_n\}) \Rightarrow (C_1, \ldots, C_n)}$$

2.2.4 Static semantics of type expressions

The static semantics of type expressions is given by inference rules:

$$\frac{}{C, S \Rightarrow \textbf{type} \vdash S \Rightarrow \textbf{type}} \qquad \frac{C \vdash T_1 \Rightarrow \textbf{type} \quad \cdots \quad C \vdash T_n \Rightarrow \textbf{type}}{C \vdash \{l_1 : T_1, \ldots, l_n : T_n\} \Rightarrow \textbf{type}}$$

2.3 Dynamic semantics

The dynamic semantics is given by judgements:

- $E \vdash e \Rightarrow v$ 'In the environment E, expression e returns value v.'
- $E \vdash D \Rightarrow E'$ 'In the environment E, declaration D returns environment E'.'
- $E \vdash (p \Rightarrow v) \Rightarrow \sigma$ 'In the environment E, binding pattern p to value v returns substitution σ.'
- $E \vdash (p \Rightarrow v) \Rightarrow \textbf{fail}$ 'In the environment E, binding pattern p to value v fails.'

An *environment* is a multiset of bindings for function identifi ers:

$$\begin{array}{rcl} E & ::= & f \Rightarrow \lambda p\,.\,e \\ & | & \\ & | & E,E \end{array}$$

with the same restrictions as contexts. Let environment overriding $E \oplus E'$ be defi ned as for contexts. Let $E \vdash f \Rightarrow \lambda p\,.\,e$ be defi ned as for contexts.

The *values* are normal forms for expressions:

$$\begin{array}{rcl} v & ::= & cv \\ & | & \{l_1 = v_1, \ldots, l_n = v_n\} \end{array}$$

A *substitutions* is a list of bindings for variable identifi ers:

$$\begin{array}{rcl} \sigma & ::= & x := v \\ & | & \\ & | & \sigma,\sigma \end{array}$$

Let $e[\sigma]$ be the usual defi nition of syntactic substitution (but note that values are closed expressions, so no alpha-conversion is necessary).

2.3.1 Dynamic semantics of expressions

The dynamic semantics of expressions is given by inference rules:

$$\frac{\begin{array}{l} E \vdash e \Rightarrow v \\ E \vdash (p \Rightarrow v) \Rightarrow \sigma \\ E \vdash e'[\sigma] \Rightarrow v' \end{array}}{E \vdash \textbf{case}\ e\ \textbf{of}\ p \to e' \mid \cdots\ \textbf{end} \Rightarrow v'} \qquad \frac{\begin{array}{l} E \vdash e \Rightarrow v \\ E \vdash (p \Rightarrow v) \Rightarrow \textbf{fail} \\ E \vdash \textbf{case}\ e\ \textbf{of}\ \cdots\ \textbf{end} \Rightarrow v' \end{array}}{E \vdash \textbf{case}\ e\ \textbf{of}\ p \to e' \mid \cdots\ \textbf{end} \Rightarrow v'}$$

$$\frac{E \vdash e \Rightarrow v}{E \vdash ce \Rightarrow cv} \qquad \frac{\begin{array}{l} E \vdash f \Rightarrow \lambda p\,.\,e' \\ E \vdash e \Rightarrow v \\ E \vdash (p \Rightarrow v) \Rightarrow \sigma \\ E \vdash e'[\sigma] \Rightarrow v' \end{array}}{E \vdash fe \Rightarrow v'}$$

$$\frac{E \vdash e_i \Rightarrow v_i \quad \cdots \quad E \vdash e_i \Rightarrow v_i}{E \vdash \{l_1 = e_1, \ldots, l_n = e_n\} \Rightarrow \{l_1 = v_1, \ldots, l_n = v_n\}}$$

$$\frac{E \vdash e \Rightarrow \{l_1 = v_1, \ldots, l_n = v_n\} \qquad E \vdash e' \Rightarrow v'}{E \vdash e.\{l_i = e_i\} \Rightarrow \{l_1 = v_1, \ldots, l_i = v' \ldots, l_n = v_n\}}$$

Note that the **error** statement has no inference rules: in this simple semantics, reaching an error state is the same as diverging. We may wish to use a finer semantics for the final standard.

2.3.2 Dynamic semantics of declarations

The dynamic semantics of declarations is given by inference rules:

$$\frac{}{E \vdash (\textbf{datatype} \ldots \textbf{endtype}) \Rightarrow ()}$$

$$\frac{}{E \vdash (\textbf{function}\ f\,p : T := e\ \textbf{endfun}) \Rightarrow (f \Rightarrow \lambda p\,.\,e)}$$

$$\frac{}{E \vdash () \Rightarrow ()} \qquad \frac{E \vdash D_1 \Rightarrow E_1 \qquad E \vdash D_2 \Rightarrow E_2}{E \vdash D_1 D_2 \Rightarrow E_1, E_2}$$

Note that the rule for function declarations is only sound because there are no partially applied functions: if there were, we would have to include closures in the semantics.

2.3.3 Dynamic semantics of patterns

The dynamic semantics of patterns is given by inference rules:

$$\frac{}{E \vdash ((x : T) \Rightarrow v) \Rightarrow (x := v)} \qquad \frac{}{E \vdash (\textbf{any}\ T \Rightarrow v) \Rightarrow ()}$$

$$\frac{E \vdash (p \Rightarrow v) \Rightarrow \sigma}{E \vdash (cp \Rightarrow cv) \Rightarrow \sigma} \qquad \frac{c \neq c'}{E \vdash (cp \Rightarrow c'v) \Rightarrow \textbf{fail}}$$

$$\frac{E \vdash (p_1 \Rightarrow v_1) \Rightarrow \sigma_1 \quad \cdots \quad E \vdash (p_n \Rightarrow v_n) \Rightarrow \sigma_n}{E \vdash (\{l_1 = p_1, \ldots, l_n = p_n\} \Rightarrow \{l_1 : v_1, \ldots, l_n : v_n\}) \Rightarrow (\sigma_1, \ldots, \sigma_n)}$$

$$\frac{E \vdash (p_i \Rightarrow v_i) \Rightarrow \textbf{fail}}{E \vdash (\{l_1 = p_1, \ldots, l_n = p_n\} \Rightarrow \{l_1 : v_1, \ldots, l_n : v_n\}) \Rightarrow \textbf{fail}}$$

3 Behavioural language

In this section, we present a simple behavioural language based on current LOTOS, but using the core functional language for data rather than ACT ONE. We show how it can be given a static and dynamic semantics in a similar style to the core functional language (except that the operational semantics is given by a labelled transition system rather than a 'big step' convergence relation).

Not all of the features of LOTOS are present in this behavioural language. Some (such as nesting of declarations) are best treated by the modules facility. Most (such as allowing **any** in **exit** statements) can be included without difficulty.

3.1 Syntax

The terminals extend those of Section 2.1 with:

symbol domain	*meaning*	*abbreviations*
Gate	gate identifi ers	g
Proc	process identifi ers	P

A *specification* is:

$$S \quad ::= \quad \textbf{specification}\ P\ \textbf{where}\ D\ \textbf{endspec}$$

This syntax is simpler than that in existing LOTOS, but we can view the specifi cation:

$$\textbf{specification}\ P \ldots\ \textbf{behaviour}\ B\ \textbf{where}\ D\ \textbf{endspec}$$

as an abbreviation for:

$$\textbf{specification}\ P\ \textbf{where process}\ P \ldots := B\ \textbf{endproc}\ D\ \textbf{endspec}$$

The only important difference between this form and the existing LOTOS specifi cation is that P can be used as a process identifi er in D, which it cannot in existing LOTOS. (This is in line with an existing suggested enhancement to LOTOS [1].)

The *declarations* are extended:

$$\begin{array}{lcl} D & ::= & \ldots \text{as in Section 2.1} \ldots \\ & | & \textbf{process}\ [g_1 : \textbf{gate}\ T_1, \ldots, g_n : \textbf{gate}\ T_n]\, p : \textbf{exit}\ T := B\ \textbf{endproc} \end{array}$$

The *behaviours* are:

$$\begin{array}{lcl} B & ::= & \textbf{stop} \\ & | & \textbf{exit}\ o \\ & | & go[e]; B \\ & | & B_1 \,[\,]\, B_2 \\ & | & B_1 \,|[g_1, \ldots, g_n]|\, B_2 \\ & | & \textbf{hide}\ g : \textbf{gate}\ T\ \textbf{in}\ B \\ & | & \textbf{choice}\ p \,[\,]\, B \\ & | & B_1 \gg \textbf{accept}\ p\ \textbf{in}\ B_2 \\ & | & \textbf{case}\ e\ \textbf{of}\ p_1 \to B_1 \mid \cdots \mid p_n \to B_n\ \textbf{endcase} \\ & | & P[g_1, \ldots, g_n]e \end{array}$$

The *offers* are:

$$\begin{array}{lcl} o & ::= & !e \\ & | & ?p \\ & | & co \\ & | & \{l_1 = o_1, \ldots, l_n = o_n\} \end{array}$$

Note that the syntax for **exit** has been simplifi ed to use the same offers as for communication. This makes the semantics simpler, but for the fi nal version we should use the existing syntax, writing ‘**exit** (e, **any** T)’ rather than ‘**exit** (!e, ?**any** T)’.

3.2 Static semantics

The static semantics assumes the existence of two “standard” types:

datatype Bool := true | false **endtype**
datatype None := **endtype**

(using the syntactic sugar for constant constructors given in [2].) In the full language, these types will be defined in modules in the standard library.

The static semantics extends that of Section 2.2 with judgements:

- $C \vdash B \Rightarrow \textbf{exit}\ T$ ‘In context C, behaviour B has functionality **exit** T.’
- $C \vdash (o \Rightarrow T) \Rightarrow C'$ ‘In the context C, binding offer o to type T gives context C'.’
- $\vdash S \Rightarrow C$ ‘The specification S is well-typed, with typing given by context C.’

Contexts are extended with process and gate identifiers:

$$\begin{array}{rcl} C & ::= & \ldots\text{as in Section 2.2}\ldots \\ & | & P \Rightarrow [\textbf{gate}\ T_1, \ldots, \textbf{gate}\ T_n] \rightarrow T \rightarrow \textbf{exit}\ T' \\ & | & g \Rightarrow \textbf{gate}\ T \end{array}$$

In this semantics, we treat the functionality ‘**noexit**’ as a synonym for ‘**exit** None’ and ‘**exit**’ as a synonym for ‘**exit** $\{\}$’. This simplifies the semantics, but is *not* compatible with existing specifications, for example it fails to type ‘**stop** || **exit** 0’ (this example was provided by Charles Pecheur). This problem is left for future work.

3.2.1 Static semantics of specifications

The static semantics of specifications is given by the inference rule:

$$\frac{\begin{array}{l} C \vdash D \Rightarrow C \\ C \vdash (P \Rightarrow [\textbf{gate}\ T_1, \ldots, \textbf{gate}\ T_n] \rightarrow T \rightarrow \textbf{exit}\ T') \end{array}}{\begin{array}{l} \vdash (\textbf{specification}\ P\ \textbf{where}\ D\ \textbf{endspec}) \\ \qquad \Rightarrow (P \Rightarrow [\textbf{gate}\ T_1, \ldots, \textbf{gate}\ T_n] \rightarrow T \rightarrow \textbf{exit}\ T') \end{array}}$$

Note that this rule includes the fact that the declaration D is allowed to be recursive.

3.2.2 Static semantics of declarations

The static semantics of declarations extends that of Section 2.2.2 with:

$$\frac{\begin{array}{l} C \vdash T_1 \Rightarrow \textbf{type} \quad \cdots \quad C \vdash T_n \Rightarrow \textbf{type} \\ C \vdash (p \Rightarrow T') \Rightarrow C' \\ C \oplus (g_1 \Rightarrow \textbf{gate}\ T_1, \ldots, g_n \Rightarrow \textbf{gate}\ T_n) \oplus C' \vdash B \Rightarrow \textbf{exit}\ T \end{array}}{\begin{array}{l} C \vdash (\textbf{process}\ P[g_1 : \textbf{gate}\ T_1, \ldots, g_n : \textbf{gate}\ T_n]\, p : \textbf{exit}\ T := B\ \textbf{endproc}) \\ \qquad \Rightarrow (P \Rightarrow [\textbf{gate}\ T_1, \ldots, \textbf{gate}\ T_n] \rightarrow T' \rightarrow \textbf{exit}\ T) \end{array}}$$

3.2.3 Static semantics of behaviours

The static semantics of behaviours is given by inference rules:

$$\frac{}{C \vdash \textbf{stop} \Rightarrow \textbf{exit}\ \text{None}} \qquad \frac{C \vdash (o \Rightarrow T) conv C'}{C \vdash \textbf{exit}\ o \Rightarrow \textbf{exit}\ T} \qquad \frac{\begin{array}{l} C \vdash g \Rightarrow \textbf{gate}\ T \\ C \vdash (o \Rightarrow T) \Rightarrow C' \\ C \oplus C' \vdash e \Rightarrow \text{Bool} \\ C \oplus C' \vdash B \Rightarrow \textbf{exit}\ T' \end{array}}{C \vdash go[e]; B \Rightarrow \textbf{exit}\ T'}$$

$$\frac{\begin{array}{l} C \vdash B_1 \Rightarrow \textbf{exit}\ T \qquad C \vdash B_2 \Rightarrow \textbf{exit}\ T \\ C \vdash g_1 \Rightarrow \textbf{gate}\ T_1 \quad \cdots \quad C \vdash g_n \Rightarrow \textbf{gate}\ T_n \end{array}}{C \vdash B_1 \,|[g_1, \ldots, g_n]|\, B_2 \Rightarrow \textbf{exit}\ T}$$

$$\frac{C \vdash B_1 \Rightarrow \textbf{exit}\ T \qquad C \vdash B_2 \Rightarrow \textbf{exit}\ T}{C \vdash B_1 \,[]\, B_2 \Rightarrow \textbf{exit}\ T}$$

$$\frac{\begin{array}{l} C \vdash T \Rightarrow \textbf{type} \\ C \oplus (g \Rightarrow \textbf{gate}\ T) \vdash B \Rightarrow \textbf{exit}\ T' \end{array}}{C \vdash \textbf{hide}\ g : \textbf{gate}\ T\ \textbf{in}\ B \Rightarrow \textbf{exit}\ T'} \qquad \frac{\begin{array}{l} C \vdash (p \Rightarrow T) \Rightarrow C' \\ C \oplus C' \vdash B \Rightarrow \textbf{exit}\ T' \end{array}}{C \vdash \textbf{choice}\ p \,[]\, B \Rightarrow \textbf{exit}\ T'}$$

$$\frac{\begin{array}{l} C \vdash B \Rightarrow \textbf{exit}\ T \\ C \vdash (p \Rightarrow T) \Rightarrow C' \\ C \oplus C' \vdash B' \Rightarrow \textbf{exit}\ T' \end{array}}{C \vdash B \gg \textbf{accept}\ p\ \textbf{in}\ B' \Rightarrow \textbf{exit}\ T'}$$

$$\frac{\begin{array}{l} C \vdash e \Rightarrow T \\ C \vdash (p_i \Rightarrow T) \Rightarrow C_i \\ C \oplus C_0 \vdash B_0 \Rightarrow \textbf{exit}\ T' \quad \cdots \quad C \oplus C_n \vdash B_n \Rightarrow \textbf{exit}\ T' \end{array}}{C \vdash \textbf{case}\ e\ \textbf{of}\ p_0 \to B_0 \mid \cdots \mid p_n \to B_n\ \textbf{endcase} \Rightarrow \textbf{exit}\ T'}$$

$$\frac{\begin{array}{l} C \vdash P \Rightarrow [\textbf{gate}\ T_1, \ldots, \textbf{gate}\ T_n] \to T \to \textbf{exit}\ T' \\ C \vdash g_1 \Rightarrow \textbf{gate}\ T_1 \quad \cdots \quad C \vdash g_n \Rightarrow \textbf{gate}\ T_n \\ C \vdash e \Rightarrow T \end{array}}{C \vdash P[g_1, \ldots, g_n]e \Rightarrow \textbf{exit}\ T'}$$

3.2.4 Static semantics of offers

The static semantics of offers is given by inference rules:

$$\frac{C \vdash e \Rightarrow T}{C \vdash (!e \Rightarrow T) \Rightarrow ()} \qquad \frac{C \vdash (p \Rightarrow T) \Rightarrow C'}{C \vdash (?p \Rightarrow T) \Rightarrow C'}$$

$$\frac{\begin{array}{l} C \vdash c \Rightarrow T \to S \\ C \vdash (o \Rightarrow T) \Rightarrow C' \end{array}}{C \vdash (co \Rightarrow S) \Rightarrow C'}$$

$$\frac{C \vdash (o_1 \Rightarrow T_1) \Rightarrow C_1 \quad \cdots \quad C \vdash (o_n \Rightarrow T_n) \Rightarrow C_n}{C \vdash (\{l_1 = o_1, \ldots, l_n = o_n\} \Rightarrow \{l_1 : T_1, \ldots, l_n : T_n\}) \Rightarrow (C_1, \ldots, C_n)}$$

3.3 Dynamic semantics

The dynamic semantics extends that of Section 2.3 with judgements:

- $E \vdash B \xrightarrow{a} B'$ 'In the context C and the environment E, B can perform an a action and reduce to B'.'
- $E \vdash (o \Rightarrow v) \Rightarrow \sigma$ 'In environment E, binding offer o to value v returns substitution σ.'

The *actions* are:

$$\begin{array}{rcl} a & ::= & i \\ & | & gv \\ & | & \delta v \end{array}$$

Environments are extended with process identifiers and the type information of constructors:

$$\begin{array}{rcl} E & ::= & \ldots\text{as in Section 2.3}\ldots \\ & | & P \Rightarrow \lambda[g_1,\ldots,g_n]\,.\,\lambda p\,.\,B \\ & | & c \Rightarrow (T \to S) \end{array}$$

The type information for constructors is needed because constructs such as **choice** and input can 'invent' new elements of any type. In order to make sure that processes are type-safe, this means we have to be able to type any value.

Substitutions are extended with gate identifier bindings:

$$\begin{array}{rcl} \sigma & ::= & \ldots\text{as in Section 2.3}\ldots \\ & | & g_1 := g_2 \end{array}$$

Note that this semantics uses gate substitution rather than gate relabelling. This is simpler (since gate bindings and value bindings are treated identically) but does require α-conversion of gates bound by **hide**. It is left for a future decision whether α-conversion or explicit relabelling should be used in E-LOTOS.

3.3.1 Dynamic semantics of specifications

The dynamic semantics of specifications is given by inference rules:

$$\frac{\begin{array}{l}\vdash D \Rightarrow E \\ E \vdash B \xrightarrow{a} B'\end{array}}{\textbf{specification}\ P\ \textbf{where}\ D\ \textbf{endspec} \vdash B \xrightarrow{a} B'}$$

For any $g'_1,\ldots,g'_n$ and v, the specification $S = \textbf{specification}\ P\ \textbf{where}\ D\ \textbf{endspec}$ determines a labelled transition system rooted at $S \vdash P[g'_1,\ldots,g'_n]v \xrightarrow{a_1} B_1 \xrightarrow{a_2} B_2 \ldots$

3.3.2 Dynamic semantics of declarations

The dynamic semantics of declarations extends that of Section 2.3.2 with:

$$\frac{}{\begin{array}{l} E \vdash (\mathbf{process}\ P[g_1 : \mathbf{gate}\ T_1, \ldots, g_n : \mathbf{gate}\ T_n]\, p : \mathbf{exit}\ T := B\ \mathbf{endproc}) \\ \qquad \Rightarrow (P \Rightarrow \lambda[g_1, \ldots, g_n] . \lambda p . B) \end{array}}$$

and replaces the rule for datatype declarations by:

$$\frac{}{\begin{array}{l} E \vdash (\mathbf{datatype}\ S := c_1\ \mathbf{of}\ T_1 \mid \cdots \mid c_n\ \mathbf{of}\ T_n\ \mathbf{endtype}) \\ \qquad \Rightarrow (c_1 \Rightarrow T_1 \rightarrow S, \ldots, c_n \Rightarrow T_n \rightarrow S) \end{array}}$$

3.3.3 Dynamic semantics of behaviours

The dynamic semantics of behaviours is given by inference rules:

$$\frac{E \vdash (o \Rightarrow v) \Rightarrow \sigma}{E \vdash \mathbf{exit}\ o \xrightarrow{\delta v} \mathbf{stop}} \qquad \frac{\begin{array}{l} E \vdash (o \Rightarrow v) \Rightarrow \sigma \\ E \vdash e[\sigma] \Rightarrow \text{true} \end{array}}{E \vdash go[e]; B \xrightarrow{gv} B[\sigma]}$$

$$\frac{E \vdash B_i \xrightarrow{a} B'_i}{E \vdash B_1 [\,] B_2 \xrightarrow{a} B'_i}$$

$$\frac{\begin{array}{l} E \vdash B_1 \xrightarrow{gv} B'_1 \qquad E \vdash B_2 \xrightarrow{gv} B'_2 \\ g \in \vec{g} \end{array}}{E \vdash B_1 |[\vec{g}]| B_2 \xrightarrow{gv} B'_1 |[\vec{g}]| B'_2} \qquad \frac{E \vdash B_1 \xrightarrow{\delta v} B'_1 \qquad E \vdash B_2 \xrightarrow{\delta v} B'_2}{E \vdash B_1 |[\vec{g}]| B_2 \xrightarrow{\delta v} B'_1 |[\vec{g}]| B'_2}$$

$$\frac{\begin{array}{l} E \vdash B_1 \xrightarrow{gv} B'_1 \\ g \notin \vec{g} \end{array}}{E \vdash B_1 |[\vec{g}]| B_2 \xrightarrow{gv} B'_1 |[\vec{g}]| B_2} \qquad \frac{\begin{array}{l} E \vdash B_2 \xrightarrow{gv} B'_2 \\ g \notin \vec{g} \end{array}}{E \vdash B_1 |[\vec{g}]| B_2 \xrightarrow{gv} B_1 |[\vec{g}]| B'_2}$$

$$\frac{E \vdash B_1 \xrightarrow{i} B'_1}{E \vdash B_1 |[\vec{g}]| B_2 \xrightarrow{i} B'_1 |[\vec{g}]| B_2} \qquad \frac{E \vdash B_2 \xrightarrow{i} B'_2}{E \vdash B_1 |[\vec{g}]| B_2 \xrightarrow{i} B_1 |[\vec{g}]| B'_2}$$

$$\frac{E \vdash B \xrightarrow{gv} B'}{E \vdash \mathbf{hide}\ g : \mathbf{gate}\ T\ \mathbf{in}\ B \xrightarrow{i} \mathbf{hide}\ g : \mathbf{gate}\ T\ \mathbf{in}\ B'}$$

$$\frac{\begin{array}{l} E \vdash B \xrightarrow{a} B' \\ a \neq gv \end{array}}{E \vdash \mathbf{hide}\ g : \mathbf{gate}\ T\ \mathbf{in}\ B \xrightarrow{g'v} \mathbf{hide}\ g : \mathbf{gate}\ T\ \mathbf{in}\ B'}$$

$$\frac{\begin{array}{l} E \vdash (p \Rightarrow v) \Rightarrow \sigma \\ E \vdash B[\sigma] \xrightarrow{a} B' \end{array}}{E \vdash \mathbf{choice}\ p [\,] B \xrightarrow{a} B'}$$

$$\frac{\begin{array}{l} E \vdash B_1 \xrightarrow{a} B'_1 \\ a \neq \delta v \end{array}}{E \vdash B_1 \gg \mathbf{accept}\ p\ \mathbf{in}\ B_2 \xrightarrow{a} B'_1 \gg \mathbf{accept}\ p\ \mathbf{in}\ B_2}$$

$$\frac{E \vdash B_1 \xrightarrow{\delta v} B_1' \qquad E \vdash (p \Rightarrow v) \Rightarrow \sigma}{E \vdash B_1 \gg \textbf{accept } p \textbf{ in } B_2 \xrightarrow{i} B_2[\sigma]}$$

$$\frac{E \vdash e \Rightarrow v \qquad E \vdash (p \Rightarrow v) \Rightarrow \sigma \qquad E \vdash B[\sigma] \xrightarrow{a} B'}{E \vdash \textbf{case } e \textbf{ of } p \rightarrow B \mid \cdots \textbf{ end} \xrightarrow{a} B'}$$

$$\frac{E \vdash e \Rightarrow v \qquad E \vdash (p \Rightarrow v) \Rightarrow \textbf{fail} \qquad E \vdash \textbf{case } e \textbf{ of } \cdots \textbf{ end} \xrightarrow{a} B'}{E \vdash \textbf{case } e \textbf{ of } p \rightarrow e' \mid \cdots \textbf{ end} \xrightarrow{a} B'}$$

$$\frac{E \vdash P \Rightarrow \lambda[g_1, \ldots, g_n] . \lambda p . B \qquad E \vdash e \Rightarrow v \qquad E \vdash (p \Rightarrow v) \Rightarrow \sigma \qquad E \vdash B[\sigma, g_1 := g_1', \ldots, g_n := g_n'] \xrightarrow{a} B'}{E \vdash P[g_1', \ldots, g_n']e \xrightarrow{a} B'}$$

3.3.4 Dynamic semantics of offers

The dynamic semantics of offers is given by inference rules:

$$\frac{E \vdash e \Rightarrow v}{E \vdash (!e \Rightarrow v) \Rightarrow ()} \qquad \frac{E \vdash (p \Rightarrow v) \Rightarrow \sigma}{E \vdash (?p \Rightarrow v) \Rightarrow \sigma} \qquad \frac{E \vdash (o \Rightarrow v) \Rightarrow \sigma}{E \vdash (co \Rightarrow cv) \Rightarrow \sigma}$$

$$\frac{E \vdash (o_1 \Rightarrow v_1) \Rightarrow \sigma_1 \quad \cdots \quad E \vdash (o_n \Rightarrow v_n) \Rightarrow \sigma_n}{E \vdash (\{l_1 = o_1, \ldots, l_n = o_n\} \Rightarrow \{l_1 = v_1, \ldots, l_n = v_n\}) \Rightarrow (\sigma_1, \ldots, \sigma_n)}$$

3.3.5 Dynamic semantics of type checking

Because of **choice** and input, we have to be able to type-check values at 'run-time', which means carrying type-checking information around. The dynamic semantics of type-checking values is given by inference rules:

$$\frac{E \vdash c \Rightarrow T \rightarrow S \qquad E \vdash v \Rightarrow T}{E \vdash cv \Rightarrow S} \qquad \frac{E \vdash v_1 \Rightarrow T_1 \quad \cdots \quad E \vdash v_n \Rightarrow T_n}{E \vdash \{l_1 = v_1, \ldots, l_n = v_n\} \Rightarrow \{l_1 : T_1, \ldots, l_n : T_n\}}$$

Moreover, we have to perform type-checking during pattern-matching, otherwise **choice** and communication can produce type errors:

$$\frac{E \vdash v \Rightarrow T}{E \vdash ((x : T) \Rightarrow v) \Rightarrow (x := v)} \qquad \frac{E \vdash v \Rightarrow T}{E \vdash (\textbf{any } T \Rightarrow v) \Rightarrow ()}$$

4 Introducing data abstraction

In Sections 2 and 3 declarations are 'flat', and there is no method for hiding implementation details. One of the goals for E-LOTOS is to investigate module systems to implement such hiding, and we will consider some of its effects here.

For the sake of simplicity, we will not consider an entire module system, but instead just add a new construct for local declarations. This presents many of the problems of giving a semantics for data abstraction, for example we can define a 'NatSet' type implementing sets of natural numbers with:

```
local
  datatype NatList := nil | cons of Nat * NatList endtype
  function
     nil merge ns := ns
    | ms merge nil := ms
    | (m cons ms) merge (n cons ns) :=
       if m < n
       then m cons (ms merge (n cons ns))
       else n cons ((m cons ms) merge ns)
       endif
  endfun
  function
     m in nil := false
    | m in (n cons ns) := (m = n) or (m > n and m in ns)
  endfun
in
  datatype NatSet := hidden of NatList endtype
  function singleton n := hidden (n cons nil) endfun
  function (hidden ms) union (hidden ns) := hidden (ms merge ns) endfun
  function m member (hidden ns) := m in ns endfun
endloc
```

(This specification uses some of the syntactic sugar for the core language discussed in [2].)

The dynamic semantics given below will type-check this as producing the context:

$$\begin{array}{rcl}
\text{NatSet} & \Rightarrow & \textbf{type} \\
\text{hidden} & \Rightarrow & \text{NatList} \rightarrow \text{NatSet} \\
\text{singleton} & \Rightarrow & \text{Nat} \rightarrow \text{NatSet} \\
\text{union} & \Rightarrow & \text{NatSet} * \text{NatSet} \rightarrow \text{NatSet} \\
\text{member} & \Rightarrow & \text{Nat} * \text{NatSet} \rightarrow \text{Bool}
\end{array}$$

Note that in this context, 'NatSet' is visible, but 'NatList' is not, so the implementation of 'NatSet' is hidden.

If we allow abstract datatypes such as 'NatSet' in communications, then the operational semantics of Section 3.3 has two problems:

1. Using a hidden input such as:

hide g : **gate** NatSet **in** g?x; **exit** x

we can nondeterministically generate any element of 'NatSet', including elements such as 'hidden (cons 1 (cons 0 nil))' which violate the invariant that 'NatSet' is implemented as a sorted list.

2. Synchronization on output also presents problems, since the definition in Section 3.3.3 (which allows output expressions to synchronize when they have the same normal form) causes the following to deadlock:

   g!(singleton 0); **exit** |[g]| g!((singleton 0) union (singleton 0)); **exit**

   since 'singleton 0' has normal form 'hidden (cons 0 nil)' whereas '(singleton 0) union (singleton 0)' has normal form 'hidden (cons 0 (cons 0 nil))'.

In this section we discuss how abstract types such as 'NatSet' can be integrated into LOTOS, in three different ways:

1. Abstract datatypes can be treated in the same way as concrete datatypes, which is simple but produces the problems noted above.
2. Abstract datatypes can be barred from communication (and similar problem cases such as **choice**) which prevents any problems from arising, but is very restrictive.
3. Specifiers can decide the communication possibilities of abstract types. This is the most flexible solution, but does require a user-specified function to be called every time a synchronization or hiding occurs.

One of the decisions to be taken in designing the E-LOTOS abstract types system is which (if any) of the above solutions to adopt.

4.1 Common ground between the 3 options

The three options are all based on the same syntactic extension: allowing local declarations. This has syntax:

$$\begin{array}{rcl} D & ::= & \ldots\text{as in Section 3.1}\ldots \\ & | & \textbf{local}\ D_1\ \textbf{in}\ D_2\ \textbf{endloc} \end{array}$$

static semantics:

$$\frac{C, C_1 \vdash D_1 \Rightarrow C_1 \qquad C, C_1 \vdash D_2 \Rightarrow C_2}{C \vdash \textbf{local}\ D_1\ \textbf{in}\ D_2\ \textbf{endloc} \Rightarrow C_2}$$

and dynamic semantics:

$$\frac{E \vdash D_1 \Rightarrow E_1 \qquad E \vdash D_2 \Rightarrow E_2}{E \vdash \textbf{local}\ D_1\ \textbf{in}\ D_2\ \textbf{endloc} \Rightarrow E_1, E_2}$$

Note that this "flattening" of a declaration to produce an environment may require α-conversion, for example the declaration:

local function foo (x,y) := x **endfun**
in function bar x := foo x **endfun endloc**
local function foo (x,y) := y **endfun**
in function baz x := foo x **endfun endloc**

produces the environment:

$$\begin{array}{rcl} \mathrm{foo}_1 & \Rightarrow & \lambda(x,y)\,.\,x \\ \mathrm{bar} & \Rightarrow & \lambda x\,.\,\mathrm{foo}_1 x \\ \mathrm{foo}_2 & \Rightarrow & \lambda(x,y)\,.\,y \\ \mathrm{baz} & \Rightarrow & \lambda x\,.\,\mathrm{foo}_2 x \end{array}$$

Since there is no overloading in this fragment of the language, such α-conversion can be carried out syntactically, without requiring any type information.

4.2 Option 1: Abstract datatypes treated as concrete datatypes

The first option is not to change the semantics of Section 3.3 at all. This is simple, but breaks data abstraction as discussed above.

4.3 Option 2: Abstract datatypes banned from communication

The second option is to use the static semantics to ban abstract datatypes from communications and **choice**. There are a number of ways in which this can be achieved, but probably the simplest is to add a new 'abstract type' declaration which specifies that a datatype is not for use in communications.

This restriction is still fairly simple, but is very restrictive: all types are declared as either concrete (all their structure is visible) or abstract (none of their structure is visible).

4.3.1 Option 2: Syntax

One possible syntax for abstract type declarations is:

$$\begin{array}{rcl} D & ::= & \ldots\text{as in Section 3.1}\ldots \\ & | & \textbf{abstype}\ S := c_1\ \textbf{of}\ T_1 \mid \cdots \mid c_n\ \textbf{of}\ T_n\ \textbf{endtype} \end{array}$$

For example, the declaration of the 'NatSet' type would now be:

abstype NatSet := hidden **of** NatList **endtype**

4.3.2 Option 2: Static semantics

The static semantics is enriched with judgements:

- $C \vdash T \Rightarrow$ **concrete** 'In the context C, T is a concrete type.'

and contexts are similarly enriched:

$$\begin{array}{rcl} C & ::= & \ldots\text{as in Section 3.2}\ldots \\ & \mid & S \Rightarrow \textbf{concrete} \end{array}$$

A type is then defined to be concrete iff it is a concrete sort, or a record of concrete types:

$$\frac{}{C, S \Rightarrow \textbf{concrete} \vdash S \Rightarrow \textbf{concrete}} \qquad \frac{C \vdash T_1 \Rightarrow \textbf{concrete} \quad \cdots \quad C \vdash T_n \Rightarrow \textbf{concrete}}{C \vdash \{l_1 : T_1, \ldots, l_n : T_n\} \Rightarrow \textbf{concrete}}$$

The static semantics for abstract datatype declaration is:

$$\frac{C \vdash T_1 \Rightarrow \textbf{type} \quad \cdots \quad C \vdash T_n \Rightarrow \textbf{type}}{\begin{array}{l} C \vdash (\textbf{abstype}\, S := c_1 \textbf{ of } T_1 \mid \cdots \mid c_n \textbf{ of } T_n \textbf{ endtype}) \\ \qquad \Rightarrow (S \Rightarrow \textbf{type}, c_1 \Rightarrow T_1 \rightarrow S, \ldots, c_n \Rightarrow T_n \rightarrow S) \end{array}}$$

The static semantics of datatype declaration says that a datatype is concrete iff all of its components are. This is achieved by adding the inference rule:

$$\frac{C \vdash T_1 \Rightarrow \textbf{concrete} \quad \cdots \quad C \vdash T_n \Rightarrow \textbf{concrete}}{\begin{array}{l} C \vdash (\textbf{datatype}\, S := c_1 \textbf{ of } T_1 \mid \cdots \mid c_n \textbf{ of } T_n \textbf{ endtype}) \\ \qquad \Rightarrow (S \Rightarrow \textbf{type}, S \Rightarrow \textbf{concrete}, c_1 \Rightarrow T_1 \rightarrow S, \ldots, c_n \Rightarrow T_n \rightarrow S) \end{array}}$$

We replace the condition '$T \Rightarrow$ **type**' by '$T \Rightarrow$ **concrete**' in the places where we do not wish to allow abstract types:

$$\frac{\begin{array}{l} C \vdash T_1 \Rightarrow \textbf{concrete} \quad \cdots \quad C \vdash T_n \Rightarrow \textbf{concrete} \\ C \vdash (p \Rightarrow T') \Rightarrow C' \\ C \oplus (g_1 \Rightarrow \textbf{gate } T_1, \ldots, g_n \Rightarrow \textbf{gate } T_n) \oplus C' \vdash B \Rightarrow \textbf{exit } T \end{array}}{\begin{array}{l} C \vdash (\textbf{process } P[g_1 : \textbf{gate } T_1, \ldots, g_n : \textbf{gate } T_n]\, p : \textbf{exit } T := B \textbf{ endproc}) \\ \qquad \Rightarrow (P \Rightarrow [\textbf{gate } T_1, \ldots, \textbf{gate } T_n] \rightarrow T' \rightarrow \textbf{exit } T) \end{array}}$$

$$\frac{\begin{array}{l} C \vdash T \Rightarrow \textbf{concrete} \\ C \oplus (g \Rightarrow \textbf{gate } T) \vdash B \Rightarrow \textbf{exit } T' \end{array}}{C \vdash \textbf{hide } g : \textbf{gate } T \textbf{ in } B \Rightarrow \textbf{exit } T'} \qquad \frac{\begin{array}{l} C \vdash T \Rightarrow \textbf{concrete} \\ C \vdash (p \Rightarrow T) \Rightarrow C' \\ C \oplus C' \vdash B \Rightarrow \textbf{exit } T' \end{array}}{C \vdash \textbf{choice } p\, [\,]\, B \Rightarrow \textbf{exit } T'}$$

$$\frac{\begin{array}{l} C \vdash (o \Rightarrow T) \Rightarrow C' \\ C \vdash T \Rightarrow \textbf{concrete} \end{array}}{C \vdash \textbf{exit } o \Rightarrow \textbf{exit } T}$$

For example, the 'NatSet' **abstype** declaration is not **concrete**, and so we cannot use 'NatSet' inside **choice** or communication.

4.3.3 Option 2: Dynamic semantics

There is no change required to the dynamic semantics of Section 3.3.

4.4 Option 3: Specifier given control

The final option is to allow the specifier to state how a type can be used in communications. Again, there are a number of ways this can be specified, but one is to allow the specifier to say when a value of a datatype is 'valid' (and so can be produced by **hide** or **choice**) and when two output values can synchronize.

4.4.1 Option 3: Syntax

One syntax for allowing specifiers to define the validity and synchronization possibilities of a type is:

$$\begin{array}{rcl} D & ::= & \ldots\text{as in Section 3.1}\ldots \\ & \mid & \textbf{datatype}\, S := c_1 \textbf{ of } T_1 \mid \cdots \mid c_n \textbf{ of } T_n \\ & & \textbf{valid}\, x := e \\ & & \textbf{sync}\, (x_1, x_2) := e \\ & & \textbf{endtype} \end{array}$$

We can replace **datatype** declarations with the above extension, by regarding:

$$\textbf{datatype}\, S := c_1 \textbf{ of } T_1 \mid \cdots \mid c_n \textbf{ of } T_n \textbf{ endtype}$$

as syntactic sugar for (assuming we can find an appropriate semantics for equality, which we will ignore for the moment):

$$\begin{array}{l} \textbf{datatype} \\ \quad S := c_1 \textbf{ of } T_1 \mid \cdots \mid c_n \textbf{ of } T_n \\ \quad \textbf{valid}\, x := \text{true} \\ \quad \textbf{sync}\, (x_1, x_2)[x_1 = x_2] := x_1 \\ \textbf{endtype} \end{array}$$

For example, the communication rules for 'NatSet' are:

- A list is valid iff it is sorted.
- Two lists can synchronize if they contain the same elements in the same order.

These can be specified as (using the same syntactic sugar as for defining pattern-matching functions):

```
local
  ...
  function
      sorted (x cons (y cons ys)) := (x <= y) and sorted (y cons ys)
    | sorted any := true
  endfun
  function
      (x cons (y cons ys)) combine zs [x=y] := (y cons ys) combine zs
    | xs combine (y cons (z cons zs)) [y=z] := xs combine (z cons zs)
    | (x cons xs) combine (y cons ys) [x=y] := xs combine ys
  endfun
in
  datatype
    NatSet := hidden of NatList
    valid (hidden xs) := sorted xs
    sync (hidden xs, hidden ys) := hidden (xs combine ys)
  endtype
  ...
endloc
```

With this semantics, **hide** and **choice** can only generate sorted lists, for example:

$$(\textbf{choice}\ (x : \text{NatSet})\ [\,]\ \textbf{exit}\ x) \xrightarrow{\delta\text{hidden(cons(0,cons(1,nil)))}} \textbf{stop}$$

but:

$$(\textbf{choice}\ (x : \text{NatSet})\ [\,]\ \textbf{exit}\ x) \not\xrightarrow{\delta\text{hidden(cons(1,cons(0,nil)))}} \textbf{stop}$$

Similarly, values can be synchronised on output to produce the result given by the **sync** specification, for instance:

$$\begin{aligned}&g!(\text{singleton } 0); \textbf{exit}\,|[g]|\,g!((\text{singleton } 0) \text{ union } (\text{singleton } 0)); \textbf{exit}\\ &\quad \xrightarrow{g!\text{hidden(cons(0,nil))}} \textbf{exit}\,|[g]|\,\textbf{exit}\end{aligned}$$

This option is the most flexible of the three, but it does require the specifier to provide 'reasonable' **valid** and **sync** functions, and it does require the execution of arbitrary user code whenever a **choice** occurs, and whenever synchronization on output happens.

4.4.2 Option 3: Static semantics

The only addition to the static semantics of Section 3.2 is to type-check the extended datatype declarations:

$$\frac{\begin{array}{l} C \vdash T_i \Rightarrow \textbf{type} \quad \cdots \quad C \vdash T_i \Rightarrow \textbf{type} \\ C \oplus (x \Rightarrow S) \vdash e_1 \Rightarrow \text{Bool} \qquad C \oplus (x_1 \Rightarrow S, x_2 \Rightarrow S) \vdash e_2 \Rightarrow S \end{array}}{\begin{array}{l} C \vdash (\textbf{datatype}\ S := c_1\ \textbf{of}\ T_1 \mid \cdots \mid c_n\ \textbf{of}\ T_n \\ \quad \textbf{valid}\ x := e_1\ \textbf{sync}\ (x_1, x_2) := e_2\ \textbf{endtype}) \\ \qquad \Rightarrow (S \Rightarrow \textbf{type}, c_1 \Rightarrow T_1 \to S, \ldots, c_n \Rightarrow T_n \to S) \end{array}}$$

4.4.3 Option 3: Dynamic semantics

The dynamic semantics we will give is rather different from that in Section 3.3. In that semantics, there is no distinction between input and output, but for abstract datatypes there is a great difference between them.

We do this by decorating the actions a process can perform with !'s and ?'s indicating whether the data was being output or input. We replace the actions in Section 3.3 by:

$$\begin{array}{rcl} a & ::= & i \\ & | & gm \\ & | & \delta m \end{array}$$

where m ranges over *messages*:

$$\begin{array}{rcl} m & ::= & cm \\ & | & \{l_1 = m_1, \ldots, l_n = m_n\} \\ & | & !v \\ & | & ?v \end{array}$$

We view messages up to the equivalence given by:

$$\begin{array}{rcl} \{l_1 = !(v_1 : T_1), \ldots, l_n = !(v_n : T_n)\} & = & !(\{l_1 = v_1, \ldots, l_n = !v_n\} : \{l_1 : T_1, \ldots, l_n : T_n\}) \\ \{l_1 = ?(v_1 : T_1), \ldots, l_n = ?(v_n : T_n)\} & = & ?(\{l_1 = v_1, \ldots, l_n = !v_n\} : \{l_1 : T_1, \ldots, l_n : T_n\}) \end{array}$$

The dynamic semantics extends that of Section 2.3 with judgements:

- $E \vdash B \xrightarrow{a} B'$ ‘In the environment E, B can perform an a action and reduce to B'.’
- $E \vdash (o \Rightarrow m) \Rightarrow \sigma$ ‘In the environment E, binding offer o to message m returns substitution σ.’
- $E \vdash$ **valid** $v \Rightarrow$ true ‘In the environment E, v is a valid value.’
- $E \vdash$ **valid** $m \Rightarrow$ true ‘In the environment E, m is a valid message.’
- $E \vdash$ **sync** $(v_1, v_2) \Rightarrow v$ ‘In the environment E, values v_1 and v_2 can synchronize together to produce value v.’
- $E \vdash$ **sync** $(m_1, m_2) \Rightarrow m$ ‘In the environment E, messages m_1 and m_2 can synchronize together to produce message m.’

We have to extend the environments to include the **valid** and **sync** declarations:

$$\begin{array}{rcl} E & ::= & \ldots \text{as in Section 3.3} \ldots \\ & | & \textbf{valid}[S] \Rightarrow \lambda x \,.\, e \\ & | & \textbf{sync}[S] \Rightarrow \lambda(x_1, x_2) \,.\, e \end{array}$$

For example, the environment generated by the ‘NatSet’ declaration is:

$$\begin{array}{rcl} \text{nil} & \Rightarrow & \text{NatList} \\ \text{cons} & \Rightarrow & \text{Nat} * \text{NatList} \rightarrow \text{NatList} \\ \text{hidden} & \Rightarrow & \text{NatList} \rightarrow \text{NatSet} \\ \textbf{valid}[\text{NatList}] & \Rightarrow & \lambda x \,.\, \text{true} \\ \textbf{sync}[\text{NatList}] & \Rightarrow & \lambda(x_1, x_2)[x_1 = x_2] \,.\, x_1 \\ \textbf{valid}[\text{NatSet}] & \Rightarrow & \lambda x \,.\, \text{sorted } x \\ \textbf{sync}[\text{NatSet}] & \Rightarrow & \lambda(x_1, x_2) \,.\, x_1 \text{ combine } x_2 \\ & \vdots & \end{array}$$

Dynamic semantics of declarations. We extend the semantics of Section 3.3.2 to take account of the **valid** and **sync** declarations:

$$\begin{array}{l} \hline E \vdash (\textbf{datatype}\ S := c_1\ \textbf{of}\ T_1 \mid \cdots \mid c_n\ \textbf{of}\ T_n \\ \quad \textbf{valid}\ x := e_1\ \textbf{sync}\ (x_1, x_2) := e_2\ \textbf{endtype}) \\ \qquad \Rightarrow (c_1 \Rightarrow T_1 \rightarrow S, \ldots, c_n \Rightarrow T_n \rightarrow S, \textbf{valid}[S] \Rightarrow \lambda x \,.\, e_1, \textbf{sync}[S] \Rightarrow \lambda(x_1, x_2) \,.\, e_2) \end{array}$$

Dynamic semantics of behaviours. This is the same as in Section 3.3.3, except for the following rules for termination, communication, synchronization, hiding, choice and sequential composition:

$$\frac{E \vdash (o \Rightarrow m) \Rightarrow \sigma}{E \vdash \textbf{exit}\ o \xrightarrow{\delta m} \textbf{stop}} \qquad \frac{\begin{array}{l} E \vdash (o \Rightarrow m) \Rightarrow \sigma \\ E \vdash e[\sigma] \Rightarrow \text{true} \end{array}}{E \vdash go[e]; B \xrightarrow{gm} B[\sigma]}$$

$$\frac{\begin{array}{l} E \vdash B_1 \xrightarrow{gm_1} B_1' \qquad E \vdash B_2 \xrightarrow{gm_2} B_2' \\ E \vdash \textbf{sync}\ (m_1, m_2) \Rightarrow m \\ g \in \vec{g} \end{array}}{E \vdash B_1 \,|[\vec{g}]|\, B_2 \xrightarrow{gm} B_1' \,|[\vec{g}]|\, B_2'} \qquad \frac{\begin{array}{l} E \vdash B_1 \xrightarrow{\delta v_1} B_1' \qquad E \vdash B_2 \xrightarrow{\delta v_2} B_2' \\ E \vdash \textbf{sync}\ (v_1, v_2) \Rightarrow v \end{array}}{E \vdash B_1 \,|[\vec{g}]|\, B_2 \xrightarrow{\delta v} B_1' \,|[\vec{g}]|\, B_2'}$$

$$\frac{\begin{array}{l} E \vdash B \xrightarrow{gm} B' \\ E \vdash \textbf{valid}\ m \Rightarrow \text{true} \end{array}}{E \vdash \textbf{hide}\ g : \textbf{gate}\ T\ \textbf{in}\ B \xrightarrow{i} \textbf{hide}\ g : \textbf{gate}\ T\ \textbf{in}\ B'}$$

$$\frac{\begin{array}{l} E \vdash (p \Rightarrow v) \Rightarrow \sigma \\ E \vdash \textbf{valid}\ v \Rightarrow \text{true} \\ E \vdash B[\sigma] \xrightarrow{a} B' \end{array}}{E \vdash \textbf{choice}\ p\,[\,]\,B \xrightarrow{a} B'} \qquad \frac{\begin{array}{l} E \vdash B_1 \xrightarrow{\delta m_1} B_1' \\ E \vdash (?p \Rightarrow m_2) \Rightarrow \sigma \\ E \vdash \textbf{sync}\ (m_1, m_2) \Rightarrow m \\ E \vdash \textbf{valid}\ m \Rightarrow \text{true} \end{array}}{E \vdash B_1 \gg \textbf{accept}\ p\ \textbf{in}\ B_2 \xrightarrow{i} B_2[\sigma]}$$

Dynamic semantics of offers. The new dynamic semantics of offers is given by inference rules:

$$\frac{E \vdash e \Rightarrow v}{E \vdash (!e \Rightarrow !v) \Rightarrow ()} \qquad \frac{E \vdash (p \Rightarrow v) \Rightarrow \sigma}{E \vdash (?p \Rightarrow ?v) \Rightarrow \sigma} \qquad \frac{E \vdash (o \Rightarrow m) \Rightarrow \sigma}{E \vdash (co \Rightarrow cm) \Rightarrow \sigma}$$

$$\frac{E \vdash (o_1 \Rightarrow m_1) \Rightarrow \sigma_1 \quad \cdots \quad E \vdash (o_n \Rightarrow m_n) \Rightarrow \sigma_n}{E \vdash (\{l_1 = o_1, \ldots, l_n = o_n\} \Rightarrow \{l_1 = m_1, \ldots, l_n = m_n\}) \Rightarrow (\sigma_1, \ldots, \sigma_n)}$$

Dynamic semantics of value validation. The semantics of value validation is to apply the user-specifi ed **valid** function recursively:

$$\frac{\begin{array}{l} E \vdash \textbf{valid}\ v \Rightarrow \text{true} \\ E \vdash c \Rightarrow T \to S \\ E \vdash \textbf{valid}[S] \Rightarrow \lambda x \,.\, e \\ E \vdash e[x := cv] \Rightarrow \text{true} \end{array}}{E \vdash \textbf{valid}\ cv \Rightarrow \text{true}} \qquad \frac{E \vdash \textbf{valid}\ v_1 \Rightarrow \text{true} \quad \cdots \quad E \vdash \textbf{valid}\ v_n \Rightarrow \text{true}}{E \vdash \textbf{valid}\ \{l_1 = v_1, \ldots, l_n = v_n\} \Rightarrow \text{true}}$$

Dynamic semantics of message validation. We can inductively defi ne message validation from value validation:

$$\frac{}{E \vdash \textbf{valid}\ !v \Rightarrow \text{true}} \qquad \frac{E \vdash \textbf{valid}\ v \Rightarrow \text{true}}{E \vdash \textbf{valid}\ ?v \Rightarrow \text{true}} \qquad \frac{E \vdash \textbf{valid}\ m \Rightarrow \text{true}}{E \vdash \textbf{valid}\ cm \Rightarrow \text{true}}$$

$$\frac{E \vdash \textbf{valid}\ m_1 \Rightarrow \text{true} \quad \cdots \quad E \vdash \textbf{valid}\ m_n \Rightarrow \text{true}}{E \vdash \textbf{valid}\ \{l_1 = m_1, \ldots, l_n = m_n\}) \Rightarrow \text{true}}$$

Note that output messages do *not* need to be validated, only input messages. This means that a protocol which ensures that every input is matched by at least one output will never need to call the **valid** functions.

Dynamic semantics of value synchronization. The semantics of value synchronization is to apply the user-specified **sync** function on sorts, and to recursively descend through records:

$$\frac{E \vdash c_1 \Rightarrow T \to S \qquad E \vdash c_2 \Rightarrow T \to S \qquad E \vdash \mathbf{sync}[S] \Rightarrow \lambda(x_1, x_2) \,.\, e \qquad E \vdash e[x_1 := c_1 v_1, x_2 := c_2 v_2] \Rightarrow v}{E \vdash \mathbf{sync}(c_1 v_1, c_2 v_2) \Rightarrow v}$$

$$\frac{E \vdash \mathbf{sync}(v_1, v_1') \Rightarrow v_1'' \quad \cdots \quad E \vdash \mathbf{sync}(v_n, v_n') \Rightarrow v_n''}{E \vdash \mathbf{sync}(\{l_1 = v_1, \ldots, l_n = v_n\}, \{l_1 = v_1', \ldots, l_n = v_n'\}) \Rightarrow \{l_1 = v_1'', \ldots, l_n = v_n''\}}$$

Dynamic semantics of message synchronization. We can inductively define message validation from value validation:

$$\frac{E \vdash \mathbf{sync}(v, v') \Rightarrow v''}{E \vdash \mathbf{sync}(!v, !v') \Rightarrow !v''} \qquad \frac{}{E \vdash \mathbf{sync}(?v, !v) \Rightarrow !v}$$

$$\frac{}{E \vdash \mathbf{sync}(!v, ?v) \Rightarrow !v} \qquad \frac{}{E \vdash \mathbf{sync}(?v, ?v) \Rightarrow ?v}$$

$$\frac{E \vdash \mathbf{sync}(m, !v) \Rightarrow m'}{E \vdash \mathbf{sync}(cm, !cv) \Rightarrow cm'} \qquad \frac{E \vdash \mathbf{sync}(m, ?v) \Rightarrow m'}{E \vdash \mathbf{sync}(cm, ?cv) \Rightarrow cm'}$$

$$\frac{E \vdash \mathbf{sync}(!v, m) \Rightarrow m'}{E \vdash \mathbf{sync}(!cv, cm) \Rightarrow cm'} \qquad \frac{E \vdash \mathbf{sync}(?v, m) \Rightarrow m'}{E \vdash \mathbf{sync}(?cv, cm) \Rightarrow cm'}$$

$$\frac{E \vdash \mathbf{sync}(m, m') \Rightarrow m''}{E \vdash \mathbf{sync}(cm, cm') \Rightarrow cm''}$$

$$\frac{E \vdash \mathbf{sync}(m_1, m_1') \Rightarrow m_1'' \quad \cdots \quad E \vdash \mathbf{sync}(m_n, m_n') \Rightarrow m_n''}{E \vdash \mathbf{sync}(\{l_1 = m_1, \ldots, l_n = m_n\}, \{l_1 = m_1', \ldots, l_n = m_n'\}) \Rightarrow \{l_1 = m_1'', \ldots, l_n = m_n''\}}$$

Note that input messages do *not* need to be synchronized, only output messages. This means that a protocol which ensures that processes will never try to synchronize on output will never need to call the **sync** functions.

5 Gates as first-class citizens

As an example of including abstract datatypes into specifications, we will show how gates can be regarded as ‘first class citizens’—thus increasing the expressive power of LOTOS, as discussed in [2].

We will show two ways in which this can be done: one based on Option 2 from the previous section, which does not allow gates to be communicated; the other based on Option 3 from the previous section, which allows gates to be communicated, thus gaining the expressive power of the π-calculus.

5.1 Common ground between the 2 options

The two possibilities have the same syntax and (almost) the same static semantics. Both of them simplify the previous syntax by abolishing the difference between gate identifiers and identifiers of any other type. For example, a process which spawns a set of one-place buffer processes can be defined:

datatype
  InOutList := nil | cons **of** (Nat **gate** * Nat **gate**) * InOutList
endtype
process
    Cell (in,out) := in?(x:Nat); out!x; Cell (in,out)
endproc
process
    Cells nil := **stop**
  | Cells ((in,out) cons gs) := Cell (in,out) ||| Cells gs
endproc

This has type:

$$\begin{array}{rcl} \text{InOutList} & \Rightarrow & \textbf{type} \\ \text{nil} & \Rightarrow & \{\} \to \text{InOutList} \\ \text{cons} & \Rightarrow & (\text{Nat } \textbf{gate} * \text{Nat } \textbf{gate}) \to \text{InOutList} \\ \text{Cell} & \Rightarrow & \text{Nat } \textbf{gate} * \text{Nat } \textbf{gate} \to \textbf{noexit} \\ \text{Cells} & \Rightarrow & \text{InOutList} \to \textbf{noexit} \end{array}$$

Although this uses a different syntax for pattern-matching gate parameters, the existing syntax (using square brackets) can be maintained for backward compatibility.

5.1.1 Syntax

We combine the syntactic categories ‘Gate’ and ‘Var’ (although for clarity we will continue to use g to range over variables of gate type).

We add a new type constructor for gates:

$$\begin{array}{rcl} T & ::= & \ldots\text{as in Section 2.1}\ldots \\ & | & \textbf{gate}\ T \end{array}$$

We simplify process specification by unifying gate- and data-parameters into one pattern:

$$\begin{array}{rcl} D & ::= & \ldots\text{as in Section 3.1, except}\ldots \\ & | & \textbf{process}\ Pp : \textbf{exit}\ T := B\ \textbf{endproc} \end{array}$$

and similarly we can simplify process instantiation by combining gate- and data-instantiation:

$$\begin{array}{rcl} B & ::= & \ldots\text{as in Section 3.1, except}\ldots \\ & | & Pe \end{array}$$

An extension we will not investigate for the moment is allowing gate expressions rather than gate identifiers, for example:

```
process Split (in,even,odd) :=
  in?x;
  (if iseven x then even else odd endif)!x ;
  Split (in,even,odd)
endproc
```

Such processes do not add any expressive power to the language (since they can be coded up using appropriate **let** statements) and add to its complexity.

5.1.2 Static semantics

The static semantics for the language with gates as first-class citizens is simpler than that in Section 3.2.

Contexts are:

$$\begin{array}{rcl} C & ::= & \ldots \text{as in Section 2.2} \ldots \\ & | & P \Rightarrow T_1 \rightarrow \textbf{exit } T_2 \end{array}$$

The static semantics for the new process declarations and instantiation are simpler than in Section 3.2:

$$\frac{\begin{array}{l} C \vdash D \Rightarrow C \\ P \Rightarrow T \rightarrow \textbf{exit } T' \end{array}}{\vdash (\textbf{specification } P \textbf{ where } D \textbf{ endspec}) \Rightarrow (P \Rightarrow T \rightarrow \textbf{exit } T')}$$

$$\frac{\begin{array}{l} C \vdash (p \Rightarrow T) \Rightarrow C' \\ C \oplus C' \vdash B \Rightarrow \textbf{exit } T' \end{array}}{C \vdash (\textbf{process } Pp : \textbf{exit } T' := B \textbf{ endproc}) \Rightarrow (P \Rightarrow T \rightarrow \textbf{exit } T')}$$

$$\frac{\begin{array}{l} C \vdash P \Rightarrow (T \rightarrow \textbf{exit } T') \\ C \vdash e \Rightarrow T \end{array}}{C \vdash Pe \Rightarrow \textbf{exit } T'}$$

5.1.3 Dynamic semantics

The dynamic semantics for the extended functional language is the same as in Section 2.3, with the addition of a new normal form for gates:

$$\begin{array}{rcl} v & ::= & \ldots \text{as in Section 2.3} \ldots \\ & | & g \end{array}$$

The environment a reduction is carried out in is enriched with the names of the free gates:

$$\begin{array}{rcl} E & ::= & \ldots \text{as in Section 3.3} \ldots \\ & | & g \Rightarrow \textbf{gate } T \end{array}$$

The dynamic semantics for data expressions containing gates then just needs a rule saying that if a gate is free then it is in normal form:

$$\frac{E \vdash g \Rightarrow \textbf{gate}\ T}{E \vdash g \Rightarrow g}$$

The dynamic semantics for the new form of process instantiation is the same as for function application:

$$\frac{\begin{array}{l} E \vdash P \Rightarrow \lambda p\,.\,B \\ E \vdash e \Rightarrow v \\ E \vdash (p \Rightarrow v) \Rightarrow \sigma \\ E \vdash B[\sigma] \xrightarrow{a} B' \end{array}}{E \vdash Pe \xrightarrow{a} B'}$$

5.2 Option 1: Non-mobile gates

The semantics for non-mobile gates is simple: we just add one type rule to the semantics in Section 4.3.2. This type rule says that any data communicated on a gate must be concrete:

$$\frac{C \vdash T \Rightarrow \textbf{concrete}}{C \vdash \textbf{gate}\ T \Rightarrow \textbf{type}}$$

Note that we do *not* add a rule saying that gates are concrete types. This stops them being communicated between processes, so giving a non-mobile semantics.

5.3 Option 2: Mobile gates

The static semantics for mobile gates is also simple: we allow any type to be used in gates (including other gates, for example):

$$\frac{C \vdash T \Rightarrow \textbf{type}}{C \vdash \textbf{gate}\ T \Rightarrow \textbf{type}}$$

The dynamic semantics, however, is more complex, since we have to deal with π-calculus scope extrusion, for example:

$$\begin{array}{c} \textbf{hide}\ g\ \textbf{in}\ (g?x; B \,|[g]|\, \textbf{hide}\ h\ \textbf{in}\ g!h; B') \\ \xrightarrow{i} \textbf{hide}\ g, h\ \textbf{in}\ (B[x := h] \,|[g]|\, B') \end{array}$$

As in the lts semantics for the π-calculus [4], we allow labels to contain the names of hidden gates, for example:

$$\begin{array}{c} \textbf{hide}\ h\ \textbf{in}\ g!h; B' \\ \xrightarrow{g(\textbf{hide}\ h\ \textbf{in}\ !h)} B' \end{array}$$

This gives the syntax of messages as being:

$$\begin{array}{rcl} m & ::= & \ldots\text{as in Section 4.4.3}\ldots \\ & | & \textbf{hide}\ g : \textbf{gate}\ T\ \textbf{in}\ m \end{array}$$

We view messages up to the structural congruence for scope extrusion (see, for example, [3]):

$$\begin{aligned}&\{l_1 = m_1, \ldots, l_i = \textbf{hide}\ g : \textbf{gate}\ T\ \textbf{in}\ m_i, \ldots, l_n = m_n\}\\&\qquad = \textbf{hide}\ g : \textbf{gate}\ T\ \textbf{in}\ \{l_1 = m_1, \ldots, l_i = m_i, \ldots, l_n = m_n\}\\&c(\textbf{hide}\ g : \textbf{gate}\ T\ \textbf{in}\ m)\\&\qquad = \textbf{hide}\ g : \textbf{gate}\ T\ \textbf{in}\ cm\end{aligned}$$

with appropriate uses of α-conversion to avoid capturing free names. Define the syntactic sugar:

$$\begin{aligned}&\textbf{hide}\ g_1, \ldots, g_n : \textbf{gate}\ T_1, \ldots, T_n\ \textbf{in}\ m\\&\qquad = \textbf{hide}\ g_1 : \textbf{gate}\ T_1\ \textbf{in}\ \cdots\ \textbf{hide}\ g_n : \textbf{gate}\ T_n\ \textbf{in}\ m\\&\textbf{hide}\ g_1, \ldots, g_n : \textbf{gate}\ T_1, \ldots, T_n\ \textbf{in}\ B\\&\qquad = \textbf{hide}\ g_1 : \textbf{gate}\ T_1\ \textbf{in}\ \cdots\ \textbf{hide}\ g_n : \textbf{gate}\ T_n\ \textbf{in}\ B\end{aligned}$$

We can then extend the dynamic semantics of Section 4.4.3 to cope with gate mobility.

Since communication can now output fresh gates using scope extrusion, computations have to extend their scope. A computation of a behaviour is therefore a series of reductions:

$$E_1 \vdash B_1 \xrightarrow{a_1} B_2 \quad \cdots \quad E_n \vdash B_n \xrightarrow{a_n} B_{n+1}$$

where:

$$E_{i+1} = \begin{cases} E_i, \vec{g} : \textbf{gate}\ \vec{T} & \text{if } a_i = g(\textbf{hide}\ \vec{g} : \textbf{gate}\ \vec{T}\ \textbf{in}\ m) \text{ and } m \text{ contains no } \textbf{hide}\text{s} \\ E_i & \text{otherwise} \end{cases}$$

we can write such a computation as:

$$E_1 \vdash B_1 \xrightarrow{a_1} B_2 \xrightarrow{a_2} \cdots \xrightarrow{a_n} B_{n+1}$$

For example, one possible computation is:

$$\begin{aligned} g : \textbf{gate gate}\ \text{Nat} \ \vdash\ & (\textbf{hide}\ h : \textbf{gate}\ \text{Nat}\ \textbf{in}\ g!h; h!0; \textbf{stop})\\ & \xrightarrow{g(\textbf{hide}\ h:\textbf{gate}\ \text{Nat}\ \textbf{in}\ !h)} (h!0; \textbf{stop})\\ & \xrightarrow{h!0} \textbf{stop}\end{aligned}$$

since:

$$\begin{aligned} g : \textbf{gate gate}\ \text{Nat} \ \vdash\ & (\textbf{hide}\ h : \textbf{gate}\ \text{Nat}\ \textbf{in}\ g!h; h!0; \textbf{stop})\\ & \xrightarrow{g(\textbf{hide}\ h:\textbf{gate}\ \text{Nat}\ \textbf{in}\ !h)} (h!0; \textbf{stop})\\ g : \textbf{gate gate}\ \text{Nat}, h : \textbf{gate}\ \text{Nat} \ \vdash\ & (h!0; \textbf{stop})\\ & \xrightarrow{h!0} \textbf{stop}\end{aligned}$$

Dynamic semantics of behaviours We add a condition to the dynamic semantics for communication in Section 4.4.3 to ensure that any communication is on a free gate:

$$\frac{\begin{array}{l}E \vdash g \Rightarrow \textbf{gate}\ T\\ E \vdash (o \Rightarrow m) \Rightarrow \sigma\\ E \vdash e[\sigma] \Rightarrow \text{true}\end{array}}{E \vdash go[e]; B \xrightarrow{gm} B[\sigma]}$$

We replace the dynamic semantics for hiding in Section 4.4.3 with rules which incorporate scope extrusion:

$$\frac{E, g \Rightarrow \textbf{gate}\ T \vdash B \xrightarrow{a} B' \qquad g \notin a}{E \vdash \textbf{hide}\ g : \textbf{gate}\ T\ \textbf{in}\ B \xrightarrow{a} \textbf{hide}\ g : \textbf{gate}\ T\ \textbf{in}\ B'}$$

$$\frac{E, g \Rightarrow \textbf{gate}\ T \vdash B \xrightarrow{g'm} B' \qquad g \neq g' \qquad g \in m}{E \vdash \textbf{hide}\ g : \textbf{gate}\ T\ \textbf{in}\ B \xrightarrow{g'(\textbf{hide}\ g:\textbf{gate}\ T\ \textbf{in}\ m)} B'}$$

$$\frac{E, g \Rightarrow \textbf{gate}\ T \vdash B \xrightarrow{\delta m} B' \qquad g \in m}{E \vdash \textbf{hide}\ g : \textbf{gate}\ T\ \textbf{in}\ B \xrightarrow{\delta(\textbf{hide}\ g:\textbf{gate}\ T\ \textbf{in}\ m)} B'}$$

$$\frac{E, g \Rightarrow \textbf{gate}\ T \vdash B \xrightarrow{g(\textbf{hide}\ \vec{g}:\textbf{gate}\ \vec{T}\ \textbf{in}\ m)} B' \qquad E, g \Rightarrow \textbf{gate}\ T \vdash \textbf{valid}\ m \Rightarrow \text{true}}{E \vdash \textbf{hide}\ g : \textbf{gate}\ T\ \textbf{in}\ B \xrightarrow{i} \textbf{hide}\ g : \textbf{gate}\ T, \vec{g} : \textbf{gate}\ \vec{T}\ \textbf{in}\ B'}$$

We replace the sequential composition rule for **accept** in Section 4.4.3 with a rules which includes scope extrusion:

$$\frac{E \vdash B_1 \xrightarrow{\delta m_1} B_1' \qquad E \vdash (?p \Rightarrow m_2) \Rightarrow \sigma \qquad E \vdash \textbf{sync}\ (m_1, m_2) \Rightarrow \textbf{hide}\ \vec{g} : \textbf{gate}\ \vec{T}\ \textbf{in}\ m \qquad E \vdash \textbf{valid}\ m \Rightarrow \text{true}}{E \vdash B_1 \gg \textbf{accept}\ p\ \textbf{in}\ B_2 \xrightarrow{i} \textbf{hide}\ \vec{g} : \textbf{gate}\ \vec{T}\ \textbf{in}\ B_2[\sigma]}$$

Dynamic semantics of validation We do not add any extra rules for validation. In particular, this means that *no* value message containing gates is valid. This is very important, as it stops behaviours from being able to 'guess' the names of gates which are outside their scope.

Dynamic semantics of synchronization The dynamic semantics for synchronization of gates is given by inference rules:

$$\frac{E \vdash g \Rightarrow \textbf{gate}\ T}{E \vdash \textbf{sync}\ (g, g) \Rightarrow g}$$

$$\frac{E, g \Rightarrow \textbf{gate}\ T \vdash \textbf{sync}\ (m_1, m_2) \Rightarrow m}{E \vdash \textbf{sync}\ (\textbf{hide}\ g : \textbf{gate}\ T\ \textbf{in}\ m_1, m_2) \Rightarrow \textbf{hide}\ g : \textbf{gate}\ T\ \textbf{in}\ m}$$

$$\frac{E, g \Rightarrow \textbf{gate}\ T \vdash \textbf{sync}\ (m_1, m_2) \Rightarrow m}{E \vdash \textbf{sync}\ (m_1, \textbf{hide}\ g : \textbf{gate}\ T\ \textbf{in}\ m_2) \Rightarrow \textbf{hide}\ g : \textbf{gate}\ T\ \textbf{in}\ m}$$

This allows behaviours to synchronize on gates they have in common, and to exchange new gates, for example:

$$\begin{aligned} &\textbf{hide}\ g\ \textbf{in}\ ((\textbf{hide}\ h_1\ \textbf{in}\ g(!h, !h_1, ?x_2); B_1)\ |[g]|\ (\textbf{hide}\ h_2\ \textbf{in}\ g(!h, ?x_1, !h_2); B_2)) \\ &\quad \xrightarrow{i} \textbf{hide}\ g, h_1, h_2\ \textbf{in}\ (B_1[x_2 := h_2]\ |[g]|\ B_2[x_1 := h_1]) \end{aligned}$$

Encoding the π-calculus This semantics is powerful enough to encode the π-calculus, although not in a very pleasant way. The coding is given by introducing a new type for one-to-many signals:

datatype OneToMany := yes
valid x := false
sync (x,y) := **error**
endtype

Since these signals cannot be ‘guessed’ or synchronized, any communication involving the sending of a OneToMany signal must be one-to-many. These can be used to code up a monadic π-calculus expression using a global channel ‘it’:

π-calculus term	*translation*
0	**stop**
$B \mid B'$	$B \,|[\mathrm{it}]|\, B'$
$x!y\,.\,B$	$\mathrm{it}\{\mathrm{chan} = !x, \mathrm{data} = !y, \mathrm{send} = !\mathrm{yes}, \mathrm{ack} = ?\mathbf{any}\}; B$
$x?y\,.\,B$	$\mathrm{it}\{\mathrm{chan} = !x, \mathrm{data} = ?y, \mathrm{send} = ?\mathbf{any}, \mathrm{ack} = !\mathrm{yes}\}; B$
$\nu x\,.\,B$	**hide** x : **gate** None **in** B

This translation works by using gates just as capabilities: all communication happens along the gate ‘it’, and the other gates are just used to control which synchronizations can happen. Since one-to-many send and acknowledge channels are used, communication is guaranteed to be one-to-one.

This translation is not very pleasant, since it puts all communication onto one global channel. For example the π-calculus reduction:

$$\nu x\,.\,((\nu y\,.\,x!y\,.\,B_1) \mid x?z\,.\,B_2) \xrightarrow{\tau} \nu x, y\,.\,(B_1 \mid B_2[z := y])$$

becomes the LOTOS reduction:

$$\begin{aligned}&\mathbf{hide}\ x\ \mathbf{in}\ (\mathbf{hide}\ y\ \mathbf{in}\ (\mathrm{it}\{\mathrm{chan} = !x, \mathrm{data} = !y, \mathrm{send} = !\mathrm{yes}, \mathrm{ack} = ?\mathrm{yes}\}; B_1)\\ &\quad |[\mathrm{it}]|\ \mathrm{it}\{\mathrm{chan} = !x, \mathrm{data} = ?z, \mathrm{send} = ?\mathrm{yes}, \mathrm{ack} = !\mathrm{yes}\}; B_2)\\ &\quad\xrightarrow{\mathrm{it}\ \mathbf{hide}\ x,y\ \mathbf{in}\ \{\mathrm{chan}=!x,\mathrm{data}=!y,\mathrm{send}=!\mathrm{yes},\mathrm{ack}=!\mathrm{yes}\}} B_1 \,|[\mathrm{it}]|\, B_2[z := y]\end{aligned}$$

A better translation can be given if we are allowed to introduce a new concurrency operator $B_1 \mid B_2$ with the operational semantics:

$$\frac{B_1 \,||\, B_2 \xrightarrow{a} B_1' \,||\, B_2'}{B_1 \mid B_2 \xrightarrow{a} B_1' \mid B_2'} \qquad \frac{B_1 \,|||\, B_2 \xrightarrow{a} B_1' \,|||\, B_2'}{B_1 \mid B_2 \xrightarrow{a} B_1' \mid B_2'}$$

We then translate the monadic π-calculus as:

π-calculus term	*translation*
0	**stop**
$B \mid B'$	$B \mid B'$
$x!y\,.\,B$	$x\{\mathrm{send} = !\mathrm{name}\ y, \mathrm{ack} = ?\mathrm{yes}\}; B$
$x?y\,.\,B$	$x\{\mathrm{send} = ?\mathrm{name}\ y, \mathrm{ack} = !\mathrm{yes}\}; B$
$\nu x\,.\,B$	**hide** x : **gate** $\{\mathrm{send} : \mathrm{Name}, \mathrm{ack} : \mathrm{OneToMany}\}$ **in** B

using the datatype 'Name' defi ned:

datatype Name := name **of gate** { send : Name, ack : OneToMany }
valid x := false
sync (x,y) := **error**
endtype

For example the π-calculus reduction:

$$\begin{aligned} &\nu x\,.\,((\nu y\,.\,x!y\,.\,B_1) \mid x?z\,.\,B_2) \\ &\quad \xrightarrow{\tau} \nu x, y\,.\,(B_1 \mid B_2[z := y]) \end{aligned}$$

becomes the LOTOS reduction:

$$\begin{aligned} &\mathbf{hide}\ x\ \mathbf{in}\ (\mathbf{hide}\ y\ \mathbf{in}\ (x\{\text{send} = !\text{name}\ y, \text{ack} = ?\text{yes}\}; B_1) \\ &\quad \mid x\{\text{send} = ?\text{name}\ y, \text{ack} = !\text{yes}\}; B_2) \\ &\qquad \xrightarrow{i} \mathbf{hide}\ x, y\ \mathbf{in}\ (B_1 \mid B_2[z := y]) \end{aligned}$$

We conjecture that this translation is adequate (although it might not be fully abstract).

6 Conclusions

In this paper we have presented semantics for LOTOS with functional data, presenting the static semantics for LOTOS in a style which will be familiar to designers of functional languages.

We investigated data abstraction, and found that it clashes with the existing semantics of communication in LOTOS, and suggested three possible approaches to rectify the problem. Two of the approaches allow for gates to be treated as data, which makes the semantics much simpler, and allows a greater expressive power.

Further work to be carried out includes: deciding which semantics for data abstraction to adopt; integrating data abstraction with modularization; and investigating polymorphism, overloading, subtyping, and other variants to the type system.

References

[1] Hubert Garavel. Six improvements to the process part of LOTOS. In *Working Draft on Enhancements to LOTOS*, ISO/IEC JTC1/SC21/WG1 N1349, Annexe K. 1994.

[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, Annexe A. 1995.

[3] Robin Milner. Functions as processes. *Math. Struct. in Comput. Science*, 2:119–141, 1992.

[4] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile proceses. *Inform. and Comput.*, 100(1):1–77, 1992.

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