E-LOTOS core language

Alan Jeffrey
University of Sussex

Guy Leduc
University of Liège

1996/09/20

Abstract

This paper presents an integrated core data and behaviour language for the new LOTOS standard. It is not intended to be directly usable for specifications, but some additional syntax sugar can be defined to make it more usable and compatible with existing specifications. The language is first-order, monomorphic, strongly typed and allows subtyping. It supports concurrency, real-time, exception handling, pattern-matching and some imperative features.

Contents

1 Introduction

The ISO formal language LOTOS [1, 6] is composed of a process algebra part (based on CCS [10] and CSP [4]) to describe behaviours, and an algebraic language (ACT ONE [3]) to describe the abstract data types. This language is mathematically well-defined and expressive: it allows the description of concurrency, nondeterminism, synchronous and asynchronous communications. It supports various levels of abstraction and provides several specification styles. Good tools exist to support specification, verification and code generation. Despite these positive features, this language is currently under revision in ISO [11] because feedback from users has indicated that the usefulness of the language is limited by certain characteristics relating both to technical capabilities and user-friendliness of the language.

Two main enhancements address datatypes and time. There is no notion of quantitative time in standard LOTOS, which precludes any precise description of real-time systems. Furthermore, the LOTOS algebraic datatypes are not user-friendly and suffer from several limitations such as the semi-decidability of equational specifications, the lack of modularity and the inability to define partial operations.

For example, a simple router of packets containing a data field and an address field might be defined in standard LOTOS:

```
process Router [in, left, right] : noexit :=
  in?p:packet;
  (
       [getdest(p) = L] → left!getdata(p);Router [in, left, right]
    [] [getdest(p) = R] → right!getdata(p);Router [in, left, right]
  )
endproc
```

This definition suffers from some problems of readability for non-LOTOS experts (for example the use of selection predicates and choice rather than a **case** construct) but is quite understandable compared to the definition of the packet datatype:

```
type Packet is
  sorts
    packet, dest
  opns
    mkpacket : dest, data → packet
    getdest : packet → dest
    getdata : packet → data
    L :  → dest
    R :  → dest
  eqns forall p:packet, de:dest, da:data
    ofsort packet mkpacket (getdest (p), getdata (p)) = p
    ofsort dest getdest (mkpacket (de, da)) = de
    ofsort data getdata (mkpacket (de, da)) = da
endtype
```

This can be compared with the equivalent process declaration in the core language presented here:

```
process Router [in(packet), left(data), right(data)] : exit (none) is
  local
    var p:packet
  in
    in(?p);
    case p.de is
        L → left(!p.da)
      | R → right(!p.da)
    endcase;
    Router [in, left, right]
  endloc
endproc
```

with the corresponding data type declarations:

```
type dest is L | R endtype
type packet is (de ⇒ dest, da ⇒ data) endtype
```

Note that:

- The gates in the Router process are explicitly typed.
- We can use field projection to access the fields of the packet, rather than using hand-crafted selection functions.
- The scope of the variables de and da are made explicit by a **local** variable declaration.
- The **case** statement is made explicit, rather than implicit using selection predicates and choice.
- We have moved the recursive call outside the **case** statement, avoiding the need to duplicate it.
- The definition of the 'dest' type as a union, and the 'packet' type as a record is made explicit, and much shorter.

The revised LOTOS language is a two-layer language. The higher layer is the user-level language, and addresses all the concerns related to the user-friendliness and expressive power of the language. The lower layer is the core-level language which we will present in this paper. The user-level language is mapped to the core-level language using a combination of syntax sugar (described in this paper) and static semantics (to resolve issues such as overloading, and not described in this paper).

The static and dynamic semantics of the core-level language is formally defined in this document. The static semantics is based on judgements such as $\mathcal{C} \vdash E \Rightarrow \textbf{exit}\ (T)$ meaning 'in context $\mathcal{C}$ expression E has result type T' for example:

$$1 \Rightarrow \text{float}, \text{x} \Rightarrow \text{float}, / \Rightarrow (\text{float}, \text{float}) \rightarrow \textbf{exit}\ (\text{float}) \vdash 1/\text{x} \Rightarrow \textbf{exit}\ (\text{float})$$

means 'in a context where 1 and x are floats, and / is a function from pairs of floats to floats, then the expression $1/\text{x}$ has result type float'. The static semantics includes:

- User-definable record, union types, and recursive types.
- Subtyping (for example we could allow integers as a subtype of floats).
- Imperative write-once variables, with a static semantics which ensures that every variable is written before read, and that shared variables cannot be used for communication between processes.
- Gates are explicitly typed (but we can use subtyping to provide the power of standard LOTOS untyped gates).

The dynamic semantics is based on judgements such as $\mathcal{E} \vdash E \xrightarrow{\alpha(N)} E'$ meaning 'in environment $\mathcal{E}$ expression E reduces (with action $\alpha(N)$) to E''. For expressions, possible values of α are an exception X or a successful termination action δ. For example the expression $1/2$ terminates with value 0.5:

$$\vdash 1/2 \xrightarrow{\delta(0.5)} \textbf{block}$$

and $1/0$ raises the exception Div:

$$\vdash 1/0 \xrightarrow{\text{Div}()} \textbf{block}$$

The dynamic semantics includes:

- Behaviours communicating on gates with other behaviours.
- Behaviours or expressions raising exceptions, which may be trapped by exception handlers.
- Behaviours with real-time semantics.

In fact, the semantics of expressions is given by treating expressions as a subclass of behaviours: expressions can only perform exception or termination actions, and cannot communicate on gates, or have any real-time behaviour. Unifying expressions and behaviours in this way allows for a much simpler and uniform semantics.

The language described in this paper is based on previous proposals for real-timed LOTOS [9] and LOTOS with functional datatypes [8, 7]. Many of the language features, especially the imperative features, are based on the proposed user-level language [5].

2 Basic concepts

2.1 Declarations

A specification in the core language is given as a sequence of *declarations* (future revisions will include a module system to structure these declarations, but for the moment we will think of them as a sequence).

These declarations come in three flavours: *type* declarations, *function* declarations, and *process* declarations. In the core language, all type and constructor identifiers must be unique—all treatment of overloading is left to the user language.

Type declarations A type declaration is either a *type synonym* or a *datatype* declaration. A type synonym declares a new type identifier for an existing type. For example we can declare a type 'point' synonymous with a record of floats as:

type point **is**
  $(\mathrm{x} \Rightarrow \mathrm{float}, \mathrm{y} \Rightarrow \mathrm{float})$
endtype

and we can declare a recursive data type of integer lists as:

type intlist **is**
    nil
  | cons(int, intlist)
endtype

Type synonyms can be used interchangably, for example the following declarations are the same:

type colpixel **is**
  $(\mathrm{pt} \Rightarrow \mathrm{point}, \mathrm{col} \Rightarrow \mathrm{colour})$
endtype
type colpixel$'$ **is**
  $(\mathrm{pt} \Rightarrow (\mathrm{x} \Rightarrow \mathrm{float}, \mathrm{y} \Rightarrow \mathrm{float}), \mathrm{col} \Rightarrow \mathrm{colour})$
endtype

We can use colpixel and colpixel$'$ as the same type (for example any function expecting a colpixel will accept a colpixel$'$). More succincltly, type equality is *structural* not by *name*.

Data type declarations define new types, listing all the *constructors* for that type. Since there can be more than one constructor, we can define *union* types, for example:

type pdu **is**
  send(packet, bit) | ack(bit)
endtype

It is possible to define recursive data types, such as the datatype of lists above.

The core language does *not* provide a mechanism for defining parameterized types—this is left for the module system.

Function declarations A function declaration defines a new function, which can be used in data expressions. For example:

function reflect (?p:point) : point **is**
  $(\mathrm{x} \Rightarrow \mathrm{p.y}, \mathrm{y} \Rightarrow \mathrm{p.x})$
endfun

The function parameters are given as a list of typed variables—in core E-LOTOS we always decorate binding occurrences of variables with ?. A function can have more than one input parameter, and can return a record of results, for example (we will fill in the details later):

```
function partition (?x : int, ?xs : intlist) : (intlist, intlist) is
  local
    var less : intlist, gtr : intlist
  init
    less := all of xs less than x;
    gtr := all of xs greater than x
  in
    (less, gtr)
  endloc
endfun
```

This function can be called (for example):

```
function quicksort (?xs : intlist) is
  case xs is
      nil →
        nil
    | cons(?y, ?ys) →
        local
          var l : intlist, g : intlist
        init
          (?l, ?g) := partition (y, ys)
        in
          append (quicksort (l), cons (y, quicksort (g)))
        endloc
  endcase
endfun
```

This style of function is very common, so we provide some syntax sugar for it, using out parameters. For example, the partition function could have been written:

```
function partition (in ?x : int, ?xs : intlist, out less : intlist, gtr : intlist) is
  less := all of xs less than x;
  gtr := all of xs greater than x
endfun
```

and then used in quicksort as:

```
partition (y, ys, ?l, ?g)
```

rather than:

```
(?l, ?g) := partition (y, ys)
```

It is possible to bind a variable to the entire argument list of a function—this is useful if the function is a wrapper to other functions, for example:

```
function F (?all as int, intlist) : intlist is
  F1 all;
  F2 all
endfun
```

has the same semantics as:

> **function** F (?x:int, ?xs:intlist) : intlist **is**
> F_1 (x, xs);
> F_2 (x, xs)
> **endfun**

By default, the whole argument list is bound to a special variable $argv, so we could have written:

> **function** F (int, intlist) : intlist **is**
> F_1 $argv;
> F_2 $argv
> **endfun**

Functions may raise exceptions (described below) which have to be declared, for example:

> **function** hd (?xs:intlist) : intlist **raises** [Hd] **is**
>   **case** xs **is**
>   nil → **raise** Hd
> | cons(?x, **any**) → x
>   **endcase**
> **endfun**

When such a function is called, the Hd exception is instantiated, for example the following will raise the exception Foo:

> hd (nil) [Foo]

Most often, we use the same exception name as in the declaration:

> hd (nil) [Hd]

This acts as a visual reminder that the hd function can raise the exception Hd.

Exceptions can be typed, for example:

> **function** foo () **raises** [Foo(string)] **is**
>   **raise** Foo("Hello world")
> **endfun**

Any untyped exceptions are assumed to have type ().

Note that in the core language, function declarations are just syntax sugar for a subclass of process declaration.

Process declarations Process declarations in the core language are very similar to function declarations: they have parameter lists, in and out parameters, result type (indicated with an **exit** annotation) and a list of typed exception parameters.

However, there are two important differences between functions and processes: processes can have real-time behaviour, and they can communicate on gates. For example, a simple counter process is defined:

> **process** Counter [up(), down()] **is**
>   up; (down ||| Counter [up, down])
> **endproc**

By default, gates have type (**etc**), which allows communication of arbitrary data, for compatibility with existing LOTOS.

Process behaviours are discussed further in Section 2.4.

2.2 Typing

Type expressions We have already seen a number of type expressions, for example:

- The data type intlist, and the type synonym point are both *type identifiers*.
- The type $(\mathrm{x} \Rightarrow \mathrm{float}, \mathrm{y} \Rightarrow \mathrm{float})$ is a *record type* with *fields* x and y.
- The type $(\mathrm{int}, \mathrm{intlist})$ is a *pair type*: in fact this is syntax sugar for the record type $(\$1 \Rightarrow \mathrm{int}, \$2 \Rightarrow \mathrm{intlist})$.
- The type $()$ is the trivial record with no fields.

Record types can be *extensible*, for example the type $(\mathrm{name} \Rightarrow \mathrm{string}, \mathbf{etc})$ is a record type with at least one field, but which can be extended to have others.

In addition to type identifiers and record types, we have two special types:

- The empty type **none** with no values, used to give the functionality of processes such as **stop** or **Counter** which never terminate.
- The universal type **any** which is a supertype of every other type, used to give a type for gates which can communicate data of any type, for compatibility with existing LOTOS.

Subtyping The core language supports *subtyping*, for example we could have integers as a subtype of floats. The built-in subtyping is on records: we allow a record type $(\mathbf{etc})$ which is a supertype of any other record. For example, the type $(\mathrm{name} \Rightarrow \mathrm{string}, \mathbf{etc})$ is a record with at least one field 'name' of type string. This record type can be extended to many subtypes, for example $(\mathrm{name} \Rightarrow \mathrm{string}, \mathrm{age} \Rightarrow \mathrm{int}, \mathbf{etc})$ or $(\mathrm{name} \Rightarrow \mathrm{string}, \mathrm{age} \Rightarrow \mathrm{int})$. Note the difference between these last two types: the former can be extended with further fields, where the latter cannot.

We include a special **none** type, which has no values. The type **none** is the most specialised type, and **any** is the most general type. Since a record type with a **none** field cannot have any values, we can identify it with **none**, for example the pair type $(\mathbf{none}, \mathrm{int})$ has no values, so is equivalent to the type **none**. This means that the one-element record type $(\mathbf{none})$ is the most specialized record type, and $(\mathbf{etc})$ is the most general.

For example, **stop** is a behaviour of type **exit** $(\mathbf{none})$, meaning that it will never terminate. Since $(\mathbf{none})$ is the least general record type, we can use **stop** wherever a process of any record type is required.

Similarly, if G is a gate of type $\mathbf{gate}(\mathbf{etc})$ then we can communicate values of any type along G—this is the same semantics as the existing untyped gates in standard LOTOS.

2.3 Data expressions

In contrast to standard LOTOS (which has a separation between processes and functions), the core language presented here considers functions to be restricted forms of processes (with no communication or real-time capabilities). The language of expressions is therefore very similar to the language of behaviours, and shares many features such as pattern-matching, exception raising and handling, and imperative features.

Normal forms A *normal form* is a data expression which cannot be reduced any further. For example $1+1$ is not in normal form, but 2 is. A normal form is one of the following:

- A primitive constant, such as `"Hello world"` or 2, for one of the built-in types. We will not consider any of the primitive constants further in this paper, and leave this until the standard libraries are to be defined.
- A variable, such as x or gtr.
- A record of normal forms, such as $(\mathrm{x} \Rightarrow 1.5, \mathrm{y} \Rightarrow -3.14)$, $()$ or $(5, \mathrm{nil}())$ (which is just syntax sugar for $(\$1 \Rightarrow 5, \$2 \Rightarrow \mathrm{nil}()))$.
- A constructor applied to a normal form, such as $\mathrm{nil}()$ or $\mathrm{cons}(5, \mathrm{nil}())$.

We will let N range over normal forms, and (RN) range over record normal forms.

Pattern-matching The expression language includes a **case** operation, which allows branching depending on the value of an expression, for example we can find the head of a list with:

> **case** xs **is**
>   nil $\rightarrow$ **raise** Hd
> | cons(?x,**any**) $\rightarrow$ x
> **endcase**

This case operation consists of a value to branch on (in this case xs) together with a list of possibilities, given by *patterns*. If the list is empty, then the first pattern will match, and the Hd exception will be raised. If the list is non-empty, then the second pattern will match, x will be *bound to* the head of the list, and will then be returned as the result.

Case expressions are evaluated by evaluating the expression to normal form, and then attempting to match the resuling value against each pattern from top to bottom until a match is found. If the value does not match any pattern (which cannot occur in the above example), a special Match exception is raised.

Note that cons(?x,**any**) is a structured pattern. At the highest level, we find the list constructor cons, built from a record pattern that includes the elementary patterns ?x and **any**. For a list to match this pattern, it has to have the form cons(hd,tl).

When a list matches the pattern cons(?x,**any**), the variable x is bound to the head of the list, for example producing the substitution [x $\Rightarrow$ hd]. Since substitutions have the same syntax as records, we will make a pun between record normal forms and substitutions.

We also allow expressions in patterns, which are evaluated when the pattern is matched, and match any value equal to the result. This is most often used to match against constants, for example:

> **case** x **is**
>   !0 $\rightarrow$ "zero"
> | **any** $\rightarrow$ "nonzero"
> **endcase**

Sometimes, it is useful to match against an expression, for example we can check to see if a list is a palindrome (using a function which reverses a list) with:

> **case** xs **is**
>   !reverse(xs) $\rightarrow$ "palindrome"
> | **any** $\rightarrow$ "nonpalindrome"
> **endcase**

The main use of matching against expressions is in communication, as we shall see in Section 2.4.

Patterns can be explicitly typed, which is useful in the presence of subtyping. For example, if int is a subtype of float, then we can construct a **case** statement to decide whether a value is an integer or not:

> **case** x:float **is**
>   **any**:int $\rightarrow$ "integer"
> | **any** $\rightarrow$ "noninteger"
> **endcase**

Again, the main use for explicitly typed patterns is in communication.

A pattern is one of the following:

- A bound variable, for example ?x.
- A free expression for example !0 or !reverse(xs).

- The wildcard pattern **any**.
- A record pattern, for example (x ⇒ ?px, y ⇒ ?py), (), or (?x, **any**) (which is just syntax sugar for ($1 ⇒ ?x, $2 ⇒ **any**)).
- An extensible record pattern, for example (x ⇒ ?px, **etc**), (**etc**), or (?x, **etc**) where **etc** is a pattern which matches any other fields. Note the difference between (?x, **any**) and (?x, **etc**): the former will only match tuples with two fields where the latter will match tuples with any (positive) number of fields.
- A record pattern with an **as** clause to bind part of the record, for example (?all **as** ?x, **etc**) or (?x, ?all **as etc**).
- A constructor applied to a pattern, for example nil() or cons(?x, **any**)
- An explicitly typed pattern, for example ?y : int.

It is easy to define operators such as **if**-statements as syntax sugar on top of the case operator, for example the expression:

if E **then** E_1 **else** E_2 **endif**

can be expanded to:

case E **is**
  true → E_1
| **any** → E_2
endcase

Exceptions Expressions can raise exceptions, in order to signal an error of some kind, for example when we attempt to take the head of an empty list:

function hd (?xs : intlist) : intlist **raises** [Hd] **is**
  case xs **is**
    nil → **raise** Hd
  | cons(?x, **any**) → x
  endcase
endfun

Exceptions either propogate to top level, or are *trapped* by an exception handler. For example we can declare a function:

function hd0 (?xs : intlist) : intlist **is**
  trap
    exception Hd **is** 0 **endexn**
  in
    hd (xs) [Hd]
  endtrap
endfun

Then hd0 (cons(a, as)) returns a, and hd0 (nil) returns 0, since the Hd exception raised by hd is trapped by the exception handler.

Exceptions can be typed, for example:

```
trap
  exception Error (?code:int) is
    case code is
        !0 → "minor error"
      | !1 → "major error"
      | any → raise Unknown (code)
    endcase
  endexn
in
  ...
endtrap
```

We can declare more than one exception in a single trap operator, for example:

```
trap
  exception Foo is E1 endexn
  exception Bar is E2 endexn
in
  E
endtrap
```

Note that Foo and Bar are only trapped in E, *not* in either E_1 or E_2. So if E raises Foo or Bar, then it will be handled, but if E_1 or E_2 raises Foo or Bar then it will not.

In addition, we can write a 'handler' for the successful termination of an expression, for example:

```
trap
  exception ParseError is 0 endexn
  exit (?x:string) is string2int (x) [ParseError] endexit
in
  E
endtrap
```

This is useful in the case where we want any ParseError exception raised by E to be trapped, but *not* any ParseError exception raised by the call to string2int. It is impossible to write this without the capability to handle successful termination—of the two obvious 'solutions', one does not type-check:

```
string2int (
  trap
    exception ParseError is 0 endexn
  in
    E
  endtrap
) [ParseError]
```

and the other traps the ParseError exception raised by string2int:

```
trap
  exception ParseError is 0 endexn
in
  string2int (E) [ParseError]
endtrap
```

The **trap** operator both declares and traps the exception—this means it is impossible for an exception to escape outside its scope. This can be contrasted with a language such as SML where exception declaration and handling are separated, so it is possible for exceptions to escape their scope:

```
local
  exception Foo
in
  raise Foo
end
```

Note that the only way in which an exception can be observed by its environment is by trapping it—it is impossible for expressions to synchronize on exceptions.

Nondeterminism In the presence of exceptions, order of evaluation becomes important, for example depending on the order of evaluation we can get different exceptions raised by the expression:

```
(raise Foo, raise Baz)
```

The semantics given in this paper is nondeterministic: record expressions are evaluated in parallel, so in the above example there is a race condition between the Foo and Baz exceptions. This means that the data expression language is nondeterministic, for example a 'coin tossing' random boolean generator is:

```
trap
  exception Foo (?b:bool) is b endexn
in
  (raise Foo (true), raise Foo (false))
endtrap
```

Since the data expression language contains nondeterminism, we include an explicit nondeterministic expression **any** T which nondeterministically generates a value of type T. For example the above coin tossing expression is equivalent to **any** bool.

Imperative features The data expression language is functional, but supports a language of record expressions which mimics an imperative language with write-once variables. For example, the imperative expression:

```
?x := 0; ?y := "hello world";
```

is equivalent to the behaviour:

```
exit (x => 0, y => "hello world")
```

The simplest imperative expression is an assignment $P := E$, where P is an irrefutable pattern and E an expression, for example:

```
?x := 4
```

Since there is an expression on the right of an assignment, we can assign non-trivial expressions to patterns, for example a random number generator is:

```
?x := any int
```

As we remarked earlier, we allow the use of out parameters as syntax sugar for assignment, for example:

```
partition (y, ys, ?l, ?g)
```

is shorthand for:

(?l,?g) := partition (y,ys)

There is a sequential composition operator whose syntax is E_1 ; E_2. It is like the LOTOS enabling operator because it combines two expressions, but it has a slightly different semantics: it does not perform an internal **i** action.

The **local** operator is used to restrict the scope of variables, with syntax **local var** *LV* **in** *E* **endloc**, where *LV* is a list of typed variables. For example:

```
local
  var x:int
in
  ?x:=E; x*x
endloc
```

has the same semantics as $E*E$ (as long as E is deterministic). Optionally, some of the local variables can be initialized with an **init** section, for example we could have written:

```
local
  var x:int
init
  ?x:=E
in
  x*x
endloc
```

An iteration (or loop) operator is included in the core language. This operator is justified in the core language for two reasons:

- It was decided to include one in the user-level language.
- It allows recursive processes to be specified without using explicit process identifiers.

Loops with local variables can be declared—these local variables can be initialized, and should then be assigned to on each iteration of the loop. A loop can be broken with a **break** command. For example, an imperative function to sum a list of numbers can be defined:

```
function sum (?xs:intlist) : int is
  loop(int)
    var ys:intlist,total:int
  init
    ?ys:=xs; ?total:=0
  in
    case ys is
        nil → break (total)
      | cons(?z,?zs) → ?total := total+z; ?ys := zs
    endcase
  endloop
endfun
```

This loop construct is defined in terms of a simpler unbreakable loop with syntax **loop forever var** *LV* **init** E_1 **in** E_2. The similarity to the syntax of local variables is not accidental, since (up to strong bisimulation) we have:

loop forever var *LV* **init** E_1 **in** E_2
  = **local var** *LV* **init** E_1 **in loop forever var** *LV* **init** E_2 **in** E_2

```
function partition (?x:int,?xs:intlist) : (intlist,intlist) is
  loop((intlist,intlist))
    var less:intlist,gtr:intlist,rest:intlist
  init
    less := nil; gtr := nil; rest := xs
  in
    case rest is
        nil →
          break ((less,gtr))
      | cons(?y,?ys) [y < x] →
          ?less := cons(y,less); ?gtr := gtr; ?rest := ys
      | cons(?y,?ys) →
          ?less := less; ?gtr := cons(y,gtr); ?rest := ys
    endcase
  endloop
endfun
```

Figure 1: The imperative version of partition

The breakable loop is then defined using exception handling, for example the above loop is shorthand for:

```
trap
  exception Inner (?x:int) is x endexn
in
  loop forever
    var ys:intlist,total:int
  init
    ?ys:=xs; ?total:=0
  in
    case ys is
        nil → raise Inner (total)
      | cons(?z,?zs) → ?total := total + z; ?ys := zs
    endcase
  endloop
endtrap
```

We also allow named loops, so that you can break a loop other than the innermost one, for example:

```
loop fred in ...
  loop janet in ...
    if b then break fred ...
```

As an example of the imperative features, an imperative definition of quicksort partitioning is given in Figure 1. It can be compared with the functional definition given in Figure 2.

```
function partition (?x:int, ?xs:intlist) : (intlist, intlist) is
  case xs is
      nil →
        (nil, nil)
    | cons(?y, ?ys) →
        local
          var less:intlist, gtr:intlist
        init
          (?less, ?gtr) := partition (x, ys)
        in
          if x > y
          then (cons(y, less), gtr)
          else (less, cons(y, gtr))
          endif
        endloc
  endcase
endfun
```

Figure 2: The functional version of partition

Static semantics The static semantics for expressions is given by translating them into the behaviour language described below. For expressions which do not assign to variables, the typing is given by judgements:

$$\mathcal{C} \vdash E \Rightarrow \textbf{exit}\ (T)$$

meaning 'in context $\mathcal{C}$, expression E has result type T'. The context $\mathcal{C}$ gives the type for each of the free identifiers used in E, for example we can deduce:

$$\mathrm{x} \Rightarrow \mathrm{int},\ * \Rightarrow (\mathrm{int}, \mathrm{int}) \rightarrow \textbf{exit}\ (\mathrm{int}) \quad \vdash \quad \mathrm{x} * \mathrm{x} \Rightarrow \textbf{exit}\ (\mathrm{int})$$

meaning 'in a context where x is an integer and $*$ is a function from pairs of integers to integers, then $\mathrm{x} * \mathrm{x}$ returns an integer'.

Expressions which assign to variables but do not return a result have typing given by judgements:

$$\mathcal{C} \vdash E \Rightarrow \textbf{exit}\ (V_1 \Rightarrow T_1, \ldots, V_n \Rightarrow T_n)$$

meaning 'in context $\mathcal{C}$, expression E assigns to variables V_1 through to V_n the types T_1 through to T_n'. For example we can deduce:

$$2 \Rightarrow \mathrm{int} \quad \vdash \quad ?\mathrm{x}{:=}2 \Rightarrow \textbf{exit}\ (\mathrm{x} \Rightarrow \mathrm{int})$$

meaning 'in a context where 2 is an integer, then ?x:=2 assigns an integer to the variable x'.

Expressions which both assign to variables and return a result have typing given by judgements:

$$\mathcal{C} \vdash E \Rightarrow \textbf{exit}\ (T,\ V_1 \Rightarrow T_1, \ldots, V_n \Rightarrow T_n)$$

which combines the above two semantics. For example:

$$2 \Rightarrow \mathrm{int},\ * \Rightarrow (\mathrm{int}, \mathrm{int}) \rightarrow \textbf{exit}\ (\mathrm{int}) \quad \vdash \quad ?\mathrm{x}{:=}2;\ \mathrm{x} * \mathrm{x} \Rightarrow \textbf{exit}\ (\mathrm{int}, \mathrm{x} \Rightarrow \mathrm{int})$$

meaning 'in a context where 2 is an integer and $*$ is a function from pairs of integers to integers, then ?x:=2; $\mathrm{x} * \mathrm{x}$ assigns an integer to the variable x and returns an integer'.

Note that x is not free in the expression ?x:=2; $\mathrm{x} * \mathrm{x}$ since it is bound by the assignment statement. This is reflected in the type judgement above, which does not require x to be in the context.

Dynamic semantics The dynamic semantics of data expressions is defined by the translation into behaviour expressions. There are two ways in which a data expression can have observable behaviour: either it terminates successfully, or it raises an exception.

Expressions which terminate successfully with a value have dynamic semantics given by judgements:

$$\mathcal{E} \vdash E \xrightarrow{\delta(N)} E'$$

meaning 'in environment $\mathcal{E}$, the expression E returns normal form N and then behaves like E''. As it happens, E' will always be an expression with no behaviour, since an expression cannot do anything after terminating, but we use this notation for symmetry with the case of exception raising. The context gives the bindings of function identifiers, and other similar static information required at run-time. For example:

$$\vdash \quad 2 * 2 \xrightarrow{\delta(4)} \textbf{block}$$

maning 'the expression $2*2$ returns the value 4 and then has no observable behaviour'.

Expressions which terminate successfully having assigned values to variables have dynamic semantics given by judgements:

$$\mathcal{E} \vdash E \xrightarrow{\delta(V_1 \Rightarrow N_1, \ldots, V_n \Rightarrow N_n)} E'$$

meaning 'in context $\mathcal{E}$, the expression E assigns normal forms N_1 through to N_n to variables V_1 through to V_n'. For example:

$$\vdash \quad \texttt{?x:=2} \xrightarrow{\delta(\mathtt{x} \Rightarrow 2)} \textbf{block}$$

meaning 'the expression `?x:=2` terminates, having assigned the value 2 to the variable x, and then has no observable behaviour'.

Expressions which both assign to variables and return a result have dynamic semantics given by judgements:

$$\mathcal{E} \vdash E \xrightarrow{\delta(N, V_1 \Rightarrow N_1, \ldots, V_n \Rightarrow N_n)} E'$$

combining the two semantics, for example:

$$\vdash \quad \texttt{?x:=2;}\ \mathtt{x} * \mathtt{x} \xrightarrow{\delta(4, \mathtt{x} \Rightarrow 2)} \textbf{block}$$

Similarly, the semantics of exceptions is given by judgements:

$$\mathcal{E} \vdash E \xrightarrow{X(N)} E'$$

For example:

$$\textbf{raise}\ X(1) \quad \xrightarrow{X(1)} \quad \textbf{block}$$

The semantics is defined formally in Section 14.

2.4 Behaviour expressions

Some knowledge of LOTOS is assumed in this paper. However, for completeness, we provide the syntax of Basic LOTOS (i.e. LOTOS without datatypes) together with some brief explanations.

$$B ::= \textbf{stop} \mid \textbf{exit} \mid \Pi\,[G^*] \mid G; B \mid \textbf{i}; B \mid B \;[]\; B \mid B \mid [G^*] \mid B \mid \textbf{hide}\ G^*\ \textbf{in}\ B \mid B \gg B \mid B\ [> B$$

The semantics is as follows:

- Deadlock: **stop** is an inactive behaviour.

- Termination: **exit** is a behaviour that terminates successfully. It performs an action on gate δ and then deadlocks.
- Process instantiation: $\Pi[\vec{G}]$ instantiates the previously delared process definition with parameters $\vec{G}$.
- Action-prefix: $G;B$ is a behaviour that first performs action G and then behaves like B.
- Internal action-prefix: $\mathbf{i};B$ is a behaviour that first performs the internal action **i** and then behaves like BB.
- External choice: B_1 [] B_2 is a process that can behave either like B_1 or like B_2 depending on the environment.
- Parallelism: $B_1 \mid [\vec{G}] \mid B_2$ is the parallel composition of B_1 and B_2 with synchronisation on the gates in $\vec{G}$.
- Abstraction: **hide** $G*$ **in** B hides in behaviour B all the actions from the set $\vec{G}$, i.e. it renames them into **i**.
- Enabling: $B_1 \gg B_2$ is the sequential composition of B_1 and B_2, i.e. B_2 can start when B_1 has terminated successfully.
- Disabling: B_1 [> B_2 allows B_2 to disable B_1 provided B_1 has not terminated successfully.

The main differences between this language and the core language that we have designed are as follows:

- Actions are particular behaviours and the two forms of sequential composition (action-prefix and enabling) are unified.
- New features are added such as pattern-matching, exceptions, assignment, time and other operators (e.g. an explicit renaming operator).

The behaviour language can be seen as an extension of the data language with communication between parallel processes and real-time features.

Communication Behaviours can communicate on *gates*. The simplest communicating process is one which synchronizes on a gate G: this is just written G. Such synchronizations can then be sequentially composed, for example a behaviour which alternates between in and out actions is:

> **loop forever in**
>   in; out
> **endloop**

Behaviours can also send or receive data on gates, for example a one-place integer buffer is:

> **loop forever**
>   **var** x:int
> **in**
>   in(?x); out(!x)
> **endloop**

Here the variable x is *bound* by the communication on the in gate, and is *free* in the communication on the out gate. The resulting behaviour copies integers from the in gate to the out gate.

When synchronizing on a gate, you can specify any pattern to synchronize on, for example:

$$G(\text{age} \Rightarrow !28, \text{name} \Rightarrow ?\text{na}, \text{address} \Rightarrow (\text{number} \Rightarrow ?\text{no}, \text{street} \Rightarrow !\texttt{"Acacia Ave"}, \textbf{etc}))$$

will synchronize on any person aged 28 living in Acacia Avenue, and will bind the variables na and no appropriately. This use of patterns in communications is the main reason for allowing ? and ! in patterns.

You can also specify a *selection predicate* specifying whether a synchronization should be allowed, for example to select anyone in their 20s living on Acaica Avenue, you might say:

G(age ⇒ `?a`, name ⇒ `?`na, address ⇒ (number ⇒ `?`no, street ⇒ `!"Acacia Ave"`, **etc**))
  [20 ≤ a **andalso** a ≤ 29]

Gate parameters are given in process declarations, for example:

process Buffer [in(int), out(int)] : **exit** (**none**) **is**
  loop forever
    var x : int
  in
    in(?x); out(!x)
  endloop
endproc

Gates may be typed: by default each gate has type (**etc**), so can communicate data of any type, for example:

process OverloadingExample [overloaded] (!x : int, !y : bool) **is**
  overloaded(?x : int);
  overloaded(?y : bool)
endproc

The first communication on the overloaded gate has to be of type integer, and the second has to be of type boolean.

We can use **as** patterns to match against all or some of a record. This is particularly useful when the record is extensible, for example we can write a simple router capable of handling any type of data as:

process Router [in(de ⇒ dest, **etc**), left, right] : **exit** (**none**) **is**
  local
    var destination: dest, data: (**etc**)
  in
    in(de ⇒ ?destination, ?data **as etc**);
    case destination **is**
        L → left!data
      | R → right!data
    endcase;
    Router [in, left, right]
  endloc
endproc

Concurrency Concurrent behaviours can synchronize on their communications. For example, two behaviours which are forced to synchronize on all communictions are:

    G(address ⇒ (number ⇒ ?no, street ⇒ `!"Acacia Ave"`, **etc**), **etc**)
  || G(age ⇒ !28, name ⇒ ?na, address ⇒ **any**)

Since the two behaviours are forced to synchronize on the gate G, this has the same semantics as:

G(age ⇒ !28, name ⇒ ?na, address ⇒ (number ⇒ ?no, street ⇒ `!"Acacia Ave"`, **etc**))

Data may be communicated in both direcions in a synchronization, for example:

$$
\begin{array}{ll}
& G(\text{age} \Rightarrow \texttt{!}28, \text{name} \Rightarrow \texttt{?na}, \textbf{etc}); B_1 \\
\texttt{||} & G(\text{age} \Rightarrow \texttt{?a}, \text{name} \Rightarrow \texttt{!"Fred"}, \textbf{etc}); B_2
\end{array}
$$

has the same semantics as:

$$
\begin{array}{l}
G(\text{age} \Rightarrow \texttt{!}28, \text{name} \Rightarrow \texttt{!"Fred"}, \textbf{etc}); \\
(\texttt{?na:="Fred"};\ B_1)\ \texttt{||}\ (\texttt{?a:=28};\ B_2)
\end{array}
$$

Parallel behaviours have to synchronize on termination, for example the following will terminate immediately, after setting variables x and y:

x:=1 || y:=2

Two behaviours which have no synchronizations at all (apart from synchronizing on termination) are:

```
    overloaded(?x:int)
||| overloaded(?y:bool)
```

This will communicate twice on the overloaded gate: once inputting an integer, and once inputting a boolean, but the order is unspecifi ed. Once both inputs have happened, the process can terminate. This process has the same semantics as:

```
   overloaded(?x:int); overloaded(?y:bool)
[] overloaded(?y:bool); overloaded(?x:int)
```

Note that the variables bound by concurrent processes are all the variables bound by the components, and that (since variables are write-once) there is no possibility of communication by shared variables.

Time Behaviours have real-time capabilities, given by three constructs:

- a time type, with addition and comparisons on times,
- a **wait** operator, to introduce delays, and
- an extended communication opertor, which is sensitive to delay.

The time datatype is a total order with addition. We shall let d range over values of type time.

The delay operator is just written **wait**(d) which delays by time d and then terminates. For example a behaviour which communicates on gate G every time unit is:

```
loop forever in
  G;
  wait(1)
endloop
```

We can delay by an arbitrary time expression **wait**(E), for example:

```
loop forever
  var x:time
in
  G(?x);
  wait(x)
endloop
```

Since time expressions may be nondeterministic, we have a simple way to write a nondeterministic delay:

loop forever in
  G ;
  wait(**any** time)
endloop

Communications can be made sensitive to time by adding a @*P* annotation, which matches the pattern *P* to the time at which the communication happens (measured from when the communication was enabled). For example:

G(?x:int)@?t[t < 3]

is a behaviour that agrees to accept an integer value (to be bound to variable x), provided that less than 3 time units have passed, whereas:

G(?x:int)@!3

is similar, but the action can only occur at time 3, because the pattern variable has been replaced by a pattern value !3. This behaviour has the same semantics as:

local
  var t:time
in
  G(?x:int)@?t[t = 3]
endloc

The time features are directly inspired by ET-LOTOS [9] but are adapted it to fit with other new paradigms of the language, such as:

- action is a behaviour,
- sequential composition does not generate an **i** action,
- the presence of pattern-matching,
- the presence of exception raising and handling.

Urgency An important concept is *urgency*: a behaviour is urgent if it cannot delay—for example if there is a computation which must be performed immediately. For example, sequential composition is urgent—once the first behaviour terminates, control is immediatly passed to the second without delay. For example, consider the process:

loop forever in
  loop forever in tick **endloop** [> **wait**(1) ;
  loop forever in tock **endloop** [> **wait**(1)
endloop

This will perform any number of 'tick' actions during the first time interval, then at time 1 control is handed over, and any number of 'tock' actions is performed until time 2, and so on. Each of the hand-over is urgent, so we know it is impossible for a 'tick' action to happen in an even time interval, or a 'tock' action to happen in an odd time interval.

In the core language, the urgent actions are:

- Internal (**i**) actions, whether written explicitly or caused by hiding.
- Exception raising (*X*) actions.

- Termination (δ) actions.

All of these actions happen immediately, for example it is impossible for the *G* action to be delayed in the behaviour:

> **i** ; (*G* [] **exit**) ; **raise** *X*

However, there is one exception to the urgency of these actions: it is possible for a termination to be delayed by a parallel behaviour. For example the following behaviour will terminate at time 2:

> **wait**(1) **;** **exit** || **wait**(2) **;** **exit**

The urgent semantics of exceptions given here is basically the same as the 'signals' model of Timed CSP [2].

Hiding The syntax for hiding is like in existing LOTOS, except that the (declared) gates are typed. For example in:

> **hide** mid (int) **in**
> Buffer [in,mid] || Buffer [mid,out]
> **endhide**

a new mid gate is declared, which can communicate integers, and is then replaced by internal **i** actions. This operator preserves the property of urgency of all **i**, and allows the modelling of urgency on hidden synchronization. This means that on can express that a synchronization should occur as soon as made possible by all the processes involved. For example the behaviour:

> **hide** *G* **in**
> **wait**(1) ; *G* ; B_1
> || **wait**(2) ; *G* ; B_2
> **endhide**

has the same semantics as:

> **wait**(2) ; **i** ;
> **hide** *G* **in**
> B_1 || B_2
> **endhide**

The hidden *G* occurs after 2 time units, which is as soon as both processes can perform *G*.

The behaviour:

> **hide** *G* **in**
> *G*@?t[$t \geq 3$] ; *B*
> **endhide**

has the same semantics as:

> **wait**(3) ; `?t:=3`; **i** ;
> **hide** *G* **in** *B* **endhide**

Again the earliest possible time for *G* to occur is after 3 time units.

The behaviour:

> **hide** *G* **in**
> *G*@?t[$t > 3$] ; *B*
> **endhide**

has two possible semantics depending on whether the type time is discrete or dense. If time is a synonym for natural number (discrete time), the behaviour has the same semantics as:

wait(4) ; ?t:=4; **i**;
hide *G* **in** *B* **endhide**

because 4 is the smallest natural number strictly greater than 3. On the other hand, if time is a synonym for rational number (dense time), the behaviour has the same semantics as:

wait(3) ; **block**

The reason why this process timestops after 3 time units without even performing the hidden *G* is because there is no smallest rational (or earliest time) strictly greater than 3.

Having to hide synchronizations to make them occur as soon as possible is sometimes criticized, because there are cases where one would like to still observe those gates. The problem here lies in the interpretation of the word 'observation'. Observing requires interaction, and interaction may lead to interference. Clearly, we would like to show the interaction to the environment without allowing it to interfere. There is a nice solution to this problem. It suffices to raise an exception (signal) immediately after the occurrence of the hidden interaction as follows. Consider two processes, Producer and Consumer, that want to synchronise on the sync event as soon as they are both ready to do so. We add a special monitoring process that synchronizes with them and sends a signal just after sync occurred:

Producer	:=	**wait**(**any** time) ; sync ; Producer
Consumer	:=	sync ; **wait**(**any** time) ; Consumer
Monitoring	:=	sync ; **signal** yes ; Monitoring
System	:=	**hide** sync **in** (Producer || Consumer || Monitoring)

The **signal** operator is the same as **raise** except that it allows computation to carry on after the exception has been raised: **raise** *X* is shorthand for **signal** *X* ; **block**.

Time nondeterminism In our model, time is nondeterministic. This means that there are behaviours that do not age in a predictive manner, because they can possibly reach different states after aging of a well-defined time. Consider the following example:

(?x := true [] ?x := false) ; **wait**(2)

After one unit of time has passed, this process will either be:

?x := true ; **wait**(1)

or:

?x := false ; **wait**(1)

This time nondeterminism is unavoidable if we want to have sequential composition not introduce an **i** action.

Actually, this nondeterminism has some advantages. It gives us for free a way to express nondeterministic delays that do not rely on internal actions. The next example better illustrates this point—after a delay, the behaviour:

wait(**any** time)

will have become:

wait(*d*)

for some *d*.

There are two shortcomings for having time to be nondeterministic:

- The hiding rule expressing urgency on hidden actions is more complex, as would be any inference rule with an negative premise. Hopefully, this is the only negative premise of the language.

- B_1 `[]` B_2 is not equal to `x :=` **any** bool`;` **if** x **then** B_1 **else** B_2, because, in the latter, time resolves the choice. The latter expression is very very close to a nondeterministic choice, but is only equivalent to it after some arbitrarily small (but non zero) delay. Indeed, after a zero delay, the choice is not resolved (yet).

An alternative semantics would be to introduce explicit β-reductions to indicate where time nondeterminism has happened—this would make the semantics for time simpler, but at the cost of introducing another form of reduction. This is left for further work.

Renaming An explicit renaming operator is introduced in the language. It allows one to rename observable actions into observable actions, or exceptions into exceptions.

Renaming an observable action into another observable action may be much more powerful than one might think at first, because it allows one to do more than just renaming gate names. For example, it can be used to change the structure of events occurring at a gate (adding or removing attributes), or to merge or split gates.

The simplest form of renaming just renames one gate to another:

rename
  gate G (x ⇒ ?i:int) **is** G'(x ⇒ !i)**endgate**
in
  B
endren

Note the syntactic similarity between renaming and function declaration or exception trapping. This form of renaming is so common that we provide a shorthand for it:

rename
  G (x ⇒ int) **is** G'
in
  B
endren

We can remove a field from a gate:

rename
  G (x ⇒ ?i:int, y ⇒ **any**:bool) **is** G'(!i)
in
  B
endren

We can add a field to a gate:

rename
  G (x ⇒ ?i:int) **is** G'(x ⇒ !i, y ⇒ !true)
in
  B
endren

We can merge two gates G' and G'' into a single gate G:

rename
  G' (x ⇒ ?i:int) **is** G(x ⇒ !i, y ⇒ !true)
  G'' (x ⇒ ?i:int) **is** G(x ⇒ !i, y ⇒ !false)
in
  B
endren

We can rename exceptions in a similar way.

Static semantics The static semantics for behaviour expressions is very similar to that of data expressions, and is given by judgements:

$$\mathcal{C} \vdash B \Rightarrow \textbf{exit}\ (\vec{V} \Rightarrow \vec{T})$$

For example:

$$G \Rightarrow \textbf{gate any} \vdash (G(?\mathrm{x}:\mathrm{int})\ |||\ G(?\mathrm{y}:\mathrm{bool})) \Rightarrow \textbf{exit}\ (\mathrm{x} \Rightarrow \mathrm{int}, \mathrm{y} \Rightarrow \mathrm{bool})$$

Dynamic semantics The dynamic semantics of data expressions is given by two kinds of reduction:

- Successful termination $\mathcal{E} \vdash E \xrightarrow{\delta(RN)} E'$.
- Exception raising $\mathcal{E} \vdash E \xrightarrow{X(RN)} E'$.

The dynamic semantics of behaviour expressions extends this with three new kinds of judgement:

- Internal actions $\mathcal{E} \vdash B \xrightarrow{\mathbf{i}()} B'$.
- Communication $\mathcal{E} \vdash B \xrightarrow{G(RN)} B'$.
- Delay $\mathcal{E} \vdash B \xrightarrow{\varepsilon(d)} B'$.

For example (up to strong bisimulation):

$$\begin{array}{lll}
\mathbf{i};\ G(?\mathrm{t});\mathbf{wait}(\mathrm{t}) & \xrightarrow{\mathbf{i}()} & G(?\mathrm{t});\mathbf{wait}(\mathrm{t}) \\
& \xrightarrow{G(3)} & ?\mathrm{t}:=3;\ \mathbf{wait}(3) \\
& \xrightarrow{\varepsilon(2)} & ?\mathrm{t}:=3;\ \mathbf{wait}(1) \\
& \xrightarrow{\varepsilon(1)} & ?\mathrm{t}:=3;\ \mathbf{wait}(0) \\
& \xrightarrow{\delta(\mathrm{t}\Rightarrow 3)} & \mathbf{block}
\end{array}$$

The urgency of internal, exception and terminatin actions is given by the properties:

- No behaviour B can offer both $\xrightarrow{\mathbf{i}()}$ and $\xrightarrow{\varepsilon(d)}$.
- No behaviour B can offer both $\xrightarrow{X(RN)}$ and $\xrightarrow{\varepsilon(d)}$.
- No behaviour B can offer both $\xrightarrow{\delta(RN)}$ and $\xrightarrow{\varepsilon(d)}$.

For example:

$$\texttt{?t:=3} \xrightarrow{\delta(\mathrm{t}\Rightarrow 3)} \textbf{block}$$

but:

$$\texttt{?t:=3} \stackrel{\varepsilon(d)}{\not\longrightarrow}$$

However, in order to get the correct synchronization semantics for termination, we have to allow terminated processes to age when placed in a parallel context. Consider the following example:

$$\texttt{?t:=3} \mid\mid \textbf{wait}(2)\texttt{;?y:=true}$$

We would like this to have semantics (up to strong bisimulation):

$$\begin{array}{lcl} \texttt{?t:=3} \mid\mid \textbf{wait}(2)\texttt{;?y:=true} & \xrightarrow{\varepsilon(1)} & \texttt{?t:=3} \mid\mid \textbf{wait}(1)\texttt{;?y:=true} \\ & \xrightarrow{\varepsilon(1)} & \texttt{?t:=3} \mid\mid \textbf{wait}(0)\texttt{;?y:=true} \\ & \xrightarrow{\delta(\mathrm{t}\Rightarrow 3,\, \mathrm{y}\Rightarrow \mathrm{true})} & \textbf{block} \end{array}$$

In order to achieve this, we allow terminated processes to age in a parallel composition. The alternative would be to treat δ actions (and sequential composition) in the same way as gates (and hiding), but this would have introduced many negative premises into the semantics (for example sequential composition and exception handling), which we have tried to avoid. The semantics presented here only uses negative premises in the semantics of hiding.

3 Overview

3.1 Syntax

The terminals of the abstract syntax are:

identifier domain	*meaning*	*abbreviation*
Var	variable identifier	*V*
Typ	type identifier	*S*
Con	constructor identifier	*C*
Proc	process identifier	Π
Gat	gate identifier	*G*
Exc	exception identifier	*X*

In addition, we define the following non-terminals as syntax sugar:

symbol domain	*meaning*	*abbreviation*	*sugar for*
Fun	function identifier	*F*	Π

The non-terminals are:

symbol domain	*meaning*	*abbreviation*
SCon	special constant	*K*
Decl	declaration	*D*
TyExp	type expression	*T*
RTyExp	record type expression	*RT*
Val	value expression	*N*
RVal	record value expression	*RN*
Pat	pattern	*P*
RPat	record pattern	*RP*
RVar	record of variables	*RV*
Behav	behaviour expression	*B*
BMatch	behaviour match	*BM*

In addition, we define the following non-terminals as syntax sugar:

symbol domain	*meaning*	*abbreviation*	*sugar for*
LocVar	local variables	LV	$RV : RT$
InPar	in parameters	IP	$RP : RT$
Exp	expression	E	B
RExp	record expression	RE	B
EMatch	expression match	EM	BM

In the grammars, non-primitive constructs (which are defined in terms of syntactic sugar for primitives) are marked with a '$\star$'. These grammars omit any **end**-keywords, which should be included in the concrete grammar.

3.2 Static semantics

The static semantics is given by a series of judgements, such as $\mathcal{C} \vdash B \Rightarrow \mathbf{exit}(RT)$ meaning 'in context $\mathcal{C}$, behaviour B has result type (RT)'. The context gives the bindings for any free identifiers, and is given by the grammar:

$\mathcal{C}$	$::=$	$V \Rightarrow T$	*variable*	(C_c1)
	$\mid$	$S \Rightarrow \mathbf{type}$	*type*	(C_c2)
	$\mid$	$S \equiv T$	*type equivalence*	(C_c3)
	$\mid$	$C \Rightarrow (RT) \rightarrow S$	*constructor*	(C_c4)
	$\mid$	$\Pi \Rightarrow [(\mathbf{gate}(RT))^*](RT)[(\mathbf{exn}(RT))^*] \rightarrow \mathbf{exit}(T)$	*process identifer*	(C_c5)
	$\mid$	$G \Rightarrow \mathbf{gate}(RT)$	*gate*	(C_c6)
	$\mid$	$X \Rightarrow \mathbf{exn}(RT)$	*exception*	(C_c7)
	$\mid$		*trivial*	(C_c8)
	$\mid$	$\mathcal{C}, \mathcal{C}$	*disjoint union*	(C_c9)

where each identifer only has one binding.

We shall write $\mathcal{C}_1; \mathcal{C}_2$ for context over-riding (with all the bindings of $\mathcal{C}_2$, and any bindings from $\mathcal{C}_1$ not overridden by $\mathcal{C}_2$).

Note that the grammar for record types overlaps with that of contexts. Whenever RT does not contain any occurrences of **etc**, we shall allow RT to range over contexts (for example in the type rule for sequential composition in Section 12.11).

3.3 Dynamic semantics

The dynamic semantics is given by a series of judgements, such as $\mathcal{E} \vdash B \xrightarrow{\delta(RN)} B'$ meaning 'in environment $\mathcal{E}$, behaviour B terminates with result (RN)'. The environment gives the bindings for free identifers, and is given by the grammar:

$\mathcal{E}$	$::=$	$S \equiv T$	*type equivalence*	(E_c1)
	$\mid$	$C \Rightarrow (RT) \rightarrow S$	*constructor*	(E_c2)
	$\mid$	$\Pi \Rightarrow \lambda[(G(RT))^*](RP : RT)[(X(RT))^*] \rightarrow B$	*process identifer*	(E_c3)
	$\mid$		*trivial*	(E_c4)
	$\mid$	$\mathcal{E}, \mathcal{E}$	*disjoint union*	(E_c5)

Note that environments have to carry type information. This is because LOTOS relies on run-time typing for much of its semantics, for example the semantics of the nondeterministic expression **any** T depends on the type rules for T.

The semantics for expressions with free variables uses *substitution* to replace free variables with values. The grammar for substitutions is given by:

σ	$::=$	$V \Rightarrow N$	*singleton*	$(\sigma_c 1)$
	$\mid$		*trivial*	$(\sigma_c 2)$
	$\mid$	σ, σ	*disjoint union*	$(\sigma_c 3)$

where each variable is only bound once. We write $B[\sigma]$ for B with all free variables replaced by values given by σ with the usual α-conversion to avoid binding free variables.

Note that the grammar for substitutions is the same as the grammar for record values *RN*, so we will use them interchangably (for example in the dynamic semantics of sequential composition in Section 12.11.

3.4 Syntax sugar

Many of the constructs in the core language are defined as syntax sugar, for example **if**-statements are defined as syntax sugar for **case**-statement.

In this paper, we do not give the semantics for terms defined by syntax sugar.

4 Declarations

4.1 Overview

Syntax

D	$::=$	**type** S **is** T	*type synonym*	$(\mathrm{D}_c 1)$
	$\mid$	**type** S **is** $C[(RT)]\ (\mid C[(RT)])^*$	*type declaration*	$(\mathrm{D}_c 2)$
	$\mid$	**process** $\Pi\ [\llbracket G[(RT)](,G[(RT)])^* \rrbracket]\ [(IP)]\ [:\ $**exit**$(T)]$ [**raises** $[\llbracket X[(RT)](,X[(RT)])^* \rrbracket]$] **is** B	*process declaration*	$(\mathrm{D}_c 3)$
	$\star\mid$	**process** $\Pi\ [\llbracket G[(RT)](,G[(RT)])^* \rrbracket]$ ([**in** IP] [**out** LV]) [**raises** $[\llbracket X[(RT)](,X[(RT)])^* \rrbracket]$] **is** B	*process with in/out parameters*	$(\mathrm{D}_c 4)$
	$\star\mid$	**function** $F\ [(IP)]\ [:\ T]$ [**raises** $[\llbracket X[(RT)](,X[(RT)])^* \rrbracket]$] **is** E	*function declaration*	$(\mathrm{D}_c 5)$
	$\star\mid$	**function** F ([**in** IP] [**out** LV]) [**raises** $[\llbracket X[(RT)](,X[(RT)])^* \rrbracket]$] **is** E	*function with in/out parameters*	$(\mathrm{D}_c 6)$
	$\mid$	DD	*declaration sequence*	$(\mathrm{D}_c 7)$
	$\mid$		*empty declaration*	$(\mathrm{D}_c 8)$

Static semantics

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

Dynamic semantics

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

Syntax sugar The **function** decalarations are synonymous with the equivalent **process** declaration.

4.2 Type synonym

Syntax

$$\textbf{type}\ S\ \textbf{is}\ T$$

Static semantics

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

Dynamic semantics

$$\frac{}{\mathcal{E} \vdash (\textbf{type}\ S\ \textbf{is}\ T) \Rightarrow (S \equiv T)}$$

4.3 Type declaration

Syntax

$$\textbf{type}\ S\ \textbf{is}\ C[\texttt{(}RT\texttt{)}]\ (\texttt{|}\ C[\texttt{(}RT\texttt{)}])^*$$

The default constructor argument type is `()`.

Static semantics

$$\frac{\mathcal{C} \vdash \texttt{(}RT_1\texttt{)} \Rightarrow \textbf{type} \quad \cdots \quad \mathcal{C} \vdash \texttt{(}RT_n\texttt{)} \Rightarrow \textbf{type}}{\mathcal{C} \vdash (\textbf{type}\ S\ \textbf{is}\ C_1\texttt{(}RT_1\texttt{)}\ \texttt{|}\ \cdots\ \texttt{|}\ C_n\texttt{(}RT_n\texttt{)}) \Rightarrow (S \Rightarrow \textbf{type}, C_1 \Rightarrow \texttt{(}RT_1\texttt{)} \rightarrow S, \ldots, C_n \Rightarrow \texttt{(}RT_n\texttt{)} \rightarrow S)}$$

Dynamic semantics

$$\frac{}{\mathcal{E} \vdash (\textbf{type}\ S\ \textbf{is}\ C_1\texttt{(}RT_1\texttt{)}\ \texttt{|}\ \cdots\ \texttt{|}\ C_n\texttt{(}RT_n\texttt{)}) \Rightarrow (C_1 \Rightarrow \texttt{(}RT_1\texttt{)} \rightarrow S, \ldots, C_n \Rightarrow \texttt{(}RT_n\texttt{)} \rightarrow S)}$$

4.4 Process declaration

Syntax

$$\textbf{process}\ \Pi\ [\texttt{[}[G[\texttt{(}RT\texttt{)}](\texttt{,}G[\texttt{(}RT\texttt{)}])^*]\texttt{]}]\ [\texttt{(}IP\texttt{)}]\ [\texttt{:}\ \textbf{exit}\texttt{(}T\texttt{)}]\ [\textbf{raises}\ [\texttt{[}[X[\texttt{(}RT\texttt{)}](\texttt{,}X[\texttt{(}RT\texttt{)}])^*]\texttt{]}]\ \textbf{is}\ B$$

The default gate list is `[]`, the default gate type is (**etc**), the default in pararameter is `()`, the default result type is **exit**(**none**), the default exception list is `[]` and the default exception type is `()`.

Static semantics

$$\frac{\begin{array}{r} \mathcal{C},G_1 \Rightarrow \textbf{gate}(RT_1),\ldots,G_m \Rightarrow \textbf{gate}(RT_m), \\ RT',X_1 \Rightarrow \textbf{exn}(RT'_1),\ldots,X_n \Rightarrow \textbf{exn}(RT'_n) \vdash B \Rightarrow \textbf{exit}(T) \end{array} \quad \begin{array}{l} \mathcal{C} \vdash (RT_1) \Rightarrow \textbf{type} \quad \cdots \quad \mathcal{C} \vdash (RT_m) \Rightarrow \textbf{type} \\ \mathcal{C} \vdash (RT) \Rightarrow \textbf{type} \\ \mathcal{C} \vdash (RT'_1) \Rightarrow \textbf{type} \quad \cdots \quad \mathcal{C} \vdash (RT'_n) \Rightarrow \textbf{type} \\ \mathcal{C} \vdash ((RP) \Rightarrow (RT)) \Rightarrow (RT') \end{array}}{\begin{array}{l} \mathcal{C} \vdash (\textbf{process}\ \Pi\ [G_1(RT_1),\ldots,G_m(RT_m)]\ (RP:RT) \\ \quad : \textbf{exit}(T)\ \textbf{raises}\ [X_1(RT'_1),\ldots,X_n(RT'_n)]\ \textbf{is}\ B) \\ \quad\quad \Rightarrow (\Pi \Rightarrow [\textbf{gate}(RT_1),\ldots,\textbf{gate}(RT_m)]\,(RT) \\ \quad\quad\quad [\textbf{exn}(RT'_1),\ldots,\textbf{exn}(RT'_n)] \rightarrow \textbf{exit}(T)) \end{array}}$$

Dynamic semantics

$$\frac{}{\begin{array}{l} \mathcal{E} \vdash (\textbf{process}\ \Pi\ [\vec{G}(\vec{RT})]\ (RP:RT)\ :\ \textbf{exit}\ (T)\ \textbf{raises}\ [\vec{X}(\vec{RT'})]\ \textbf{is}\ B) \\ \quad \Rightarrow (\Pi \Rightarrow \lambda\ [\vec{G}(\vec{RT})]\ (RP:RT)\ [\vec{X}(\vec{RT'})] \rightarrow B) \end{array}}$$

4.5 Process declaration with in/out parameters

Syntax

$$\textbf{process}\ \Pi\ [[[G[(RT)](,G[(RT)])^*]]]\ ([\textbf{in}\ IP]\ [\textbf{out}\ LV])\ [\textbf{raises}\ [[[X[(RT)](,X[(RT)])^*]]]\ \textbf{is}\ B$$

The default gate list is $[]$, the default gate type is $(\textbf{etc})$, the default in parameter list is **in** (), the default out parameter list is **out** (), the default exception list is $[]$ and the default exception type is $()$.

Syntax sugar

$$\left(\begin{array}{l} \textbf{process}\ \Pi\ [\vec{G}(\vec{RT})] \\ \quad (\textbf{in}\ IP\ \textbf{out}\ RV:RT) \\ \quad \textbf{raises}\ [\vec{X}(\vec{RT'})]\ \textbf{is} \\ \quad\quad B \end{array}\right) \stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{process}\ \Pi\ [\vec{G}(\vec{RT})] \\ \quad (IP)\ :\ \textbf{exit}((RT)) \\ \quad \textbf{raises}\ [\vec{X}(\vec{RT'})]\ \textbf{is} \\ \quad\quad \textbf{local var}\ RV:RT \\ \quad\quad \textbf{init}\ B \\ \quad\quad \textbf{in exit}((RV)) \end{array}\right)$$

4.6 Function declaration

Syntax

$$\textbf{function}\ F\ [(IP)]\ [:\ T]\ [\textbf{raises}\ [[[X[(RT)](,X[(RT)])^*]]]\ \textbf{is}\ E$$

The default in pararameter is $()$, the default result type is **none**, the default exception list is $[]$ and the default exception type is $()$.

Syntax sugar

$$\left(\begin{array}{l} \textbf{function}\ F\ (IP)\ :\ T \\ \quad \textbf{raises}\ [\vec{X}(\vec{RT})]\ \textbf{is} \\ \quad\quad E \end{array}\right) \stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{process}\ F\ (IP)\ :\ \textbf{exit}(T) \\ \quad \textbf{raises}\ [\vec{X}(\vec{RT})]\ \textbf{is} \\ \quad\quad E \end{array}\right)$$

4.7 Function declaration with in/out parameters

Syntax

$$\textbf{function}\ F\ \texttt{(}[\textbf{in}\ IP]\ [\textbf{out}\ LV]\texttt{)}\ [\textbf{raises}\ [\texttt{[}[X[\texttt{(}RT\texttt{)}](\texttt{,}X[\texttt{(}RT\texttt{)}])^*]\texttt{]}]\ \textbf{is}\ E$$

Syntax sugar The default in parameter list is **in** (), the default out parameter list is **out** (), the default exception list is `[]` and the default exception type is `()`.

$$\left(\begin{array}{l}\textbf{function}\ F\ \texttt{(}\textbf{in}\ IP\ \textbf{out}\ RV : RT\texttt{)}\\ \quad \textbf{raises}\ \texttt{[}\vec{X}\texttt{(}\vec{RT}\texttt{)]}\ \textbf{is}\\ \quad\quad E\end{array}\right) \stackrel{\mathsf{def}}{=} \left(\begin{array}{l}\textbf{function}\ F\ \texttt{(}IP\texttt{)}\ \texttt{:}\ \texttt{(}RT\texttt{)}\\ \quad \textbf{raises}\ \texttt{[}\vec{X}\texttt{(}\vec{RT}\texttt{)]}\ \textbf{is}\\ \quad\quad \textbf{local var}\ RV : RT\\ \quad\quad \textbf{init}\ E\\ \quad\quad \textbf{in}\ \texttt{(}RV\texttt{)}\end{array}\right)$$

5 Type expressions

5.1 Overview

Syntax

T	$::=$	S	*type identifier*	(T_c1)
	$\mid$	$\texttt{(}RT\texttt{)}$	*records*	(T_c2)
	$\mid$	**none**	*empty type*	(T_c3)
	$\mid$	**any**	*universal type*	(T_c4)

Static semantics

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

$$\mathcal{C} \vdash T \sqsubseteq T'$$

Subtyping is a preorder:

$$\frac{}{\mathcal{C} \vdash T \sqsubseteq T}$$

$$\frac{\mathcal{C} \vdash T \sqsubseteq T' \qquad \mathcal{C} \vdash T' \sqsubseteq T''}{\mathcal{C} \vdash T \sqsubseteq T''}$$

We write $T \equiv T'$ for $T \sqsubseteq T'$ and $T' \sqsubseteq T$. We will write:

$$\mathcal{C} \vdash T_1 \sqcup T_2 \Rightarrow T \qquad \mathcal{C} \vdash T_1 \sqcap T_2 \Rightarrow T$$

whenever (up to $\equiv$) T_1 and T_2 have a least upper bound (respectively greatest lower bound) T.

Dynamic semantics

$$\mathcal{E} \vdash T \sqsubseteq T'$$

In each case, the judgements are the same as for the static semantics, so we omit them.

5.2 Type identifier

Syntax

S

Static semantics

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

$$\frac{}{\mathcal{C}, S \equiv T \vdash S \equiv T}$$

5.3 Record type

Syntax

(RT)

Static semantics

$$\frac{\mathcal{C} \vdash RT \Rightarrow \textbf{record}}{\mathcal{C} \vdash (RT) \Rightarrow \textbf{type}}$$

$$\frac{\mathcal{C} \vdash RT \sqsubseteq RT'}{\mathcal{C} \vdash (RT) \sqsubseteq (RT')}$$

5.4 Empty type

Syntax

none

Static semantics

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

$$\frac{}{\mathcal{C} \vdash \textbf{none} \sqsubseteq T}$$

$$\frac{}{\mathcal{C} \vdash \textbf{none} \equiv (V \Rightarrow \textbf{none}, RT)}$$

5.5 Universal type

Syntax

any

Static semantics

$$\frac{}{\mathcal{C} \vdash \mathbf{any} \Rightarrow \mathbf{type}}$$

$$\frac{}{\mathcal{C} \vdash T \sqsubseteq \mathbf{any}}$$

6 Record type expressions

6.1 Overview

Syntax

RT	$::=$	$V \Rightarrow T$	*singleton*	(RT_c1)
	$\mid$	$\mathbf{etc}$	*universal record*	(RT_c2)
	$\mid$		*trivial*	(RT_c3)
	$\mid$	$RT\ ,RT$	*disjoint union*	(RT_c4)
	$\star\mid$	$T(\ ,T)^*$	*tuple*	(RT_c5)

Static semantics

$$\mathcal{C} \vdash RT \Rightarrow \mathbf{record}$$

$$\mathcal{C} \vdash RT \sqsubseteq RT'$$

Subtyping is a preorder:

$$\frac{}{\mathcal{C} \vdash RT \sqsubseteq RT}$$

$$\frac{\mathcal{C} \vdash RT \sqsubseteq RT' \qquad \mathcal{C} \vdash RT' \sqsubseteq RT''}{\mathcal{C} \vdash RT \sqsubseteq RT''}$$

We write $RT \equiv RT'$ for $RT \sqsubseteq RT'$ and $RT' \sqsubseteq RT$.

Dynamic semantics

$$\mathcal{E} \vdash RT \sqsubseteq RT'$$

In each case, the judgements are the same as for the static semantics, so we omit them.

6.2 Singleton record

Syntax

$$V \Rightarrow T$$

Static semantics

$$\frac{\mathcal{C} \vdash T \Rightarrow \textbf{type}}{\mathcal{C} \vdash (V \Rightarrow T) \Rightarrow \textbf{record}}$$

$$\frac{\mathcal{C} \vdash T \sqsubseteq T'}{\mathcal{C} \vdash (V \Rightarrow T) \sqsubseteq (V \Rightarrow T')}$$

6.3 Universal record

Syntax

etc

Static semantics

$$\frac{}{\mathcal{C} \vdash \textbf{etc} \Rightarrow \textbf{record}}$$

$$\frac{}{\mathcal{C} \vdash RT \sqsubseteq \textbf{etc}}$$

6.4 Empty record

Syntax

$()$

Static semantics

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

6.5 Record disjoint union

Syntax

RT, RT

Static semantics

$$\frac{\mathcal{C} \vdash RT_1 \Rightarrow \textbf{record} \qquad \mathcal{C} \vdash RT_2 \Rightarrow \textbf{record}}{\mathcal{C} \vdash RT_1, RT_2 \Rightarrow \textbf{record}} [RT_1 \text{ and } RT_2 \text{ have disjoint fields}]$$

$$\frac{\mathcal{C} \vdash RT_1 \sqsubseteq RT_1' \qquad \mathcal{C} \vdash RT_2 \sqsubseteq RT_2'}{\mathcal{C} \vdash RT_1, RT_2 \sqsubseteq RT_1', RT_2'}$$

$$\frac{}{\mathcal{C} \vdash RT_1, RT_2 \equiv RT_2, RT_1}$$

$$\frac{}{\mathcal{C} \vdash (RT_1, RT_2), RT_3 \equiv RT_1, (RT_2, RT_3)}$$

$$\frac{}{\mathcal{C} \vdash (), RT \equiv RT}$$

6.6 Tuple

Syntax

$T(\,,T)^*$

Syntax sugar

$T_1\,,\,\ldots\,,T_n \stackrel{\text{def}}{=} \$1 \Rightarrow T_1\,,\,\ldots\,,\$n \Rightarrow T_n$

7 Value expressions

Syntax

$$
\begin{array}{rcll}
N & ::= & K & \textit{primitive constant} \quad (\mathrm{N}_c1)\\
& | & V & \textit{variables} \quad (\mathrm{N}_c2)\\
& | & (RN) & \textit{record values} \quad (\mathrm{N}_c3)\\
& | & C\,[N] & \textit{constructor application} \quad (\mathrm{N}_c4)
\end{array}
$$

Static semantics

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

$$\frac{\mathcal{C} \vdash N \Rightarrow T \qquad \mathcal{C} \vdash T \sqsubseteq T'}{\mathcal{C} \vdash N \Rightarrow T'}$$

Dynamic semantics

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

$$\frac{\mathcal{E} \vdash N \Rightarrow T \qquad \mathcal{E} \vdash T \sqsubseteq T'}{\mathcal{E} \vdash N \Rightarrow T'}$$

In each case, the judgements are the same as for the static semantics, so we omit them.

7.1 Primitive constants

Syntax

K

Static semantics In this paper we will not discuss the static semantics of primitives—this is left to the design of the standard libraries.

7.2 Variables

Syntax

V

Static semantics

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

7.3 Record values

Syntax

(RN)

Static semantics

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

7.4 Constructor application

Syntax

$C\ [N]$

The default argument is `()`.

Static semantics

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

8 Record value expressions

Syntax

$$\begin{array}{llll} RN & ::= & V \Rightarrow N & \textit{singleton}\,(\mathrm{RN}_c1) \\ & | & & \textit{trivial}\,(\mathrm{RN}_c2) \\ & | & RN\,,RN & \textit{disjoint union}\,(\mathrm{RN}_c3) \\ & \star\,| & N(\,,N)^* & \textit{tuple}\,(\mathrm{RN}_c4) \end{array}$$

Static semantics

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

$$\frac{\mathcal{C} \vdash RN \Rightarrow RT \quad \mathcal{C} \vdash RT \sqsubseteq RT'}{\mathcal{C} \vdash RN \Rightarrow RT'}$$

Dynamic semantics

$\mathcal{E} \vdash RN \Rightarrow RT$

$$\frac{\mathcal{E} \vdash RN \Rightarrow RT \quad \mathcal{E} \vdash RT \sqsubseteq RT'}{\mathcal{E} \vdash RN \Rightarrow RT'}$$

In each case, the judgements are the same as for the static semantics, so we omit them.

8.1 Singleton record

Syntax

$V \Rightarrow N$

Static semantics

$$\frac{\mathcal{C} \vdash N \Rightarrow T}{\mathcal{C} \vdash (V \Rightarrow N) \Rightarrow (V \Rightarrow T)}$$

8.2 Empty record

Syntax

$()$

Static semantics

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

8.3 Record disjoint union

Syntax

RN, RN

Static semantics

$$\frac{\mathcal{C} \vdash RN_1 \Rightarrow RT_1 \qquad \mathcal{C} \vdash RN_2 \Rightarrow RT_2}{\mathcal{C} \vdash RN_1, RN_2 \Rightarrow RT_1, RT_2}[RN_1 \text{ and } RN_2 \text{ have disjoint fields}]$$

8.4 Tuple

Syntax

$N(,N)^*$

Syntax sugar

$N_1, \ldots, N_n \stackrel{\mathsf{def}}{=} \$1 \Rightarrow N_1, \ldots, \$n \Rightarrow N_n$

9 Patterns

Syntax

P	$::=$	(RP)	*records*	(P_c1)
	$\mid$	**any**	*wildcard*	(P_c2)
	$\mid$	$?V$	*variable*	(P_c3)
	$\mid$	$!E$	*expression*	(P_c4)
	$\mid$	$C\ [P]$	*constructor application*	(P_c5)
	$\mid$	$P : T$	*explicit typing*	(P_c6)

Static semantics

$\mathcal{C} \vdash (P \Rightarrow T) \Rightarrow (RT)$

Dynamic semantics

$\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN)$

$\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow \textbf{fail}$

9.1 Expression pattern

Syntax

$!E$

Static semantics

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

Dynamic semantics

$$\frac{\mathcal{E} \vdash E \xrightarrow{\delta(N')} E'}{\mathcal{E} \vdash (\,!E \Rightarrow N) \Rightarrow ()}[N = N']$$

$$\frac{\mathcal{E} \vdash E \xrightarrow{\delta(N')} E'}{\mathcal{E} \vdash (\,!E \Rightarrow N) \Rightarrow \textbf{fail}}[N \neq N']$$

$$\frac{\mathcal{E} \vdash E \xrightarrow{X\,(RN)} E'}{\mathcal{E} \vdash (\,!E \Rightarrow N) \Rightarrow \textbf{fail}}$$

9.2 Variable binding

Syntax

$?V$

Static semantics

$$\frac{}{\mathcal{C} \vdash (?V \Rightarrow T) \Rightarrow (V \Rightarrow T)}$$

Dynamic semantics

$$\frac{}{\mathcal{E} \vdash (?V \Rightarrow N) \Rightarrow (V \Rightarrow N)}$$

9.3 Record pattern

Syntax

(RP)

Static semantics

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

$$\frac{\mathcal{C} \vdash (RP \Rightarrow (\vec{V} \Rightarrow \textbf{any})) \Rightarrow (RT)}{\mathcal{C} \vdash ((RP) \Rightarrow \textbf{any}) \Rightarrow (RT)}$$

These rules require that if $(RT) \sqsubseteq T$ then either $T \equiv (RT')$ and $RT \sqsubseteq RT'$ or $T \equiv$ **any**.

Dynamic semantics

$$\frac{\mathcal{E} \vdash (RP \Rightarrow RN') \Rightarrow (RN)}{\mathcal{E} \vdash ((RP) \Rightarrow N) \Rightarrow (RN)}[N = (RN')]$$

$$\frac{\mathcal{E} \vdash (RP \Rightarrow RN') \Rightarrow \textbf{fail}}{\mathcal{E} \vdash ((RP) \Rightarrow N) \Rightarrow \textbf{fail}}[N = (RN')]$$

$$\frac{}{\mathcal{E} \vdash ((RP) \Rightarrow N) \Rightarrow \textbf{fail}}[N \neq (RN')]$$

9.4 Constructor application

Syntax

C $[P]$

The default pattern is `()`.

Static semantics

$$\frac{\mathcal{C} \vdash C \Rightarrow (RT) \rightarrow S \quad \mathcal{C} \vdash S \sqsubseteq T \quad \mathcal{C} \vdash (P \Rightarrow (RT)) \Rightarrow (RT')}{\mathcal{C} \vdash (C\ P \Rightarrow T) \Rightarrow (RT')}$$

Dynamic semantics

$$\frac{\mathcal{E} \vdash (P \Rightarrow (RN)) \Rightarrow (RN')}{\mathcal{E} \vdash (C\ P \Rightarrow N) \Rightarrow (RN')}[N = C(RN)]$$

$$\frac{\mathcal{E} \vdash (P \Rightarrow (RN)) \Rightarrow \textbf{fail}}{\mathcal{E} \vdash (C\ P \Rightarrow N) \Rightarrow \textbf{fail}}[N = C(RN)]$$

$$\frac{}{\mathcal{E} \vdash (C\ P \Rightarrow N) \Rightarrow \textbf{fail}}[N \neq C\ N']$$

9.5 Explicit typing

Syntax

$P : T$

Static semantics

$$\frac{\mathcal{C} \vdash T \Rightarrow \textbf{type} \quad \mathcal{C} \vdash T \sqsubseteq T' \quad \mathcal{C} \vdash (P \Rightarrow T) \Rightarrow (RT)}{\mathcal{C} \vdash (P : T \Rightarrow T') \Rightarrow (RT)}$$

Dynamic semantics

$$\frac{\mathcal{E} \vdash N \Rightarrow T \quad \mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN)}{\mathcal{E} \vdash (P : T \Rightarrow N) \Rightarrow (RN)}$$

$$\frac{\mathcal{E} \vdash N \Rightarrow T \quad \mathcal{E} \vdash (P \Rightarrow N) \Rightarrow \textbf{fail}}{\mathcal{E} \vdash (P : T \Rightarrow N) \Rightarrow \textbf{fail}}$$

$$\frac{\mathcal{E} \vdash N \Rightarrow T' \quad \mathcal{E} \vdash T \sqcap T' \Rightarrow \textbf{none}}{\mathcal{E} \vdash (P : T \Rightarrow N) \Rightarrow \textbf{fail}}$$

These rules require:

1. **none** to have no elements, and
2. a separability condition: if $\mathcal{E} \vdash N \Rightarrow$ **any** and $\mathcal{E} \vdash T \Rightarrow$ **type** then $\mathcal{E} \vdash N \Rightarrow T$ or $\mathcal{E} \vdash N \Rightarrow T'$ and $\mathcal{E} \vdash T \sqcap T' \Rightarrow$ **none**.

9.6 Wildcard

Syntax

any

Static semantics

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

Dynamic semantics

$$\frac{}{\mathcal{E} \vdash (\mathbf{any} \Rightarrow N) \Rightarrow ()}$$

10 Record patterns

Syntax

RP	$::=$	$V \Rightarrow P$	*singleton*	(RP_c1)
	$\mid$	**etc**	*wildcard*	(RP_c2)
	$\mid$	P **as** RP	*record match*	(RP_c3)
	$\mid$		*trivial*	(RP_c4)
	$\mid$	$RP\,,RP$	*disjoint union*	(RP_c5)
	$\star\mid$	$P(\,,P)^*$	*tuple*	(RP_c6)

with the restriction that **etc** can occur at most once in any record pattern (this is to bar ambiguous patterns such as (?x **as etc**, ?y **as etc**)).

Static semantics

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

Dynamic semantics

$$\mathcal{E} \vdash (RP \Rightarrow RN) \Rightarrow (RN')$$

$$\mathcal{E} \vdash (RP \Rightarrow RN) \Rightarrow \mathbf{fail}$$

10.1 Singleton record pattern

Syntax

$V \Rightarrow P$

Static semantics

$$\frac{\mathcal{C} \vdash (P \Rightarrow T) \Rightarrow (RT)}{\mathcal{C} \vdash ((V \Rightarrow P) \Rightarrow (V \Rightarrow T)) \Rightarrow (RT)}$$

Dynamic semantics

$$\frac{\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN)}{\mathcal{E} \vdash ((V \Rightarrow P) \Rightarrow (V \Rightarrow N)) \Rightarrow (RN)}$$

$$\frac{\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow \textbf{fail}}{\mathcal{E} \vdash ((V \Rightarrow P) \Rightarrow (V \Rightarrow N)) \Rightarrow \textbf{fail}}$$

10.2 Record wildcard

Syntax

etc

Static semantics

$$\frac{}{\mathcal{C} \vdash (\textbf{etc} \Rightarrow RT) \Rightarrow ()}$$

Dynamic semantics

$$\frac{}{\mathcal{E} \vdash (\textbf{etc} \Rightarrow RN) \Rightarrow ()}$$

10.3 Empty record pattern

Syntax

$()$

Static semantics

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

Dynamic semantics

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

10.4 Record match

Syntax

P **as** *RP*

Static semantics

$$\frac{\begin{array}{l}\mathcal{C} \vdash (P \Rightarrow (RT)) \Rightarrow (RT_1) \\ \mathcal{C} \vdash (RP \Rightarrow RT) \Rightarrow (RT_2)\end{array}}{\mathcal{C} \vdash (P \textbf{ as } RP \Rightarrow RT) \Rightarrow (RT_1, RT_2)} [RT_1 \text{ and } RT_2 \text{ have disjoint fields}]$$

Dynamic semantics

$$\frac{\begin{array}{l}\mathcal{E} \vdash (P \Rightarrow (RN)) \Rightarrow (RN_1) \\ \mathcal{E} \vdash (RP \Rightarrow RN) \Rightarrow (RN_2)\end{array}}{\mathcal{E} \vdash (P \textbf{ as } RP \Rightarrow RN) \Rightarrow (RN_1, RN_2)}$$

10.5 Record disjoint union

Syntax

RP, RP

Static semantics

$$\frac{\mathcal{C} \vdash (RP_1 \Rightarrow RT_1) \Rightarrow (RT_1') \qquad \mathcal{C} \vdash (RP_2 \Rightarrow RT_2) \Rightarrow (RT_2')}{\mathcal{C} \vdash (RP_1, RP_2 \Rightarrow RT_1, RT_2) \Rightarrow (RT_1', RT_2')} [RT_1' \text{ and } RT_2' \text{ have disjoint fields}]$$

Dynamic semantics

$$\frac{\mathcal{E} \vdash (RP_1 \Rightarrow RN_1) \Rightarrow (RN_1') \qquad \mathcal{E} \vdash (RP_2 \Rightarrow RN_2) \Rightarrow (RN_2')}{\mathcal{E} \vdash (RP_1, RP_2 \Rightarrow RN_1, RN_2) \Rightarrow (RN_1', RN_2')}$$

$$\frac{\mathcal{E} \vdash (RP_1 \Rightarrow RN_1) \Rightarrow \textbf{fail}}{\mathcal{E} \vdash (RP_1, RP_2 \Rightarrow RN_1, RN_2) \Rightarrow \textbf{fail}}$$

$$\frac{\mathcal{E} \vdash (RP_2 \Rightarrow RN_2) \Rightarrow \textbf{fail}}{\mathcal{E} \vdash (RP_1, RP_2 \Rightarrow RN_1, RN_2) \Rightarrow \textbf{fail}}$$

10.6 Tuple

Syntax

$P(\,,P)^*$

Syntax sugar

$$P_1, \ldots, P_n \stackrel{\text{def}}{=} \$1 \Rightarrow P_1, \ldots, \$n \Rightarrow P_n$$

11 Records of variables

Syntax

RV	$::=$	$V \Rightarrow V$	*singleton* (RV_c1)
	$\mid$		*trivial* (RV_c2)
	$\mid$	RV, RV	*disjoint union* (RV_c3)
	$\star \mid$	$V(\,,V)^*$	*tuple* (RV_c4)

Static semantics

$$\mathcal{C} \vdash (RV \Rightarrow RT) \Rightarrow (RT')$$

$$\frac{\mathcal{C} \vdash (RV \Rightarrow RT) \Rightarrow (RT') \quad \mathcal{C} \vdash RT' \equiv RT''}{\mathcal{C} \vdash (RV \Rightarrow RT) \Rightarrow (RT'')}$$

Dynamic semantics

$$\mathcal{E} \vdash (RV \Rightarrow RT) \Rightarrow (RT')$$

In each case, the judgements are the same as for the static semantics, so we omit them.

11.1 Singleton record variable

Syntax

$$V \Rightarrow V$$

Static semantics

$$\frac{}{\mathcal{C} \vdash ((V \Rightarrow V') \Rightarrow (V \Rightarrow T)) \Rightarrow (V' \Rightarrow T)}$$

11.2 Empty record variables

Syntax

$$()$$

Static semantics

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

11.3 Record disjoint union

Syntax

$$RV, RV$$

Static semantics

$$\frac{\mathcal{C} \vdash (RV_1 \Rightarrow RT_1) \Rightarrow (RT_1') \qquad \mathcal{C} \vdash (RV_2 \Rightarrow RT_2) \Rightarrow (RT_2')}{\mathcal{C} \vdash (RV_1, RV_2 \Rightarrow RT_1, RT_2) \Rightarrow (RT_1', RT_2')} [RT_1' \text{ and } RT_2' \text{ have disjoint fi elds}]$$

11.4 Tuple

Syntax

$$V\,(\,,V\,)^{*}$$

Syntax sugar

$$V_1\,,\,\ldots\,,V_n \stackrel{\text{def}}{=} \$1 \Rightarrow V_1\,,\,\ldots\,,\$n \Rightarrow V_n$$

12 Behaviour expressions

12.1 Overview

Syntax

B	$::=$	$G\ [P]\ [\texttt{@}P]\ [\texttt{[}E\texttt{]}]\ [\mathbf{start}\texttt{(}N\texttt{)}]$	*action*	(B_c1)
	$\mid$	$\mathbf{i}[\texttt{()}]$	*internal action*	(B_c2)
	$\mid$	$\mathbf{exit}[\texttt{(}RN\texttt{)}]$	*termination*	(B_c3)
	$\mid$	$\mathbf{exit}\texttt{(}\mathbf{any}\ T\texttt{)}$	*nondeterministic termination*	(B_c4)
	$\mid$	$\mathbf{signal}\ X\ [E]$	*raising exception*	(B_c5)
	$\mid$	$\mathbf{stop}$	*inaction*	(B_c6)
	$\mid$	$\mathbf{block}$	*time block*	(B_c7)
	$\mid$	$\mathbf{wait}\texttt{(}E\texttt{)}$	*delay*	(B_c8)
	$\mid$	$P\ \texttt{:=}\ E$	*assignment*	(B_c9)
	$\mid$	$B\ \texttt{;}\ B$	*sequential composition*	(B_c10)
	$\mid$	$B\ \texttt{[>}\ B$	*disabling*	(B_c11)
	$\mid$	$B\ \texttt{\|\|}\ B$	*synchronization*	(B_c12)
	$\mid$	$B\ \texttt{\|[}[G(\,,G)^{*}]\texttt{]\|}\ B$	*concurrency*	(B_c13)
	$\mid$	$B\ \texttt{[]}\ B$	*choice*	(B_c14)
	$\mid$	$\mathbf{choice}\ P\ [\mathbf{after}\texttt{(}N\texttt{)}]\ \texttt{[]}\ B$	*choice over values*	(B_c15)
	$\mid$	$\mathbf{trap}\ (\mathbf{exception}\ X[\texttt{(}IP\texttt{)}]\ \mathbf{is}\ B)^{*}\ [\mathbf{exit}\ [P]\ \mathbf{is}\ B]\ \mathbf{in}\ B$	*trap*	(B_c16)
	$\mid$	$\mathbf{case}\ E[\texttt{:}T]\ \mathbf{is}\ BM$	*case*	(B_c17)
	$\mid$	$\mathbf{local\ var}\ LV\ [\mathbf{init}\ B]\ \mathbf{in}\ B$	*variable declaration*	(B_c18)
	$\mid$	$\mathbf{hide}\ G[\texttt{(}RT\texttt{)}](\,,G[\texttt{(}RT\texttt{)}])^{*}\ \mathbf{in}\ B$	*gate hiding*	(B_c19)
	$\mid$	$\mathbf{rename}\ (G[\texttt{(}IP\texttt{)}]\ \mathbf{is}\ G\ [P]\ \mid\ X[\texttt{(}IP\texttt{)}]\ \mathbf{is}\ X\ [E])^{*}\ \mathbf{in}\ B$	*renaming*	(B_c20)
	$\mid$	$\Pi\ [\texttt{[}[G(\,,G)^{*}]\texttt{]}]\ [E]\ [\texttt{[}[X(\,,X)^{*}]\texttt{]}]$	*process instantiation*	(B_c21)
	$\mid$	$\mathbf{loop\ forever}\ [\mathbf{var}\ LV]\ [\mathbf{init}\ B]\ \mathbf{in}\ B$	*iteration*	(B_c22)
	$\star\mid$	$B\ \texttt{\|\|\|}\ B$	*interleaving*	(B_c23)
	$\star\mid$	$\mathbf{exit}\texttt{(}RE\texttt{)}$	*successful termination*	(B_c24)
	$\star\mid$	$\mathbf{raise}\ X\ E$	*raising exception*	(B_c25)
	$\star\mid$	$\mathbf{if}\ E\ \mathbf{then}\ B\ [\mathbf{else}\ B]$	*if-then-else*	(B_c26)
	$\star\mid$	$\Pi\ [\texttt{[}[G(\,,G)^{*}]\texttt{]}]\ \texttt{(}RE, RP\texttt{)}\ [\texttt{[}[X(\,,X)^{*}]\texttt{]}]$	*in/out process instantiation*	(B_c27)
	$\star\mid$	$\mathbf{loop}\ [X]\ [\texttt{(}T\texttt{)}]\ [\mathbf{var}\ LV]\ [\mathbf{init}\ B]\ \mathbf{in}\ B$	*breakable iteration*	(B_c28)

$\star \mid$ **break** $[X]$ $[(E)]$ *breaking iteration* (B_c29)

Static semantics

$\mathcal{C} \vdash B \Rightarrow \mathbf{exit}(RT)$

$$\frac{\mathcal{C} \vdash B \Rightarrow \mathbf{exit}(RT) \qquad \mathcal{C} \vdash RT \sqsubseteq RT'}{\mathcal{C} \vdash B \Rightarrow \mathbf{exit}(RT')}$$

Untimed dynamic semantics

$\mathcal{E} \vdash B \xrightarrow{\mu(RN)} B'$

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

Timed dynamic semantics

$\mathcal{E} \vdash B \xrightarrow{\varepsilon(d)} B'$

We shall write $\mathcal{E} \vdash B \xrightarrow{\mu(RN@d)} B'$ when either:

- $\mathcal{E} \vdash B \xrightarrow{\mu(RN)} B'$ and $d = 0$, or
- $\mathcal{E} \vdash B \xrightarrow{\varepsilon(d)} B''$ and $\mathcal{E} \vdash B'' \xrightarrow{\mu(RN)} B'$

Requirements on the time domain:

1. The only closed normal forms of type time are the special constants ranged over by d.
2. The time domain is a commutative cancellative monoid $+$ with unit 0.
3. The order given by $d_1 \leq d_2$ iff $\exists d \,.\, d_1 + d = d_2$ is a total order.

Since time is a commutative cancellative monoid, it satisfies the properties:

$$d_1 + d_2 = d_2 + d_1 \qquad \text{if } d_1 + d = d_2 + d \text{ then } d_1 = d_2 \qquad d_1 + (d_2 + d_3) = (d_1 + d_2) + d_3 \qquad d + 0 = d = 0 + d$$

We assume a type bool declared:

type bool **is** true | false

12.2 Action

Syntax

G $[P]$ $[\texttt{@}P]$ $[\texttt{[}E\texttt{]}]$ $[\mathbf{start}(N)]$

Default values are `()`, `@`**any**, `[`true`]` and **start**`(0)` respectively.

Static semantics

$$\frac{\begin{array}{l} \mathcal{C} \vdash G \Rightarrow \textbf{gate}\,(RT) \\ \mathcal{C} \vdash (P_1 \Rightarrow (RT)) \Rightarrow (RT_1) \\ \mathcal{C} \vdash (P_2 \Rightarrow \text{time}) \Rightarrow (RT_2) \\ \mathcal{C}; RT_1, RT_2 \vdash E \Rightarrow \textbf{exit}(\text{bool}) \\ \mathcal{C} \vdash N \Rightarrow \text{time} \end{array}}{\mathcal{C} \vdash G\; P_1\; \texttt{@}P_2\; \texttt{[}E\texttt{]}\; \textbf{start}(N) \Rightarrow \textbf{exit}(RT_1, RT_2)} [RT_1 \text{ and } RT_2 \text{ have disjoint fields}]$$

Untimed dynamic semantics

$$\frac{\begin{array}{l} \mathcal{E} \vdash (P_1 \Rightarrow (RN)) \Rightarrow (RN_1) \\ \mathcal{E} \vdash (P_2 \Rightarrow d) \Rightarrow (RN_2) \\ \mathcal{E} \vdash E[RN_1, RN_2] \xrightarrow{\delta(\text{true})} E' \end{array}}{\mathcal{E} \vdash G\; P_1\; \texttt{@}P_2\; \texttt{[}E\texttt{]}\; \textbf{start}(d) \xrightarrow{G(RN)} \textbf{exit}(RN_1, RN_2)}$$

Timed dynamic semantics

$$\frac{}{\mathcal{E} \vdash G\; P_1\; \texttt{@}P_2\; \texttt{[}E\texttt{]}\; \textbf{start}(d) \xrightarrow{\varepsilon(d')} G\; P_1\; \texttt{@}P_2\; \texttt{[}E\texttt{]}\; \textbf{start}(d + d')} [0 < d']$$

12.3 Internal action

Syntax

i[()]

Static semantics

$$\frac{}{\mathcal{C} \vdash \textbf{i}() \Rightarrow \textbf{exit}()}$$

Untimed dynamic semantics

$$\frac{}{\mathcal{E} \vdash \textbf{i}() \xrightarrow{\textbf{i}()} \textbf{exit}()}$$

Timed dynamic semantics None.

12.4 Termination

Syntax

exit [(RN)]

The default termination value is `()`.

Static semantics

$$\frac{\mathcal{C} \vdash RN \Rightarrow RT}{\mathcal{C} \vdash \mathbf{exit}(RN) \Rightarrow \mathbf{exit}(RT)}$$

Untimed dynamic semantics

$$\frac{}{\mathcal{E} \vdash \mathbf{exit}(RN) \xrightarrow{\delta(RN)} \mathbf{block}}$$

Timed dynamic semantics None.

12.5 Nondeterministic termination

Syntax

$\mathbf{exit}(\mathbf{any}\ T)$

Static semantics

$$\frac{\mathcal{C} \vdash T \Rightarrow \mathbf{type}}{\mathcal{C} \vdash \mathbf{exit}(\mathbf{any}\ T) \Rightarrow \mathbf{exit}(T)}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash N \Rightarrow T}{\mathcal{E} \vdash \mathbf{exit}(\mathbf{any}\ T) \xrightarrow{\delta(N)} \mathbf{block}}$$

Timed dynamic semantics None.

12.6 Signalling

Syntax

$\mathbf{signal}\ X\ [E]$

The default expression is `()`.

Static semantics

$$\frac{\mathcal{C} \vdash E \Rightarrow \mathbf{exit}((RT)) \qquad \mathcal{C} \vdash X \Rightarrow \mathbf{exn}(RT)}{\mathcal{C} \vdash \mathbf{signal}\ X\ E \Rightarrow \mathbf{exit}()}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash E \xrightarrow{\delta((RN))} E'}{\mathcal{E} \vdash \mathbf{signal}\ X\ E \xrightarrow{X\,(RN)} \mathbf{exit}()}$$

$$\frac{\mathcal{E} \vdash E \xrightarrow{X'(RN)} E'}{\mathcal{E} \vdash \mathbf{signal}\ X\ E \xrightarrow{X'(RN)} \mathbf{signal}\ X\ E'}$$

Timed dynamic semantics None.

12.7 Inaction

Syntax

stop

Static semantics

$$\frac{}{\mathcal{C} \vdash \textbf{stop} \Rightarrow \textbf{exit}(\textbf{none})}$$

Untimed dynamic semantics None.

Timed dynamic semantics

$$\frac{}{\mathcal{E} \vdash \textbf{stop} \xrightarrow{\varepsilon(d)} \textbf{stop}}[0 < d]$$

12.8 Time block

Syntax

block

Static semantics

$$\frac{}{\mathcal{C} \vdash \textbf{block} \Rightarrow \textbf{exit}\ (\textbf{none})}$$

Untimed dynamic semantics None.

Timed dynamic semantics None.

12.9 Delay

Syntax

wait (E)

Static semantics

$$\frac{\mathcal{C} \vdash E \Rightarrow \textbf{exit}(\text{time})}{\mathcal{C} \vdash \textbf{wait}\ (E) \Rightarrow \textbf{exit}()}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash E \xrightarrow{\delta(0)} E'}{\mathcal{E} \vdash \textbf{wait}\ (E) \xrightarrow{\delta()} \textbf{block}}$$

$$\frac{\mathcal{E} \vdash E \xrightarrow{XN} E'}{\mathcal{E} \vdash \textbf{wait}\ (E) \xrightarrow{XN} \textbf{wait}\ (E')}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash E \xrightarrow{\delta(d+d')} E'}{\mathcal{E} \vdash \textbf{wait}\ (E) \xrightarrow{\varepsilon(d)} \textbf{wait}\ (d')}[0 < d]$$

12.10 Assignment

Syntax

$P := E$

The pattern must be irrefutable.

Static semantics

$$\frac{\begin{array}{l}\mathcal{C} \vdash E \Rightarrow \textbf{exit}(T) \\ \mathcal{C} \vdash (P \Rightarrow T) \Rightarrow (RT)\end{array}}{\mathcal{C} \vdash P := E \Rightarrow \textbf{exit}(RT)}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash E \xrightarrow{X(RN)} E'}{\mathcal{E} \vdash P := E \xrightarrow{X(RN)} P := E'}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash E \xrightarrow{\delta(N)} E' \\ \mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN)\end{array}}{\mathcal{E} \vdash P := E \xrightarrow{\delta(RN)} \textbf{block}}$$

Timed dynamic semantics None.

12.11 Sequential composition

Syntax

$B\ ;\ B$

Static semantics

$$\frac{\begin{array}{c}\mathcal{C} \vdash B_1 \Rightarrow \textbf{exit}(RT_1) \\ \mathcal{C}; RT_1 \vdash B_2 \Rightarrow \textbf{exit}(RT_2)\end{array}}{\mathcal{C} \vdash B_1\ ;\ B_2 \Rightarrow \textbf{exit}(RT_1, RT_2)}[RT_1 \text{ and } RT_2 \text{ have disjoint fi elds}]$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{a(RN)} B_1'}{\mathcal{E} \vdash B_1 \;;\; B_2 \xrightarrow{a(RN)} B_1' \;;\; B_2}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN_1)} B_1' \\ \mathcal{E} \vdash B_2[RN_1] \xrightarrow{a(RN_2)} B_2'\end{array}}{\mathcal{E} \vdash B_1 \;;\; B_2 \xrightarrow{a(RN_2)} \mathbf{exit}(RN_1) \;;\; B_2'}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN_1)} B_1' \\ \mathcal{E} \vdash B_2[RN_1] \xrightarrow{\delta(RN_2)} B_2'\end{array}}{\mathcal{E} \vdash B_1 \;;\; B_2 \xrightarrow{\delta(RN_1,RN_2)} \mathbf{block}}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d)} B_1'}{\mathcal{E} \vdash B_1 \;;\; B_2 \xrightarrow{\varepsilon(d)} B_1' \;;\; B_2}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN_1@d_1)} B_1' \\ \mathcal{E} \vdash B_2[RN_1] \xrightarrow{\varepsilon(d_2)} B_2'\end{array}}{\mathcal{E} \vdash B_1 \;;\; B_2 \xrightarrow{\varepsilon(d_1+d_2)} \mathbf{exit}(RN_1) \;;\; B_2'}$$

12.12 Disabling

Syntax

B [> B

Static semantics

$$\frac{\mathcal{C} \vdash B_1 \Rightarrow \mathbf{exit}(RT) \qquad \mathcal{C} \vdash B_2 \Rightarrow \mathbf{exit}(RT)}{\mathcal{C} \vdash B_1 \;[\!>\; B_2 \Rightarrow \mathbf{exit}(RT)}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{a(RN)} B_1'}{\mathcal{E} \vdash B_1 \;[\!>\; B_2 \xrightarrow{a(RN)} B_1' \;[\!>\; B_2}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN)} B_1'}{\mathcal{E} \vdash B_1 \;[\!>\; B_2 \xrightarrow{\delta(RN)} B_1'}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{\mu(RN)} B_2'}{\mathcal{E} \vdash B_1 \;[\!>\; B_2 \xrightarrow{\mu(RN)} B_2'}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\varepsilon(d)} B_2'}{\mathcal{E} \vdash B_1 \; [> B_2 \xrightarrow{\varepsilon(d)} B_1' \; [> B_2'}$$

12.13 Synchronization

Syntax

$B \; || \; B$

Static semantics

$$\frac{\mathcal{C} \vdash B_1 \Rightarrow \mathbf{exit}(RT_1) \qquad \mathcal{C} \vdash B_2 \Rightarrow \mathbf{exit}(RT_2)}{\mathcal{C} \vdash B_1 \; || \; B_2 \Rightarrow \mathbf{exit}(RT_1, RT_2)} [RT_1 \text{ and } RT_2 \text{ have disjoint fi elds}]$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\mathbf{i}()} B_1'}{\mathcal{E} \vdash B_1 \; || \; B_2 \xrightarrow{\mathbf{i}()} B_1' \; || \; B_2}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{\mathbf{i}()} B_2'}{\mathcal{E} \vdash B_1 \; || \; B_2 \xrightarrow{\mathbf{i}()} B_1 \; || \; B_2'}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{G(RN)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{G(RN)} B_2'}{\mathcal{E} \vdash B_1 \; || \; B_2 \xrightarrow{G(RN)} B_1' \; || \; B_2'}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{X(RN)} B_1'}{\mathcal{E} \vdash B_1 \; || \; B_2 \xrightarrow{X(RN)} B_1' \; || \; B_2}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{X(RN)} B_2'}{\mathcal{E} \vdash B_1 \; || \; B_2 \xrightarrow{X(RN)} B_1 \; || \; B_2'}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN_1)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\delta(RN_2)} B_2'}{\mathcal{E} \vdash B_1 \; || \; B_2 \xrightarrow{\delta(RN_1, RN_2)} B_1' \; || \; B_2'}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\varepsilon(d)} B_2'}{\mathcal{E} \vdash B_1 \; || \; B_2 \xrightarrow{\varepsilon(d)} B_1' \; || \; B_2'}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN@d)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\varepsilon(d+d')} B_2'}{\mathcal{E} \vdash B_1 \; || \; B_2 \xrightarrow{\varepsilon(d+d')} \mathbf{exit}(RN) \; || \; B_2'} [0 < d']$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d+d')} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\delta(RN@d)} B_2'}{\mathcal{E} \vdash B_1 \; || \; B_2 \xrightarrow{\varepsilon(d+d')} B_1' \; || \; \mathbf{exit}(RN)} [0 < d']$$

12.14 Concurrency

Syntax

B |[$[G(,G)^*]$]| B

Static semantics

$$\frac{\mathcal{C} \vdash B_1 \Rightarrow \mathbf{exit}(RT_1) \qquad \mathcal{C} \vdash B_2 \Rightarrow \mathbf{exit}(RT_2) \\ \mathcal{C} \vdash G_1 \Rightarrow \mathbf{gate}(RT_1') \quad \cdots \quad \mathcal{C} \vdash G_n \Rightarrow \mathbf{gate}(RT_n')}{\mathcal{C} \vdash B_1 \;|[G_1, \ldots, G_n]|\; B_2 \Rightarrow \mathbf{exit}(RT_1, RT_2)} [RT_1 \text{ and } RT_2 \text{ have disjoint fields}]$$

Untimed dynamic semantics

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

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

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{G_i(RN)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{G_i(RN)} B_2'}{\mathcal{E} \vdash B_1 \;|[\vec{G}]|\; B_2 \xrightarrow{G_i(RN)} B_1' \;|[\vec{G}]|\; B_2'}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN_1)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\delta(RN_2)} B_2'}{\mathcal{E} \vdash B_1 \;|[\vec{G}]|\; B_2 \xrightarrow{\delta(RN_1, RN_2)} B_1' \;|[\vec{G}]|\; B_2'}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\varepsilon(d)} B_2'}{\mathcal{E} \vdash B_1 \;|[\vec{G}]|\; B_2 \xrightarrow{\varepsilon(d)} B_1' \;|[\vec{G}]|\; B_2'}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN@d)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\varepsilon(d+d')} B_2'}{\mathcal{E} \vdash B_1 \;|[\vec{G}]|\; B_2 \xrightarrow{\varepsilon(d+d')} \mathbf{exit}(RN) \;|[\vec{G}]|\; B_2'} [0 < d']$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d+d')} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\delta(RN@d)} B_2'}{\mathcal{E} \vdash B_1 \;|[\vec{G}]|\; B_2 \xrightarrow{\varepsilon(d+d')} B_1' \;|[\vec{G}]|\; \mathbf{exit}(RN)} [0 < d']$$

12.15 Choice

Syntax

B [] B

Static semantics

$$\frac{\mathcal{C} \vdash B_1 \Rightarrow \mathbf{exit}(RT) \qquad \mathcal{C} \vdash B_2 \Rightarrow \mathbf{exit}(RT)}{\mathcal{C} \vdash B_1 \;[]\; B_2 \Rightarrow \mathbf{exit}(RT)}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\mu(RN)} B_1'}{\mathcal{E} \vdash B_1 \;[]\; B_2 \xrightarrow{\mu(RN)} B_1'}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{\mu(RN)} B_2'}{\mathcal{E} \vdash B_1 \;[]\; B_2 \xrightarrow{\mu(RN)} B_2'}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\varepsilon(d)} B_2'}{\mathcal{E} \vdash B_1 \;[]\; B_2 \xrightarrow{\varepsilon(d)} B_1' \;[]\; B_2'}$$

12.16 Choice over values

Syntax

$$\textbf{choice}\ P\ [\textbf{after}(N)]\ []\ B$$

The default value is **after**(0).

Static semantics

$$\frac{\begin{array}{c} \mathcal{C} \vdash (P \Rightarrow \textbf{any}) \Rightarrow (RT) \\ \mathcal{C} \vdash N \Rightarrow \text{time} \\ \mathcal{C}; RT \vdash B \Rightarrow \textbf{exit}(RT') \end{array}}{\mathcal{C} \vdash \textbf{choice}\ P\ \textbf{after}(N)\ []\ B \Rightarrow \textbf{exit}(RT')}$$

Untimed dynamic semantics

$$\frac{\begin{array}{l} \mathcal{E} \vdash (N \Rightarrow \textbf{any}) \\ \mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN) \\ \mathcal{E} \vdash B[RN] \xrightarrow{a(RN'@d)} B' \end{array}}{\mathcal{E} \vdash \textbf{choice}\ P\ \textbf{after}(d)\ []\ B \xrightarrow{a(RN')} B'}$$

Note: this semantics is the only place where the timed semantics is used in the untimed semantics, thus breaking the stratification which is useful in proving the semantics well-defined.

Timed dynamic semantics

$$\frac{\forall N\,.\,((\mathcal{E} \vdash N \Rightarrow \textbf{any} \text{ and } \mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN)) \text{ implies } B[RN] \xrightarrow{\varepsilon(d+d')})}{\mathcal{E} \vdash \textbf{choice}\ P\ \textbf{after}(d)\ []\ B \xrightarrow{\varepsilon(d')} \textbf{choice}\ P\ \textbf{after}(d+d')\ []\ B}[0 < d']$$

Note: in the presence of time nondeterminism, this operator does *not* behave as a generalization of []. For example:

$$(\textbf{choice any after}(0)\ []\ ?y := \textbf{any}\ \text{bool};\ G(!y) \xrightarrow{\varepsilon(1)} (\textbf{choice any after}(1)\ []\ ?y := \textbf{any}\ \text{bool};\ G(!y)$$

For these reasons, this semantics is highly undesirable, and it may be better to replace:

$$\textbf{choice}\ ?x{:}T\ []\ B$$

by

$$\textbf{local var}\ \mathtt{x}{:}T\ \textbf{init}\ ?x := \textbf{any}\ T\ \textbf{in}\ B$$

12.17 Trap

Syntax

$$\textbf{trap}\ (\textbf{exception}\ X\ [(IP)]\ \textbf{is}\ B)^*\ [\textbf{exit}\ [P]\ \textbf{is}\ B]\ \textbf{in}\ B$$

The default input parameter is `()` and the default exit pattern is `()`.

Static semantics

$$\frac{\begin{array}{c} \mathcal{C} \vdash (RT_1) \Rightarrow \textbf{type} \quad \cdots \quad \mathcal{C} \vdash (RT_n) \Rightarrow \textbf{type} \\ \mathcal{C} \vdash (RP_1 \Rightarrow RT_1) \Rightarrow (RT_1') \quad \cdots \quad \mathcal{C} \vdash (RP_n \Rightarrow RT_n) \Rightarrow (RT_n') \\ \mathcal{C}; RT_1' \vdash B_1 \Rightarrow \textbf{exit}(RT) \quad \cdots \quad \mathcal{C}; RT_n' \vdash B_n \Rightarrow \textbf{exit}(RT) \\ \mathcal{C}; X_1 \Rightarrow \textbf{exn}(RT_1), \ldots, X_n \Rightarrow \textbf{exn}(RT_n) \vdash B \Rightarrow \textbf{exit}(RT) \end{array}}{\begin{array}{l} \mathcal{C} \vdash (\textbf{trap exception}\ X_1(RP_1 : RT_1)\ \textbf{is}\ B_1 \\ \quad \ldots\ \textbf{exception}\ X_n(RP_n : RT_n)\ \textbf{is}\ B_n\ \textbf{in}\ B) \\ \quad\quad \Rightarrow \textbf{exit}(RT) \end{array}}$$

$$\frac{\begin{array}{c} \mathcal{C} \vdash (RT_1) \Rightarrow \textbf{type} \quad \cdots \quad \mathcal{C} \vdash (RT_n) \Rightarrow \textbf{type} \\ \mathcal{C} \vdash (RP_1 \Rightarrow RT_1) \Rightarrow (RT_1') \quad \cdots \quad \mathcal{C} \vdash (RP_n \Rightarrow RT_n) \Rightarrow (RT_n') \\ \mathcal{C}; RT_1' \vdash B_1 \Rightarrow \textbf{exit}(RT) \quad \cdots \quad \mathcal{C}; RT_n' \vdash B_n \Rightarrow \textbf{exit}(RT) \\ \mathcal{C}; X_1 \Rightarrow \textbf{exn}(RT_1), \ldots, X_n \Rightarrow \textbf{exn}(RT_n) \vdash B \Rightarrow \textbf{exit}(RT') \\ \mathcal{C} \vdash (P \Rightarrow (RT')) \Rightarrow (RT'') \\ \mathcal{C}; RT'' \vdash B' \Rightarrow \textbf{exit}(RT) \end{array}}{\begin{array}{l} \mathcal{C} \vdash (\textbf{trap exception}\ X_1(RP_1 : RT_1)\ \textbf{is}\ B_1 \\ \quad \ldots\ \textbf{exception}\ X_n(RP_n : RT_n)\ \textbf{is}\ B_n \textbf{exit}\ P\ \textbf{is}\ B'\ \textbf{in}\ B) \\ \quad\quad \Rightarrow \textbf{exit}(RT) \end{array}}$$

Untimed dynamic semantics Here $\vec{\mu}$ ranges over X and **exit** (which we consider to be equal to δ).

$$\frac{\mathcal{E} \vdash B \xrightarrow{\mu(RN)} B'}{\mathcal{E} \vdash (\textbf{trap}\ \vec{\mu}\ (\vec{RP} : \vec{RT})\ \textbf{is}\ \vec{B}\ \textbf{in}\ B) \xrightarrow{\mu(RN)} (\textbf{trap}\ \vec{\mu}\ (\vec{RP} : \vec{RT})\ \textbf{is}\ \vec{B}\ \textbf{in}\ B)} [\mu \notin \vec{\mu}]$$

$$\frac{\begin{array}{l} \mathcal{E} \vdash B \xrightarrow{\mu_i(RN)} B' \\ \mathcal{E} \vdash ((RP_i) \Rightarrow (RN)) \Rightarrow (RN') \\ \mathcal{E} \vdash B_i[RN'] \xrightarrow{\mu(RN'')} B_i' \end{array}}{\mathcal{E} \vdash (\textbf{trap}\ \vec{\mu}\ (\vec{RP} : \vec{RT})\ \textbf{is}\ \vec{B}\ \textbf{in}\ B) \xrightarrow{\mu(RN'')} B_i'}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B \xrightarrow{\varepsilon(d)} B'}{\mathcal{E} \vdash (\textbf{trap}\ \vec{\mu}\ (\vec{RP} : \vec{RT})\ \textbf{is}\ \vec{B}\ \textbf{in}\ B) \xrightarrow{\varepsilon(d)} (\textbf{trap}\ \vec{\mu}\ (\vec{RP} : \vec{RT})\ \textbf{is}\ \vec{B}\ \textbf{in}\ B')}$$

$$\frac{\begin{array}{l} \mathcal{E} \vdash B \xrightarrow{\mu_i(RN@d)} B' \\ \mathcal{E} \vdash ((RP_i) \Rightarrow (RN)) \Rightarrow (RN') \\ \mathcal{E} \vdash B_i[RN'] \xrightarrow{\varepsilon(d')} B_i' \end{array}}{\mathcal{E} \vdash (\textbf{trap}\ \vec{\mu}\ (\vec{RP} : \vec{RT})\ \textbf{is}\ \vec{B}\ \textbf{in}\ B) \xrightarrow{\varepsilon(d+d')} B_i'}$$

12.18 Case

Syntax

$\textbf{case}\ E\,[:T]\ \textbf{is}\ BM$

The default type is the principal type of E (*note* this requires static information).

The match is required to be exhaustive. If it is not, a default **any** $\rightarrow$ **raise** Match clause is added.

Static semantics

$$\frac{\mathcal{C} \vdash E \Rightarrow \textbf{exit}(T) \qquad \mathcal{C} \vdash (BM \Rightarrow T) \Rightarrow \textbf{exit}(RT)}{\mathcal{C} \vdash \textbf{case}\ E : T\ \textbf{is}\ BM \Rightarrow \textbf{exit}(RT)}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash E \xrightarrow{\delta(N)} E' \qquad \mathcal{E} \vdash (BM \Rightarrow N) \xrightarrow{\mu(RN)} B}{\mathcal{E} \vdash \textbf{case}\ E : T\ \textbf{is}\ BM \xrightarrow{\mu(RN)} B}$$

$$\frac{\mathcal{E} \vdash E \xrightarrow{X(RN)} E'}{\mathcal{E} \vdash \textbf{case}\ E : T\ \textbf{is}\ BM \xrightarrow{X(RN)} \textbf{case}\ E' : T\ \textbf{is}\ BM}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash E \xrightarrow{\delta(N)} E' \qquad \mathcal{E} \vdash (BM \Rightarrow N) \xrightarrow{\varepsilon(d)} B}{\mathcal{E} \vdash \textbf{case}\ E : T\ \textbf{is}\ BM \xrightarrow{\varepsilon(d)} B}$$

12.19 Variable declaration

Syntax

$\textbf{local var}\ LV\ [\textbf{init}\ B]\ \textbf{in}\ B$

The default initialization section is **init exit**.

Static semantics

$$\frac{\mathcal{C} \vdash (RV \Rightarrow RT) \Rightarrow (RT_1, RT_2) \qquad \mathcal{C} \vdash B_1 \Rightarrow \textbf{exit}(RT_1) \qquad \mathcal{C}; RT_1 \vdash B_2 \Rightarrow \textbf{exit}(RT_2, RT')}{\mathcal{C} \vdash \textbf{local var}\ RV : RT\ \textbf{init}\ B_1\ \textbf{in}\ B_2 \Rightarrow \textbf{exit}(RT')}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{a(RN)} B_1'}{\mathcal{E} \vdash \textbf{local var } RV : RT \textbf{ init } B_1 \textbf{ in } B_2 \xrightarrow{a(RN)} \textbf{local var } RV : RT \textbf{ init } B_1' \textbf{ in } B_2}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN_1)} B_1' \qquad \mathcal{E} \vdash B_2[RN_1] \xrightarrow{a(RN_2)} B_2'}{\mathcal{E} \vdash \textbf{local var } RV : RT \textbf{ init } B_1 \textbf{ in } B_2 \xrightarrow{a(RN_2)} \textbf{local var } RV : RT \textbf{ init exit}(RN_1) \textbf{ in } B_2'}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN_1)} B_1' \qquad \mathcal{E} \vdash B_2[RN_1] \xrightarrow{\delta(RN_2, RN)} B_2' \qquad \mathcal{E} \vdash (RN_1, RN_2) \Rightarrow (RT_1, RT_2) \qquad \mathcal{E} \vdash (RV \Rightarrow RT) \Rightarrow (RT_1, RT_2)}{\mathcal{E} \vdash \textbf{local var } RV : RT \textbf{ init } B_1 \textbf{ in } B_2 \xrightarrow{\delta(RN)} \textbf{block}}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d)} B_1'}{\mathcal{E} \vdash \textbf{local var } RV : RT \textbf{ init } B_1 \textbf{ in } B_2 \xrightarrow{\varepsilon(d)} \textbf{local var } RV : RT \textbf{ init } B_1' \textbf{ in } B_2}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN_1 @ d_1)} B_1' \qquad \mathcal{E} \vdash B_2[RN_1] \xrightarrow{\varepsilon(d_2)} B_2'}{\mathcal{E} \vdash \textbf{local var } RV : RT \textbf{ init } B_1 \textbf{ in } B_2 \xrightarrow{\varepsilon(d_1 + d_2)} \textbf{local var } RV : RT \textbf{ init exit}(RN_1) \textbf{ in } B_2'}$$

12.20 Gate hiding

Syntax

$\textbf{hide } G[(RT)](, G[(RT)])^* \textbf{ in } B$

The default gate type is $(\textbf{etc})$.

Static semantics

$$\frac{\mathcal{C} \vdash (RT_1) \Rightarrow \textbf{type} \quad \cdots \quad \mathcal{C} \vdash (RT_n) \Rightarrow \textbf{type} \qquad \mathcal{C}; G_1 \Rightarrow \textbf{gate}(RT_1), \ldots, G_n \Rightarrow \textbf{gate}(RT_n) \vdash B \Rightarrow \textbf{exit}(RT)}{\mathcal{C} \vdash \textbf{hide } G_1(RT_1), \cdots, G_n(RT_n) \textbf{ in } B \Rightarrow \textbf{exit}(RT)}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B \xrightarrow{a(RN)} B'}{\mathcal{E} \vdash \textbf{hide } \vec{G}(\vec{RT}) \textbf{ in } B' \xrightarrow{a(RN)} \textbf{hide } \vec{G}(\vec{RT}) \textbf{ in } B'}[a \notin \vec{G}]$$

$$\frac{\mathcal{E} \vdash B \xrightarrow{G_i(RN)} B' \qquad \mathcal{E} \vdash (RN) \Rightarrow (RT_i)}{\mathcal{E} \vdash \textbf{hide } \vec{G}(\vec{RT}) \textbf{ in } B' \xrightarrow{\mathbf{i}()} \textbf{hide } \vec{G}(\vec{RT}) \textbf{ in } B'}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B \xrightarrow{\varepsilon(d)} B' \textbf{ refusing } \vec{G}(\vec{RT})}{\mathcal{E} \vdash \textbf{hide } \vec{G}(\vec{RT}) \textbf{ in } B' \xrightarrow{\varepsilon(d)} \textbf{hide } \vec{G}(\vec{RT}) \textbf{ in } B'}$$

where $\mathcal{E} \vdash B \xrightarrow{\varepsilon(d)} B'$ **refusing** $\vec{G}(\vec{RT})$ if we can find a path $\{B_{d'} \mid 0 \leq d' \leq d\}$ such that:

- $B = B_0$ and $B' = B_d$.
- $\mathcal{E} \vdash B_{d'} \xrightarrow{\varepsilon(d'')} B_{d'+d''}$.
- There is no $\mathcal{E} \vdash (RN) \Rightarrow (RT_i)$ and $d' < d$ such that $\mathcal{E} \vdash B_{d'} \xrightarrow{G_i(RN)} B''$.

Note: this more complex definition of hiding is required in the presence of time nondeterminism.

12.21 Renaming

Syntax

$$\textbf{rename } (G\ [(IP)]\ \textbf{is}\ G\ [P] \ \mid\ X\ [(IP)]\ \textbf{is}\ [\textbf{signal}]\ X\ [E])^* \textbf{ in } B$$

The default gate input parameter is (**etc**), the default gate pattern is !$argv, the default exception input parameter is (), and the default exception value is $argv.

Static semantics

$$\frac{\begin{array}{c} \mathcal{C} \vdash (RT_1) \Rightarrow \textbf{type} \quad \cdots \quad \mathcal{C} \vdash (RT_m) \Rightarrow \textbf{type} \\ \mathcal{C} \vdash (RP_1 \Rightarrow RT_1) \Rightarrow (RT_1'') \quad \cdots \quad \mathcal{C} \vdash (RP_m \Rightarrow RT_m) \Rightarrow (RT_m'') \\ \mathcal{C}; RT_1'' \vdash G_1'\ P_1 \Rightarrow \textbf{exit}\ () \quad \cdots \quad \mathcal{C}; RT_m'' \vdash G_m'\ P_m \Rightarrow \textbf{exit}\ () \\ \mathcal{C} \vdash (RT_1') \Rightarrow \textbf{type} \quad \cdots \quad \mathcal{C} \vdash (RT_m') \Rightarrow \textbf{type} \\ \mathcal{C} \vdash (RP_1' \Rightarrow RT_1') \Rightarrow (RT_1''') \quad \cdots \quad \mathcal{C} \vdash (RP_n' \Rightarrow RT_n') \Rightarrow (RT_n''') \\ \mathcal{C}; RT_1''' \vdash \textbf{signal}\ X_1'\ E_1 \Rightarrow \textbf{exit}\ () \quad \cdots \quad \mathcal{C}; RT_n''' \vdash \textbf{signal}\ X_n'\ E_n \Rightarrow \textbf{exit}\ () \\ \mathcal{C}; G_1 \Rightarrow \textbf{gate}\ (RT_1), \ldots, G_m \Rightarrow \textbf{gate}\ (RT_m), \\ X_1 \Rightarrow \textbf{exn}\ (RT_1'), \ldots, X_m \Rightarrow \textbf{exn}\ (RT_n') \vdash B \Rightarrow \textbf{exit}(RT) \end{array}}{\begin{array}{l} \mathcal{C} \vdash (\textbf{rename} \\ \qquad G_1(RP_1 : RT_1)\ \textbf{is}\ G_1'\ P_1\ \cdots \\ \qquad G_m(RP_m : RT_m)\ \textbf{is}\ G_m'\ P_m \\ \qquad X_1(RP_1' : RT_1')\ \textbf{is signal}\ X_1'\ E_1\ \cdots \\ \qquad X_m(RP_n' : RT_n')\ \textbf{is signal}\ X_n'\ E_n \\ \quad \textbf{in}\ B) \Rightarrow \textbf{exit}(RT) \end{array}}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B \xrightarrow{\mu(RN)} B'}{\mathcal{E} \vdash (\textbf{rename}\ \vec{\mu}\ (\vec{RP} : \vec{RT})\ \textbf{is}\ \vec{B}\ \textbf{in}\ B) \xrightarrow{\mu(RN)} (\textbf{rename}\ \vec{\mu}\ (\vec{RP} : \vec{RT})\ \textbf{is}\ \vec{B}\ \textbf{in}\ B')}[\mu \notin \vec{\mu}]$$

$$\frac{\begin{array}{l} \mathcal{E} \vdash B \xrightarrow{\mu_i(RN)} B' \\ \mathcal{E} \vdash (((RP_i) : (RT_i)) \Rightarrow (RN)) \Rightarrow (RN') \\ \mathcal{E} \vdash B_i[RN'] \xrightarrow{\mu(RN'')} B_i' \end{array}}{\mathcal{E} \vdash (\textbf{rename}\ \vec{\mu}\ (\vec{RP} : \vec{RT})\ \textbf{is}\ \vec{B}\ \textbf{in}\ B) \xrightarrow{\mu(RN'')} (\textbf{rename}\ \vec{\mu}\ (\vec{RP} : \vec{RT})\ \textbf{is}\ \vec{B}\ \textbf{in}\ B')}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B \xrightarrow{\varepsilon(d)} B'}{\mathcal{E} \vdash (\textbf{rename } \vec{\mu}\ (\vec{RP}:\vec{RT})\ \textbf{is } \vec{B} \textbf{ in } B) \xrightarrow{\varepsilon(d)} (\textbf{rename } \vec{\mu}\ (\vec{RP}:\vec{RT})\ \textbf{is } \vec{B} \textbf{ in } B')}$$

12.22 Process instantiation

Syntax

$\Pi\ [\,[\![G(,G)^*]\!]\,]\ [E]\ [\,[\![X(,X)^*]\!]\,]$

The default gate and exception lists are the empty list `[]`, and the default argument is `()`.

Static semantics

$$\frac{\begin{array}{l}\mathcal{C} \vdash \Pi \Rightarrow [\textbf{gate}(RT_1), \ldots, \textbf{gate}(RT_m)]\,(RT')\,[\textbf{exn}(RT_1'), \ldots, \textbf{exn}(RT_n')] \rightarrow \textbf{exit}(RT) \\ \mathcal{C} \vdash G_1 \Rightarrow \textbf{gate}(RT_1) \quad \cdots \quad \mathcal{C} \vdash G_m \Rightarrow \textbf{gate}(RT_m) \\ \mathcal{C} \vdash E \Rightarrow \textbf{exit}((RT')) \\ \mathcal{C} \vdash X_1 \Rightarrow \textbf{exn}(RT_1') \quad \cdots \quad \mathcal{C} \vdash X_m \Rightarrow \textbf{exn}(RT_n')\end{array}}{\mathcal{C} \vdash \Pi\ [G_1, \ldots, G_m]\ E\ [X_1, \ldots, X_n] \Rightarrow \textbf{exit}(RT)}$$

Untimed dynamic semantics

$$\frac{\begin{array}{l}\mathcal{E} \vdash \Pi \Rightarrow \lambda[\vec{G}'(\vec{RT})]\,(RP:RT)\,[\vec{X}'(\vec{RT}')] \rightarrow B \\ \mathcal{E} \vdash (\textbf{rename } \vec{G}'(\vec{RT})\ \textbf{is } \vec{G}\ \vec{X}'(\vec{RT}')\ \textbf{is } \vec{X} \textbf{ in case } E:(RT)\ \textbf{is } (RP) \rightarrow B) \xrightarrow{\mu(RN)} B'\end{array}}{\mathcal{E} \vdash \Pi\ [\vec{G}]\ E\ [\vec{X}] \xrightarrow{\mu(RN)} B'}$$

Timed dynamic semantics

$$\frac{\begin{array}{l}\mathcal{E} \vdash \Pi \Rightarrow \lambda[\vec{G}'(\vec{RT})]\,(RP:RT)\,[\vec{X}'(\vec{RT}')] \rightarrow B \\ \mathcal{E} \vdash (\textbf{rename } \vec{G}'(\vec{RT})\ \textbf{is } \vec{G}\ \vec{X}'(\vec{RT}')\ \textbf{is } \vec{X} \textbf{ in case } E:(RT)\ \textbf{is } (RP) \rightarrow B) \xrightarrow{\varepsilon(d)} B'\end{array}}{\mathcal{E} \vdash \Pi\ [\vec{G}]\ E\ [\vec{X}] \xrightarrow{\varepsilon(d)} B'}$$

12.23 Iteration

Syntax

loop forever [**var** *LV*] [**init** *B*] **in** *B*

The default local variables are **var** () and the default initialization is **init exit**.

Static semantics

$$\frac{\begin{array}{r}\mathcal{C} \vdash (RV \Rightarrow RT) \Rightarrow (RT_1, RT_2) \\ \mathcal{C} \vdash B_1 \Rightarrow \textbf{exit}(RT_1) \\ \mathcal{C}; RT_1 \vdash B_2 \Rightarrow \textbf{exit}(RT_1, RT_2)\end{array}}{\mathcal{C} \vdash \textbf{loop forever var } RV:RT \textbf{ init } B_1 \textbf{ in } B_2 \Rightarrow \textbf{exit}(\textbf{none})}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash \textbf{local var } RV : RT \textbf{ init } B_1 \textbf{ in } (\textbf{loop forever var } RV : RT \textbf{ init } B_2 \textbf{ in } B_2) \xrightarrow{\mu N} B}{\mathcal{E} \vdash \textbf{loop forever var } RV : RT \textbf{ init } B_1 \textbf{ in } B_2 \xrightarrow{\mu N} B}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash \textbf{local var } RV : RT \textbf{ init } B_1 \textbf{ in } (\textbf{loop forever var } RV : RT \textbf{ init } B_2 \textbf{ in } B_2) \xrightarrow{\varepsilon(d)} B}{\mathcal{E} \vdash \textbf{loop forever var } RV : RT \textbf{ init } B_1 \textbf{ in } B_2 \xrightarrow{\varepsilon(d)} B}$$

12.24 Interleaving

Syntax

$B \mid\mid\mid B$

Syntax sugar

$B_1 \mid\mid\mid B_2 \stackrel{\mathsf{def}}{=} B_1 \mid [\,] \mid B_2$

12.25 Termination

Syntax

$\textbf{exit}(RE)$

Syntax sugar

$\textbf{exit}(RE) \stackrel{\mathsf{def}}{=} RE$

12.26 Raising exception

Syntax

$\textbf{raise } X\ E$

Syntax sugar

$\textbf{raise } X\ E \stackrel{\mathsf{def}}{=} \textbf{signal } X\ E \,;\ \textbf{block}$

12.27 If-then-else

Syntax

$\textbf{if } E \textbf{ then } B\ [\textbf{else } B]$

The default **else** clause is **exit**.

Syntax sugar

$\textbf{if } E \textbf{ then } B_1 \textbf{ else } B_2 \stackrel{\mathsf{def}}{=} \textbf{case } E : \mathrm{bool} \textbf{ is } \mathrm{true} \rightarrow B_1 \mid \mathrm{false} \rightarrow B_2$

12.28 Process instantiation with in/out parameters

Syntax

$$\Pi\ [\texttt{[}[G(\texttt{,}G)^*]\texttt{]}]\ \texttt{(}RE\texttt{,}RP\texttt{)}\ [\texttt{[}[X(\texttt{,}X)^*]\texttt{]}]$$

The default gate and exception lists are the empty list `[]`.

Syntax sugar

$$(\Pi\ \texttt{[}\vec{G}\texttt{]}\ \texttt{(}RE\texttt{,}RP\texttt{)}\ \texttt{[}\vec{X}\texttt{]}) \stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{trap exit}\ \texttt{(?x)}\ \textbf{is}\ \texttt{(}RP\texttt{)}\ \texttt{:=}\ \texttt{x} \\ \textbf{in}\ \Pi\ \texttt{[}\vec{G}\texttt{]}\ \texttt{(}RE\texttt{)}\ \texttt{[}\vec{X}\texttt{]} \end{array} \right)$$

12.29 Breakable iteration

Syntax

$$\textbf{loop}\ [X]\ [\texttt{(}T\texttt{)}]\ [\textbf{var}\ LV]\ [\textbf{init}\ B]\ \textbf{in}\ B$$

The default exception name is inner, the default local variable declaration is **var** (), and the default initalization is **init exit**.

Syntax sugar

$$\left(\begin{array}{l} \textbf{loop}\ X \\ \quad \textbf{var}\ LV \\ \quad \textbf{init}\ B_1 \\ \quad \textbf{in}\ B_2 \end{array} \right) \stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{trap} \\ \quad \textbf{exception}\ X\ \textbf{is exit} \\ \textbf{in loop forever} \\ \quad \textbf{var}\ LV \\ \quad \textbf{init}\ B_1 \\ \quad \textbf{in}\ B_2 \end{array} \right)$$

$$\left(\begin{array}{l} \textbf{loop}\ X\texttt{(}T\texttt{)} \\ \quad \textbf{var}\ LV \\ \quad \textbf{init}\ B_1 \\ \quad \textbf{in}\ B_2 \end{array} \right) \stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{trap} \\ \quad \textbf{exception}\ X\texttt{(?x:}T\texttt{)}\ \textbf{is exit}\ \texttt{(x)} \\ \textbf{in loop forever} \\ \quad \textbf{var}\ LV \\ \quad \textbf{init}\ B_1 \\ \quad \textbf{in}\ B_2 \end{array} \right)$$

12.30 Breaking iteration

Syntax

$$\textbf{break}\ [X]\ [\texttt{(}E\texttt{)}]$$

The default exception name is inner.

Syntax sugar

$$\textbf{break}\ X\ \texttt{(}E\texttt{)} \stackrel{\text{def}}{=} \textbf{raise}\ X\ \texttt{(}E\texttt{)}$$

$$\textbf{break}\ X \stackrel{\text{def}}{=} \textbf{raise}\ X$$

13 Behaviour pattern-matching

13.1 Overview

Syntax

$$\begin{array}{lcll} BM & ::= & P\ [E] \rightarrow B & \textit{single match}\ \ (\mathrm{M}_c1) \\ & \mid & BM \mid BM & \textit{multiple match}\ \ (\mathrm{M}_c2) \end{array}$$

Static semantics

$$\mathcal{C} \vdash (BM \Rightarrow T) \Rightarrow \textbf{exit}(RT)$$

Dynamic semantics

$$\mathcal{E} \vdash (BM \Rightarrow N) \Rightarrow \textbf{fail}$$

$$\mathcal{E} \vdash (BM \Rightarrow N) \xrightarrow{\alpha(RN)} B$$

$$\alpha ::= \mu \mid \varepsilon$$

13.2 Single match

Syntax

$$P\ [[E]] \rightarrow B$$

The default selection predicate is [true].

Static semantics

$$\frac{\begin{array}{c}\mathcal{C} \vdash (P \Rightarrow T) \Rightarrow (RT) \\ \mathcal{C}; RT \vdash E \Rightarrow \textbf{exit}(\mathrm{bool}) \\ \mathcal{C}; RT \vdash B \Rightarrow \textbf{exit}(RT')\end{array}}{\mathcal{C} \vdash ((P\ [E] \rightarrow B) \Rightarrow T) \Rightarrow \textbf{exit}(RT')}$$

Dynamic semantics

$$\frac{\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow \textbf{fail}}{\mathcal{E} \vdash ((P\ [E] \rightarrow B) \Rightarrow N) \Rightarrow \textbf{fail}}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN) \\ \mathcal{E} \vdash E[RN] \xrightarrow{\delta(\mathrm{false})} E'\end{array}}{\mathcal{E} \vdash ((P\ [E] \rightarrow B) \Rightarrow N) \Rightarrow \textbf{fail}}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN) \\ \mathcal{E} \vdash E[RN] \xrightarrow{X(RN)} E'\end{array}}{\mathcal{E} \vdash ((P\ [E] \rightarrow B) \Rightarrow N) \Rightarrow \textbf{fail}}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN) \\ \mathcal{E} \vdash E[RN] \xrightarrow{\delta(\mathrm{true})} E' \\ \mathcal{E} \vdash B[RN] \xrightarrow{\alpha(RN')} B'\end{array}}{\mathcal{E} \vdash ((P\ [E] \rightarrow B) \Rightarrow N) \xrightarrow{\alpha(RN')} B'}$$

13.3 Multiple match

Syntax

$BM \mid BM$

Static semantics

$$\frac{\mathcal{C} \vdash (BM_1 \Rightarrow T) \Rightarrow \textbf{exit}(RT) \qquad \mathcal{C} \vdash (BM_2 \Rightarrow T) \Rightarrow \textbf{exit}(RT)}{\mathcal{C} \vdash ((BM_1 \mid BM_2) \Rightarrow T) \Rightarrow \textbf{exit}(RT)}$$

Dynamic semantics

$$\frac{\mathcal{E} \vdash (BM_1 \Rightarrow N) \xrightarrow{\alpha(RN)} B}{\mathcal{E} \vdash ((BM_1 \mid BM_2) \Rightarrow N) \xrightarrow{\alpha(RN)} B}$$

$$\frac{\mathcal{E} \vdash (BM_1 \Rightarrow N) \Rightarrow \textbf{fail} \quad \mathcal{E} \vdash (BM_2 \Rightarrow N) \xrightarrow{\alpha(RN)} B}{\mathcal{E} \vdash ((BM_1 \mid BM_2) \Rightarrow N) \xrightarrow{\alpha(RN)} B}$$

$$\frac{\mathcal{E} \vdash (BM_1 \Rightarrow N) \Rightarrow \textbf{fail} \quad \mathcal{E} \vdash (BM_2 \Rightarrow N) \Rightarrow \textbf{fail}}{\mathcal{E} \vdash ((BM_1 \mid BM_2) \Rightarrow N) \Rightarrow \textbf{fail}}$$

14 Expressions

14.1 Overview

Syntax

E	$\star ::=$	**block**	*time block*	(E_c1)
	$\star \mid$	$P := E$	*assignment*	(E_c2)
	$\star \mid$	$E \; ; \; E$	*sequential composition*	(E_c3)
	$\star \mid$	**trap** (**exception** X [(IP)] **is** E)* [**exit** [(P)] **is** E] **in** E	*trap*	(E_c4)
	$\star \mid$	**local var** LV [**init** E] **in** E	*variable declaration*	(E_c5)
	$\star \mid$	**rename** (X[(IP)] **is** X [E])* **in** E	*renaming*	(E_c6)
	$\star \mid$	**loop forever** [**var** LV] [**init** E] **in** E	*iteration*	(E_c7)
	$\star \mid$	**raise** X E	*raising exception*	(E_c8)
	$\star \mid$	**case** E[$: T$] **is** EM	*case*	(E_c9)
	$\star \mid$	**if** E **then** E [**else** E]	*if-then-else*	(E_c10)
	$\star \mid$	F [E] [$[\![X(,X)^*]\!]$]	*function instantiation*	(E_c11)
	$\star \mid$	F (RE, RP) [$[\![X(,X)^*]\!]$]	*in/out function instantiation*	(E_c12)
	$\star \mid$	**loop** [X] [(T)] [**var** LV] [**init** E] **in** E	*breakable iteration*	(E_c13)
	$\star \mid$	**break** [X] [(E)]	*breaking iteration*	(E_c14)
	$\star \mid$	N	*value*	(E_c15)
	$\star \mid$	**any** T	*nondeterministic termination*	(E_c16)

$\star\,\mid$	(RE)	*record expression*	(E_c17)
$\star\,\mid$	$C\,[E]$	*constructor application*	(E_c18)
$\star\,\mid$	E **andalso** E	*conjunction*	(E_c19)
$\star\,\mid$	E **orelse** E	*disjunction*	(E_c20)
$\star\,\mid$	$E = E$	*equality*	(E_c21)
$\star\,\mid$	E <> E	*inequality*	(E_c22)
$\star\,\mid$	$E\ .\ V$	*select field*	(E_c23)
$\star\,\mid$	$E\ :\ T$	*explicit typing*	(E_c24)

Syntax sugar Note that this entire syntactic category is syntax sugar.

We translate each expression of type T into a behaviour of type **exit**(T), maintaining the invariant that each expression is only capable of performing termination (δ) or exception (X) transitions, and not internal (**i**), gate (G) or delay (ε) transitions.

Most of the translations are straightforward, since they are the same as the behaviour parts. We only give the non-trivial translations here.

14.2 Value

Syntax

N

Syntax sugar

$N \stackrel{\mathsf{def}}{=} \mathbf{exit}(N)$

14.3 Nondeterministic termination

Syntax

any T

Syntax sugar

$\mathbf{any}\ T \stackrel{\mathsf{def}}{=} \mathbf{exit}(\mathbf{any}\ T)$

14.4 Record expression

Syntax

(RE)

Syntax sugar

$(RE) \stackrel{\mathsf{def}}{=} \mathbf{trap\ exit}\ ?\mathrm{x}\ \mathbf{is}\ (x)\ \mathbf{in}\ RE$

14.5 Constructor application

Syntax

$C\ [E]$

The default argument is `()`.

Syntax sugar

$C\ E \stackrel{\text{def}}{=} \textbf{case}\ E\ \textbf{is}\ \texttt{?}x \longrightarrow C\ x$

14.6 Conjunction

Syntax

$E\ \textbf{andalso}\ E$

Syntax sugar

$E_1\ \textbf{andalso}\ E_2 \stackrel{\text{def}}{=} \textbf{if}\ E_1\ \textbf{then}\ E_2\ \textbf{else}\ \text{false}$

14.7 Disjunction

Syntax

$E\ \textbf{orelse}\ E$

Syntax sugar

$E_1\ \textbf{orelse}\ E_2 \stackrel{\text{def}}{=} \textbf{if}\ E_1\ \textbf{then}\ \text{true}\ \textbf{else}\ E_2$

14.8 Equality

Syntax

$E\ \texttt{=}\ E$

Syntax sugar

$E_1\ \texttt{=}\ E_2 \stackrel{\text{def}}{=} \textbf{case}\texttt{(}E_1, E_2\texttt{)}\ \textbf{is}\texttt{(?}x, \texttt{?}y\texttt{)} \longrightarrow \textbf{case}\ x\ \textbf{is}\ \texttt{!}y \longrightarrow \text{true}\ \texttt{|}\ \textbf{any} \longrightarrow \text{false}$

14.9 Inequality

Syntax

$E\ \texttt{<>}\ E$

Syntax sugar

$E_1\ \texttt{<>}\ E_2 \stackrel{\text{def}}{=} \textbf{if}\ E_1\ \texttt{=}\ E_2\ \textbf{then}\ \text{false}\ \textbf{else}\ \text{true}$

14.10 Field select

Syntax

$E\ .\ V$

Syntax sugar

$E\ .\ V \stackrel{\text{def}}{=} \textbf{case}\ E\ \textbf{is}\ (V \Rightarrow ?x, \textbf{etc}) \rightarrow x$

14.11 Explicit typing

Syntax

$E\ :\ T$

Syntax sugar

$E : T \stackrel{\text{def}}{=} \textbf{case}\ E : T\ \textbf{is}\ ?x \rightarrow x$

15 Record expressions

15.1 Syntax

RE	$\star ::=$	$V \Rightarrow E$	*singleton* (RE_c1)
	$\star \mid$		*trivial* (RE_c2)
	$\star \mid$	RE , RE	*disjoint union* (RE_c3)
	$\star \mid$	$E(, E)^*$	*tuple* (RE_c4)

15.2 Syntax sugar

Note that this entire syntactic category is syntax sugar.

Each record expression of type RT is translated into a behaviour of type $\textbf{exit}(RT)$.

15.3 Singleton record

Syntax

$V \Rightarrow E$

Syntax sugar

$V \Rightarrow E \stackrel{\text{def}}{=} ?V\ := E$

15.4 Empty record

Syntax

$()$

Syntax sugar

$() \stackrel{\text{def}}{=} \textbf{exit}$

15.5 Record disjoint union

Syntax

$RE\,,RE$

Syntax sugar

$RE_1\,,RE_2 \stackrel{\text{def}}{=} RE_1\,|||\,RE_2$

15.6 Record tuple

Syntax

$E(\,,E)^*$

Syntax sugar

$E_1\,,\,\ldots\,,E_n \stackrel{\text{def}}{=} \$1 \Rightarrow E_1\,,\,\ldots\,,\$n \Rightarrow E_n$

16 Expression pattern-matching

16.1 Overview

Syntax

EM	$\star ::=$	$P\ [E] \rightarrow E$	*single match*(M_c1)
	$\star\,|$	$EM\,|\,EM$	*multiple match*(M_c2)

Syntax sugar Note that this entire syntactic category is syntax sugar.

Expression pattern-matches trivially translate into behaviour pattern-matches.

17 In parameters

17.1 Overview

Syntax

IP	$\star ::=$	$V \Rightarrow [P:]T$	*singleton* (IP_c1)
	$\star\,|$	**etc**	*wildcard* (IP_c2)
	$\star\,|$	P **as** IP	*record match* (IP_c3)
	$\star\,|$		*trivial* (IP_c4)
	$\star\,|$	$IP\,,IP$	*disjoint union* (IP_c5)
	$\star\,|$	$[P:]T(\,,[P:]T)^*$	*tuple* (IP_c6)

with the restriction that **etc** can occur at most once.

Syntax sugar Note that this entire syntactic category is syntax sugar.

Each parameter list is translated to a typed record pattern of the form \$argv **as** RP : RT

17.2 Singleton parameter list

Syntax

$$V \Rightarrow [P:]T$$

The default pattern is **any**.

Syntax sugar

$$(V \Rightarrow P : T) \stackrel{\text{def}}{=} \$\text{argv } \textbf{as } (V \Rightarrow P) : (V \Rightarrow T)$$

17.3 Wildcard

Syntax

etc

Syntax sugar

$$\textbf{etc} \stackrel{\text{def}}{=} \$\text{argv } \textbf{as etc} : \textbf{etc}$$

17.4 Record match

Syntax

$$P \textbf{ as } IP$$

Syntax sugar

$$P \textbf{ as } \$\text{argv } \textbf{as } RP : RT \stackrel{\text{def}}{=} \$\text{argv } \textbf{as } P \textbf{ as } RP : RT$$

17.5 Trivial parameter list

Syntax

$$()$$

Syntax sugar

$$() \stackrel{\text{def}}{=} \$\text{argv } \textbf{as } () : ()$$

17.6 Parameter list disjoint union

Syntax

$$IP, IP$$

Syntax sugar

$$((\$\text{argv}\ \textbf{as}\ RP_1\ :\ RT_1), (\$\text{argv}\ \textbf{as}\ RP_1\ :\ RT_1)) \stackrel{\text{def}}{=} (\$\text{argv}\ \textbf{as}\ RP_1, RP_2\ :\ RT_1, RT_2)$$

17.7 Tuple parameter list

Syntax

$$[P:]T(,[P:]T)^*$$

The default pattern is **any**.

Syntax sugar

$$(P_1:T_1, \ldots, P_n:T_n) \stackrel{\text{def}}{=} (\$1 \Rightarrow P_1:T_1, \ldots, \$n \Rightarrow P_n:T_n)$$

18 Local variables

18.1 Overview

Syntax

LV	$\star ::=$	$V \Rightarrow V:T$	*singleton*(LV$_c$1)
	$\star \mid$		*trivial*(LV$_c$2)
	$\star \mid$	LV, LV	*disjoint union*(LV$_c$3)
	$\star \mid$	$V:T(,V:T)^*$	*tuple*(LV$_c$4)

Syntax sugar Note that this entire syntactic category is syntax sugar.

Each local variable list is translated into a typed variable list of the form $RV:RT$.

18.2 Singleton variable list

Syntax

$$V \Rightarrow V:T$$

Syntax sugar

$$(V \Rightarrow V':T) \stackrel{\text{def}}{=} (V \Rightarrow V')\ :\ (V \Rightarrow T)$$

18.3 Trivial variable list

Syntax

$$()$$

Syntax sugar

$$() \stackrel{\text{def}}{=} ()\ :\ ()$$

18.4 Variable list disjoint union

Syntax

$LV \, , LV$

Syntax sugar

$$((RV_1 : RT_1), (RV_2 : RT_2)) \stackrel{\mathsf{def}}{=} (RV_1, RV_2 \, : \, RT_1, RT_2)$$

18.5 Tuple variable list

Syntax

$V : T \, (\, , V : T)^*$

Syntax sugar

$$(V_1 : T_1 \, , \, \ldots \, , V_n : T_n) \stackrel{\mathsf{def}}{=} (\$1 \Rightarrow V_1 : T_1 \, , \, \ldots \, , \$n \Rightarrow V_n : T_n)$$

19 Further work

There are a number of features still missing from the language, some of which might be added into the core language:

- We may wish to add a 'parallel composition over values' operator in the same style as the current 'choice over values' operator.
- There have been requests for the ability to form *n*-out-of-*m* communication channels as well as the current *n*-out-of-*n* channels.
- An additional suspend/resume operator has been requested.
- The ability to rename a gate or exception to more than one other gate would be useful.
- Support for write-many variables would be useful.
- The ability to call functions declared with named parameters, using positional arguments. (At the moment functions are either declared with positional or named arguments, and the two styles cannot be mixed).

We need to check a number of semantic properties for the language, for example: principal typing, type safety, stratification, and bisimulation as a congruence.

The relationship between the core language and the user-level language needs to be clarified.

References

[1] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[2] J. Davies, D. Jackson, and S. Schneider. Broadcast communication for real-time processes. In J. Vytopil, editor, *Proc. Formal Techniques in Real-Time and Fault-Tolerant systems*, pages 149–170. Springer-Verlag, 1992. LNCS 571.

[3] H. Ehrig and B. Mahr. Fundamentals of algebraic specification. *Bull. Euro. Assoc. Theoret. Comp. Sci.*, 6, 1985.

[4] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[5] H. Garavel and M. Sighireanu. *French-Romanian Integrated Proposal for the User Language of E-LOTOS* Input document (KC3) to the ISO/IEC JTC1/SC21/WG7/E-LOTOS meeting in Kansas City, May 1996.

[6] ISO. *LOTOS, a Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1989. IS 8807.

[7] A. Jeffrey. A core data and behaviour language for E-LOTOS. Input document (KC1) to the ISO/IEC JTC1/SC21/WG7/E-LOTOS meeting in Kansas City, May 1996.

[8] A. 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 N1053, chapter Annexe A. 1995.

[9] L. Léonard and G. Leduc. An introduction to ET-LOTOS for the description of time-sensitive systems. *Computer Networks and ISDN Systems*, 1996. To appear.

[10] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[11] J. Quemada, editor. *Working draft on Enhancements to LOTOS*. ISO/IEC JTC1/SC21/WG7 N1053. 1994.