E-LOTOS:
Tutorial and Semantics

Alberto Verdejo

Departamento de Sistemas Informáticos y Programación
Escuela Superior de Informática
Universidad Complutense de Madrid
29th May 2004

Supervised by: Luis Fernando Llana-Daz
Narciso Mart-Oliet

Contents

Part I

Tutorial

This tutorial introduces E-LOTOS by describing all the features of the language, and by showing its expressive power. We show how E-LOTOS can be used to specify systems, their behaviour, and the values they manipulate, as well as how the specifier can modularize systems. The tutorial includes some *small* examples that show how E-LOTOS features are used to specify common problems, usual generic data types, well-known concurrent programming problems, as well as to describe hardware components.

E-LOTOS is still a draft, under revision. So this is still an unfinished document, it has to be completed when the ISO standard will be definitively approved.

1 Introduction

> If cars had improved at the rate of computers in the same time period, a Rolls Royce would now cost 10 dollars and get a billion miles per gallon. (Unfortunately, it would probably also have a 200-page manual telling how to open the door.)
>
> – Andrew S. Tanenbaum

LOTOS (Language Of Temporal Ordering Specification) [ISO88] is a Formal Description Technique[1] developed within ISO for the formal specification of open distributed systems. It is based on the intuitive and well known *black box* analogy [Mil89] where systems are described as black boxes with buttons, that represent their entire capability of communication. The basic idea is that systems can be specified by defining the temporal relation among the interactions that constitute the *externally* observable behaviour of a system. As stated in [Mil89] "the behaviour of a system is exactly what is observable, and to observe a system is exactly to communicate with it."

LOTOS is composed of a process algebra part (based on CCS [Mil89] and CSP [Hoa85]) to describe systems, and an algebraic language (ACT ONE [EM85]) to describe the abstract data types. LOTOS has proven very successful in specifying protocols and services (examples can be found in [vEVD89, Gam90, Mun91, Pec92, BL93, GH93]). However, LOTOS has several limitations related to its expressive power and structuring capabilities, besides user-friendliness. For these reasons, LOTOS is currently under revision in ISO [Que98], in the Work Item "Enhancements to LOTOS," giving rise to a revised language called E-LOTOS (Enhanced LOTOS). E-LOTOS has the same structure as LOTOS. It is formed of a behavioural process algebra part, which inherits some operators from LOTOS, generalizes others, and adds new operators; and of a functional data definition part which allows constructive definitions of data types and functions for manipulating them, and is thought to be more user friendly.

Among the enhancements introduced in E-LOTOS, the most important ones are:

- the notion of quantitative time: in E-LOTOS we can define the exact time at which events (by adding annotations to actions, see Section 2.1) or behaviours (by using **wait** statements, see Section 2.16) may occur,
- the data part, both the definition of new data types and the construction of values of predefined types, and
- modularity, which allows the definition of types, functions, and processes in separate modules, by controlling their visibility by means of module interfaces, and the definition of generic modules, useful for code reuse. One module can use another one by means of importation clauses.

Although in this tutorial we describe all the operators of E-LOTOS, the inherited operators and the new ones, now we concentrate on the latter. Among the new operators, we have:

[1] "Formal Description Techniques (FDTs) are methods of defining the behaviour of an (information processing) system in a language with formal syntax and semantics, instead of a natural language as English." [ISO88]

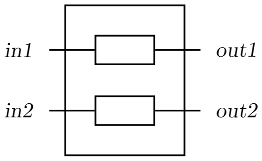
- the sequential composition operator that allows to concatenate two processes. This operator unifies both the action prefix operator and the enabling operator of LOTOS.
- the general parallel operator that allows the synchronization of n among m processes.
- the suspend/resume operator, which generalizes the disabling operator of LOTOS by allowing a disabled process to be resumed by the disabling process.
- operators to raise exceptions and to handle them.
- the renaming operator which allows to rename actions and exceptions, and to modify their parameters.

E-LOTOS has several imperative features, which have been introduced in order to make the job of specifying systems easier to the user of this programming paradigm. Assignment, declaration of local variables, and several iterative operators, are some of these features. We describe them in Section 2.19.

In the following two chapters we describe the Base Language, that is, the part of E-LOTOS used to write the behaviour of processes, Chapter 2, and the data types, Chapter 3. In Chapter 4 we describe the Module Language, that is, the language used to modularize the specifications. Chapter 5 includes several examples that show how E-LOTOS can be applied to specify different kinds of systems.

1.1 An example: two positions register

In order to show the basic ideas of E-LOTOS specification, let us begin with as simple example. Coming back to the idea of the black boxes, let us imagine that we want to specify a two positions register, graphically described by



At this level, we only know that the register has four gates (buttons of the black box) *in1*, *in2*, *out1*, and *out2*. A specification of the behaviour of this register would be a description of when the buttons (or gates) are available to be pressed. Generally, some gates are *input gates* (*in1* and *in2*) and values can go through them, and other gates are *output gates* (*out1* and *out2*). If the register has the following restrictions:

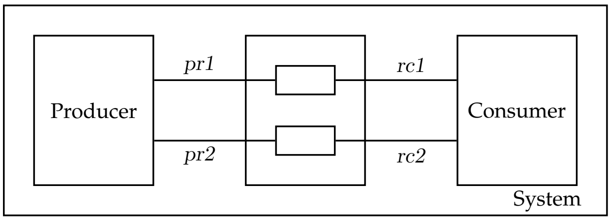
- the register is initially empty,
- both data have to get in before any one of them gets out,
- data through gate *in1* have to get in before data through gate *in2*, and
- data have to get out in the same order as they get in,

we would describe it as

```
process Register [in1:data,in2:data,out1:data,out2:data] is
  var x1:data,x2:data in
    in1(?x1); in2(?x2); out1(!x1); out2(!x2)
  endvar
endproc
```

This example will be used during the tutorial and, then, all the details will be given (see Section 2.3). Note how the order of the communications through gates are expressed (using the sequential composition operator ;) and how the values communicated are represented with variables.

If we imagine other two black boxes representing a producer, that produces two values and saves them in the register, and a consumer that takes the values from the register, we can put them together, communicating:



The complete system is described in E-LOTOS as follows:

```
process System is
    hide pr1:data,pr2:data,rc1:data,rc2:data in
        conc
            Producer [pr1,pr2]()
        |[pr1,pr2]|
            Register [pr1,pr2,rc1,rc2]()
        |[rc1,rc2]|
            Consumer [rc1,rc2]()
        endconc
    endhide
endproc
```

where Producer and Consumer are processes specifying the behaviour of the producer and the consumer (see Section 2.7). They are put together with the Register, running in parallel, and communicating through gates. The whole system has no gate offered to the environment, because it represents a complete, closed system. All its behaviour is described and occurs internally.

1.2 Variables

Regarding the construction of values and the use of variables, E-LOTOS was thought as a functional language, in the sense that there is no idea of state and variables are given a value only once. When a variable is given a value, this value is substituted for the variable in the successive behaviour. However, this idea changed during the design of the language and, although values are still built with constructors as in functional language, E-LOTOS has *write-many* variables, that is, variables that can be assigned several times. Variables are declared with the **var** operator (Section 2.19). When a variable is declared, it is given a type and there is the possibility to give it an initial value. We have seen a variable declaration in the Register process:

```
var x1:data,x2 : data in
  in1(?x1); in2(?x2); out1(!x1); out2(!x2)
endvar
```

A variable can have a value of its declared type or a value of a subtype of its declared type, and this value can be changed during the execution of a specification.

The introduction of this kind of variables has affected the use of several operators. Thus, we will describe how variables can be used when several processes are composed by using any of these operators.

1.3 Time in E-LOTOS

Time is a very important aspect in concurrent, distributed systems. Although it is not present in classical process algebras, there have been many proposals to introduce time in process algebras [RR86, NS91, Yi91, BB93, Sch95]. The introduction of time in LOTOS has been dealt in [LL94, dFLL$^+$95].

The introduction of time is one of the most important enhancements of E-LOTOS. The specific features that E-LOTOS provides to manage time are three. The first one is that the specification writer can describe *when* the actions a process performs may occur, and this can be described with great flexibility, as we show in Section 2.1.

The second feature that E-LOTOS provides is the **wait** instruction, that, given a duration time d, represents that the process idles the time d.

The type time is the third feature introduced in E-LOTOS with respect to control of time. In E-LOTOS ([Que98]), no data type time is defined, but only the properties that it has to fulfill are enumerated. So, each *implementation* of the language can define a different time, provided that these requirements are fulfilled. The properties are:

- the time domain is a commutative, cancellative monoid with addition + and unit 0. Thus, it satisfies the properties:
  - $d_1 + d_2 = d_2 + d_1$
  - if $d_1 + d = d_2 + d$ then $d_1 = d_2$
  - $d_1 + (d_2 + d_3) = (d_1 + d_2) + d_3$
  - $d + 0 = 0 + d = d$

  where d_1, d_2, and d are variables over the time domain.
- the order given by $d_1 \leq d_2$ if and only if $\exists d \,.\, d_1 + d = d_2$ is a total order.

For a description of the different time domains used in the literature see, for example, [NS91]. At first glance, it seems that it is better to have a continuous time domain. But, as shown in [LDV99], it has several problems, for example, we can define processes that freezes time. It is enough to have a discrete time domain and define the *unit of time* as small as we want. This is not a restriction decision since all computers are ruled by an internal discrete clock. In this tutorial we assume that the type time has been declared as a synonym of the type nat of natural numbers.

Although time is foreign to a process in the sense that the process cannot control time and has to *coexist* with it, the introduction of time has affected the meaning of the different operators of E-LOTOS. We will describe in the following sections how these operators behave in relation to time. When we say that a process *lets time pass*, we mean that the process can be idle (doing nothing but waiting) a concrete period of time. Thus, when we say that a process does not let time pass, we mean that the process *must* do something immediately. This is related to the notion of *urgency*. When an action is *urgent* it must be performed as soon as possible. This means that when an urgent action is enabled, it must be performed unless other action was performed without consuming time. Urgency is used to guarantee the progress of a system: if there were no urgent actions the systems could idle forever. For example, the special internal action **i** (Section 2.4) is urgent. We can specify which actions are urgent at certain level in order to ensure the evolution of the system. This is very useful when different components of a system are communicating, as we will see in Section 2.14, where the hiding operator is described. Not only actions may be urgent, there are other constructions in E-LOTOS, for example assignments, that are also urgent. Due to urgency, one could write specifications that *blocks* time. This is an undesirable and counterintuitive feature. The idea of time blocking is that a behaviour is performing internal actions urgently, thus time is not allowed to pass (see Section 2.6).

In E-LOTOS time is *deterministic*. As stated in [NS91], it is usually admitted that when a process P is idle (does not perform any action) for some duration d, then the resulting behaviour is completely determined from P and d. The avoidance of time nondeterminism has also affected several operators. We will describe how these operators have been affected.

We write the symbol ⏱ in the margin when we speak about time in relation with the operator which is being described.

2 Base Language for processes

In this chapter we are going to show the constructions of E-LOTOS used to describe the behaviour of systems. In Chapter 3 we will explain those constructions used to describe the values and types of data that these systems can manage.

We try to give this description from the simple operators to the more complex ones; however, in order to show examples, sometimes we have to include operators that are explained later.

All the constructions are illustrated with small examples that show a concrete use of each one. In Chapter 5 we will introduce more complex examples.

2.1 Actions

In E-LOTOS a concurrent system is specified as a process that is composed of other processes interacting with each other. This interaction between processes is carried out by means of *actions*, which represent synchronization or communication between processes through *gates*.

The E-LOTOS syntax to indicate that a process is carrying out an action is as follows:

$$G\ [\texttt{(}P_1\texttt{)}]\ [\texttt{@}P_2]\ [\ \texttt{[}\ E\ \texttt{]}\ ]$$

where the components between [ ] are optional, G is the name of a gate, P_1 and P_2 are patterns (see Section 3.6) and E is a boolean expression.

The simplest action is a synchronization in which only the name of the gate, G, is indicated.

Value passing

If the action stands for a communication in which there is an information exchange among the different processes that are communicating, that information is indicated by the first pattern, P_1. Although the different patterns of E-LOTOS are shown later on in Section 3.6, now we will introduce some of them by means of examples. For example, if a process has a gate called *outP* and it wants to communicate on it the value 3, in E-LOTOS we should indicate it by using the behaviour[1]

$$outP\texttt{(!3)}$$

Although perhaps the first interesting example of this tutorial should be

$$outP\texttt{(! "Hello, world!")}^2$$

[1] We use the word “process” to refer to both an abstract entity which can communicate with other processes and the declaration (implementation) of this entity in E-LOTOS, and the word “behaviour” to refer to terms constructed by combining the different operators of the language in order to describe the behaviour of processes.

[2] `"Hello, world!"` is a constant of the predefined type `string`, Section 3.1.

If we want to communicate several values at a time we have to build what is called in E-LOTOS terminology a *record* of values, a list of transmitted values enclosed between parentheses. For example, if the process wants to communicate both values 3 and `"Hello, world!"`, we write

$$\mathit{outP}\texttt{(!(3,"Hello, world!"))}$$

When we write the record of values that a gate communicates, we can give a name to each value (that is, to each field of the record)[3]. Thus we can write

$$\mathit{outP}\texttt{(!(}\mathit{value}\texttt{ => }3\texttt{,}\mathit{greeting}\texttt{ => "Hello, world!"))}$$

If the process has another gate, called *inP*, on which data is received and is saved in the variable x, then we write

$$\mathit{inP}\texttt{(?}x\texttt{)}$$

If the process receives a record of values through gate *inP*, we can save the whole record value in the variable x as above, or we can use different variables to save the different fields of the record,

$$\mathit{inP}\texttt{((}\mathit{value}\texttt{ => ?}v\texttt{,}\mathit{greeting}\texttt{ => ?}g\texttt{))}$$

In order to prevent errors, in E-LOTOS there is the possibility (but not the obligation) of typing the different components which appear in a process specification. In particular, we can type the gates of a process, so a gate can only communicate values of that type.

In the previous example, if the process receives only integers on the gate *inP*, we can write

$$\mathit{inP}\texttt{(?}x\texttt{:}\mathsf{int}\texttt{)}$$

A process that takes a record with an integer and a string looks like

$$\mathit{inP}\texttt{ ((?}x\texttt{:}\mathsf{int}\texttt{, ?}y\texttt{:}\mathsf{string}\texttt{))}$$

We can also specify a *selection predicate* in an action, which specifies the conditions that the transferred values have to fulfill. For example, if a process only should receive integers smaller than 10, we would write

$$\mathit{inP}\texttt{(?}x\texttt{:}\mathsf{int}\texttt{) [}x\texttt{<}10\texttt{]}$$

⏱ Timed constraints

It is assumed that the execution of an action takes no time, that is, actions are *atomic* and *durationless*. However, communications can be made sensitive to time by adding the `@` P_2 annotation, which pattern-matches the pattern P_2 to the time when the action happens, measured from the time when the communication was enabled. We can use this, joined with the use of selection predicates, to control the time when actions may be performed.

The behaviour

$$\mathit{inP}\texttt{(?}x\texttt{:}\mathsf{int}\texttt{) @?}t\texttt{ [}t\texttt{<}5\texttt{]}$$

specifies an action that receives an integer that is bound to the variable x provided that less than 5 units of time have passed, whereas the action

$$\mathit{inP}\texttt{(?}x\texttt{:}\mathsf{int}\texttt{) @!}5$$

[3] In this case, the gate has to be declared as having the corresponding record type, see Section 3.2.

can only occur 5 units of time after the action *inP* had been enabled. Let us explain how the different patterns of E-LOTOS can be used to represent different timed constraints. If we use the pattern ?t, then variable t will be bound to the time at which the action happens, and the selection predicate is which imposes the restriction (t<5); but if we use the pattern !5, then the value 5 is compared with the time at which the action happens, and the action will be possible only if this time is also 5.

Anyone of the three parameters that come with a gate in an action is optional. When any of them is not present, a default value is used. So, the pattern P_1 is (), an empty record, by default; the pattern P_2 is **any**:time, that means any value of type time; and the expression E is *true*, i.e., no conditions are required.

There is a special action, the internal action **i**, which will be studied in Section 2.4. Intuitively, it represents an action made by the process without the knowledge of its observer.

2.2 Sequential composition

We can compose two behaviours in sequence with the sequential composition operator ";". B_1;B_2 behaves first as B_1. When B_1 has finished then it continues as B_2. A sequential composition B_1;B_2 finishes when B_2 does.

With this new operator we can specify a behaviour with two gates, *inP* and *outP*, which receives an integer on gate *inP*, saves it in the integer variable x, and then sends this integer through gate *outP*:

```
inP(?x); outP(!x)
```

The sequential composition operator has greater precedence than any other binary operator. Thus, when we write B_1;B_2[]B_3 (where [] is the selection operator, see next section) we mean (B_1;B_2)[]B_3. Anyway, we can use parentheses in order to clarify behaviours or to force precedence. Binary operators are right associative, so B_1;B_2;B_3 means B_1;(B_2;B_3).

Although the complete syntax of process declarations will be fully covered in Section 2.20, we are going to use it in examples to make them easier to understand. For the time being, it is enough to say that in a process declaration the *formal* gates which the process can communicate through are declared (like formal parameters are in a procedure declaration in Pascal), and also its behaviour (as a procedure body). In the process instantiation the *actual* gates are given.

Let us declare a process whose behaviour is as in our last example:

```
process Buffer1 [inP:int,outP:int] is
  var x:int in
    inP(?x); outP(!x)
  endvar
endproc
```

Note that in gate declarations, each gate is typed with the type of values the gate can communicate.

Processes may be recursive. In this way, a process that is continuously receiving integers through a gate and sending them through another, may be specified as

```
process Buffer2 [inP:int,outP:int] is
  var x:int in
    inP(?x:int); outP(!x); Buffer2[inP,outP]()
  endvar
endproc
```

In a process instantiation, the lists of actual gates and actual parameters (a process can also have a list of parameters) have to be given, although any of them may be empty. Thus, we have to write here Buffer2[*inP*,*outP*]() instead of only Buffer2[*inP*,*outP*].

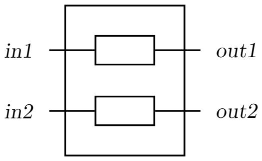
The behaviour B_1;B_2 lets time pass if B_1 does, or if B_1 has finished and B_2 lets time pass.

For LOTOS users, we have to note that LOTOS has two sequential composition operators: the action prefix operator (";") and the enabling operator (">>"). E-LOTOS has unified both of them in the sequential composition operator ";" we have just explained, and there is a particular case, where the left behaviour consists of an action, for example *outP*(!3);Buffer2[*inP*,*outP*]().

2.3 Selection operator

The selection operator "[]" denotes a choice between two possible behaviours. In this way, the behaviour B_1[]B_2 (where B_1 and B_2 are also behaviours) may perform actions either from B_1 or from B_2. The choice is solved (in principle) when B_1[]B_2 interacts with its environment, which is defined by another behaviour. If the environment offers a first action of B_1, then B_1 is chosen and B_2 is forgotten; otherwise, if the environment offers a first action of B_2 then B_2 is chosen and B_1 is forgotten. We see below what happens if the action offered by the environment is a first action of both B_1 and B_2.

For example, let us suppose we want to specify the two positions register that we introduce in Section 1.1, graphically described by



which can communicate with its environment through the gates *in1*, *in2*, *out1*, and *out2*. The register is initially empty and both data have to get in before any one of them gets out; besides, data through gate *in1* have to get in before data through gate *in2*; and data have to get out in the same order as they get in. Then the following process describes this behaviour

```
process Register1 [in1:data,in2:data,out1:data,out2:data] is
  var x1:data,x2:data in
    in1(?x1); in2(?x2); out1(!x1); out2(!x2)
  endvar
endproc
```

where data is the type of the values the register can save.

But if the first element can go out without waiting for the second element, then we can specify the register behaviour with the selection operator as

```
process Register2 [in1:data,in2:data,out1:data,out2:data] is
  var x1:data,x2:data in
    in1(?x1);
    ( in2(?x2); out1(!x1)
     []
      out1(!x1); in2(?x2)
    );
    out2(!x2)
  endvar
endproc
```

E-LOTOS grammar forces the user to put parentheses where a behaviour is ambiguous[4], so the user cannot mix binary operators without parentheses. Too many parentheses may lead to unreadable specifications, so E-LOTOS also has a *bracketed* syntax for these operators, suggested by Ed Brinksma in his thesis [Bri88]. The bracketed syntax for the selection operator is as follows:

$$\textbf{sel}\ B_1\,\texttt{[]}\ \ldots\ \texttt{[]}\,B_n\ \textbf{endsel}$$

where if some B_i use a binary operator (but `;`), it has to use it with parentheses.

Nondeterministic choice

If we have the behaviour $B_1\,\texttt{[]}\,B_2$ and the environment offers a first observable action of both, then the process is chosen nondeterministically. To show this, let us see an example with vending machines[5]. Suppose we have a machine (Machine 1) whose behaviour is specified as

$$\text{Machine 1} \quad \equiv \quad \texttt{(}\ \textit{insert10}\texttt{;}\ \textit{take\_coffee}\ \texttt{)}\ \texttt{[]}\ \texttt{(}\ \textit{insert5}\texttt{;}\ \textit{take\_milk}\ \texttt{)}$$

This machine offers to the client (its environment) the choice between inserting a 10 units coin or a 5 units coin. If the client inserts a 10 units coin then the machine evolves to a state where only coffee is offered, but if the client inserts a 5 units coin, then the machine only offers milk. So, the client can (indirectly) choose if he is going to take coffee or milk by inserting the right coin.

Let us suppose now that we have another machine (Machine 2) that is like the last one but serves a more expensive milk, which costs 10 units. If we modify (the behaviour of) Machine 1 getting

$$\text{Machine 2} \quad \equiv \quad \texttt{(}\ \textit{insert10}\texttt{;}\ \textit{take\_coffee}\ \texttt{)}\ \texttt{[]}\ \texttt{(}\ \textit{insert10}\texttt{;}\ \textit{take\_milk}\ \texttt{)}$$

then we will not obtain the desired behaviour. Now the client has to insert a 10 units coin independently of what he wants. Once the client has inserted the coin, Machine 2 evolves to one of two possible states: one in which it only offers coffee, or one in which it only offers milk, and the client can only take what the machine offers. To which state the machine evolves depends on a nondeterministic choice. It is said that the machine has evolved *internally* (without knowledge of the environment) to one of these states. Therefore, Machine 2 is nondeterministic.

So, if we want to specify a machine that behaves as Machine 1, that is, that offers coffee and milk after the client has inserted a coin, but with more expensive milk, we have to write instead:

$$\text{Machine 3} \quad \equiv \quad \textit{insert10}\texttt{;}\ \texttt{(}\ \textit{take\_cofee}\ \texttt{[]}\ \textit{take\_milk}\ \texttt{)}$$

Once the client has inserted a coin, Machine 3 offers both coffee and milk, so the client can choose what he wants.

Another way of introducing nondeterminism is with internal actions, **i**, which we will present in the next section.

Variables and selection operator

Regarding variables, we have to say that in a selection $B_1\,\texttt{[]}\,B_2$, variables which are modified only by B_1 or only by B_2 must be initialized, that is, they have to be given a value, before the selection, and the modification must preserve the type of the variables. In this way, it can be assured that, independently of the behaviour executed, the set of initialized variables after the selection is always the same, and we

[4] The sequential composition operator is an exception to this rule, and this is the reason why we have said above that this operator has the greatest precedence.

[5] This example has been extracted from [LFHH].

know their type. Variables which are modified by both behaviours may have been initialized previously or not, but both behaviours have to assign them a value of the same type.

Thus, the behaviours

```
inP(?x:int);
(  inP(?x:bool)                         inP(?x:int)
 [] outP(!x)            and           [] inP(?x:bool)
)
```

are not allowed, because:

- in the first one, although variable x is initialized before the selection operator by means of the behaviour *inP*`(?`x`:int)`, only one of the subbehaviours of the selection operator, *inP*`(?`x`:bool)`, modifies it, and it changes the type of x;
- in the second one, both subbehaviours, *inP*`(?`x`:int)` and *inP*`(?`x`:bool)`, modify variable x, but they do not give it the same type.

Note that although a variable has been declared, it may have been declared of the universal type **any** (see Section 3.2), so it can have values of any type.

Time determinism

Let us consider the following behaviour[6]

```
(?x:=5 [] ?x:=2); wait(1);P[...]
```

It is time nondeterministic: intuitively, when one unit of time has passed and process P is executed, variable x may have either the value 5 or the value 2.

Time nondeterminism is an undesirable feature, and E-LOTOS forbid it requiring that when we build a behaviour B_1 `[]` B_2, both behaviours B_1 and B_2 have to be “guarded,” that is, they have to perform an action on a gate (or raise an exception as we will see in Section 2.15) before they finish; this action decides which branch of the selection operator is selected.

The behaviour B_1 `[]` B_2 lets time pass if both B_1 and B_2 do so.

2.4 Internal action

The internal action, **i**, is an action that a process can carry out to evolve in an autonomous manner, without being observed by the environment. Together with the selection operator it is used to model nondeterminism. It is also used to represent hidden actions, as we will see in Section 2.14.

Let us suppose now that our two positions register may loose information, that is, it is possible that an element gets in but never gets out. So, once an element has got in there are two possible cases: the element is offered or the element is lost, missing the chance of taking it out. We may describe this behaviour as follows

[6] We will describe the **wait** instruction in Section 2.16, and the assignment in Section 2.19

```
process Register3 [in1:data,in2:data,out1:data,out2:data] is
  var x1:data,x2:data in
    in1(?x1:data);
    sel
      in2(?x2:data); out1(!x1)
    []
      out1(!x1); in2(?x2:data)
    []
      i; in2(?x2:data)
    endsel;
    sel out2(!x2) [] i endsel
  endvar
endproc
```

although we will see below (at the end of this section) that it is not right enough (due to urgency of internal action).

Once the first element has got in, there are three cases:

1. the second element gets in and then the first gets out,
2. the first element gets out and then the second gets in, or
3. the first element is lost and then the second gets in.

It seems that the case where the second element gets in and then the first element is lost lacks. But since losing an element is not observable by the environment, it is the same that the first element is lost before or after the second gets in.

Nondeterministic behaviours

As commented above, internal actions can be used to specify nondeterminism because they can be executed in an autonomous way. The behaviour

$$\textit{coffee}\ \texttt{[]}\ \textit{milk}$$

offers the possibility of taking either coffee or milk. However, the behaviour

$$(\ \mathbf{i};\textit{coffee}\ )\ \texttt{[]}\ \textit{milk}$$

always offers coffee but only *may* offer milk, that is, if the environment asks for synchronization on gate *milk*, the synchronization could be either accepted or rejected, in a nondeterministic way. Instead, if the client asks for synchronization on gate *coffee*, he always will succeed.

On the other hand, the behaviour

$$(\ \mathbf{i};\textit{coffee}\ )\ \texttt{[]}\ (\ \mathbf{i};\textit{milk}\ )$$

may not be able to synchronize on any of its gates, depending on an internal choice.

Urgency

The internal action **i** is *urgent*, that is, it cannot idle. This means that when an internal action can be performed, it *must* be performed unless another action is performed without delay. So the process Register3 we have written above has not the meaning we want. Due to the urgency of the internal action, if actions on gate *in2* or *out1* are not possible immediately by the environment, then the internal action must be carried out. So, now, the behaviour specifies that if the second element does not get in immediately or the first element does not get out immediately, then the first one is lost, which is not the desired behaviour. In the behaviour

```
( i;coffee ) [] milk
```

if the environment wants to synchronize after some period of time (greater than 0) on gate *milk*, it will never have success.

We can specify the desired behaviour by introducing a nondeterministic internal election between losing the element or not losing the element. The right **Register4** process is:

```
process Register4 [in1:data,in2:data,out1:data,out2:data] is
  var x1:data,x2:data in
    in1(?x1:data);
    ( i;(  in2(?x2:data); out1(!x1)
         [] out1(!x1); in2(?x2:data)
        )
      []
        i; in2(?x2:data)
    );
    (  i; out2(!x2)
      [] i
    )
  endvar
endproc
```

2.5 Successful termination

Successful termination in E-LOTOS is represented by the behaviour **null**. This behaviour terminates immediately (it is urgent), without doing anything.

2.6 Inaction and time block

There are two behaviours in E-LOTOS for representing anomalous processes: **stop** and **block**. Both behaviours do not do any communication, and they do not terminate either. The difference is that **block** *blocks* time, whereas **stop** can delay any time. Both represent a *pathological* behaviour, a deadlocked process, that is, an undesirable specification, a *broken* system. Therefore, they should not be used in a correct specification in parallel with other behaviours.

For example, the behaviour

```
stop || wait(1);outP(!3)
```

can perform a communication on gate *outP* after one unit of time, but it cannot finish (as we will see in the next section). However, the behaviour

```
block || wait(1);outP(!3)
```

cannot do anything, because it cannot idle one unit of time, since **block** *freezes* time.

The behaviour **block** can be defined with help of the urgency of the internal action:

```
process Block is
    i;Block[ ]()
endproc
```

2.7 Parallel composition operator

We are going to study in this section, and the following ones, five different ways of composing processes in parallel.

The first possibility is using the parallel operator, whose syntax is

$$B_1 \ |[\ G_1,G_2,\ldots,G_n \ ]| \ B_2$$

where B_i stands for behaviours and G_j for gates.

The behaviour B_1 |[$G_1,\ldots,G_n$]| B_2 offers observable actions of both B_1 and B_2 as long as they are not actions carried out on a gate in the given list $G_1, G_2, \ldots, G_n$. In order to carry out an action in the given list, both behaviours have to perform it simultaneously. If one behaviour can communicate on a gate in the list, but the other cannot, then the first one has to wait for the other, if it can. When both behaviours offer an action on the same gate, then the communication will be possible if the patterns associated with both actions match (see Section 3.6). In such a case the whole behaviour will offer the action on the common gate, and once this happens both behaviours will continue their way until the next synchronization. This kind of synchronization is called *multi-way synchronization*, and it is useful when more than one behaviour have to synchronize on a common gate.

To show the use of this operator, we are going to add components to the register example, introducing two new processes: a producer that fills the positions and a consumer that empties them. These processes could be

```
process Producer [p1:data,p2:data] is
    (* produce first value *)
    p1(!val1);
    (* produce second value *)
    p2(!val2)
endproc
```

where *val*1 and *val*2 are two constants of type data, and

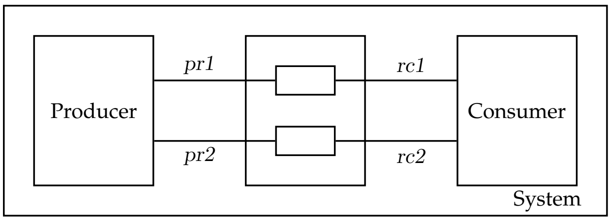
```
process Consumer [c1:data,c2:data] is
  var v1:data,v2:data in
    c1(?v1);
    (* consume first value *)
    c2(?v2)
    (* consume second value *)
  endvar
endproc
```

In E-LOTOS, comments are enclosed within (* and *).

Now we can put together these processes and the register to model the complete system. Our consumer only receives two elements and is not prepared for the loss of elements, so we use the register version that does not loose elements.

We use gates *pr1* (producer-register 1), *pr2*, *rc1* (register-consumer 1), and *rc2* to represent the communication between the processes. These gates, and the communications carried on them, are not interesting outside the *complete* system, so the environment cannot synchronize on them.

The system may be drawn as



The complete E-LOTOS system is:

```
process System1 is
    hide pr1:data,pr2:data,rc1:data,rc2:data in
        conc
            Producer [pr1,pr2]()
        |[pr1,pr2]|
            Register2 [pr1,pr2,rc1,rc2]()
        |[rc1,rc2]|
            Consumer [rc1,rc2]()
        endconc
    endhide
endproc
```

So the register must synchronize on its *input* actions (*pr1* and *pr2*) with the producer, and must synchronize on its *output* actions (*rc1* and *rc2*) with the consumer. Here we have used the bracketed syntax of the parallel composition operator,

$$\textbf{conc}\ B_1\ \mid[\ldots]\mid\ \ldots\ \mid[\ldots]\mid\ B_n\ \textbf{endconc}$$

(where the parallel composition operator is right associative, like any other binary operator) and the **hide** operator that, intuitively, is used to show that the gates that connect the register with the producer and the consumer are not observable for someone that sees the whole system. We will discuss this operator in Section 2.14.

In the behaviour $B_1\ \mid[\ G_1,\ldots,G_n\ ]\mid\ B_2$, B_1 and B_2 must assign disjoint global variables. There is no *share memory* that processes running in parallel can use. Communication has to be explicit, that is, B_1 and B_2 can communicate only through gates $G_1, \ldots, G_n$. Besides, if both behaviours assigned the same variable, we would not know the last value when the whole behaviour finished.

The behaviour

$$B_1\ \mid[\ G_1,\ldots,G_n\ ]\mid\ B_2$$

finishes when B_1 and B_2 do. It is said that both behaviours B_1 and B_2 must synchronize on their termination.

⏱ $B_1\ \mid[\ G_1,\ldots,G_n\ ]\mid\ B_2$ lets time pass if both B_1 and B_2 do so, or if one of them has finished (and it is waiting for the other) and the other lets time pass.

2.8 Interleaving operator

When the list of gates on which two parallel processes have to synchronize is empty, E-LOTOS has an abbreviated form of the parallel operator, the *interleaving* operator, "|||",

$$B_1\ |||\ B_2 \quad = \quad B_1\ \mid[]\mid\ B_2.$$

As a consequence, this operator generates the merging of the actions of both B_1 and B_2, in such a way that the actions belonging to each one of them remain in the same order.

As a matter of fact, there is an occasion when the processes have to synchronize: termination. As for the previous parallel operator, the interleaving B_1 ||| B_2 of B_1 and B_2 finishes when both processes do it.

We may use this operator to model, in an easier way than we did before, the fact that the register can receive the second element either before or after the first one has got out:

```
process Register5 [in1:data,in2:data,out1:data,out2:data] is
  var x1 : data,x2:data in
    in1(?x1:data);
    (   in2(?x2:data)
      |||
        out1(!x1)
    );
    out2(!x2)
  endvar
endproc
```

The bracketed syntax of the interleaving operator is as follows:

$$\textbf{inter}\ B_1\ \texttt{|||}\ \ldots\ \texttt{|||}\ B_n\ \textbf{endinter}$$

Being an abbreviation, the interleaving operator has the same features as the parallel operator regarding variables and time.

2.9 Synchronization operator

This operator is used when the two behaviours composed in parallel have to synchronize on every observable action. They also have to synchronize on the termination. Its unbracketed syntax is

$$B_1\ \texttt{||}\ B_2$$

and the bracketed syntax is

$$\textbf{fullsync}\ B_1\ \texttt{||}\ \ldots\ \texttt{||}\ B_n\ \textbf{endfullsync}$$

Note that

$$B_1\ \texttt{||}\ B_2 \quad = \quad B_1\ \texttt{|[}\ \text{“all gates of } B_1 \text{ and } B_2\text{”}\ \texttt{]|}\ B_2.$$

Using both this operator and the interleaving operator we can specify again the complete system that puts together the producer, the consumer, and the register. Producer and consumer have no common gate, so their parallel evolutions are independent of each other, and we can put them together using the interleaving operator. The resulting behaviour has to synchronize on all gates with the register, so we compose them with the synchronization operator, in the following way:

```
process System2 is
  hide pr1:data,pr2:data,rc1:data,rc2:data in
      (  Producer [pr1,pr2]()
         |||
          Consumer [rc1,rc2]()
      )
    ||
      Register5 [pr1,rc1,pr2,rc2]()
  endhide
endproc
```

Regarding assignments of variables and time, the synchronization operator behaves like the parallel operator.

2.10 General parallel operator

The three parallel operators we have seen ("||", "|||", and "|[...]|") are not easy to use when we want to describe some possible networks of communicating processes. Let us suppose we have a set of m processes, $P_1, P_2, \ldots, P_m$, and we want to specify a synchronization scheme where, in order to execute any action on a given gate, we need the collaboration of *any collection* of n of these processes. When $n = 2$ this means that any process P_i can communicate on that gate with any other process P_j $(i \neq j)$.

The specification in E-LOTOS of this kind of "n among m" synchronization using only the kind of operators we have already introduced, is not easy. To solve this problem, in E-LOTOS a new generalized parallel operator has been introduced, which allows to directly express networks of processes.

The syntax of this new operator is:

```
par G1#n1, ... ,Gp#np in
   [Γ1] -> B1
|| ...
|| [Γm] -> Bm
endpar
```

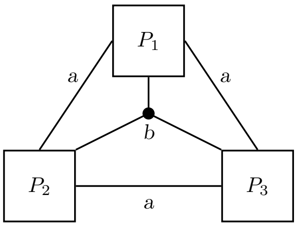
where n_i's are positive natural numbers, $p \geq 0$ and Γ_i's are gate lists $\Gamma_i = G_{i1}, \ldots, G_{ir_i}$.

The intuitive meaning of this operator is as follows: the whole behaviour can perform an action on a gate G, if:

- There is a component B_i such that gate G is not in its *synchronization list* Γ_i which performs it; or
- There are some components synchronizing in the following way:
    - if gate G is in the *list of degrees* (G_1#n_1, ... ,G_p#n_p) with degree n, then n whatever components B_i such that $G \in \Gamma_i$ synchronizes on G.
    - if gate G is not in the list of degrees, then all the behaviours B_i having G in their synchronization list Γ_i synchronizes on it.

The behaviour finishes when all the behaviours B_i have finished.

With this new operator we can specify, in an easy way, systems like the one graphically represented by



where on gate a two processes have to synchronize, and on gate b all the three processes have to synchronize, in the following way:

```
par a#2 in
     [a,b] -> P1
  || [a,b] -> P2
  || [a,b] -> P3
endpar
```

Regarding variables and time, the general parallel operator behaves as the other parallel operators: all the behaviours B_i have to assign pairwise disjoint variables, and it lets time pass if all B_i do, or if some of them have finished and all the unfinished ones let time pass.

2.11 Parallel over values operator

Sometimes we have a behaviour that may depend on a variable, and we want to put together, running in parallel, some instantiations of this common behaviour each one with a different value of this variable. For example, we may have a process that describes the behaviour of a node of a network which is identified by a unique identifier, and we may want to build a network with several nodes each one with a different identifier.

We can specify this kind of behaviour in E-LOTOS with the parallel over values operator. Its syntax is

par P **in** N ||| B **endpar**

where P is a pattern, N is a list of values (the type of N has to be List, a predefined type of E-LOTOS, see Section 3.1) these values can be matched against P, and B is a behaviour that may depend on the variables in P. The represented behaviour is the interleaving of a series of instantiations of B, one for each value of N. For example, if we have a process Node[...](*id*:IdType,...), we can specify a network of five nodes with different identifiers in the following way:

```
par ?x in [1,2,3,4,5] |||
  Node[...](x,...)
endpar
```

In other parallel operators, we have said that it is not allowed that two behaviours composed in parallel assign to the same variable. Here, it is the same behaviour which is used as a template to make several instantiations. If this template behaviour assigned to a global variable, then the different instantiations would do, and we would have several behaviours in parallel assigning to the same global variable. In order to forbid this situation, the template behaviour in a parallel over values cannot assign any global variable. Of course, it can assign local variables declared inside the template behaviour. In this way, each instantiation would have its own local variables.

Regarding time, the parallel over values operator lets time pass if all the instantiations do, or if some of them have finished and all the unfinished instantiations let time pass.

2.12 Disabling operator

The disabling operator "[>" models an interruption of a process by another. So the behaviour B_1 [> B_2 models the fact that at any point during the execution of B_1 there is a choice between doing a next action of B_1 or a first action of B_2. Once an action of B_2 is carried out, B_2 continues evolving and the remaining behaviour of B_1 is forgotten. But if B_1 finishes without interruption, the whole behaviour finishes and B_2 is not able to interrupt any more.

Coming back to the register example, we can use the disabling operator to model the fact that the register fails after 50 units of time if it has not finished yet its "normal" behaviour:

```
process Register6 [in1:data,in2:data,out1:data,out2:data] is
  var x1:data,x2:data in
    ( in1(?x1);
    ( in2(?x2) ||| out1(!x1) );
    out2(!x2) )
    [> ( wait(50); i )        (* the register fails *)
  endvar
endproc
```

where we have used the internal action **i** to represent the *internal* decision of the register to fail, and the **wait** operator (Section 2.16) to express when the register fails.

The disabling operator also has a bracketed syntax which is as follows:

$$\textbf{dis}\ B_1\,\texttt{[>}\ldots\,\texttt{[>}B_n\ \textbf{enddis}$$

Regarding variables, the disabling operator behaves like the selection operator, that is, in the behaviour B_1 [>B_2, when only B_1 or B_2 modifies a variable, it must be initialized before B_1 [>B_2; and if a variable is modified by both of them, they have to assign a value of the same type.

In order to keep internal action urgent and time deterministic, behaviour B_2 is required to be guarded in the disabling behaviour B_1 [> B_2. For technical reasons, B_1 is not required to be guarded (see Section 7.13 for a complete discussion on this subject).

The behaviour B_1 [>B_2 let time pass if both B_1 and B_2 do.

2.13 Suspend/Resume operator

The suspend/resume operator "[X>" is an extension of the disabling operator (Section 2.12) which allows the resumption of the interrupted behaviour. If during the evolution of the behaviour B_1 [X> B_2 B_1 is interrupted by B_2 then B_1 is suspended until B_2 resumes it through exception X. Then B_1 continues its evolution with the possibility of being interrupted again. B_2 is always restarted after a resumption.

Although exceptions and their handling will be seen in Section 2.15, it is enough to know now that an exception X can be raised with **signal** X.

Going on with our register example, if it can recover itself after an unexpected failure and continue as nothing wrong has happened, then we could specify it as follows:

```
process Register7 [in1:data,in2:data,out1:data,out2:data] is
  var x1:data,x2:data in
    ( in1(?x1);
    ( in2(?x2) ||| out1(!x1) );
    out2(!x2) )
    [Ok> ( wait(50); i; Repair[](); signal Ok )
  endvar
endproc
```

With this operator we can specify more complex interruption mechanisms, where, for example, a behaviour controls the evolution of others. Let us consider the next process

```
( ( B1 [cont1> stop1; start1;signal cont1 ) ||| (B2 [cont2> stop2; start2;signal cont2 ) )
|[stop1,stop2,start1,start2]|
B3
```

The behaviour B_3 controls B_1 and B_2 through gates *stop1*, *start1*, *stop2*, and *start2*. B_3 can stop the evolution of B_1 using the gate *stop1* and can restart it using *start1*. Using instead *stop2* and *start2* it can control B_2.

Regarding variables and time, the suspend/resume operator behaves like the disabling operator.

2.14 Hiding operator

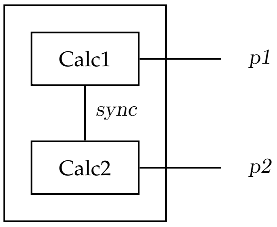
The hiding operator allows the abstraction of the internal operation of a process, by hiding those actions that are considered internal to it, and which are considered of no interest at certain level of detail. To do it, the hiding operator transforms observable actions into internal actions, hiding them to the environment, and making them urgent actions as we will see below.

The syntax of the hiding operator is

$$\textbf{hide}\ G_1[:T_1], \ldots, G_n[:T_n]\ \textbf{in}\quad B\quad \textbf{endhide}$$

where the G_i's are gates declared in B by the **hide** operator, so they can be used only by B. Types T_i are **any** by default, that is, if we do not declare the type of a gate, it can communicate values of any type.

Let us suppose that the producer used before is composed of two *subprocesses*, each of one produces a value.



In order to produce the values in the desired order, the components have to synchronize, but this synchronization (or the way it is done) is not interesting from the point of view of the whole system, so we hide it. Using the **hide** operator the producer would become

```
process Producer2 [p1:data,p2:data] is
    hide sync in
        Calculation[p1, sync](1)
      |[ sync ]|
        Calculation[p2, sync](2)
    endhide
endproc
```

where we have used two different instantiations of the same process Calculation, given by

```
process Calculation [pd:data,sync](turn:int) is
    case turn is
          !1 -> pd(!val1); sync
        | !2 -> sync; pd(!val2)
    endcase
endproc
```

where *val*1 and *val*2 are two constants of type data.

The process Calculation has a parameter *turn* of type int which is used to distinguish whether the process has to communicate the calculated value either before or after it synchronizes on gate *sync*. The **case** operator (Section 3.6) is used to define conditional behaviours. In this case, if *turn* is equal to 1 then the process first communicates the value and then offers a synchronization on gate *sync*, and if *turn* is 2 the process first synchronizes and then communicates its value.

Urgency

The behaviour **hide** G_1:T_1**,** . . . **,**G_n:T_n **in** B **endhide** lets a time d pass if B lets time d pass and B cannot offer a communication on a hidden gate ($G_1, \ldots, G_n$) in a time less than d. So, the **hide** operator transforms the actions on hidden gates in *urgent* actions, that is, they are performed as soon as they are enabled. Thus, when we hide the gate *sync* in the example above, we achieve two aims: first, this action cannot be observed outside the producer, and, second, communication on that gate must be performed as soon as possible, because *sync* is an urgent action. We will see another example in Section 2.16.

2.15 Exceptions and their handling

Exceptions are nowadays recognized as a desirable programming feature for dealing with errors and other abnormal situations. Concerning parallel languages, the importance of exceptions combined with concurrency has been pointed out by Berry in relation to the language ESTEREL [Ber93]. That proposal was adapted to the framework of process algebras by Nicollin and Sifakis in their "Algebra of Timed Processes," ATP [NS94]. However, none of the standardized Formal Description Techniques (SDL, LOTOS, and ESTELLE) supports exceptions. In E-LOTOS exceptions and their handling have been introduced through the **trap** operator (handling) and **raise** and **signal** instructions (raising).

Beginning with the **trap** operator, its syntax is:

```
trap
  exception $X_1$[($P_1$)[:$T_1$]] is $B_1$ endexn
    ...
  exception $X_n$[($P_n$)[:$T_n$]] is $B_n$ endexn
  [exit [$P_{n+1}$] is $B_{n+1}$ endexit]
in $B$
endtrap
```

where B stands for the "normal" behaviour and $X_1, \ldots, X_n$ are the exceptions that can be raised from B. Each exception X_i can be raised together with a value of type T_i (which is () by default) that will be trapped with the pattern P_i (which is () by default); and will be handled by behaviour B_i that defines how the behaviour evolves after the exception has been raised (and trapped). If the **exit** clause is present, when the behaviour B finishes and returns a value, this value is matched against the pattern P_{n+1} and the behaviour B_{n+1} is executed.

Taking the words of H. Garavel in [GS96], the **trap** operator behaves like a "watchdog." The normal behaviour B evolves until it raises an exception X_i. Then, B is aborted and the exception handler associated to X_i starts its execution, once the value raised together with the exception have been assigned to the corresponding pattern.

An exception may be raised with the **signal** or **raise** instructions whose syntax is

signal X [(E)]

raise X [(E)]

These instructions are quite similar; both raise an exception (possibly with associated values E), and if the exception is trapped, then there is no difference. But if the exception is not trapped[7], then the behaviour after a **signal** instruction can be executed, whereas if the exception has been raised with a **raise**, then the whole behaviour is blocked. In fact, **raise** X(E) is syntactic sugar for **signal** X(E); **block**.

In [GS96] some algebraic properties of the **trap** operator are shown together with the relationship with other operators which are derived from it.

We can use exceptions for example when our register fails. If it cannot recover itself, it finishes raising an exception *Error*.

```
process Register8 [in1:data,in2:data,out1:data,out2:data] raises [Error] is
  var x1:data,x2:data in
    ( in1 ?x1:data;
     ( in2 ?x2:data ||| out1 !x1 );
     out2 !x2 )
    [> ( wait(50); i; signal Error )
  endvar
endproc
```

Note how it is indicated with **raises** [*Error*] that this process can raise an exception.

If the producer receives the message that an error has occurred through gate *anError*, then we can handle exception *Error* with a communication with the producer through this gate:

```
   Producer3[pr1,pr2,anError]()
  |[pr1,pr2,anError]|
   trap
     exception Error is anError endexn
   in
     Register8[pr1,pr2]()[Error]
   endtrap
```

[7] In order to raise an exception X, it must be declared as an exception, so we could think that we can only raise exceptions declared in a **trap**, but as we will see in Section 4.4 a specification can also declare exceptions that are not trapped and are propagated to the top level, that is, to the level of the environment of the system.

2.16 Delay instruction

We have already seen how to make gates sensitive to the time when they occur (see Section 2.1). Now we present another construction related to time: the delay operator **wait**.

The behaviour

wait(E)

is idle while the time indicated by the expression E (which must have type time) passes, and then finishes.

For example, if our register has a delay in such a way that a received value is not able to get out before three units of time have passed since it was received, we could specify it as

```
process Register9 [in1:data,in2:data,out1:data,out2:data] is
  var x1:data,x2:data in
    hide canGetIn2 in
          in1(?x1); CanGetIn2; wait(3); out1(!x1)
        |[canGetIn2]|
          canGetIn2; in2(?x2) ; wait(3); out2(!x2)
    endhide
  endvar
endproc
```

By using the **hide** operator and a synchronization on gate *canGetIn2*, we have described in a different way that before the fact that data received through gate *in1* has to get in before data through gate *in2*. In this way it is easier to specify that we need to let pass three units of time before the corresponding value can get out.

If we do not know exactly the delay of the register, but we know a lower bound, *min*, of this delay, we can specify that the time passed between the moment the element gets in and the moment the element is able to get out is not less than this value, in the following way:

```
in1(?x1);
?t := any time [min <= t]; wait(t);
out1(!x1)
```

where we have used a nondeterministic assignment, whose meaning will be explained in Section 2.19. We have that once an element has got in through gate *in1*, an arbitrary value of type time greater than or equal to *min* is assigned to variable t, and we have to wait for this time, and then a communication on gate *out1* is offered.

Let us see now another simple, classical example[8] which uses both a time annotation on actions and the **wait** operator. We have a sender process that, after having transmitted a message, waits for an acknowledgment. If this acknowledgment does not arrive during a certain time then the sender retransmits the same message; otherwise, it waits for another message to be transmitted. The process that specifies the sender behaviour is as follows:

```
process Sender1[req:message,trans:message,ack] is
  var m:message in
    req(?m);
    Send_Message1[trans,ack](m);
    Sender1[req,trans,ack]
  endvar
endproc
```

where we have defined

[8] This example has been adapted from [LL94].

```
process Send_Message1[trans:message,ack](m:message) is
  var t:time in
    trans(!m);
    (  ack @?t [t<waiting_time]
    [] wait(waiting_time);Send_Message1[trans,ack](m)
    )
  endvar
endproc
```

Note that if the acknowledgment does not arrive before *waiting_time* units of time have passed, the action on gate *ack* becomes impossible, and only a new transmission (of the same value) is possible (via the recursive call to Send_Message1).

Suppose that we want to separate the time constraints from the rest of the behaviour. We can have a process Timer that controls these constraints and interacts with the sender:

```
process Timer[setT:time,reset,timeout] is
  var t:time in
    setT(?t);
    (  reset
    [] wait(t);timeout
    );
    Timer[setT,reset,timeout]()
  endvar
endproc
```

and modified processes:

```
process Sender2[req:message,trans:message,ack,setT:time,reset,timeout] is
  var m:message in
    req(?m);
    Send_Message2[trans,ack,setT,reset,timeout](m);
    Sender2[req,trans,ack,setT,reset,timeout]
  endvar
endproc
```

and

```
process Send_Message2[trans:message,ack,setT:time,reset,timeout](m:message) is
  trans(!m);setT(!waiting_time);
  (  ack;reset
  [] timeout;Send_Message2[trans,ack,setT,reset,timeout](m)
  )
endproc
```

And then the whole sender could be like

```
  Sender2[req,trans,ack,setT,reset,timeout]
|[setT,reset,timeout]|
  Timer[setT,reset,timeout]
```

The problem is that in this last behaviour, the synchronization on gates *setT*, *reset*, and *timeout* is forced, but it is not forced the time at which they occur. We have to hide these gates in order to make the actions on them urgent actions, that is, the synchronizations on these gates occur as soon as they are possible. Thus, the sender can be

```
process Sender3 [ req:message, trans:message, ack ] is
  hide setT:time, reset, timeout in
      Sender2 [ req, trans, ack, setT, reset, timeout ]
    |[setT, reset, timeout]|
      Timer [ setT, reset, timeout ]
  endhide
endproc
```

2.17 Renaming operator

In E-LOTOS we can rename actions by means of an explicit renaming operator which allows to rename observable actions into observable actions, and exceptions into exceptions, and also allows to modify the offers of the actions performed by a behaviour.

For example, let us suppose we have two behaviours B_1 and B_2 which we cannot modify since they belong to a private library of processes. If we compose them in parallel and they have to communicate with each other, we have a problem if the messages sent by any of them, say B_1, are not in the same format as B_2 is expecting. For example, B_1 could send a message $G(!E)$ through the gate G but B_2 could expect messages of the form $G(!f(header, E, trailer))$, where f is a function to build packets from messages by adding them a header and a trailer. We can solve this problem by means of the renaming operator.

An explicit renaming operator is also useful for verification purposes, when it is desirable to rename all non-interesting (for the verification of the property to be checked) observable actions into some particular action, different of the internal action (so hiding cannot be used).

The syntax of this operator is as follows:

```
rename
  gate $G_1[(P_1)[:T_1]]$ is $G'_1[(P'_1)]$
    ...
  gate $G_m[(P_m)[:T_m]]$ is $G'_m[(P'_m)]$
  signal $X_1[(P''_1)[:T'_1]]$ is $X'_1[(E_1)]$
    ...
  signal $X_n[(P''_n)[:T'_n]]$ is $X'_n[(E_n)]$
in $B$
endren
```

It represents the behaviour B where the gates $G_1, \ldots, G_m$ and the exceptions $X_1, \ldots, X_n$ are renamed into $G'_1, \ldots, G'_m$ and $X'_1, \ldots, X'_n$, respectively. These gates and exceptions are declared by the **rename** behaviour, so they can be used only in B. P_i's are patterns used as follows: values associated to renamed gates or exceptions are bounded to variables in P_i's, and then these variables can be used in the corresponding new gates (patterns P'_i) or exceptions (expressions E_i).

The simplest renaming is the one that only renames one gate into another:

```
rename
  gate inP(?a):int is intro(!a)
in
  B
endren
```

In this way, if behaviour B offers a communication on gate *inP* of an integer, then this value is bound to a and the whole behaviour will offer a communication on gate *intro* of the same integer a.

However, gate renaming may be more powerful than a simple change of name. We can also use it to change the structure (type of values) of actions offered by a process. For example we can remove a record field:

```
rename
  gate inP ((x => ?a, y => ?b)):recordXY is inP(!(x => a))
in
  B
endren
```

where recordXY is a type defined by the user (in E-LOTOS anonymous types cannot be used) as

```
type recordXY is
  (x => int, y => bool)
endtype
```

Note that there are two gates called *inP*: the left one is the new declared gate that can be used in B; and the right one is a gate which have to be declared outside this **rename**, and cannot be used by B. If B performs the communication *inP* `(!(`x `=> 3,`y `=>` *true*`))` on (local) gate *inP*, the variable a will be bound to the value 3, and the variable b will be bound to the value *true*, and the whole rename behaviour will perform the communication *inP* `(!(`x `=> 3))` on (global) gate *inP*.

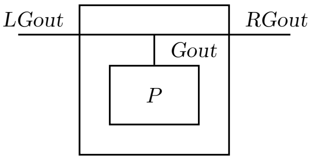
We can add a field:

```
rename
  gate inP ((x => ?a:int)):recordX is inP (!(x => a,y => true))
in
  B
endren
```

We can even change the values that are being communicated. For example, we can solve the problem at the beginning of this section in the following way

```
 rename
  gate G (?e):message is G(!f(header,e,trailer))
 in
  B1
 endren
||
 B2
```

We can also split a gate G into two gates G_1 and G_2, depending on the values G carries on. For example, let us suppose we have defined a process P that sends values through a gate *Gout* indicating whether they have to go to the right or to the left side, by using actions like *Gout*`(!(`*left*`,`*dat*`))` or like *Gout*`(!(`*right*`,`*dat*`))` (where *left* and *right* are values of an enumerated type (see Section 3.1) and *dat* is a variable with a value of type data). If we want to use process P in a context where we have two different gates, *LGout* to send values to the left, and *RGout* to send values to the right,



we can use the **rename** operator as follows:

```
rename
  gate Gout ((!left,?d:data)):any is LGout(!d)
  gate Gout ((!right,?d:data)):any is RGout(!d)
in
  P[Gout]()
endren
```

We can also do the opposite, that is, we can merge two gates G' and G'' in a single gate G:

```
rename
  gate G'(?a):int is G(!(a,true))
  gate G''(?a):int is G(!(a, false))
in
  B
endren
```

Exceptions can be renamed in the same way. Let us imagine a process that raises a generic exception *Error* together with the number of the produced error. If at some time the association between numbers and errors is changed in the error table, and we have a function g that defines this change, we can rename the exception *Error* in order to vary its parameter as function g indicates:

```
rename
  signal Error (?e):int is Error (g(e))
in
  B
endren
```

2.18 Conditional operator

As in almost every programming language, E-LOTOS includes a conditional **if** – **then** – **else** operator. Its syntax is:

```
if E then B
  (elsif E then B)*
  [else B]
endif
```

where E must be a boolean expression. The behaviour associated with the first expression E that evaluates to *true* describes how the **if** – **then** – **else** behaves. The behaviour associated with the **else** clause is **null** by default, and therefore this is the behaviour when no expression E is *true*, and there is no else clause. For instance, we can use this instruction for communicating the absolute value of variable x through gate *outP*:

```
if x >= 0 then  outP(!x)
else  outP(!-x)
endif
```

2.19 Imperative features

In E-LOTOS several imperative features have been introduced in order to make the job of specifying systems easier to the user of this programming paradigm.

One of these features is assignment. E-LOTOS has *write-many* variables, that is, variables that can be assigned several times. The static semantics of the language forbids to read a variable unless it has a value. We can assign values to a variable by using the assignment instruction, whose syntax is

$$P \texttt{ := } E$$

where P is a pattern which can be pattern-matched against the value returned by expression E. The result of the assignment is the creation of a binding between the variable (or variables) in the pattern and the expression value. In this way, the assignment

$$\texttt{?}x \texttt{ := 3}$$

binds the value 3 to variable x. On the other hand, the assignment

$$\texttt{(?}x\texttt{, ?}y\texttt{) := (5, } \mathit{true}\texttt{)}$$

binds variable x with 5 and variable y with *true*.

There is another kind of assignment, the *nondeterministic assignment*, whose syntax is:

$$P \texttt{ := } \mathbf{any}\ T\ [\texttt{[} E \texttt{]}]$$

By means of it we bind the variables in pattern P with any value of type T that satisfies the condition E, which is *true* by default. For example, the assignment

$$\texttt{?}x \texttt{ := } \mathbf{any}\ \mathsf{int}\ \texttt{[} x \texttt{ < 10]}$$

binds variable x with an integer less than 10.

Another imperative feature introduced in E-LOTOS is the possibility of declaring variables, using the **var** operator (as we have seen in several examples), in order to restrict the scope of variables, and therefore also hiding global variables with the same name. The syntax of this operator is:

```
var V1:T1[:=E1], ... ,Vn:Tn[:=En] in
  B
endvar
```

where the declared variables $V_1, \ldots, V_n$ may be initialized with the corresponding expressions E_j.

The occurrences of the identifiers naming the declared variables in behaviour B refer to these new variables, hiding in the scope those variables with the same name that already exist. Changes of the values of these (local) variables do not modify anyway the external ones. On the other hand, they will not be visible outside the scope of the **var** operator .

For example, the behaviour

```
?x := 3;
var x : int in
  ?x := 5; ?y := x
endvar;
inP(!(x,y))
```

offers on the gate *inP* the pair (3,5).

As we said, declared variables may be initialized, and so the following behaviour is equivalent to the previous one:

```
?x := 3;
var x : int := 5 in
  ?y := x
endvar;
inP(!(x,y))
```

A variable V declared of type T can be assigned a value of any subtype T' of T. In the current version of E-LOTOS, when V is assigned a value of T', $T' \sqsubseteq T$, then it can only be assigned a value of a subtype of T', losing the information about the type T in the declaration. We have propose to change this, and keep the *static* type T associated with variable V (see Section 7.31.1).

E-LOTOS has also several iterative operators: **loop**, **while**, and **for**. The **loop** operator represents an infinite loop whose execution can only be stopped raising an exception with the instruction **break**. It has two versions. The first one, whose syntax is

loop B **endloop**,

represents a loop where the behaviour B is continuously executed until the predefined exception *inner* is raised (by means of **break** command, see below). The second version has syntax

```
loop X [: T] in
  B
endloop
```

and represents a *breakable* loop that can be interrupted with the exception X.

We raise this exception[9] with a **break** command, with syntax

break [X [(E)]]

where X is the name of the exception associated with the loop we can stop (and if it has no name, the default *inner* exception will be used) and E, the value associated with the exception, represents the result that the loop returns.

Note that exception X is declared together with the loop, and so we do not need to introduce an explicit handler for it. The handling of this exception consists of breaking (stopping) the loop, and returning the associated value, if any[10].

For example, using the **loop** operator we can define the recursive (infinite) process that continuously receives and sends integers, as follows:

```
process Buffer3 [inP:int,outP:int] is
  var x:int in
     loop
       inP(?x); outP(!x)
     endloop
  endvar
endproc
```

On the other hand, the buffer that receives and sends integers until it receives a 0, can be specified with a breakable **loop**:

[9] And also the predefined *inner* exception, but in this case no name has to be given.

[10] In the definition of the semantics [Que98] we see that a **trap** is included (see Section 6.8).

```
process Buffer4 [inP:int,outP:int] is
  var x:int in
    loop End in
      inP(?x); outP(!x)
      if x=0 then break End endif
    endloop
  endvar
endproc
```

E-LOTOS has also a *conditional* loop, the **while** instruction, whose syntax is:

```
while E do
  B
endwhile
```

where E is a boolean expression, and B is a behaviour which is executed until the expression E evaluates to *false*. For example, we can add all the elements in a list of integers, xs, with type **list of** int (see Section 3.1), and send the result through a gate *outP* as follows:

```
var total : int := 0,e:int in
  while not(isempty(xs)) do
    ?e := head(xs);
    ?total := total + e;
    ?xs := tail(xs)
  endwhile;
  outP(!total)
envar
```

where we have used some functions for manipulating lists, to be described in Section 3.1.

The last iterative construction of E-LOTOS is the **for** loop, whose syntax is:

```
for E1 while E2 by E3 do
  B
endfor
```

and represents the execution of the expression E_1, and then, while expression E_2 is evaluated to *true*, the execution of behaviour B followed by the evaluation of expression E_3. Thus, the **for** loop is just syntactic sugar for

```
E1;
while E2 do
  B;E3
endwhile
```

2.20 Process declaration and instantiation

We can use process declarations to give a name to a behaviour and to abstract the names of actual gates, parameters, and exceptions. Thus, processes are parameterized by a list of formal gates, a list of formal variables (parameters), and a list of formal exceptions.

The syntax of a process declaration is as follows:

$$\begin{array}{ll}
\textbf{process}\ \Pi & [\ \texttt{[}\ G_1[\texttt{:}T_1]\texttt{,}\ldots\texttt{,}G_n[\texttt{:}T_n]\ \texttt{]}\ ] \\
 & [\ \texttt{(}\ [\textbf{in}|\textbf{out}]V_1\texttt{:}T'_1\texttt{,}\ldots\texttt{,}[\textbf{in}|\textbf{out}]V_m\texttt{:}T'_m\ \texttt{)}\ ] \\
 & [\textbf{raises}\ \texttt{[}\ X_1[\texttt{:}T''_1]\texttt{,}\ldots\texttt{,}X_p[\texttt{:}T''_p]\ \texttt{]}] \\
\quad\textbf{is}\ B & \\
\textbf{endproc} &
\end{array}$$

where

- Π is a process identifier;
- $\texttt{[}\ G_1[\texttt{:}T_1]\texttt{,}\ldots\texttt{,}G_n[\texttt{:}T_n]\ \texttt{]}$ is the list of gates the process uses, which is empty (`[]`) by default;
- T_i is the type of gate G_i, that is, the type of values gate G_i can communicate. It is **any** by default, that is, a gate can communicate values of any type by default[11];
- $\texttt{(}\ [\textbf{in}|\textbf{out}]V_1\texttt{:}T'_1\texttt{,}\ldots\texttt{,}[\textbf{in}|\textbf{out}]V_m\texttt{:}T'_m\ \texttt{)}$ is the list of formal parameters, where each parameter is either an input parameter (by default), **in**, or an output parameter, **out**;
- $\texttt{[}\ X_1[\texttt{:}T''_1]\texttt{,}\ldots\texttt{,}X_p[\texttt{:}T''_p]\ \texttt{]}$ is the list of formal exceptions, empty by default;
- T''_j is the type of values associated with exception X_j, which is `()` by default, that is, by default an exception has no value associated; and
- B is the *body* of the process, the behaviour that describes how the process behaves.

Declared processes may be called by using process instantiations, whose syntax is:

$$\Pi\ \texttt{[}[GPL]\texttt{]}\ \texttt{(}[APL]\texttt{)}\ [\texttt{[}XPL\texttt{]}]$$

where

- GPL is the gate parameter list and it may be either in *positional* form, $G'_1\texttt{,}\ldots\texttt{,}G'_n$, or in *named* form, $G_{i_1}\ \texttt{=>}\ G'_{i_1}\texttt{,}\ldots\texttt{,}G_{i_s}\ \texttt{=>}\ G'_{i_s}$ (G_{i_j} are the formal gates and G'_{i_j} the actual gates). When the named form is used, it is not needed to give all the parameters by using the keyword **...** at the end, and in this case, the absent actual parameters will be added with the same names as the formal ones. For example, a process with header

  process Register `[` *in1*`:data,`*in2*`:data,`*out1*`:data,`*out2*`:data` `]` **is**

  can be instantiated as follows:

  Register `[` *in1* `=>` *inGate1* `,` *out1* `=>` *outGate1* `,`**...** `]` `()`

  and this instantiation is equivalent to this one:

  Register `[` *inGate1* `,` *in2* `,` *outGate1* `,` *out2* `]` `()`

- APL is the list of actual parameters where each parameter can be an expression (input parameter) or a pattern (output parameter), and the list may be either in positional or named form; and
- XPL is the exception parameter list, which may also be either in positional or named form.

An instantiation of a process Π stands for the body of the process where actual gates, parameters, and exceptions are substituted for formal ones.

[11] This is made for compatibility with LOTOS.

3 Base Language for data types

In this chapter we describe how the data types used in the behaviour language described in the previous chapter can be defined in E-LOTOS, and how values of these types are used.

3.1 Data types

As we have said, the part regarding the declaration and use of data types is one of those that has been changed more in E-LOTOS with respect to its predecessor LOTOS.

In LOTOS the abstract data type specification language ACT ONE [EM85] is used to declare new data types and to represent their values (called in LOTOS value expressions). This language is not too user-friendly and suffers from several limitations such as the semi-decidability of equational specifications, the lack of modularity, and the inability to define partial operations.

In E-LOTOS, ACT ONE has been substituted by a new language in which data types are declared in a similar way as they are in functional languages (ML, Haskell), and where some facilities for the use of values are given.

Predefined types. In E-LOTOS there is a set of predefined types with associated operations which are specified in the "Predefined Library" described in Chapter 7 of [Que98]. Predefined types are:

- bool: with constants *true* and *false*, and operations not, and and or among others.
- nat: the natural numbers. In E-LOTOS, we have a specific syntax (similar to the usual one) to generate values of this type (and also for the rest of the numeric types), so that we can write 1, 7, 2258 . . . instead of having to use the constructors, to write expressions like

  ```
  Succ(Succ( ...  Succ(0)...)).
  ```

  E-LOTOS also provides arithmetic operations, like +, -, *, div, mod, and comparison operations, like >, >=, !=, ...
- int: the integer numbers, with a rich syntax like naturals, and with arithmetic and comparison operations. There is also an operation to transform a positive integer to the corresponding natural (nat), and another to transform a natural to the corresponding integer (int)
- rational: the rational numbers. In addition to arithmetic and comparison operations, there are functions like round(r), to get the nearest integer value of r, ceil(r), to get the least integer value greater than or equal to r, and floor(r), to get the greatest value less than or equal to r.
- float: the real numbers, with arithmetic, comparison, and trigonometric operations.
- char: the character values, written between quote symbols (e.g. `'a'`). They represent the ISO Latin-1 characters. There are functions like tolower, toupper, isalpha, isdigit, islower, isupper.

- string: sequences of characters between double quotes (e.g. ``‘‘Hello, world!’’``). There are operations to know the number of characters of a string (length), to concatenate two strings (concat), to take any prefix or suffix of a string, to take the character in any position. There are also comparison operations, and operations to convert any natural, integer, or float to a string.
- List: lists of values of any type. It is defined as a new data type (see below)

  ```
  type List is
    nil | cons(any,List)
  endtype
  ```

  The user can define lists of a concrete type as we explain below.

Predefined type schemes. In E-LOTOS there is also a set of type schemes that are translated to usual type and function declarations, and that are used to make easier the definition of typical types, as suggested by the “rich term syntax” of [Pec94].

We can define an *enumerated* type ET with values $V_1, \ldots, V_n$ as

```
type ET is
  enum V_1, ..., V_n
endtype
```

This type has several predefined functions as a comparison operation (==), and operations to get the next value (succ) or the previous value (pred) of a given one.

For example, we can define a type colour with some colours:

```
type colour is
  enum Blue, Red, Green, Yellow, Pink
endtype
```

which is translated into a new data type (see below) with a constructor for each enumerated value:

```
type colour is
  Blue | Red | Green | Yellow | Pink
endtype
```

The language also allows record types to be easily defined and dealt with. It is possible to declare a record by giving the list of its fields together with their corresponding types. For example, we can define the record type

```
(name => string, address => string, age => nat)
```

and we can access each field with the “.” notation, for example *rec*.*name*, provided variable *rec* has the above type.

We can also define sets of values of a given type T, with syntax

```
type ST is
  set of T
endtype
```

and express extensionally values of this type, i.e., by giving the list of their elements,

$$\{e_1, \ \ldots \ ,e_n\}$$

where e_i are expressions of type T. This scheme has predefined functions to calculate the union (union), difference (diff), and intersection (inters) of two sets; to calculate the number of elements (card) of a set; and to know if an element belongs (isin) to a set; to know if a set is empty (isempty).

We can define lists of elements of type T in the following way

```
type LT is
  list of T
endtype
```

and write list values with syntax

$$[e_1, \ldots, e_n]$$

where e_i are expressions of type T. This scheme has predefined functions to know the first element (head) of a list, to remove the first element of a list (tail), to know the nth element of a list (nth), to concatenate (concat) two lists, and to know the number of elements (length) of a list.

By using these functions we can write again a behaviour that adds the elements of a list of integers, xs, and communicates the result through gate *outP*, but without destroying the given list:

```
var total : int := 0, len : int := length(xs),e:int,n:nat in
  for ?n := 1 while n<=len by ?n := n+1 do
    ?e := nth(xs,n);
    ?total := total+e;
  endwhile;
  outP(!total)
envar
```

The predefined operations on these types and their implementations are shown in Chapter 7 of [Que98].

User defined types. The user can define two kinds of types: type synonyms and new data types.

A type synonym declaration simply declares a new identifier for an existing type. For example, we can define the type Complex to represent complex numbers, as a record with two float fields:

```
type Complex is
  (real => float, imag => float)
endtype
```

The general syntax to declare a type synonym is

```
type S is (RT) endtype
```

where S is a type identifier, and *RT* is a record type, that is, a non empty list of type identifiers separated by commas, where each type can be labelled with the name of the corresponding record field. In the previous example the record has two fields: *real* and *imag*, both of type float.

It is also possible to declare a type synonym renaming an existing type, with syntax

```
type S renames T endtype
```

where T is a type identifier. For example, the type time can be declared as a synonym of the predefined type nat with the declaration

```
type time renames nat endtype
```

Type equality is *structural*, not by *name*. Thus, with the above definitions, we can use Complex and the anonymous type (*real* => float, *imag* => float) as the same type, and also nat and time.

The declaration of a new data type consists of the enumeration of all the constructors for that type, each one with the types of its arguments, separated with "|". The concrete syntax to declare a new data type is:

```
type S is
  C_1(RT_1) | ... | C_n(RT_n)
endtype
```

For example, we can define a type dest which represents the destination of a message when it reaches a router with two exits, one on the left and one on the right, as follows:

```
type dest is
  left | right
endtype
```

Also, we can define the type of messages, data messages or acknowledgment messages (for error handling), as follows:

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

We can define recursive data types, by using in some constructors the type which is being declared, as in the predefined type List described above.

The base language does not allow to declare parameterized types. This is left for the module system (see Section 4.3).

3.2 Type expressions

We have seen in the previous section how the user can *declare* new types with the facilities of the language. Now we are going to see how to write type expressions, that is, what the user can write when a type is needed, for example, when she wants to type a gate or a local variable in a behaviour.

The simplest type expression is a type *identifier*. We have already seen several examples: nat and bool are predefined type identifiers, Complex is a type synonym identifier, and dest is a new datatype identifier.

Type identifiers are the unique type expressions the user can use in order to type an E-LOTOS element. For example, when the user types a gate of a process, $G : T$, T must be a predefined or user-declared type identifier, or one of the special type identifiers, **none** and **any**, that we present below. It is the same when we want to declare a variable or a pattern type. Thus, at the user level, in E-LOTOS there are no anonymous types. For instance, the user is *not* able to declare

$$G : (\mathit{real} \texttt{ => } \mathsf{float}, \mathit{imag} \texttt{ => } \mathsf{float}),$$

although, of course, she can declare

$$G : \mathsf{Complex}$$

provided the Complex type is already declared.

We have two special types:

- **none** is the empty type, that has no values, used to give functionality to processes that never stop.

- **any** represents a wildcard type which is used to type gates which can communicate data of any type, in order to achieve compatibility with LOTOS.

We can also write record type expressions RT, although as we have seen, the user can only use them in the declaration of type synonyms. Note that a record type expression in *not* a type, and it becomes a (anonymous) type when we enclose it between parentheses, (RT). We have also seen that each component of a record can be associated with a field name. But when we give name to a field, we have to give name to all the fields. For example, in the record type expression

(x => float,y => float)

x and y are the names of the record fields. In this way, the order of the different fields is not important, but, of course, the names of the fields are. However, it is optional to give name to the different fields of a record, so a correct record type expression is

(int,float),

which represents a record with an integer and a float. In this case the order of the fields *is important.* Really, this expression is syntactic sugar for

($1 => int,$2 => float),

where E-LOTOS *invents* default names for the fields, $\$1, \$2, \ldots$, when the user does not give them. And ($1 => float,$2 => int) is a different type.

In E-LOTOS we can write *extensible* record types, with the keyword **etc** at the end of the record, meaning any set of fields. So the record type

(*message* => data, **etc**)

represents a type of records with *at least* one field called *message* of type data. This means that we have as values of this type all the record values that have *at least* a field with that name and type.

The notion of extensible records is related to record subtyping, that we detail in the next section.

3.3 Subtyping

E-LOTOS has a built-in subtyping relation between record types. A record type is subtype of another record type if the former has at least all the fields the latter has. So the record type (**etc**) is a supertype of any other record type. And the extensible record type

(*message* => data, **etc**)

has as subtypes, for example, the record types

(*message* => data, *de* => dest, **etc**) and (*message* => data, *de* => dest).

We denote that type T is a subtype of type T' by $T \sqsubseteq T'$. Whenever $T \sqsubseteq T'$ we can use values of type T where values of type T' are required.

Let us see an example that shows the usefulness of subtyping. Let us suppose we have to specify a Router process that receives packets through an input gate, and it has to direct them to one of its output gates, depending on the packet destination. So the only restriction is that packets that get into the router have to have a field indicating their destination. We can declare an extensible record type

```
type packet is
  (de => dest, etc)
endtype
```

and then a process specifying the router is the following:

```
process Router [inP:packet, leftP:packet, rightP:packet] is
  var p:packet in
    loop
      inP(?p);
      case p.de is
          left -> leftP(!p)
        | right -> rightP(!p)
      endcase
    endloop
  endvarendproc
```

The empty type **none** is subtype of any type T, **none** $\sqsubseteq T$, and it is equivalent to any record with a field of type **none** (since that field cannot have any value, the whole type has no values). The universal type **any** is supertype of any type T, that is, $T \sqsubseteq$ **any**.

3.4 Expressions

In contrast to LOTOS [ISO88], where there is a separation between processes and functions, E-LOTOS considers functions as a kind of processes. A function in E-LOTOS is any process with the following characteristics: it is deterministic; it cannot communicate (i.e. it has no gates), and so its only capabilities are to return values and raise exceptions; and has no real-time behaviour (i.e. a function is an immediately exiting process). Therefore, the expression (sub)language is very similar to that of behaviours, that we have been introducing, once the elements related to these characteristics are removed.

The simplest expressions are *normal forms*. A normal form is an expression which cannot be reduced any further. The following examples are normal forms:

- a *primitive* constant, such as 5, *true*, or "Hello, world!",
- a variable, such as x or *total*,
- a constructor of a new data type applied to normal forms, such as *right*, *nil*, *cons*(5, *nil*), and *send*(*aPacket*, *aBit*).
- a record of normal forms, such as (*real* => 4.5, *imag* => 8) or (7, 9) (that is syntactic sugar for ($1 => 7, $2 => 9)).

Starting from these simple expressions we can construct more complex ones using the operators of E-LOTOS.

Expressions can raise exceptions, provided they are in the scope of an application of the **trap** operator where they are handled, as we saw in Section 2.15:

```
trap
  exception $X_1$[($P_1$)[:$T_1$]] is $E_1$ endexn
    ...
  exception $X_n$[($P_n$)[:$T_n$]] is $E_n$ endexn
  [exit [$P_{n+1}$] is $E_{n+1}$ endexit]
in $E$
endtrap
```

However, we have to remark that the handler behaviours have been substituted by expressions.

For example, let us consider the function Head that returns the head of a list:

```
function Head (xs:intlist) : int raises [Hd] is
  var x:int := 0 in
     case xs is
          nil -> signal Hd;0
        | cons(?x,any:intlist) -> x
     endcase
  endvar
endfunc
```

Function Head declares that it may raise an exception, whose *formal* name is *Hd*. Inside the function, pattern-matching is used to distinguish whether the integer list *xs* is empty or not (patterns and pattern matching will be detailed in Section 3.6). When the list is empty the function raises the exception *Hd* (and if the exception *Hd* is not trapped, the function Head returns 0, because all the branches in a **case** statement have to return something of the same type if they finish). In this way, the user of the function is who has to give the handler of the exception. For example, if we do not want an error to be produced when the head of an empty list is consulted, but instead 0 is to be returned, we can use

```
trap
     exception Hd0 is 0 endexn
in
     Head (xs) [Hd0]
endtrap
```

that returns 0 if the list *xs* is empty, and otherwise returns the head of the list. The exception *Hd0* is the *actual* exception in the Head function call.

We can also rename exceptions raised by an expression (in this case it has no sense to rename gates, as expressions do not have gates) with all the renaming power we saw in Section 2.17. Now the syntax is

```
rename
   signal $X_1[(P''_1)[:T'_1]]$ is $X'_1[(E_1)]$
     ...
   signal $X_n[(P''_n)[:T'_n]]$ is $X'_n[(E_n)]$
in $E$
endren
```

The expression language also has a conditional expression **if** – **then** – **else**; and some imperative features, such as:

- assignment, P := E, that produces bindings between the variables in the pattern P and values of E. That expression does not return a value but produces bindings (between variables in P and the value returned by expression E).
- sequential composition, E_1 ; E_2, which returns bindings produced by E_1 not overridden by those of E_2, bindings produced by E_2 and the value returned by E_2.
- declaration of local variables, with syntax

```
var $V_1$:$T_1$[:=$E_1$], ... ,$V_n$:$T_n$[:=$E_n$] in
  $E$
endvar
```

For example, the expression

```
var x:int in
  ?x:=E; x * x
endvar
```

returns the same value as the expression $E * E$ does, and it does not produce bindings because variable x is local to the **var** declaration, and its binding does not go out of this declaration.

- different iterative constructions such as **loop**, **while**, and **for**.

Finally, E-LOTOS includes some operators that can only be applied to expressions:

- boolean conjunction E_1 **andalso** E_2 that evaluates E_1 and if its value is *true* then the value of E_2 is returned, otherwise *false* is returned. E_1 and E_2 must be of type bool, that is, they have to return a value of this type.
- boolean disjunction E_1 **orelse** E_2 that evaluates E_1 and if its value is *false* then the value of E_2 is returned, otherwise *true* is returned. E_1 and E_2 must be of type bool.
- equality operation, E_1 = E_2. E_1 and E_2 do not need to have the same type.
- inequality operation, E_1 <> E_2. E_1 and E_2 do not need to have the same type.
- select field operation, $E.V$. Assuming that expression E returns a record value with a field called V, $E.V$ returns the value associated with this field.
- explicit typing, $E:T$, that returns the value of expression E only if it is of type T. This is controlled by the static semantics at compilation time.

3.5 Function declaration and instantiation

As we can declare processes (Section 2.20), we can declare functions in order to give a name to an expression E, and abstract the names of variables (or subexpressions) and exceptions in E.

The syntax of a function declaration is as follows:

```
function F   [ ( [in|out]V1:T1, ... ,[in|out]Vm:Tm ) ] [:T]
             [raises [ X1[:T'1], ... ,Xp[:T'p] ]]
is E
endfunc
```

where

- F is a process identifier;
- ([**in**|**out**]V_1:T_1, ... ,[**in**|**out**]V_m:T_m) is the list of formal parameters, empty () by default;
- T is the type of the value returned by F, and it is an empty record () by default;
- [X_1[:T'_1], ... ,X_p[:T'_p]] is the list of formal exceptions, empty [ ] by default; and
- expression E is the body of the function F.

A function call is an expression. Its syntax is

$$F[(APL)][[XPL]]$$

where APL is the list of actual parameters and XPL is the exception parameter list, which may be either in positional or named form.

We can declare also *infix* functions

function F **infix** [([**in**|**out**]V_1:T_1,[**in**|**out**]V_2:T_2)] [:T]
[**raises** [X_1[:T'_1], ... ,X_p[:T'_p]]]
is E
endfunc

which can be called as follows:

$$AP_1\ F\ AP_2[\texttt{[}XPL\texttt{]}]$$

We can also declare *values*, which represent constant functions without parameters, with syntax

value V : T **is** E **endval**

where V is the value identifier, T is a type identifier, and the value of expression E (which has to be of type T) is what value V returns.

3.6 Patterns and pattern matching

We have already used patterns in several examples: assignments P:=E, actions G P_1 @ P_2, etc. Patterns are matched against values and can produce bindings on variables. Now we are going to describe the different kinds of patterns in E-LOTOS. A pattern has one of the following forms:

- a variable, ?x. If we try to match ?x against a value N of type T, the pattern matching succeeds if the variable x has been declared with type T' and T is a subtype of T'. In this case, x is bound to N.
- an expression, !E. This pattern can be matched against the value N only if N is the value of the expression E. In this case, the pattern matching does not produce bindings.
- a wildcard pattern, **any**:T, that matches against any value of type T without producing bindings.
- a record pattern, (RP), where RP is a list of patterns separated by commas, either in a *named* form V_1 => P_1, ... ,V_n => P_n (where $V_1, \ldots, V_n$ are the field names of the record) or in a *positional* form P_1, ... ,P_n. The pattern matching of (RP) against the value N will succeed if N has the form (RN) and RN is a list of values (possible preceded by a field name) that match against the corresponding patterns in RP. This pattern matching produces the bindings produced by matching each pattern of RP against the corresponding value in RN. Patterns in RP must bind disjoint variables.

  For example, the record pattern (*real* => ?re,*imag* => ?im) matches against the value (*real* => 2.5,*imag* => 0.3) binding variable re with the value 2.5 and variable im with the value 0.3.

  In a record pattern RP we can use the keyword **etc** at the end of RP for representing all the unspecified fields. So, the pattern (*real* => ?re,**etc**) may be matched against the value (*real* => 2.5,*imag* => 0.3) producing a binding between the variable re and the value 2.5.
- a constructor application, C[(RP)], where RP stands for the arguments of the constructor C (if needed). The pattern C(RP) matches with the value N if N has the form C(RN) and RN matches with RP, and the produced bindings are those produced by the pattern matching of each pattern against the corresponding value. For example, *cons*(?x,?l) or *cons*(!3,**any**:List).
- an explicit typing, P:T. The pattern P:T matches against the value N if N has type T and P matches against N. For example, the pattern ?l:intlist matches with the value *cons*(4,*nil*).

Patterns are also used in the **case** operator, whose syntax is

```
case E[:T] is
    P_1[[E_1]] -> B_1
        ⋮
  | P_n[[E_n]] -> B_n
endcase
```

where expressions E_i are boolean expressions, which are *true* by default.

The value of the expression E is matched sequentially against each of the clauses P_1 [E_1], ..., P_n [E_n]. A value N matches against a clause P_i [E_i] if the value matches against the pattern P_i and E_i is evaluated to *true* in the context of variables bound by the pattern-matching. The behaviour B_i associated with the first clause that matches N describes how the whole behaviour continues. If there is no clause that matches N then the predefined exception *Match* is raised.

For example, we can define a process that receives messages of type pdu and behaves in a different manner depending on the kind of message:

```
process P [inP:pdu,... ] is
   var p:packet in
      inP(?p);
      case p is
          send(?pa:paquet,?b:bit) -> (* handle data *)
        | ack(?b:bit) -> (* handle acknowledgment *)
      endcase
   endvar
endproc
```

There is also the possibility of pattern-matching between two patterns, for example, in the communication

$$inP(?x)\ ||\ inP(!3)$$

In this case the pattern ?x is pattern-matched against the pattern !3. The pattern-matching between two patterns succeeds if there is a value N such that both patterns can be matched against N. In the example this value is 3.

Although strange, we can put together two behaviours that can receives values on a gate *inP*,

$$inP(?x)\ ||\ inP(?y)$$

In this case, a value of the type of values that G can communicate is "*invented*" and both x and y are bound to this value.

4 Module Language

When large specifications are developed, it is useful to encapsulate several related data types, functions, and processes so that they can be regarded as a single unit (a *module*) with which one can work. One can also want to combine different modules, to control what objects a module shows, and to build generic modules that are parameterized by other modules.

Due to the LOTOS limited form of modularity, whose modules only encapsulate types and operations but not processes, and do not support abstraction (every object declared in a module is exported outside)[1], E-LOTOS has a new modularization system, which allows

- to define a set of related objects (types, functions, and processes),
- to control what objects the module exports (by means of interfaces),
- to include within a module the objects declared in other modules (by means of import clauses),
- to hide the implementation of some objects (by means of opaque types, functions, and processes), and
- to build generic modules.

In order to facilitate this modularization, a separation between the concept of module *interface* and definition *module* is done. An interface declares the visible objects of a module and what the user needs to know about them (the name of a data type or a function profile, for example). A module gives the definition (or implementation) of objects (visible or not).

A specification in modular E-LOTOS is a sequence of interface (Section 4.1) and module (Section 4.2) or generic module (Section 4.3) declarations, besides a specification declaration (Section 4.4).

4.1 Interfaces

As we have said above, an interface defines the visible part of the objects (types, functions, and processes) declared within a module.

The syntax of an interface declaration is as follows:

interface *int-id* [**import** *int-exp*$_1$ **,** . . . **,** *int-exp*$_n$] **is**
  i-body
endint

where *int-id* is an interface identifier, *int-exp*$_1, \ldots,$ *int-exp*$_n$ are interface expressions that we will describe below, and *i-body* is the body of the interface.

In the body of an interface, the visible parts of types, functions, and processes are given. The visible part of a type is its name and, maybe, its implementation. If the implementation of a type is not given,

[1] A critical evaluation of LOTOS data types from the user point of view can be found in [Mun91].

the type is called *opaque*, and only the functions declared in the interface can modify it. An opaque type declaration is as follows:

$$\textbf{type } T$$

If the type is not opaque, it is declared as we saw in Section 3.1.

For functions and processes, only their header is visible. Thus, in an interface function headers like

$\textbf{function } F \quad [\ \texttt{(}\ [\textbf{in}|\textbf{out}]V_1\texttt{:}T'_1\texttt{,}\ \ldots\ \texttt{,}[\textbf{in}|\textbf{out}]V_m\texttt{:}T'_m\ \texttt{)}\ ]\ [\texttt{:}T]\ [\textbf{raises}\ \texttt{[}\ X_1[\texttt{:}T''_1]\texttt{,}\ \ldots\ \texttt{,}X_p[\texttt{:}T''_p]\ \texttt{]}]$

or processes headers like

$\textbf{process } \Pi \quad [\ \texttt{[}\ G_1[\texttt{:}T_1]\texttt{,}\ \ldots\ \texttt{,}G_n[\texttt{:}T_n]\ \texttt{]}\ ]\ [\ \texttt{(}\ [\textbf{in}|\textbf{out}]V_1\texttt{:}T'_1\texttt{,}\ \ldots\ \texttt{,}[\textbf{in}|\textbf{out}]V_m\texttt{:}T'_m\ \texttt{)}\ ]$
$[\textbf{raises}\ \texttt{[}\ X_1[\texttt{:}T''_1]\texttt{,}\ \ldots\ \texttt{,}X_p[\texttt{:}T''_p]\ \texttt{]}]$

may appear.

We can also declare *values*, which represent constant functions without parameters, with syntax

$\textbf{value } V\ \texttt{:}\ T$

where V is the value identifier, and T is a type identifier.

The types of formal parameters must be either predefined, imported, or declared by the interface.

For example, we can define an interface of a module that implements our register:

```
interface Register_Interface is
  type data
  process Register [in1:data,in2:data,out1:data,out2:data]
endint
```

As we have seen, an interface may import other interfaces, and the imported interfaces are specified by means of interface expressions. An interface expression may be

- an interface identifier *int-id′*, that represents all the identifiers declared in the interface *int-id′*; or
- a renaming of an interface

  [*int-id′* **renaming** (*reninst*)]

  representing the identifiers declared in the interface *int-id′* renamed by the renaming *reninst*, explained below; or
- an explicit interface body (possibly renamed)

  [*i-body* [**renaming** (*reninst*)]]

  that represents the identifiers declared in the interface body *i-body*.

In the last two cases *reninst* is a list of renaming of

- types, $\textbf{types}\ S'_1\texttt{:=}S_1\texttt{,}\ \ldots\ \texttt{,}S'_n\texttt{:=}S_n$,
- constructors and functions, $\textbf{opns}\ F'_1\texttt{:=}F_1\texttt{,}\ \ldots\ \texttt{,}F'_m\texttt{:=}F_m$,
- processes, $\textbf{opns}\ \Pi'_1\texttt{:=}\Pi_1\texttt{,}\ \ldots\ \texttt{,}\Pi'_p\texttt{:=}\Pi_p$, or
- values, $\textbf{values}\ V'_1\texttt{:=}V_1\texttt{,}\ \ldots\ \texttt{,}V'_s\texttt{:=}V_s$.

The primed identifiers are the old names (of *int-id′* or of *i-body*) being renamed, and the unprimed identifiers are the new names (which can be used in *int-id*). For example,

[Register_Interface **renaming** (**proc** Register := Register_Nat)] represents the declarations in the interface Register_Interface where the process Register now is named Register_Nat.

The imported identifiers have to be all different, so we have to use renaming when there are conflicts.

We can define an interface for our system composed by the register, the producer, and the consumer, as follows

```
interface System_Interface import Register_Interface is
  process Producer [p1:data,p2:data]
  process Consumer [c1:data,c2:data]
endint
```

All visible objects of an interface (including the imported ones) are visible from outside and may be imported in other interfaces. The importation is transitive through interfaces.

4.2 Modules

A module specifies the implementation of a set of (related) types, functions, and processes. The objects that a module exports, that is, that may be imported by other modules, are controlled by means of interfaces.

In E-LOTOS a module declaration is as follows:

```
module mod-id [: int-exp] [import mod-exp1, ... ,mod-expn] is
   m-body
endmod
```

where *mod-id* is a module identifier; *int-exp* is an interface expression (Section 4.1) that declares the visible objects of *mod-id*, so that other modules that import *mod-id* only can use the objects declared in *int-exp*[2]; *mod-exp*$_1$, . . . ,*mod-exp*$_n$ are module expressions that we will describe below; and *m-body* is the body of the module, which is a sequence of type declarations (Section 3.1), function declarations (Section 3.5), and process declarations (Section 2.20).

The objects imported by a module have to have all different identifiers, so renaming is needed when an identifier is used more than once.

For example, we can specify the module that implements a version of the register (a register that communicates natural numbers) as follows:

```
module Register_Mod: [Register_Interface renaming (proc Register := Register_Nat)] is
  type data renames nat endtype
  process Register_Nat [in1:data,in2:data,out1:data,out2:data] is
    var x1:data,x2:data in
    in1(?x1 : data);
      ( in2(?x2 : data)
      |||
      out1(!x1) );
      out2(!x2)
    endvar
  endproc
endmod
```

[2] By default, the interface of a module is the set of objects imported from other modules and the objects declared by the module body.

A module expression may be a module identifier possibly restricted by an interface expression and renamed:

mod-id [: *int-exp*] [**renaming** (*reninst*)]

or an instantiation of a generic module as we will see in Section 4.3.

The module that implements the objects declared in System_Interface may be

```
module System_Mod import Register_Mod is
  value val1:data is 8 endval
  value val2:data is 15 endval
  process Producer [p1:data,p2:data] is
      p1(!val1); p2(!val2)
  endproc
  process Consumer [c1:data,c2:data] is
    var v1 : data,v2 : data in
      c1(?v1 : data); c2(?v2 : data)
    endvar
  endproc
endmod
```

4.3 Generic Modules

Genericity is a useful mechanism to construct specifications and for code reuse. In E-LOTOS, a generic module declaration is as follows:

```
generic gen-id ( mod-id_1:int-exp_1, ... ,mod-id_n:int-exp_n ) [: int-exp]
 [import mod-exp_1, ... ,mod-exp_m] is
  m-body
endgen
```

where *gen-id* is a generic module identifier, *mod-id*$_i$ are module identifiers, which are called the formal parameters of the generic module, and whose visible objects are declared in the interface expressions *int-exp*$_i$; *mod-exp*$_j$ are module expressions as defined in Section 4.2, and *m-body* is the body of the generic module, where the objects declared in the *int-exp*$_i$, and the imported ones, may be used.

For example, instead of implementing a different register depending on the values that the register can communicate, we can define a generic module that implements a *generic* register parameterized by the type of the values the register can communicate:

```
interface Data is
  type data
endint
generic Register_Gen (D:Data) is
  process Register [in1:data,in2:data,out1:data,out2:data] is
    var x1 : data,x2 : data in
      in1(?x1 : data);
      ( in2(?x2 : data)
      |||
      out1(!x1) );
      out2(!x2)
    endvar
  endproc
endgen
```

In order to use a generic module we have to instantiate it, by providing actual parameters, which must be modules that match the corresponding interface expression. A module matches an interface whether it implements at least the objects declared in the interface.

The syntax of a generic module instantiation is as follows:

gen-id `(` *mod-id*$_1$ `=>` *mod-exp*$_1$ `,` ... `,` *mod-id*$_n$ `=>` *mod-exp*$_n$ `)` [`:` *int-exp*] [**renaming** `(` *reninst* `)`]

where *mod-id*$_i$ are module identifiers that must be the names of the formal parameters of *gen-id*, *mod-exp*$_i$ are module expressions that must match the corresponding interface expressions in the declaration of *gen-id*; *int-exp* is an interface expression and *reninst* is a renaming clause as we saw in Section 4.1.

A generic module instantiation is a module expression, and may be the body of a module, or may appear in an importation clause.

We can instantiate our generic register, in order to build a register of natural numbers:

```
module Mod_Register_Nat is
  Register_Gen (D => NaturalNumbers renaming (types nat := data))
    renaming (proc Register := Register_Nat)
endmod
```

4.4 Specification

The entry point of an E-LOTOS description is the *specification* declaration, with syntax

[*top-dec*]
specification Σ [**import** *mod-exp*$_1$ `,` ... `,` *mod-exp*$_n$] **is**
  [**gates** $G_1 : T_1$ `,` ... `,` $G_m : T_m$]
  [**exceptions** $X_1 : T'_1$ `,` ... `,` $X_p : T'_p$]
  (**behaviour** B | **value** E)
endspec

where

- *top-dec* is a sequence of interfaces, modules, and generic modules declarations;
- Σ is the name of the specification;
- *mod-exp*$_i$ are module expressions that define the imported modules;
- $G_1 : T_1, \ldots, G_m : T_m$ is the list of gates of the whole system;
- $X_1 : T'_1, \ldots, X_p : T'_p$ is the list of exceptions that the system can raise to its environment; and
- the body of the specification may be a behaviour B, or a value E.

For example we can define the specification of the system composed of a producer, a register, and a consumer:

```
specification System import System_Mod is
  behaviour
    hide pr1:data,pr2:data,rc1:data,rc2:data in
      conc
        Producer [pr1, pr2] ()
      |[pr1, pr2]|
        Register_Nat [pr1, pr2, rc1, rc2] ()
      |[rc1, rc2]|
        Consumer [rc1, rc2]()
      endconc
    endhide
endspec
```

5 Examples

In this chapter we will discuss some well-known examples where the main E-LOTOS features are used.

5.1 Global clock

In this section we are going to specify a *global clock*, that is, a clock that measures time since it is started until it is stopped, and to which other processes running in parallel can ask what time it is. The clock is always able to communicate what time it is, and to be stopped. The process Clock is as follows:

```
process Clock[stopClock,whatTime:time](gtime:time) is
  var gt:time:=0,t:time:=0 in
      whatTime(?gt) @?t [gt=gtime+t];Clock[...](gtime+t)
    []
      stopClock
  endvar
endproc
```

We use the parameter *gtime* to measure the global time. Initially, it is set to 0 (in the initial call), and then its value always represents the time when the last question "What time is it?" was answered. When another question is asked, the clock performs the action

$$whatTime(?gt)\ @?t\ [gt{=}gtime{+}t]$$

where *gt* represents the value communicated, and although it is like an input (?*gt*), really it is the unique value such that $gt = gtime + t$ where t is the time measured since the action was enabled, that is, since the last question was answered. Therefore, *gt* represents the global time (new value for *gtime*).

At any moment, the clock can be stopped with action *stopClock*. This is necessary if the clock is running in parallel with a process that finishes and we want the whole system (the process and the clock) to finish.

5.2 FIFO queue

Now, we are going to specify a generic FIFO (*first-in, first-out*) queue, which we will use in Section 5.4.

First, we specify the characteristics that the elements of the queue must fulfill in an interface:

```
interface Data is
  type elem
endint
```

Now, we specify the generic module that describes the queue operations.

```
generic GenQueue(D:Data) is
  type queue is
      Empty
    | Add(queue,elem)
  endtype
  function addQueue(q:queue,e:elem):queue is
    Add(q,e)
  endfunc
  function front(q:queue):elem raises [EmptyQueue] is
    var e:elem is
      e:=any elem;
      case q in
          Empty -> signal EmptyQueue; e
        | Add(Empty,?e) -> e
        | Add(Add(?q,?e),any:elem) -> front(Add(q,e))
      endcase
    endvar
  endfunc
  function delete(q:queue):queue raises [EmptyQueue] is
    var e1:elem,e2:elem is
      case q in
          Empty -> signal EmptyQueue; Empty
        | Add(Empty,any:elem) -> Empty
        | Add(Add(?q,?e1),?e2) -> Add(delete(Add(q,e1)),e2)
      endcase
    endvar
  endfunc
  function isEmpty(q:queue):bool is
    case q in
        Empty -> true
      | Add(any:queue,any:elem) -> false
    endcase
  endfunc
endgen
```

And now we can instantiate this generic queue to make, for example, a queue of natural numbers:

```
module NatQueue is
  GenQueue(D => NaturalNumbers renaming(types nat := elem))
endmod
```

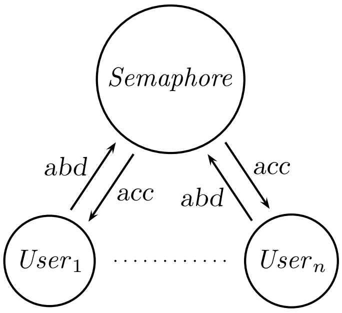
5.3 Random semaphore

We study now the classical example of the *critical section problem*, which is one of the classic concurrent programming problems. Let us suppose that a set of n processes (*users*) have to access a shared resource in mutual exclusion, i.e, the resource can be used by at most one user at a time. So the behaviour of each user should be

1. non-critical section
2. entry protocol

3. *critical section*
4. exit protocol
5. go to 1.

In order to ensure mutual exclusion, we introduce a *semaphore* process which all the users have to synchronize with, before and after entering the critical section. In this first attempt, the semaphore allows to enter the critical section to *any* of the processes that are waiting, provided none is already in. So the semaphore allows one of those users to enter the critical section and then that user notifies its exit of the critical section. The structure of the whole system could be



We can model a user as a process that continuously behaves as described above. All the users notify its exit through the same gate, and the semaphore uses the same gate to allow access to every user, so in order to distinguish users each one will have a different identifier (implemented as a parameter). A user has to provide its identifier each time it wants to access or exit the resource. An E-LOTOS specification of a user may be:

```
process User1 [acc:id,abd:id](myid:id) is
  loop
    (* non-critical section *)
    acc(!myid);
    (* use shared resource *)
    abd(!myid)
  endloop
endproc
```

where type id may be a synonym of the predefined nat type:

```
type id renames
  nat
endtype
```

We can also specify the infinite behaviour of the user with a recursive process:

```
process User2 [acc:id,abd:id](myid:id) is
  (* non-critical section *)
  acc(!myid);
  (* use shared resource *)
  abd(!myid);
  User2 [acc,abd](myid)
endproc
```

As another example, we can modify this process by making explicit the state of the user, i.e, whether it is in or out the critical section. We could write:

```
process User3 [acc:id,abd:id](myid:id) is
  var st:state := outside in
    loop
      case st is
          outside -> (* non-critical section *)
                     acc(!myid);
                     ?st := inside
        | inside -> (* use shared resource *)
                    abd(!myid);
                    ?st := outside
      endcase
    endloop
  endvar
endproc
```

where we have used a new type, state, that has two different values, *inside* and *outside*, declared as

```
type state is
   inside | outside
endtype
```

The Semaphore process uses the same gates as users do. When the resource is free, the semaphore waits for someone who wants to enter, and keeps in variable *usr* who has entered. Then, the resource is not free and the semaphore allows the *in* user (*usr*) to exit. When this user does so, the semaphore repeats its behaviour. The specification of this process is as follows:

```
process Semaphore[acc:id,abd:id] is
  var usr:nat in
    acc(?usr);
    abd(!usr);
    Semaphore[acc,abd]()
  endvar
endproc
```

We can model the entire system by instantiating several times the User1 process (each one with a different identifier) and composing them in parallel with an instantiation of the Semaphore process. We hide the gates that stand for synchronization between the users and the semaphore, in order to make them urgent and because they are of no interest from the point of view of the environment. So the behaviour of the system can be specified with:

```
hide acc:id,abd:id in
  par ?usr in [1,2,3,4,5] |||
    User1[acc,abd](usr)
  endpar
||
  Semaphore[acc,abd]
endhide
```

And the complete E-LOTOS specification would be:

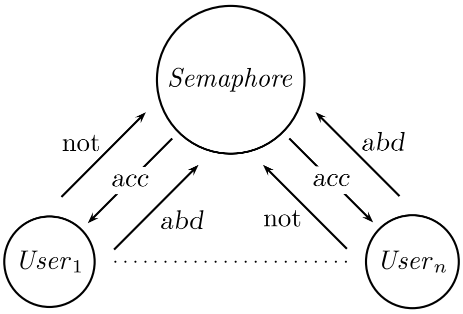
```
module CriticalSection1 is
  type id renames
    nat
  endtype
  process User1 [acc:id,abd:id](myid:id) is
    loop
      (* non-critical section *)
      acc(!myid);
      (* use shared resource *)
      abd(!myid)
    endloop
  endproc
  process Semaphore[acc:id,abd:id] is
    var usr:nat in
      acc(?usr);
      abd(!usr);
      Semaphore[acc,abd]()
    endvar
  endproc
endmod

specification RandomSemaphore import CriticalSection1 is
behaviour
  hide acc:id,abd:id in
    par ?usr in [1,2,3,4,5] |||
      User1[acc,abd](usr)
    endpar
  ||
    Semaphore[acc,abd]
  endhide
endspec
```

In this example, when there are several users waiting to access the shared resource, it is not known which of them will succeed. From this point of view, the semaphore is *random*. Any of the waiting users can be the next to access the resource. In the next section the semaphore will take care of the order in which the waiting users access the resource.

5.4 FIFO semaphore

In this section we discuss the same problem as in the last section, but in this case the waiting users will be served in the order they arrived to the critical section. In order to achieve this, each user has to notify its intention to access the critical section. So, in this case there is a new communication between the users and the semaphore:



When a user notifies to the semaphore its wish to access the critical section, the semaphore keeps the user identifier in a queue of identifiers. When the resource is free, the semaphore allows access to the user whose identifier is in the front of the queue. So, we need a queue of identifiers. We can use the generic queue of Section 5.2, by doing the following instantiation:

```
module NatQueue is
  GenQueue(D => DataTypes renaming(types id := elem))
endmod
```

where DataTypes is the module that specifies the new data types required in this example, and id should be among them.

We have to modify also the User2 process in order to notify to the semaphore its intention to access the resource:

```
process User4 [not:id,acc:id,abd:id](myid:id) is
   (* non-critical section *)
   not(!myid);acc(!myid);
   (* use shared resource *)
   abd(!myid);
   User4 [...](myid)
endproc
```

The semaphore has to allow always a user notification, and it has to insert the user identifier in the queue; if the resource is free and there is some user waiting, then it has to allow access to the user in the front of the queue; and if the resource is not free, it has to wait for the in user to go out. The semaphore specification in this case is:

```
process Semaphore[not:id,acc:id,abd:id](free:bool,q:queue,usrIn:nat) is
  var usr:nat:=0 in
    trap
      exception EmpQ is stop endexn
    in
      sel
          not(?usr);Semaphore[...](free,addQueue(q,usr),usrIn)
        []
          acc(!front(q)[EmpQ]) [free and not(isEmpty(q))];
          Semaphore[...](false,q,front(q)[EmpQ])
        []
          abd(!usrIn)[not(free)];Semaphore[...](true,delete(q)[EmpQ],0)
      endsel
    endtrap
  endvar
endproc
```

In this case, we have used a different kind of specification than in, for example, User3. We have used a recursive process and, instead of using several **if** – **then** – **else** instructions, we use a selection where each branch begins with an action with a selection predicate that specifies when it is possible to execute this branch. For example, the first branch begins with the action *not*(?*usr*) (with default selection predicate [*true*]), that is, it is always possible to take a notification from a user. The second branch begins with

acc(!front(*q*)[*EmpQ*]) [*free* and not(isEmpty(*q*))]

that is, the user who is the first of the queue can access the resource provided that the resource is free and there are users waiting. In fact, the condition not(isEmpty(*q*)) is not necessary, without care of an exception being raised. If q is empty, the pattern !front(q)[$EmpQ$] fails and it cannot match against any value, so the action is not offered.

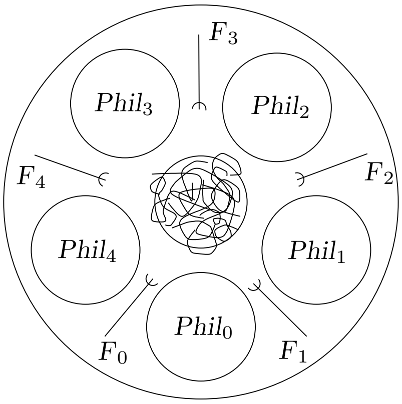
Finally, in the specification of the entire system, the only needed modification is the inclusion of the new gate *not*:

```
specification RandomSemaphore import CriticalSection2 is
behaviour
  hide not:id,acc:id,abd:id in
    par ?usr in [1,2,3,4,5] |||
      User4[...](usr)
    endpar
  ||
    Semaphore[...](true,Empty,0)
  endhide
endspec
```

where CriticalSection2 is the module that includes the declaration of types and processes used.

5.5 Dining philosophers

This problem, originally stated and solved by E.W. Dijkstra [Dij65], is set in a monastery where five monks are dedicated philosophers. Each philosopher has a room in which he can engage in thinking. There is also a common dining room, with a circular table with five plates, each labeled by the name of the philosopher who uses it, as the following figure shows:



To the left of each philosopher there is laid a fork, and in the center stands a large bowl of spaghetti, which is constantly replenished. A philosopher is expected to spend most of his time thinking, but when he feels hungry, he goes to the dining room, takes a seat, eats, and then returns to his room to think. However, the spaghetti is so entangled that two forks are needed simultaneously in order to eat.

The problem is to devise a *ritual* (protocol) that will allow the philosophers to eat. Each philosopher may use only the two forks adjacent to his plate. The protocol must satisfy the following requirements:

- mutual exclusion, that is, two philosophers cannot use the same fork simultaneously;
- freedom from deadlock and lockout, that is, absence of starvation – literally!.

Our first solution[1] consists in the philosophers having to ask for the forks in order to take them. In this way, each fork controls that only one philosopher (one that is adjacent) takes it at any time. The process that specifies the behaviour of a philosopher may be:

```
process Philosopher[sits_down,picks_up,puts_down,gets_up](id:nat) is
  (* think *)
  sits_down(!id);
  picks_up(!(id,id));  picks_up(!(id,right(id)));
  (* eat *)
  puts_down(!id);
  puts_down(!(id,right(id)));
  gets_up(!id);
  Philosopher[...](id)
endproc
```

The process is parameterized by the identifier of the philosopher that represents. When the philosopher sits down, he tries to pick up the fork on this left, picks_up(!(*id*,*id*)) (where the second value represents the number of the fork which is picked up), and when he has picked up it, he tries to pick up the fork in his right (right(*id*) represents the identifier of the fork which is on the right of the philosopher *id*). When the philosopher has two forks he eats, and then he puts down the forks and gets up.

The process that specifies the behaviour of the forks would be:

```
process Fork[picks_up,puts_down](id:nat) is
    picks_up(!(id,id));puts_down(!(id,id));Fork[...](id)
    []
    picks_up(!(left(id),id));puts_down(!(left(id),id));Fork[...](id)
endproc
```

That is, a fork allows to be picked up by the philosopher with the same identifier or by the philosopher who is on its left (left(*id*)).

And the behaviour of the dining room with five philosophers and five forks may be specified as follows:

```
hide picks_up,puts_down in
    par ?phi in[1,2,3,4,5] |||
      Philosopher[...](phi)
    endpar
  |[ picks_up,puts_down  ]|
    par ?fo in[1,2,3,4,5]  |||
      Fork[...](fo)
    endpar
endhide
```

But this solution may become deadlocked, for example, if all the philosophers get hungry at the same time, they all sit down, pick up their left fork, and try to pick up the other fork, which is not free.

The problem may be solved by adding a *footman* (which acts as the semaphore in the previous example) whose permission the philosophers have to ask for sitting down, and to whom they must communicate that they get up. The footman never allows more than four philosophers to be seated simultaneously. His behaviour may be specified with the following E-LOTOS process:

[1] As we will see below, this is not really a good solution.

```
process Footman[sits_down,gets_up](seated:nat) is
    sits_down(?id)[seated<4];Footman[...](seated + 1)
  []
    gets_up(?id);Footman[...](seated − 1)
endproc
```

Now, we have to add the footman to the whole system:

```
hide sits_down,picks_up,puts_down,gets_up in
    ( par ?phi in [1,2,3,4,5] |||
        Philosopher[...](phi)
      endpar
  |[ picks_up,puts_down ]|
      par ?fo in [1,2,3,4,5] |||
      Fork[...](fo)
      endpar )
  |[sits_down,gets_up ]|
    Footman[...](0)
endhide
```

This solution is free of deadlock, as explained in [Hoa85].

5.6 Readers and writers

This is again a mutual exclusion problem, but now there are two kinds of processes, readers and writers, sharing a resource, for example a database. Readers only examine information of the database, whereas writers modify it. A writer must have exclusive access to the database, otherwise the information could be corrupted. But any number of readers can access the database at a time, provided that there is no writer modifying it.

In this solution, there will be a manager that controls the right access to the database. In a first attempt, readers and writers have to wait for the manager's permission to access the database; when they have it, they can access the database, and then they have to notify their going out to the manager. We can specify the readers and writers behaviours in the following way:

```
process Reader1 [accR:id,abdR:id](myid:id) is
  (* other things *)
  accR(!myid);
  (* read *)
  abdR(!myid);
  Reader1 [...](myid)
endproc

process Writer1 [accW:id,abdW:id](myid:id) is
  (* other things *)
  accW(!myid);
  (* write *)
  abdW(!myid);
  Writer1 [...](myid)
endproc
```

The manager has a boolean parameter, *writing*, that controls whether there is a writer updating the database, and an integer parameter, *readers*, that controls how many readers are reading the database.

When a reader wants to access it, the manager will give permission if there is no writer. If a writer wants to access it, the manager will give permission only if there is no writer and no reader. The specification of the manager is:

```
process Manager1[accR:id,abdR:id,accW:id,abdW:id] (writing:bool,readers:nat)is
  var r:nat:=0,w:nat:=0 in
    sel
        accR(?r)[not(writing)];Manager1[...](false,readers + 1)
      []
        accW(?w)[not(writing) and (readers = 0)];Manager1[...](true,readers)
      []
        abdR(?r);Manager1[...](writing,readers − 1)
      []
        abdW(?w);Manager1[...](false,readers)
    endsel
  endvar
endproc
```

The complete system with 8 readers and 2 writers is specified as follows:

```
specification ReadersWriters1 import ReadWriteMod is
behaviour
  hide accR:id,accW:id,abdR:id,abdW:id in
      ( par ?r in [1,2,3,4,5,6,7,8] |||
          Reader1[accR,abdR](r)
        endpar
      |||
        par ?w in [1,2] |||
          Writer1[accW,abdW](w)
        endpar )
    ||
      Manager1[accR,accW,abdR,abdW](false,0)
  endhide
endspec
```

where ReadWriteMod is the module that contains the declarations of the processes Reader1, Writer1, and Manager1.

This solution suffers from the *starvation problem*. Once the readers begin using the database, they can monopolize it, without allowing any writer to access it. We can solve this problem if the manager does not allow new reader accesses when there is a writer waiting to update the database. In this way, readers will be finishing (*readers* will become 0) and the writer will be able to access.

In order to introduce this new idea, the writers have to notify their interest of accessing the database, so a new gate, *notW*, that allows communication between writers and the manager has to be included. The manager will have a new parameter, *wwriters*, to count how many writers are waiting. The specification of the readers is as before, but writers and the manager have to be modified as follows:

```
process Writer2 [notW:id,accW:id,abdW:id](myid:id) is
  (* other things *)
  notW(!myid); accW(!myid);
  (* write *)
  abdW(!myid); Writer2 [...](myid)
endproc
```

```
process Manager2[accR:id,abdR:id,notW:id,accW:id,abdW:id]
                (writing:bool,readers:nat,wwriters:nat) is
  var r:nat:=0,w:nat:=0 in
    sel
        accR(?r)[not(writing) and (wwriters=0)];
        Manager2[...](writing, readers + 1, wwriters)
      []
        notW(?w); Manager2[...](writing, readers, wwriters + 1)
      []
        accW(?w)[not(writing) and (readers=0)];
        Manager2[...](true, readers, wwriters − 1)
      []
        abdR(?r)[readers>0]; Manager2[...](writing, readers − 1, wwriters)
      []
        abdW(?w)[writing]; Manager2[...](false, readers, wwriters)
    endsel
  endvar
endproc
```

where this process will be called from the specification as

$$\text{Manager2[...]}(\mathit{false}, 0, 0).$$

This second attempt has introduced the reverse problem: now writers can be always accessing the database, not allowing readers to use it. We can solve this problem if the manager stops the notifications of new writers once it knows there are readers waiting. The writers that have already notified their intention can access the database sequentially, but no new writers can notify. In this way the parameter *wwriters* will become 0 and at least one reader will be able to access.

In this attempt, a new gate, *notR*, will be included, through which readers can notify to the manager their wish to access the database. And the manager has to count how many readers are waiting (parameter *wreaders*). The specification of the reader has to be changed to

```
process Reader3 [notR:id,accR:id,abdR:id](myid:id) is
  (* other things *)
  notR(!myid); accR(!myid);
  (* read *)
  abdR(!myid) Reader3 [...](myid)
endproc
```

The writer is the same as in the second attempt, and the new manager is:

```
process Manager3[notR:id,accR:id,abdR:id,notW:id,accW:id,abdW:id]
                (writing:bool,readers:nat,wwriters:nat,wreaders:nat) is
  var r:nat:=0,w:nat:=0 in
    sel
        notR(?r); Manager3[...](writing, readers, wwriters, wreaders + 1)
      []
        accR(?r)[not(writing) and (wwriters = 0)];
        Manager3[...](writing, readers + 1, wwriters, wreaders − 1)
      []
        notW(?w)[readers > 0 or (wreaders = 0)];
        Manager3[...](writing, readers, wwriters + 1, wreaders)
      []
        accW(?w)[not(writing) and (readers = 0)];
        Manager3[...](true, readers, wwriters − 1, wreaders)
      []
        abdR(?r); Manager3[...](writing, readers − 1, wwriters, wreaders)
      []
        abdW(?w); Manager3[...](false, readers, wwriters, wreaders)
    endsel
  endvar
endproc
```

The manager could be modified not only to know how many readers and writers are waiting, but *who* of them are waiting (as we did in Section 5.4, by means of a queue in a semaphore), so it can control the order in which they are served.

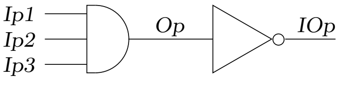
5.7 Specifying digital logic

In this section, we show how we can deal with the specification of digital logic components in E-LOTOS. It is entirely based on the work [TS94] by K. J. Turner and Richard O. Sinnott, and the subsequent work [JT97], by Ji He and K. J. Turner. There, the specification and validation of digital logic components and circuits using LOTOS are addressed, with the philosophy that it should be easy for the hardware engineer to translate a circuit schematic into a LOTOS specification, and then to analyze and verify the properties of this specification. Because of this, a component library (DILL) is introduced.

Here, our aim is simply to show how these jobs can be dealt in E-LOTOS by means of an example: a specification of a full adder.

First, we introduce some basic ideas. Digital signals are going to be modeled as two-level voltages, specified as constants of the E-LOTOS data type Bit. The constant *bit1* represents logic 1, constant *bit0* represents logic 0, and constant *bitX* represents an unknown, arbitrary or “do not care” value, used as the initial state of every signal (both inputs and outputs). This data type is specified in a module we will see below, with the definition of several logical operations on signals.

Each basic logic gate (*Inverter*, *And*, *Or*, *XOr*, ...) is modeled as an E-LOTOS process. For example, And3[*Ip1*, *Ip2*, *Ip3*, *Op*] is an E-LOTOS specification of an *And* gate with three inputs. An E-LOTOS gate (for example, *Ip1*) models a physical wire or pin, and an E-LOTOS action (for example, *Ip1*(!*bit1*)) models a signal change on the wire (here, for logic 0 to logic 1). Larger circuits can be built from basic logic gates using E-LOTOS parallel behaviours. For example, an And3 gate followed by an Inverter,



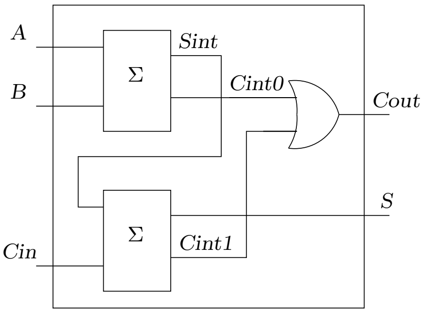
can be modeled as:

```
And3[Ip1,Ip2,Ip3,Op]() |[Op]| Inverter[Op,IOp]()
```

Thus, connecting several wires and pins is modeled as synchronization at E-LOTOS gates. We package the specification of a circuit into an E-LOTOS process in order to reuse it. The E-LOTOS gates of the process are the inputs and outputs of the circuit, and all the other E-LOTOS gates are hidden. For example, the circuit above can be specified as:

```
process And3Inverter [Ip1:Bit,Ip2:Bit,Ip3:Bit,IOp:Bit] is
  hide Op:Bit in
      And3[Ip1,Ip2,Ip3,Op]()
    |[Op]|
      Inverter[ Op,IOp ] ()
  endhide
endproc
```

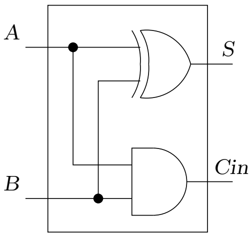
After this introduction, now we are going to specify a full-adder, as explained in [Flo94]. A full-adder accepts three inputs including an input carry and generates a sum output and an output carry. A full-adder can be built from two half-adders and an Or2 gate, as the following logic diagram shows:



Thus, we can specify a process Full-Adder as follows

```
process Full-Adder[ A,B,Cin,S,Cout ] is
  hide Sint:Bit,Cint0:Bit,Cint1:Bit in
      (
        Half-Adder[ A,B,Sint,Cint0 ] ()
      |[Sint]|
        Half-Adder[ Sint,Cin,S,Cint1 ] ()
      )
    |[Cint0,Cint1]|
      Or2[ Cint0,Cint1,Cout ] ()
  endhide
endproc
```

A half-adder accepts two binary digits on its inputs and produces two binary digits on its outputs, a sum bit and a carry bit. A half-adder can be built from an And2 gate and a XOr2 gate, connected in the following way:



We can specify it in E-LOTOS with the following process:

```
process Half-Adder [A:Bit,B:Bit,S:Bit,Cout:Bit] is
    Xor2[ A,B,S ] ()
  |[A,B]|
    And2[ A,B,Cout ] ()
endproc
```

We have reached the logic gate level. We have to specify now the logic gates Or2, XOr2, and And2. Like it is done in [TS94, JT97], we can specify a process that implements logic gates with two inputs and which is parameterized by the binary logical operation that it must implement. The process is Logic2:

```
process Logic2[Ip1:Bit,Ip2:Bit,Op:Bit](bOp:BitOp) is
  var bIn1:Bit:=bitX, bIn2:Bit:=bitX, bOut:Bit:=bitX, bOutNew:Bit:=bitX in
    loop
      sel
          Ip1(?bIn1)
        []
          Ip2(?bIn2)
        []
          ?bOutNew:=Apply2(bOp,bIn1,bIn2);
          sel
            Op(?bOut2) [(bOutNew = bitX) and (bOut = bitX) and(bOut2 <> bitX)];
            ?bOut:=bOut2
          []
            Op(!bOutNew) [(bOutNew <> bitX) and (bOutNew <> bOut)];
            ?bOut:=bOutNew
          endsel
      endsel
    endloop
  endvar
endproc
```

Variables $bIn1$, $bIn2$, and *bOut* save the state of the pins. The first two branches of the outer selection describe the possibility of new inputs. If the value of an input pin changes, it is saved in the corresponding variable. The third branch describes the output. Variable *bOutNew* saves the value of the application of the binary logical operation *bOp* to the values of the inputs. The second branch of the inner selection describe the case when the inputs are known (that is, they are not *bitX*) and then the output is also known (*bOutNew*<>*bitX*), and there is a change in the output (*bOutNew*<>*bOut*). The first branch describes the case when the output is unknown, but the output pin takes a known value from outside, for example, because there is feedback.

And now, by instantiating this process we can specify the needed logic gates:

```
process Or2[Ip1:Bit,Ip2:Bit,Op:Bit] is
  Logic2[Ip1,Ip2,Op](orOp)
endproc
process XOr2[Ip1:Bit,Ip2:Bit,Op:Bit] is
  Logic2[Ip1,Ip2,Op](xorOp)
endproc
process And2[Ip1:Bit,Ip2:Bit,Op:Bit] is
  Logic2[Ip1,Ip2,Op](andOp)
endproc
```

Finally, we have to specify the Bit data type, with the logical operations.

```
module Bit_Mod is
type Bit is
  bit1 |bit0 | bitX
endtype
```

```
function not(b:Bit):Bit Bis
  case b is
      bit0 -> bit1
    | bitX -> bitX
    | bit1 -> bit0
  endcase
endfun

function or(b1:Bit,b2:Bit):Bit is
  case (b1,b2) is
      (?b:Bit,bit0) -> b
    | (bit0,bitX) -> bitX
    | (bitX,bitX) -> bitX
    | (bit1,bitX) -> bit1
    | (?b:Bit,bit1) -> bit1
  endcase
endfun

function and(b1:Bit,b2:Bit):Bit is
  case (b1,b2) is
      (?b:Bit,bit0) -> bit0
    | (bit0,bitX) -> bit0
    | (bitX,bitX) -> bitX
    | (bit1,bitX) -> bitX
    | (?b:Bit,bit1) -> b
  endcase
endfun

function xor(b1:Bit,b2:Bit):Bit is
  case (b1,b2) is
    (?b:Bit,bit0) -> b
    | (?b:Bit,bitX) -> bitX
    | (?b:Bit,bit1) -> not(b)
  endcase
endfun

type BitOp is
  orOp | andOp | xorOp
endtype

function Apply2(bOp:BitOp,b1:Bit,b2:Bit):Bit is
  case bOp is
      orOp -> or(b1,b2)
    | andOp -> and(b1,b2)
    | xorOp -> xor(b1,b2)
  endcase
endfun
endmod
```

Part II

Semantics

In this part we define and explain the E-LOTOS semantics. In Chapter 6 we give the translation from the concrete syntax, used to write the specifications, to the abstract syntax, used to define the semantics.

In Chapter 7 we define the static semantics of the Base Language, that defines which E-LOTOS terms (in the abstract syntax) are semantically correct, and in Chapter 8 we show the dynamic semantics, that defines how an E-LOTOS specification can evolve. In Chapter 9 we show the semantics of a simple module system.

We have followed the semantics introduced in [Que98] as much as possible. However, [Que98] is a draft and is still being revised. From our point of view, it suffers from some problems which are being discussed. In this report we give our proposed solution to these problems.

6 Translation to Abstract Syntax

In this chapter we give the translation from concrete syntax (syntax used by the specifier, that is, the User Language we have seen in the previous chapters) to abstract syntax (syntax used to define the semantics of E-LOTOS).

6.1 Introduction

The aims of the translation to abstract syntax are:

- remove *syntactic sugar*, that is, remove constructs that are non-primitive in the sense that they can be expressed by using other constructs (which are primitives).
- unify functions and processes, by translating functions to processes where a return type is added (see Section 6.12).
- unify expressions and behaviours, by translating those expressions that do not have a direct translation to behaviours. For example, the expression

  ```
  trap
      exception Hd0 is 0 endexn
  in
      Head (xs) [Hd0]
  endtrap
  ```

  has a direct translation to a behaviour (translating the value 0 and the function instantiation), but

  E_1 **andalso** E_2

  has not (see Section 6.14).
- remove **out** parameters, by translating both process declarations and process instantiations.
- add default values in constructs with optional parameters.
- add new constructs and write annotations in some constructs in order to define in an easier way the semantics.

In the following sections we define how constructs that are not trivially translated to abstract syntax are translated. The constructs that are not present in the following sections are translated removing (if any) the final keywords as **endtrap**, **endhide**, etc; and adding default values for optional parameters not present.

The translation rules are given in the form

$$t_1 \quad \overset{\mathsf{abs}}{=} \quad t_2$$

meaning that the E-LOTOS term t_1 is translated to the E-LOTOS term t_2. Sometimes t_2 is not in abstract syntax (because we can use another rule to translate it), so it is assumed that an E-LOTOS term is translated by applying these rules until there are no more rules we can apply.

Bracketed syntax as **sel** B_1 `[]` B_2 `[]` ... `[]` B_n **endsel** are translated to B_1 `[]` $(B_2$ `[]` ... `[]` $B_n)$. An abstract syntax term is a *syntactic tree* and we do not need parentheses. Since we do not draw trees we will sometimes use parentheses () to ease the understanding of the abstract syntax terms but they have no meaning.

Records

In the abstract syntax, all record terms: record types RT, record patterns RP, and record of values RN, are named, that is, each field is preceded by a label. Thus, positional record terms have to be translated as follows:

$$T_1\texttt{,}\ldots\texttt{,}T_n \overset{\mathsf{abs}}{=} \$1 \texttt{=>} T_1\texttt{,}\ldots\texttt{,}\$n \texttt{=>} T_n$$

$$P_1\texttt{,}\ldots\texttt{,}P_m \overset{\mathsf{abs}}{=} \$1 \texttt{=>} P_1\texttt{,}\ldots\texttt{,}\$m \texttt{=>} P_m$$

$$N_1\texttt{,}\ldots\texttt{,}N_p \overset{\mathsf{abs}}{=} \$1 \texttt{=>} N_1\texttt{,}\ldots\texttt{,}\$p \texttt{=>} N_p$$

where $\$1, \$2, \ldots$ are the "invented" labels. The abstract syntax uses identifiers beginning with \$ when they are introduced by this translation. This is done in order to do not have conflicts with the user identifiers.

Abstract syntax context

A context $\mathcal{S}$ is used to translate process declarations and process instantiations. It has the following grammar:

$$\begin{array}{lcll r}
\mathcal{S} & ::= & \Pi \Rightarrow Pos & \textit{positional information} & (\mathcal{S}1) \\
 & \mid & \Pi \Rightarrow \textbf{gates}(\texttt{[}G_1,\ldots,G_n\texttt{]}) & \textit{formal gate list} & (\mathcal{S}2) \\
 & \mid & \Pi \Rightarrow \textbf{params}(\texttt{[}V_1,\ldots,V_p\texttt{]}) & \textit{formal parameter list} & (\mathcal{S}3) \\
 & \mid & \Pi \Rightarrow \textbf{exceptions}(\texttt{[}X_1,\ldots,X_m\texttt{]}) & \textit{formal exception list} & (\mathcal{S}4) \\
 & \mid & \mathcal{S},\mathcal{S} & \textit{union} & (\mathcal{S}5)
\end{array}$$

- $\Pi \Rightarrow Pos$ means that Pos is the set of positions of **out** parameters in the process Π, needed to remove these parameters in a process instantiation.
- $\Pi \Rightarrow$ **gates**(`[`$G_1,\ldots,G_n$`]`) means that $G_1,\ldots,G_n$ are the names of the formal gates in process Π. It is needed when the abbreviated gate list **...** is used.
- $\Pi \Rightarrow$ **params**(`[`$V_1,\ldots,V_p$`]`) means that $V_1,\ldots,V_p$ are the names of the formal parameters.
- $\Pi \Rightarrow$ **exceptions**(`[`$X_1,\ldots,X_m$`]`) means that $X_1,\ldots,X_m$ are the formal exceptions identifiers.

6.2 Actions

The abstract syntax for actions in E-LOTOS is

$$G\ \texttt{(}\$1\,\texttt{=>}\,P_1\texttt{)}\ \texttt{@}P_2\ E\ \textbf{start}\langle N\rangle.$$

That is, there are not optional parameters, and the clause $\textbf{start}\langle N\rangle$ has been added. This annotation represents the time since the action was enabled, so N must be a value of the data type time.

Note how the pattern representing the value communicated through the gate is a record pattern with one field called \$1. It is due to the form in which gates are represented in [Que98]. A gate G which has been declared of type T can communicate values which are records with one field (called \$1) of type T. Hence, the pattern, which has to be pattern-matched against the type $(\$1\,\texttt{=>}\,T)$, has to be of the form $(\$1\,\texttt{=>}\,P_1)$.

When an action is translated the default values for the optional parameters have to be given: the pattern P_1 is () by default, P_2 is **any**:time, and E is *true* (which abstract syntax is **exit**(\$1 => *true*)); and the clause $\textbf{start}\langle 0\rangle$ has to be added.

6.3 Successful termination without values

The behaviour **null** finishes immediately without producing any binding, so it is syntactic sugar for **exit**() (see next section):

$$\textbf{null} \quad \overset{\text{abs}}{=} \quad \textbf{exit}\texttt{()}.$$

6.4 Successful termination with values

exit(RN) is a new abstract syntax behaviour which is not in the User Language. It represents a behaviour that finishes immediately, returning the values indicated by RN, which is a list $V_1\,\texttt{=>}\,N_1, \ldots, V_n\,\texttt{=>}\,N_n$. It is used in the definition of the semantics to represent the bindings between variables and values produced by a behaviour, as we will see in the next chapter.

6.5 Interleaving operator

The interleaving operator is syntactic sugar for the parallel operator when the list of gates on which the processes have to synchronize is empty. Thus,

$$B_1\ \texttt{|||}\ B_2 \quad \overset{\text{abs}}{=} \quad B_1\ \texttt{|[]|}\ B_2.$$

Note that, in the abstract syntax, the parallel operator may have an empty list of gates.

6.6 Exception raising

An exception raised with the **raise** operator has the same meaning as if the exception is raised with the **signal** operator, if the exception is in the scope of a **trap** operator. But if the exception is not in the scope of a **trap** behaviour, the **raise** operator blocks time. Since any behaviour behind the **signal** operator is not executed if the raised exception is trapped, we have the following translation

$$\textbf{raise}\ X\ [\texttt{(}\ E\ \texttt{)}]\ \overset{\text{abs}}{=}\ \textbf{signal}\ X\texttt{(}\ E\ \texttt{)}\ \texttt{;}\ \textbf{block}$$

The default expression is **exit**($1 => ()). That is, when no expression is given in the specification, the value () is sent together with the exception.

6.7 Conditional operator

The **if** – **then** – **else** instruction is syntactic sugar for a **case** where the expression on which the choice is made is a boolean expression and there are two possibilities. In order to achieve this translation, an **if** – **then** – **else** behaviour is translated in two steps. First, if there are **elsif** clauses, they are removed in the following way:

$$\begin{array}{ll} & \textbf{if}\ E_1\ \textbf{then}\ B_1\ \textbf{elsif}\ E_2\ \textbf{then}\ B_2\ \textbf{else}\ B_3\ \textbf{endif} \\ \overset{\text{def}}{=} & \\ & \textbf{if}\ E_1\ \textbf{then}\ B_1\ \textbf{else}\ \textbf{if}\ E_2\ \textbf{then}\ B_2\ \textbf{else}\ B_3\ \textbf{endif}\ \textbf{endif} \end{array}$$

Second, when there is no **elsif** clause, the following translation is made:

$$\begin{array}{ll} & \textbf{if}\ E\ \textbf{then}\ B_1\ \textbf{else}\ B_2\ \textbf{endif} \\ \overset{\text{def}}{=} & \\ & \textbf{case}\ E\texttt{:}\mathsf{bool}\ \textbf{is}\ \mathit{true}\ \texttt{->}\ B_1\ \texttt{|}\ \mathit{false}\ \texttt{->}\ B_2 \end{array}$$

6.8 Imperative features

A loop like **loop** *B* **endloop** can be stopped by means of the predefined exception *inner*. This is achieved by translating the **loop** in the following way:

$$\begin{array}{ll} & \textbf{loop}\ B\ \textbf{endloop} \\ \overset{\text{def}}{=} & \\ & \textbf{trap exception}\ \mathit{inner}\texttt{(}\$1\ \texttt{=>}\ \texttt{())}\ \textbf{is exit}\texttt{()} \\ & \textbf{in loop}\ B \end{array}$$

A “named” loop, that is, a loop where the name of the exception which breaks the loop has been given, and where the type of the values that go with the exception is declared, is translated in the following way:

$$\begin{array}{ll} & \textbf{loop}\ X[\texttt{:}T]\ \textbf{in}\ B\ \textbf{endloop} \\ \overset{\text{def}}{=} & \\ & \textbf{trap exception}\ X\texttt{(}\$1\ \texttt{=>}\ \texttt{?}x\texttt{)}\texttt{:}T\ \textbf{is exit}\texttt{(}\$1\ \texttt{=>}\ x\texttt{)} \\ & \textbf{in loop}\ B \end{array}$$

where the type T is () by default.

The **break** instruction is used to stop a loop. Since the loops are translated to **trap** behaviours, the **break** instruction has to be translated to exception raising:

$$\textbf{break}\ [X\ [\texttt{(}\ E\ \texttt{)}]]\ \overset{\text{abs}}{=}\ \textbf{raise}\ X\ \texttt{(}\ E\ \texttt{)}$$

The default exception raised is the exception *inner*, and the default expression is **exit**($1 => ()).

The **while** loop is syntactic sugar for a **loop** whose body is an **if** – **then** – **else**: if the (expression) condition of the **while** is true then the body of the **while** is executed, and if it is false, then the loop is finished. Thus, the **while** is translated as follows:

$$\begin{array}{l} \textbf{while } E \textbf{ do } B \textbf{ endwhile} \\ \overset{\text{def}}{=} \\ \textbf{loop if } E \textbf{ then } B \textbf{ else break endif endloop} \end{array}$$

As we saw, the **for** loop is syntactic sugar for a **while** which has an initialization before it starts and whose body has an "increment" behaviour at the end:

$$\begin{array}{l} \textbf{for } E_1 \textbf{ while } E_2 \textbf{ by } E_3 \textbf{ do } B \textbf{ endfor} \\ \overset{\text{def}}{=} \\ E_1 \,\textbf{;}\, \textbf{while } E_2 \textbf{ do } B\textbf{;}E_3 \textbf{ endwhile} \end{array}$$

6.9 Process declaration

Regarding process declarations, there are several things to do in order to translate them to abstract syntax. First we have to record information needed later in the translation of the process instantiations. So, if the following process declaration

> **process** Π [$G_1 : T_1$, ... ,$G_n : T_n$] (FPL) **raises** [$X_1 : T''_1$, ... ,$X_m : T''_m$]
> **is** B
> **endproc**

is being translated, where

$$FPL = [\textbf{in}|\textbf{out}]\ V_1 : T'_1, \ldots, [\textbf{in}|\textbf{out}]\ V_p : T'_p$$

stands for the list of formal parameters, the following information is recorded in the context $\mathcal{S}$:

- the positions of the **out** parameters. This is needed to remove actual parameters in the process instantiation,
$$\Pi \Rightarrow \{j \mid \textbf{out}\ V_j : T_j \in FPL\}.$$
- the names of the formal gates, needed if the abbreviated gate parameter list **...** is used in the instantiation of the process,
$$\Pi \Rightarrow \textbf{gates}([G_1, \ldots, G_n]).$$
- the names of the formal parameters, for the same reason as above,
$$\Pi \Rightarrow \textbf{params}([V_1, \ldots, V_p]).$$
- the names of the formal exceptions,
$$\Pi \Rightarrow \textbf{exceptions}([X_1, \ldots, X_m]).$$

Second, we have to translate the process declaration in order to remove **out** parameters (if any). In the abstract syntax process declaration the parameter list has only **in** parameters, and a new clause indicating the "return" type of the process (formed from the **out** parameters) is added.

In the body of the process, (old) **out** parameters are translated to local variables, so a local variable declaration has to be included, and the values associated to these variables when the body of the process finishes are returned outside the process by means of an **exit** behaviour (see bellow). So, the process declaration translation is as follows:

$$
\begin{array}{ll}
 & \mathbf{process}\ \Pi\ [\ G_1 : T_1\ ,\ \ldots\ ,G_n : T_n\ ]\ (\ FPL\ )\ \mathbf{raises}\ [\,X_1 : T''_1\ ,\ \ldots\ ,X_m : T''_m\,] \\
 & \mathbf{is}\ B\ \mathbf{endproc} \\
\overset{\mathsf{def}}{\equiv} & \\
 & \mathbf{process}\ \Pi\ [\ G_1 : T_1\ ,\ \ldots\ ,G_n : T_n\ ]\ (\ I\ )\ :OT\ \mathbf{raises}\ [\,X_1 : T''_1\ ,\ \ldots\ ,X_m : T''_m\,] \\
 & \mathbf{is\ var}\ O\ \mathbf{in}\ B\ ;\ \mathbf{exit}(RV)
\end{array}
$$

where

- I stands for the formal input parameters, $I = V_{i_1} : T_{i_1}\ ,\ \ldots\ ,V_{i_k} : T_{i_k}$, where

$$\{V_{i_1} : T_{i_1}, \ldots, V_{i_k} : T_{i_k}\} = \{V_i : T_i \mid [\mathbf{in}]\ V_i : T_i \in FPL\} \quad i_1 < i_2 < \ldots < i_k$$

- OT stands for the return type of the process, $OT = (\$1 \Rightarrow T_{i_1}\ ,\ \ldots\ ,\$l \Rightarrow T_{i_l})$, where

$$\{T_{i_1}, \ldots, T_{i_l}\} = \{T_i \mid \mathbf{out}\ V_i : T_i \in FPL\} \quad i_1 < i_2 < \ldots < i_l$$

  Note that when there are no **out** parameters the type OT is `()`. In [Que98] the type **none** is used when there are no out parameters, but this gives problems because **none** is used when a process does not finish. We have proposed that when a process finishes but does not return anything (because it has no **out** parameter) it has type `()`, and it has been accepted.

- O stands for the variables that represent the **out** parameters, $O = V_{i_1} : T_{i_1}\ ,\ \ldots\ ,V_{i_l} : T_{i_l}$, where

$$\{V_{i_1} : T_{i_1}, \ldots, V_{i_l} : T_{i_l}\} = \{V_i : T_i \mid \mathbf{out}\ V_i : T_i \in FPL\} \quad i_1 < i_2 < \ldots < i_l$$

- and RV stands for the returned variables, $RV = \$1 \Rightarrow V_{i_1}\ ,\ \ldots\ ,\$l \Rightarrow V_{i_l}$, where

$$\{V_{i_1}, \ldots, V_{i_l}\} = \{V_i \mid \mathbf{out}\ V_i : T_i \in FPL\} \quad i_1 < i_2 < \ldots < i_l$$

Example 6.1 The process

```
process P [G:int](v:int,out w:int) is
  G(!v);G(?w)
endproc
```

is translated to

```
process P [G:int](v:int) raises [ ] is
  var w:int in
    G(!v);G(?w);exit($1 => w)
```

and the following information is saved in the context $\mathcal{S}$:

$P \Rightarrow \{2\}$
$P \Rightarrow \mathbf{gates}([G])$
$P \Rightarrow \mathbf{params}([v, w])$
$P \Rightarrow \mathbf{exceptions}([])$

6.10 Process instantiation

A process instantiation is translated to abstract syntax in two steps. First, if actual gates, parameters, or exceptions are given in abbreviated form, then, the non-present ones are provided, by using the information in the context $\mathcal{S}$. The following rule is used in this step:

$$\frac{\begin{array}{c}\mathcal{S} \vdash \Pi \Rightarrow \mathbf{gates}(\,\texttt{[}\,G_1,\ldots,G_n\,\texttt{]}\,)\\ \mathcal{S} \vdash \Pi \Rightarrow \mathbf{params}(\,\texttt{[}\,V_1,\ldots,V_p\,\texttt{]}\,)\\ \mathcal{S} \vdash \Pi \Rightarrow \mathbf{exceptions}(\,\texttt{[}\,X_1,\ldots,X_m\,\texttt{]}\,)\end{array}}{\Pi\ \texttt{[}GPL\texttt{]}\texttt{(}APL\texttt{)}\texttt{[}XPL\texttt{]} \ \overset{\mathsf{abs}}{=}\ \Pi\ \texttt{[}\,g_1,\ldots,g_n\,\texttt{]}\,\texttt{(}ep_1,\ldots,ep_p\texttt{)}\,\texttt{[}\,x_1,\ldots,x_m\,\texttt{]}}$$

where

- if $GPL = G_{k_1}\texttt{=>}G'_{k_1}\texttt{,}\ldots\texttt{,}G_{k_r}\texttt{=>}G'_{k_r}[\texttt{,...}]$ and $G_{k_i} \in \{G_1,\ldots,G_n\}$ then

  $$g_i = \begin{cases} G'_i & \text{iff } G_i \texttt{ => } G'_i \ \in\ GPL \\ G_i & \text{otherwise} \end{cases}$$

- if $APL = V_{j_1}\texttt{=>}AP_{j_1}\texttt{,}\ldots\texttt{,}V_{j_s}\texttt{=>}AP_{j_s}[\texttt{,...}]$ and $V_{j_i} \in \{V_1,\ldots,V_p\}$ then

  $$ep_i = \begin{cases} E'_i & \text{iff } V_i \texttt{ => } AP_i \ \in\ APL \\ P'_i & \text{iff } V_i \texttt{ => } AP_i \ \in\ APL \\ V_i & \text{otherwise} \end{cases}$$

  where AP_i must be an expression if parameter i is an **in** parameter, and a pattern if parameter i is an **out** parameter. In the latter case, the pattern AP_i must be able to be pattern matched against a value of the corresponding type.

- if $XPL = X_{l_1}\texttt{=>}X'_{l_1}\texttt{,}\ldots\texttt{,}X_{l_q}\texttt{=>}X'_{l_q}[\texttt{,...}]$ and $X_{l_i} \in \{X_1,\ldots,X_p\}$ then

  $$x_i = \begin{cases} X'_i & \text{iff } X_i \texttt{ => } X'_i \ \in\ XPL \\ X_i & \text{otherwise} \end{cases}$$

Second, the actual parameters in the position of **out** parameters (which have to be patterns) are removed. This is done by means of a **trap** behaviour that traps with an **exit** clause the returned values and assigns them to the actual **out** parameters. The following rule is applied:

$$\frac{\mathcal{S} \vdash \Pi \Rightarrow Pos}{\Pi\ \texttt{[}\ GPL\texttt{]}\ \texttt{(}\ AP_1\texttt{,}\ldots\texttt{,}AP_n\ \texttt{)}\ \texttt{[}XPL\texttt{]}\ \overset{\mathsf{abs}}{=}\ \begin{array}{l}\textbf{trap exit } \texttt{?}\$param \textbf{ is } \texttt{(}outRP\texttt{)}\ \texttt{:=}\ \textbf{exit}\texttt{(}\$1 \texttt{ => } \$param\texttt{)}\\ \textbf{in } \Pi\ \texttt{[}GPL\texttt{]}\ \texttt{(}inRE\texttt{)}\ \texttt{[}XPL\texttt{]}\end{array}}\ [Pos \neq \emptyset]$$

where

- $outRP = \$1 \texttt{ => } AP_{i_1}\texttt{,}\ldots\texttt{,}\$s \texttt{ => } AP_{i_s}$ with $i_1 < i_2 < \ldots < i_s$ and $\{i_1,\ldots,i_s\} \ = \ Pos$;
- $inRE = AP_{j_1}\texttt{,}\ldots\texttt{,}AP_{j_k}$ with $j_1 < j_2 < \ldots < j_k$ and $\{j_1,\ldots,j_k\} \ = \ \{1,\ldots,n\} \ - \ Pos$; and
- $\$param$ is a new variable, which cannot be used by the specification writer.

Example 6.2 The process declared in Example 6.1 can be instantiated as follows:

$P\texttt{[}G'\texttt{]}\texttt{(}5,\texttt{?}a\texttt{)}$

which is translated into abstract syntax to

trap exit $\texttt{?}\$param$ **is** $\texttt{(}\$1 \texttt{ => ?}a\texttt{)}\ \texttt{:=}\ $ **exit**$\texttt{(}\$1 \texttt{ => } \$param\texttt{)}$
in $P\texttt{[}G'\texttt{]}\texttt{(}5\texttt{)}\texttt{[ ]}$

6.11 Case operator

The **case** operator is translated to abstract syntax by adding a final clause to ensure that the pattern-matching is exhaustive, if all the branches (but the last one) fail then the last one (which never fails) raise the predefined exception *Match*.

case E:T **is** BM **endcase**

$\overset{\text{def}}{=}$

case E:T **is** BM | **any**:T -> **signal** *Match*

The **case** with several expressions is translated in the same way.

6.12 Function declaration

A function declaration is translated to a process declaration

function F ([**in**|**out**]V_1:T_1, . . . ,[**in**|**out**]V_m:T_m) :T **raises** [X_1:T'_1, . . . ,X_p:T'_p] **is** E **endfunc**

$\overset{\text{def}}{=}$

process F [ ] ([**in**|**out**]V_1:T_1, . . . ,[**in**|**out**]V_m:T_m,**out** $\$result$:$T$) **raises** [X_1:T'_1, . . . ,X_p:T'_p]
is ?$\$result$:= E **endproc**

where a new **out** parameter is added that represents the *result* of the function. We have propose to include this new parameter because in [Que98] the value returned by a function is not taken into account in the translation to abstract syntax.

An infix function declaration is translated in the same way.

Example 6.3 Let us see how the (very simple) function declaration

```
function two:int is
  2
endfunc
```

is translated.

First, it is translated to the following process declaration:

```
process two[ ](out $result:int) raises [ ] is
  ?$result := 2
endproc
```

which is then translated to abstract syntax as

```
process two[ ]():($1 => int) raises [ ] is
  var $result:int in
    ?$result := exit($1 => 2) ; exit($1 => $res)
```

6.13 Type expressions

Although in the User Language there is no anonymous type, that is, every type has a name, in the abstract syntax, anonymous types are accepted. Thus, a record type expression enclosed between brackets (RT) is considered a type expression.

6.14 Expressions

In the abstract syntax, expressions are considered as a kind of behaviour. An expression is a particular case of behaviour which can only raise exceptions and terminate (possibly assigning values to variables) but which cannot communicate on gates or delay. Most of the expressions can be seen as the corresponding behaviour, and others need an explicit translation. We are going to show now these latter ones.

Values

Values, that is, expressions which are in normal form, are translated by using the new abstract operator **exit**. By using this operator, a value expression N is translated to **exit**($1 => N), that is, a value expression N is a behaviour that associates the identifier $1 with the value N.

We have to note that only the values which are situated in a place of an expression (where an expression is needed) are translated in this way. For example, if we have the assignment ?p := (x => 8, y => 3), then the record (x => 8, y => 3) is an expression, so it is translated as indicated above, and the assignment in abstract syntax would be ?p := **exit**($1 => ($x$ => 8, y => 3)), where the values 8 and 3 do not need to be translated.

andalso operation

The expression E_1 **andalso** E_2 returns the value of E_2 if E_1 evaluates to *true*, and *false* otherwise. It is translated by using an **if** − **then** − **else**, that needs to be translated later.

$$E_1 \textbf{ andalso } E_2 \quad \overset{\text{abs}}{=} \quad \textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } \textit{false} \textbf{ endif}$$

orelse operation

The expression E_1 **orelse** E_2 returns the value of E_2 if E_1 evaluates to *false*, and *true* otherwise. This expression is translated as the last one by using an **if** − **then** − **else**.

$$E_1 \textbf{ orelse } E_2 \quad \overset{\text{abs}}{=} \quad \textbf{if } E_1 \textbf{ then } \textit{true} \textbf{ else } E_2 \textbf{ endif}$$

Equality operation

E-LOTOS provides an equality operation that compares the values of two expressions of any type. This is done by means of a **case** behaviour that force to evaluate the expressions and then compare them with other **case**. The translation of this equality operator is as follows:

$$\begin{array}{ll} & E_1 = E_2 \\ \overset{\text{def}}{=} & \\ & \textbf{case } (E_1\texttt{:}\textbf{any}, E_2\texttt{:}\textbf{any}) \textbf{ is } (?x, ?y)\texttt{->} \textbf{ case } x\texttt{:}\textbf{any} \textbf{ is } !y\texttt{->}\textit{true} \mid \textbf{any}\texttt{:}\textbf{any}\texttt{->}\textit{false} \end{array}$$

Inequality operation

E_1 <> E_2 returns *false* if $E_1 = E_2$ returns *true* and vice-versa. So we can translate first

$$\begin{array}{ll} & E_1 \mathrel{<>} E_2 \\ \overset{\text{def}}{=} & \\ & \textbf{if } E_1 = E_2 \textbf{ then } \textit{false} \textbf{ else } \textit{true} \textbf{ endif} \end{array}$$

and then translate both $E_1 = E_2$ and the **if** − **then** − **else**.

Select field operation

The required value is obtained by means of a **case**. If the operation E.V is correct, then the expression E has to be able to be reduced to a record of values with a field named V. This value can be matched against the pattern (V => ?x,**etc**). The value of the variable x after the pattern matching is the desired valued.

$$E\ .\ V \quad \overset{\mathsf{abs}}{=} \quad \textbf{case}\ E\texttt{:}\textbf{any is}\ \texttt{(}V\ \texttt{=>}\ \texttt{?}x\texttt{,}\textbf{etc}\texttt{)}\ \texttt{->}\ x$$

Explicit typing operation

An explicit typing E:T is translated as follows:

$$E\texttt{:}T \quad \overset{\mathsf{abs}}{=} \quad \textbf{case}\ E\texttt{:}T\ \textbf{is}\ \texttt{?}x\ \texttt{->}\ x$$

The operational semantics of the **case** operator ensures that E:T only has meaning when the expression E has type T.

Function instantiation

A function instantiation is translated to a process instantiation with a new actual parameter ?\$1 that will be associated with the value that the function body returns:

$$F\texttt{(}APL\texttt{)}\ \texttt{[}XPL\texttt{]} \quad \overset{\mathsf{abs}}{=} \quad F\texttt{(}APL\texttt{,?\$1)}\ \texttt{[}XPL\texttt{]}$$

Example 6.4 Let us see how the following use of the function in Example 6.3,

```
?x := two
```

is translated.

First, the function instantiation is translated to a process instantiation

```
?x := two[](?$1)[]
```

which is then translated to abstract syntax as

```
?x := trap exit ?$param is ($1 => ?$1) := exit($1 => $param)
        in two[]()[]
```

7 Static Semantics of Base Language

The aim of the static semantics of a language is to check decidable properties of a program, in this case a specification in E-LOTOS. The most common property is type checking, that ensures, in a strongly typed language such as E-LOTOS, that no type error will occur at execution or evaluation time. Another property checked by the E-LOTOS static semantics is that variables are always read after they have been assigned a value.

In the following sections we describe the properties checked by the static semantics in the Base Language, as well as the contexts and inference rules used for the different constructs in E-LOTOS.

7.1 Introduction

In the Base Language, static semantics checks the following properties:

- That all the objects that appear in a behaviour have the correct type. For example, that values sent through a gate have the same type as values that the gate can communicate, that a process instantiation has parameters with correct types, or that in the **case** operator all possible patterns have the right type.
- What is the *type* of a behaviour, that is, what bindings between variables and types the behaviour produces, and whether the behaviour can finish immediately or it has to carry out an action or raise an exception before finishing. Static semantics judgements that check the type of a behaviour B are as follows:

  $$\mathcal{C} \vdash B \implies \mathsf{exit} \,\langle\!|\, RT \,|\!\rangle,$$

  meaning that in the context $\mathcal{C}$ behaviour B finishes producing bindings RT, or

  $$\mathcal{C} \vdash B \implies \mathsf{guarded} \,\langle\!|\, RT \,|\!\rangle.$$

  meaning that in the context $\mathcal{C}$ behaviour B produces bindings RT and it cannot finish initially, i.e., without carrying out an action or raise an exception. As we explain below, record type terms, RT, are used to indicate the bindings between variables and types that a behaviour produces.
- That time is deterministic, by rejecting behaviours that could be time nondeterministic. Static semantics forbids behaviours like

  $$\texttt{(?}x\texttt{:=5 [] ?}x\texttt{:=2); }\textbf{wait}\texttt{(1)}$$

  that, as we saw, is time nondeterministic: when one unit of time has passed, variable x may have either value 5 or 2. This behaviour is rejected by requiring that behaviours composed with the selection operator must be guarded, i.e., that they must do an action on a gate or raise an exception before they finish (see Section 7.4).
- That variables are used in the right way, i.e., that a variable that has not been assigned a value yet is never read, and that a variable is never assigned a value of an incompatible type (see Section 7.22).

Contexts

In the previous judgements, a context $\mathcal{C}$ appears. It keeps all the information related to identifiers that have already appeared in the specification and that are *visible* in this moment. Contexts are built with this grammar:

$$
\begin{array}{lcl@{\qquad}r@{\quad}r}
\mathcal{C} & ::= & K \Rrightarrow T & \textit{predefined constant} & (\mathcal{C}1) \\
 & \mid & V \mathrel{\mathtt{=>}} T & \textit{initialized variable} & (\mathcal{C}2) \\
 & \mid & V \Rightarrow ?T & \textit{typed variable} & (\mathcal{C}3) \\
 & \mid & S \Rightarrow \mathsf{type} & \textit{type} & (\mathcal{C}4) \\
 & \mid & S \equiv T & \textit{type equivalence} & (\mathcal{C}5) \\
 & \mid & T \sqsubseteq T' & \textit{subtype} & (\mathcal{C}6) \\
 & \mid & C \Rightarrow (RT) \rightarrow S & \textit{constructor} & (\mathcal{C}7) \\
 & \mid & \Pi \Rightarrow [\,\mathsf{gate}\langle(\$1 \mathrel{\mathtt{=>}} T_1)\rangle, \ldots, \mathsf{gate}\langle(\$1 \mathrel{\mathtt{=>}} T_m)\rangle\,] & \textit{process identifier} & (\mathcal{C}8) \\
 & & \qquad (V_1 : T'_1, \ldots, V_p : T'_p) & & \\
 & & \qquad [\,\mathsf{exn}\langle(\$1 \mathrel{\mathtt{=>}} T''_1)\rangle, \ldots, \mathsf{exn}\langle(\$1 \mathrel{\mathtt{=>}} T''_n)\rangle\,] \rightarrow \mathsf{exit} \llparenthesis RT \rrparenthesis & & \\
 & \mid & \Pi \Rightarrow [\,\mathsf{gate}\langle(\$1 \mathrel{\mathtt{=>}} T_1)\rangle, \ldots, \mathsf{gate}\langle(\$1 \mathrel{\mathtt{=>}} T_m)\rangle\,] & \textit{process identifier} & (\mathcal{C}9) \\
 & & \qquad (V_1 : T'_1, \ldots, V_p : T'_p) & & \\
 & & \qquad [\,\mathsf{exn}\langle(\$1 \mathrel{\mathtt{=>}} T''_1)\rangle, \ldots, \mathsf{exn}\langle(\$1 \mathrel{\mathtt{=>}} T''_n)\rangle\,] \rightarrow \mathsf{guarded} \llparenthesis RT \rrparenthesis & & \\
 & \mid & G \Rightarrow \mathsf{gate}\langle(RT)\rangle & \textit{gate} & (\mathcal{C}10) \\
 & \mid & X \Rightarrow \mathsf{exn}\langle(RT)\rangle & \textit{exception} & (\mathcal{C}11) \\
 & \mid & & \textit{trivial} & (\mathcal{C}12) \\
 & \mid & \mathcal{C}, \mathcal{C} & \textit{disjoint union} & (\mathcal{C}13)
\end{array}
$$

where each element has the following meaning:

- $K \Rrightarrow T$ means that K is a predefined constant of type T. This information is introduced in the initial context when the predefined library of E-LOTOS is read. Any specification is checked in a context with this information.
- $V \mathrel{\mathtt{=>}} T$ means that variable V has been initialized, that is, V has been given a value, and this value has type T.
- $V \Rightarrow ?T$ means that variable V has been declared in a **var** behaviour, so it is restricted to have a value of type T.
- $S \Rightarrow \mathsf{type}$ means that S is the name given to a declared type.
- $S \equiv T$ means that types S and T are equivalent. When this information is included in the context, we use it to denote that T is the *definition* of the identifier S. We also write $T \equiv T'$ as a shorthand for $T \sqsubseteq T'$ and $T' \sqsubseteq T$.
- $T \sqsubseteq T'$ means that type T is a subtype of type T', that is, all values of type T are also values of type T'.
- $C \Rightarrow (RT) \rightarrow S$ means that C is a constructor of the datatype S, and it has arguments of type (RT).
- $\Pi \Rightarrow [\,\mathsf{gate}\langle(\$1 \mathrel{\mathtt{=>}} T_1)\rangle, \ldots, \mathsf{gate}\langle(\$1 \mathrel{\mathtt{=>}} T_m)\rangle\,]$
  $(V_1 : T'_1, \ldots, V_p : T'_p)$
  $[\,\mathsf{exn}\langle(\$1 \mathrel{\mathtt{=>}} T''_1)\rangle, \ldots, \mathsf{exn}\langle(\$1 \mathrel{\mathtt{=>}} T''_n)\rangle\,] \rightarrow \mathsf{exit} \llparenthesis RT \rrparenthesis$
  means that process Π has been declared; that it has m formal gates, that can communicate values of types T_i; that it has p formal input parameters called $V_1, \ldots, V_p$ of types T'_i; that it has n formal exceptions, which are raised together with values of types T''_i; and that the body of the process

produces bindings RT, and it may perform no action or raise no exception before finishing, that is, the body is an exit behaviour.

- $\Pi \Rightarrow$ [$\mathit{gate}\langle$($\$1$ => T_1)$\rangle$, ... , $\mathit{gate}\langle$($\$1$ => T_m)$\rangle$]
  ($V_1 : T'_1, \ldots, V_p : T'_p$)
  [$\mathit{exn}\langle$($\$1$ => T''_1)$\rangle$, ... , $\mathit{exn}\langle$($\$1$ => T''_n)$\rangle$] $\rightarrow$ $\mathit{guarded}$ $\langle\!| RT |\!\rangle$
  means the same as above but now the body of the process performs an action or raises an exception before finishing, that is, the body is a guarded behaviour.
- $G \Rightarrow \mathit{gate}\langle$($RT$)$\rangle$ means that gate G has been declared, and it can communicate values of type (RT). In [Que98] gates are handled in the following way: A gate G which has been declared of type T can communicate values which are records with one field (called $1) of type T. Thus, $G \Rightarrow \mathit{gate}\langle$($RT$)$\rangle$ is always of the form $G \Rightarrow \mathit{gate}\langle$($\$1$ => T)$\rangle$. We are going to follow this decision in order to do not separate from the proposed standard although we think that it would be clearer to save only the type T, and have things like $\mathit{gate}\langle T\rangle$. Nevertheless, we will write $\mathit{gate}\langle$($\$1$ => T)$\rangle$ when we want to do explicit that G communicates values of type T.
- $X \Rightarrow \mathit{exn}\langle$($RT$)$\rangle$ means that exception X has been declared, and it is raised together with values of type (RT). As with gates, (RT) will be always of the form ($\$1$ => T).

Operations used on contexts are the "," (disjoint union), "$\odot$" (matching union), ";" (context overriding), and "$-$" (subtraction). Their meaning is as follows:

- $\mathcal{C}_1, \mathcal{C}_2$ represents the union of the bindings of $\mathcal{C}_1$ and $\mathcal{C}_2$, and it is defined only if $\mathcal{C}_1$ and $\mathcal{C}_2$ bind different identifiers.
- $\mathcal{C}_1 \odot \mathcal{C}_2$ denotes the matching union of contexts $\mathcal{C}_1$ and $\mathcal{C}_2$. It is defined only if the common names of $\mathcal{C}_1$ and $\mathcal{C}_2$ have the same bindings in both contexts. By considering contexts as functions from identifiers to their declaration, the formal definition of the operator $\odot$ is:

$$(\mathcal{C}_1 \odot \mathcal{C}_2)(id) = \begin{cases} \mathcal{C}_1(id) & \textbf{if } id \notin \mathsf{Dom}(\mathcal{C}_2) \\ \mathcal{C}_2(id) & \textbf{if } id \notin \mathsf{Dom}(\mathcal{C}_1) \\ \mathcal{C}_1(id) & \textbf{if } \mathcal{C}_2(id) = \mathcal{C}_1(id) \end{cases}$$

  where $\mathsf{Dom}(\mathcal{C})$ denotes the identifiers bound by $\mathcal{C}$.
- $\mathcal{C}_1 ; \mathcal{C}_2$ represents the context with all the bindings of $\mathcal{C}_2$, and any binding from $\mathcal{C}_1$ not overridden by $\mathcal{C}_2$. That is,

$$(\mathcal{C}_1 ; \mathcal{C}_2)(id) = \begin{cases} \mathcal{C}_2(id) & \textbf{if } id \in \mathsf{Dom}(\mathcal{C}_2) \\ \mathcal{C}_1(id) & \textbf{if } id \notin \mathsf{Dom}(\mathcal{C}_2) \end{cases}$$

- $\mathcal{C} - \{V_1, \ldots, V_n\}$ represents the context $\mathcal{C}$ where all bindings related to variables in $\{V_1, \ldots, V_n\}$ have been removed.

The precedence of these operators is $\odot$ > , > ; > $-$.

Bindings

When a behaviour is executed it can perform actions, raise exceptions, and it can also assign variables. The assignment ?V := E produces a binding between variable V and type T of the value of the expression E. This binding, V => T, represents that variable V has been assigned a value of type T. Bindings are also produced by means of pattern-matching. When a behaviour produces more than one binding, these bindings are put together in a list V_1 => $T_1, \ldots, V_n$ => T_n which has the same syntax as a record type expression. Thus, we will use record type expressions, RT, to represent the bindings produced by a behaviour.

It is important to know the bindings produced by a behaviour B when it finishes, because this information can be used by a behaviour B' after B, for example in the behaviour $B\,;B'$. Behaviour B' knows the bindings produced by B if it is executed in a context with these bindings (see Section 7.3 for the static semantics of the sequential composition). The binding V => T can also be part of a context, so we will use RT as a record type expression, a list of bindings produced by a behaviour, and a context, interchangeably. We do the same with bindings produced by pattern-matching which are introduced in the context (see, for example, Section 7.2).

Due to the way value N is translated into the behaviour **exit**($1 => N), the identifier $1 is used to keep the value returned by a behaviour[1]. So, the behaviour **exit**($1 => 3), which is the translation of the integer value 3, finishes by producing binding $1 => int, and this means that **exit**($1 => 3) *returns* an integer (see Example 7.9 in Section 7.6). In the inference rules of the following sections, E denotes a behaviour which is a translation of an expression. When we want that this *expression* only returns a value, we require it to be of type *exit* ⟬ $1 => T ⟭. For example, E is a boolean expression if it is of type *exit* ⟬ $1 => bool ⟭.

Judgements

The relation between the predicates *guarded* and *exit* can be expressed with the following general inference rule:

$$\frac{\mathcal{C} \vdash B \implies \textit{guarded} \,\langle\!|\, RT \,|\!\rangle}{\mathcal{C} \vdash B \implies \textit{exit} \,\langle\!|\, RT \,|\!\rangle} \tag{7.14}$$

That is, *guarded* is a stronger condition. Thus, if we require a behaviour B to be an *exit* ⟬ RT ⟭ behaviour, then a *guarded* ⟬ RT ⟭ behaviour may be used instead.

By relating these judgements with the notion of subtyping (we will see the subtyping judgements below), we have the following inference rules:

$$\frac{\begin{array}{l}\mathcal{C} \vdash B \implies \textit{exit} \,\langle\!|\, RT \,|\!\rangle \\ \mathcal{C} \vdash RT \sqsubseteq RT'\end{array}}{\mathcal{C} \vdash B \implies \textit{exit} \,\langle\!|\, RT' \,|\!\rangle} \tag{7.15}$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash B \implies \textit{guarded} \,\langle\!|\, RT \,|\!\rangle \\ \mathcal{C} \vdash RT \sqsubseteq RT'\end{array}}{\mathcal{C} \vdash B \implies \textit{guarded} \,\langle\!|\, RT' \,|\!\rangle} \tag{7.16}$$

That is, if we can prove that behaviour B produces bindings RT and, by seeing RT as a record type, we can prove that RT is a subtype of RT', then we can deduce that behaviour B produces bindings RT'.

Although previous judgements are the most important ones, since they refer to behaviours, we have also other judgements, needed to check properties of E-LOTOS elements that are not behaviours, and which are used to deduce facts regarding the latter.

Thus, we have judgements to check the type of a value, as follows

$$\mathcal{C} \vdash N \Rightarrow^{\mathrm{T}} T,$$

meaning that in context $\mathcal{C}$ value N has type T. The same kind of judgement is used to deduce the (record) type of a record of values,

$$\mathcal{C} \vdash RN \Rightarrow^{\mathrm{T}} RT.$$

[1] Behaviours that *return values* are those which were expressions before translating into abstract syntax.

We can relate these judgements with the notion of subtyping by means of the following inference rules:

$$\frac{\begin{array}{c}\mathcal{C} \vdash N \Rrightarrow^{\mathbf{T}} T \\ \mathcal{C} \vdash T \sqsubseteq T'\end{array}}{\mathcal{C} \vdash N \Rrightarrow^{\mathbf{T}} T'} \tag{7.17}$$

That is, if we can deduce from the context $\mathcal{C}$ that value N has type T, and that T is a subtype of type T', then we can also prove from context $\mathcal{C}$ that value N has type T'. In other words, we can use a value of type T where a value of type T' is required, provided that $T \sqsubseteq T'$ can be proved. We have a similar rule for record types:

$$\frac{\begin{array}{c}\mathcal{C} \vdash RN \Rrightarrow^{\mathbf{T}} RT \\ \mathcal{C} \vdash RT \sqsubseteq RT'\end{array}}{\mathcal{C} \vdash RN \Rrightarrow^{\mathbf{T}} RT'} \tag{7.18}$$

As we see in Section 7.28.3 the order of the fields of a record is not important in the abstract syntax because the fields are named.

Another kind of judgement we use is that relative to pattern matching. The judgement

$$\mathcal{C} \vdash (P \ \Rrightarrow^{\mathbf{PM}} \ T) \ \mapsto \ \langle\!|\, RT \,|\!\rangle$$

means that in context $\mathcal{C}$, when we match pattern P against type T, we succeed and bindings RT (between variables in P and types in T) are produced. And there are also judgements to deal with record pattern matching:

$$\mathcal{C} \vdash (RP \ \Rrightarrow^{\mathbf{PM}} \ RT) \ \mapsto \ \langle\!|\, RT' \,|\!\rangle$$

Judgements to deal with the pattern matching of all the branches of a **case** behaviour against a type are also used:

$$\mathcal{C} \vdash (BM \ \Rrightarrow^{\mathbf{PM}} \ T) \ \Longrightarrow \ \textit{exit} \ \langle\!|\, RT \,|\!\rangle$$

$$\mathcal{C} \vdash (BM \ \Rrightarrow^{\mathbf{PM}} \ T) \ \Longrightarrow \ \textit{guarded} \ \langle\!|\, RT \,|\!\rangle$$

where BM stands for all the possibilities (patterns) and associated behaviours in a **case** body.

As we have already seen, we also have judgements that assert that a type T is a subtype of another type T', with the following form:

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

The subtyping relation is a preorder, that is, it is reflexive and transitive, which is established by the following rules:

$$\frac{}{\mathcal{C} \vdash T \sqsubseteq T} \tag{7.19}$$

$$\frac{\mathcal{C} \vdash T \sqsubseteq T' \qquad \mathcal{C} \vdash T' \sqsubseteq T''}{\mathcal{C} \vdash T \sqsubseteq T''} \tag{7.20}$$

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

Similarly, the judgement

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

states that the record type RT is a subtype of the record type RT', and record subtyping is also a preorder:

$$\frac{}{\mathcal{C} \vdash RT \sqsubseteq RT} \tag{7.21}$$

$$\frac{\mathcal{C} \vdash RT \sqsubseteq RT' \qquad \mathcal{C} \vdash RT' \sqsubseteq RT''}{\mathcal{C} \vdash RT \sqsubseteq RT''} \tag{7.22}$$

We will write $RT \equiv RT'$ as abbreviation for $RT \sqsubseteq RT'$ and $RT' \sqsubseteq RT$.

Another kind of judgements is the one that asserts that a RT is really a record type, that is, it is well-formed:

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

We will see in Section 7.28 how well-formed records are.

There is a general axiom that asserts that for any binding J from a context $\mathcal{C}$, J may be inferred from that context:

$$\frac{}{\mathcal{C}, J \vdash J} \tag{7.23}$$

New inference rules

There are some *gaps* in the semantics of pattern-matching defined in [Que98]. For example, the pattern $(x \texttt{ => } ?a, y \texttt{ => } ?b)$ cannot be matched against the type $(y \texttt{ => int}, x \texttt{ => int})$ although it is said that the order of the fields are not important when they are named.

We have propose to include (there is nothing similar in [Que98]) a rule that relates pattern-matching with type equivalence:

$$\frac{\begin{array}{l}\mathcal{C} \vdash T_1 \equiv T_2 \\ \mathcal{C} \vdash (P \ \Rightarrow^{\mathbf{PM}} \ T_1) \ \mapsto \ \langle\!| RT |\!\rangle\end{array}}{\mathcal{C} \vdash (P \ \Rightarrow^{\mathbf{PM}} \ T_2) \ \mapsto \ \langle\!| RT |\!\rangle} \tag{7.24}$$

This is needed, for example, if we want to pattern-match $(x\texttt{=>}?a, y\texttt{=>}?b)$ against the type $(y\texttt{=>int}, x\texttt{=>int})$.

We need also a rule for record type equivalence:

$$\frac{\begin{array}{l}\mathcal{C} \vdash RT_1 \equiv RT_2 \\ \mathcal{C} \vdash (RP \ \Rightarrow^{\mathbf{PM}} \ RT_1) \ \mapsto \ \langle\!| RT |\!\rangle\end{array}}{\mathcal{C} \vdash (RP \ \Rightarrow^{\mathbf{PM}} \ RT_2) \ \mapsto \ \langle\!| RT |\!\rangle} \tag{7.25}$$

We have also propose to include in [Que98] an inference rule relating the type of values a gate can communicate and subtyping. If we know that a gate G can communicate values of type T and we know that $T \sqsubseteq T'$, then we know that G communicates values of type T'.

$$\frac{\begin{array}{l}\mathcal{C} \vdash G \Rightarrow \textit{gate}\langle(\$1 \texttt{=>} T)\rangle \\ \mathcal{C} \vdash T \sqsubseteq T'\end{array}}{\mathcal{C} \vdash G \Rightarrow \textit{gate}\langle(\$1 \texttt{=>} T')\rangle} \tag{7.26}$$

This is needed, for example, to instantiate a process like Router in Section 3.3. There, we define the extensible type packet,

```
type packet is
  (de => dest, etc)
endtype
```

and Router has three gates that can communicate values of this type, that is, Router can be used to send values that are records with at least one field called *de* of type dest. Thus, if we define the type completePacket,

```
type completePacket is
  (de => dest, value => data)
endtype
```

we should be able to instantiate Router with a gate *inCompP* declared of type completePacket. But this is not possible (see Section 7.26) if we have not the rule 7.26. By using this rule we have

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathit{inCompP} \Rightarrow \mathsf{gate}\langle\,\texttt{(}\$1 \texttt{=>} \mathsf{completePacket}\texttt{)}\,\rangle \\ \mathcal{C} \vdash \mathsf{completePacket} \sqsubseteq \mathsf{packet}\end{array}}{\mathcal{C} \vdash G \Rightarrow \mathsf{gate}\langle\,\texttt{(}\$1 \texttt{=>} \mathsf{packet}\texttt{)}\,\rangle} \tag{7.26}$$

In the following sections we will comment inference rules associated with the different elements of E-LOTOS. We will show examples of applications in each case.

7.2 Actions

We begin by showing the rule associated with a communication or synchronization through a gate of the form

$$G\ \texttt{(}\$1 \texttt{=>} P_1\texttt{)}\ \texttt{@}P_2\ E\ \mathbf{start}\langle N\rangle.$$

The inference rule is as follows:

$$\frac{\begin{array}{l}\mathcal{C} \vdash G \Rightarrow \mathsf{gate}\langle\,\texttt{(}\$1 \texttt{=>} T\texttt{)}\,\rangle \\ \mathcal{C} \vdash (\texttt{(}\$1 \texttt{=>} P_1\texttt{)}\ \Rightarrow^{\mathbf{PM}}\ \texttt{(}\$1 \texttt{=>} T\texttt{)}) \ \mapsto\ \langle\!|\, RT_1 \,|\!\rangle \\ \mathcal{C} \vdash (P_2\ \Rightarrow^{\mathbf{PM}}\ \mathsf{time}) \ \mapsto\ \langle\!|\, RT_2 \,|\!\rangle \\ \mathcal{C}; RT_1, RT_2 \vdash E \implies \mathsf{exit}\,\langle\!|\, \$1 \texttt{=>} \mathsf{bool} \,|\!\rangle \\ \mathcal{C} \vdash N \Rightarrow^{\mathbf{T}} \mathsf{time}\end{array}}{\mathcal{C} \vdash G\ \texttt{(}\$1 \texttt{=>} P_1\texttt{)}\ \texttt{@}P_2\ E\ \mathbf{start}\langle N\rangle \implies \mathsf{guarded}\,\langle\!|\, RT_1, RT_2 \,|\!\rangle} \tag{7.27}$$

with the additional condition that RT_1 and RT_2 are disjoint, that is, they bind different variables.

It means that from the context $\mathcal{C}$ we can deduce that action $G\ \texttt{(}\$1 \texttt{=>} P_1\texttt{)}\ \texttt{@}P_2\ E\ \mathbf{start}\langle N\rangle$ is semantically correct and it produces bindings RT_1, RT_2, by doing an action before terminating, provided that all the following conditions are fulfilled:

1. G is a gate identifier declared in the context $\mathcal{C}$ and the gate can communicate values of type $\texttt{(}\$1\texttt{=>}T\texttt{)}$,

$$\mathcal{C} \vdash G \Rightarrow \mathsf{gate}\langle\,\texttt{(}\$1 \texttt{=>} T\texttt{)}\,\rangle.$$

2. The pattern matching of pattern $\texttt{(}\$1\texttt{=>}P_1\texttt{)}$ against type $\texttt{(}\$1\texttt{=>}T\texttt{)}$ succeeds, and it produces bindings RT_1,

$$\mathcal{C} \vdash (\texttt{(}\$1 \texttt{=>} P_1\texttt{)}\ \Rightarrow^{\mathbf{PM}}\ \texttt{(}\$1 \texttt{=>} T\texttt{)}) \ \mapsto\ \langle\!|\, RT_1 \,|\!\rangle.$$

3. Pattern P_2 matches against type time by producing bindings RT_2,

$$\mathcal{C} \vdash (P_2\ \Rightarrow^{\mathbf{PM}}\ \mathsf{time}) \ \mapsto\ \langle\!|\, RT_2 \,|\!\rangle.$$

4. In the context $\mathcal{C}; RT_1, RT_2$, that is, in the context which is the result of adding to $\mathcal{C}$ the bindings previously produced, expression E finishes and it returns *a value* of type bool,
$$\mathcal{C}; RT_1, RT_2 \vdash E \implies \textsf{\textit{exit}}\,\langle\!|\, \$1 \texttt{=>} \mathsf{bool} \,|\!\rangle.$$

5. Value N has type time,
$$\mathcal{C} \vdash N \Rightarrow^{\mathbf{T}} \mathsf{time}.$$
In fact, when the static semantics is checked, every action has the clause $\mathbf{start}\langle 0\rangle$ (included by translating to abstract syntax), so this condition becomes $0 \Rightarrow^{\mathbf{T}} \mathsf{time}$ which must be true in any context. We have propose to remove it.

Example 7.1 Let us see what kind of behaviour $outP\texttt{(!3)}$ is. This behaviour is translated to abstract syntax as follows:

$$outP\texttt{(}\$1 \texttt{=>} \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{))}\ \texttt{@}\mathbf{any} : \mathsf{time}\ \mathbf{exit}\texttt{(}\$1 \texttt{=>} \mathit{true}\texttt{)}\ \mathbf{start}\langle 0\rangle.$$

By using inference rule 7.27, we can deduce that

$$\frac{\begin{array}{l}\mathcal{C} \vdash outP \Rightarrow \textsf{\textit{gate}}\langle\texttt{(}\$1 \texttt{=>} \mathsf{int}\texttt{)}\rangle \\ \mathcal{C} \vdash \texttt{((}\$1 \texttt{=>} \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{))}\ \Rightarrow^{\mathbf{PM}}\ \texttt{(}\$1 \texttt{=>} \mathsf{int}\texttt{))}\ \mapsto\ \langle\!|\ |\!\rangle \\ \mathcal{C} \vdash (\mathbf{any} : \mathsf{time}\ \Rightarrow^{\mathbf{PM}}\ \mathsf{time})\ \mapsto\ \langle\!|\ |\!\rangle \\ \mathcal{C} \vdash \mathbf{exit}\texttt{(}\$1 \texttt{=>} \mathit{true}\texttt{)} \implies \textsf{\textit{exit}}\,\langle\!|\, \$1 \texttt{=>} \mathsf{bool} \,|\!\rangle \\ \mathcal{C} \vdash 0 \Rightarrow^{\mathbf{T}} \mathsf{time}\end{array}}{\mathcal{C} \vdash outP\texttt{(}\$1 \texttt{=>} \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{))}\ \texttt{@}\mathbf{any} : \mathsf{time}\ \mathbf{exit}\texttt{(}\$1 \texttt{=>} \mathit{true}\texttt{)}\ \mathbf{start}\langle 0\rangle \implies \textsf{\textit{guarded}}\,\langle\!|\ |\!\rangle}\ (7.27)$$

That is, $outP\texttt{(!3)}$ is a behaviour that does not finish immediately, but it has to do an action before finishing (*guarded*), and it does not return any value nor produce any bindings between variables and types ($\langle\!|\ |\!\rangle$).

The first premise is an application of the axiom 7.23, provided that the information

$$outP \Rightarrow \textsf{\textit{gate}}\langle\texttt{(}\$1 \texttt{=>} \mathsf{int}\texttt{)}\rangle$$

exists in the context $\mathcal{C}$.

The second premise is proved in Example 7.28 in Section 7.31.4. The third premise is fulfilled thanks to:

$$\frac{\mathcal{C} \vdash \mathsf{time} \Rightarrow \textsf{\textit{type}}}{\mathcal{C} \vdash (\mathbf{any} : \mathsf{time}\ \Rightarrow^{\mathbf{PM}}\ \mathsf{time})\ \mapsto\ \langle\!|\ |\!\rangle}\ (7.106)$$

Finally, the fourth premise is proved in the following way:

$$\cfrac{\cfrac{\cfrac{\mathcal{C} \vdash \mathit{true} \texttt{=>} \mathsf{bool}}{\mathcal{C} \vdash \mathit{true} \Rightarrow^{\mathbf{T}} \mathsf{bool}}\ (7.95)}{\mathcal{C} \vdash \$1 \texttt{=>} \mathit{true} \Rightarrow^{\mathbf{T}} \$1 \texttt{=>} \mathsf{bool}}\ (7.100)}{\mathcal{C} \vdash \mathbf{exit}\texttt{(}\$1 \texttt{=>} \mathit{true}\texttt{)} \implies \textsf{\textit{exit}}\,\langle\!|\, \$1 \texttt{=>} \mathsf{bool} \,|\!\rangle}\ (7.33)$$

Example 7.2 Let us study now a communication in which an integer is received, by considering the behaviour

$$inP\texttt{(?}x : \mathsf{int}\texttt{)}.$$

Its translation to abstract syntax is

$$inP\texttt{(}\$1 \texttt{=>} \texttt{?}x : \mathsf{int}\texttt{)}\ \texttt{@}\mathbf{any}\texttt{:}\mathsf{time}\ \mathbf{exit}\texttt{(}\$1 \texttt{=>} \mathit{true}\texttt{)}\ \mathbf{start}\langle 0\rangle.$$

By using rule 7.27, we can deduce that

$$\frac{\begin{array}{l}
\mathcal{C} \vdash \mathit{inP} \Rightarrow \mathsf{gate}\langle(\$1 \texttt{=>} \mathsf{int})\rangle \\
\mathcal{C} \vdash ((\$1 \texttt{=>} ?x : \mathsf{int}) \;\Rightarrow^{\mathbf{PM}}\; (\$1 \texttt{=>} \mathsf{int})) \;\mapsto\; \llparenthesis\, x \texttt{=>} \mathsf{int} \,\rrparenthesis \\
\mathcal{C} \vdash (\mathbf{any} : \mathsf{time} \;\Rightarrow^{\mathbf{PM}}\; \mathsf{time}) \;\mapsto\; \llparenthesis\;\rrparenthesis \\
\mathcal{C};\; x \texttt{=>} \mathsf{int} \vdash \mathbf{exit}(\$1 \texttt{=>} \mathit{true}) \;\Longrightarrow\; \mathsf{exit}\, \llparenthesis\, \$1 \texttt{=>} \mathsf{bool} \,\rrparenthesis \\
\mathcal{C} \vdash 0 \Rightarrow^{\mathbf{T}} \mathsf{time}
\end{array}}{\mathcal{C} \vdash \mathit{inP}(\$1 \texttt{=>} ?x : \mathsf{int})\; \texttt{@}\mathbf{any} : \mathsf{time}\; \mathbf{exit}(\$1 \texttt{=>} \mathit{true})\; \mathbf{start}\langle 0\rangle \;\Longrightarrow\; \mathsf{guarded}\, \llparenthesis\, x \texttt{=>} \mathsf{int} \,\rrparenthesis}\;(7.27)$$

that is, action $\mathit{inP}(?x : \mathsf{int})$ is a behaviour that finishes after doing an action, and it binds variable x with type int (after the execution of this action, it is known that variable x has a value of type int).

The first premise is proved by axiom 7.23, if the corresponding information is in the context $\mathcal{C}$. The second premise has the following proof:

$$\dfrac{\dfrac{\dfrac{\mathcal{C} \vdash \mathsf{int} \Rightarrow \mathsf{type} \quad \dfrac{}{\mathcal{C} \vdash \mathsf{int} \sqsubseteq \mathsf{int}}\,(7.19) \quad \dfrac{}{\mathcal{C} \vdash (?x \;\Rightarrow^{\mathbf{PM}}\; \mathsf{int}) \;\mapsto\; \llparenthesis\, x \texttt{=>} \mathsf{int} \,\rrparenthesis}\,(7.103)}{\mathcal{C} \vdash (?x\!:\!\mathsf{int} \;\Rightarrow^{\mathbf{PM}}\; \mathsf{int}) \;\mapsto\; \llparenthesis\, x \texttt{=>} \mathsf{int} \,\rrparenthesis}\,(7.110)}{\mathcal{C} \vdash (\$1 \texttt{=>} ?x : \mathsf{int} \;\Rightarrow^{\mathbf{PM}}\; \$1 \texttt{=>} \mathsf{int}) \;\mapsto\; \llparenthesis\, x \texttt{=>} \mathsf{int} \,\rrparenthesis}\,(7.112)}{\mathcal{C} \vdash ((\$1 \texttt{=>} ?x : \mathsf{int}) \;\Rightarrow^{\mathbf{PM}}\; (\$1 \texttt{=>} \mathsf{int})) \;\mapsto\; \llparenthesis\, x \texttt{=>} \mathsf{int} \,\rrparenthesis}\,(7.107)$$

where we have assumed that there is no information that restricts the type of variable x in the context $\mathcal{C}$.

The other premises are analogous to ones in the previous example.

Example 7.3 We can see now a more complicated communication through a gate, like

$$\mathit{outP}(\texttt{!}(x \texttt{=>} 2, y \texttt{=>} \mathit{true}))\; \texttt{@?}t,$$

which is translated to

$$\mathit{outP}(\$1 \texttt{=>} \texttt{!}\mathbf{exit}(\$1 \texttt{=>} (x \texttt{=>} 2, y \texttt{=>} \mathit{true})))\; \texttt{@?}t\; \mathbf{exit}(\$1 \texttt{=>} \mathit{true})\; \mathbf{start}\langle 0\rangle.$$

Let us imagine that the gate outP has been declared without type and the default type **any** has been given to it in the translation into abstract syntax, that is, the information

$$\mathit{outP} \Rightarrow \mathsf{gate}\langle(\$1 \texttt{=>} \mathbf{any})\rangle$$

is in the context $\mathcal{C}$.

We can use inference rule 7.27 to deduce that

$$\frac{\begin{array}{l}
\mathcal{C} \vdash \mathit{outP} \Rightarrow \mathsf{gate}\langle(\$1 \texttt{=>} \mathbf{any})\rangle \\
\mathcal{C} \vdash ((\$1 \texttt{=>} \texttt{!}\mathbf{exit}(\$1 \texttt{=>} (x \texttt{=>} 2, y \texttt{=>} \mathit{true}))) \;\Rightarrow^{\mathbf{PM}}\; (\$1 \texttt{=>} \mathbf{any})) \;\mapsto\; \llparenthesis\;\rrparenthesis \\
\mathcal{C} \vdash (?t \;\Rightarrow^{\mathbf{PM}}\; \mathsf{time}) \;\mapsto\; \llparenthesis\, t \texttt{=>} \mathsf{time} \,\rrparenthesis \\
\mathcal{C}; t \texttt{=>} \mathsf{time} \vdash \mathbf{exit}(\$1 \texttt{=>} \mathit{true}) \;\Longrightarrow\; \mathsf{exit}\, \llparenthesis\, \$1 \texttt{=>} \mathsf{bool} \,\rrparenthesis \\
\mathcal{C} \vdash 0 \Rightarrow^{\mathbf{T}} \mathsf{time}
\end{array}}{\begin{array}{l}\mathcal{C} \vdash \mathit{outP}(\$1 \texttt{=>} \texttt{!}\mathbf{exit}(\$1 \texttt{=>} (x \texttt{=>} 2, y \texttt{=>} \mathit{true})))\; \texttt{@?}t\; \mathbf{exit}(\$1 \texttt{=>} \mathit{true})\; \mathbf{start}\langle 0\rangle \\ \qquad \Longrightarrow\; \mathsf{guarded}\, \llparenthesis\, t \texttt{=>} \mathsf{time} \,\rrparenthesis\end{array}}\;(7.27)$$

The second premise is proved in Example 7.29 in Section 7.31.4, and the third premise is fulfilled thanks to:

$$\frac{}{\mathcal{C} \vdash (?t \;\Rightarrow^{\mathbf{PM}}\; \mathsf{time}) \;\mapsto\; \llparenthesis\, t \texttt{=>} \mathsf{time} \,\rrparenthesis}\;(7.102)$$

assuming that variable t has been declared of type time.

The other premises are analogous to those of previous examples.

7.3 Sequential composition

The rules for the sequential composition $B_1\,;B_2$ state that composition is well-formed if both B_1 and B_2 are, that it returns bindings produced by behaviour B_1 overridden by those produced by behaviour B_2, and that it is "guarded" if at least one of them is. The inference rules are as follows:

$$\frac{\begin{array}{c}\mathcal{C} \vdash B_1 \implies \textsf{\textit{exit}}\,\llparenthesis\, RT_1 \,\rrparenthesis \\ \mathcal{C}; RT_1 \vdash B_2 \implies \textsf{\textit{exit}}\,\llparenthesis\, RT_2 \,\rrparenthesis\end{array}}{\mathcal{C} \vdash B_1 \;\; ; \;\; B_2 \implies \textsf{\textit{exit}}\,\llparenthesis\, RT_1; RT_2 \,\rrparenthesis} \tag{7.28}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash B_1 \implies \textsf{\textit{exit}}\,\llparenthesis\, RT_1 \,\rrparenthesis \\ \mathcal{C}; RT_1 \vdash B_2 \implies \textsf{\textit{guarded}}\,\llparenthesis\, RT_2 \,\rrparenthesis\end{array}}{\mathcal{C} \vdash B_1 \;\; ; \;\; B_2 \implies \textsf{\textit{guarded}}\,\llparenthesis\, RT_1; RT_2 \,\rrparenthesis} \tag{7.29}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash B_1 \implies \textsf{\textit{guarded}}\,\llparenthesis\, RT_1 \,\rrparenthesis \\ \mathcal{C}; RT_1 \vdash B_2 \implies \textsf{\textit{exit}}\,\llparenthesis\, RT_2 \,\rrparenthesis\end{array}}{\mathcal{C} \vdash B_1 \;\; ; \;\; B_2 \implies \textsf{\textit{guarded}}\,\llparenthesis\, RT_1; RT_2 \,\rrparenthesis} \tag{7.30}$$

We have to note that in all three cases, the second behaviour B_2 is executed in a context where bindings RT_1 produced by the first one B_1 have been included (overriding context $\mathcal{C}$). If both behaviours are guarded, then we can use first the rule 7.14 with one of them, and then follow using rule 7.29 or 7.30 (see next example).

Example 7.4 We can use these rules to know what kind of behaviour

$$inP\texttt{(?}x : \textsf{int}\texttt{)}\,;\; outP\texttt{(!}x\texttt{)}$$

is. It is translated to abstract syntax as follows:

$$\begin{array}{l} inP\texttt{(}\$1\;\texttt{=>}\;\texttt{?}x : \textsf{int}\texttt{)}\;\texttt{@}\mathbf{any} : \textsf{time}\;\mathbf{exit}\texttt{(}\$1\;\texttt{=>}\;\mathit{true}\texttt{)}\;\mathbf{start}\langle 0\rangle\,\texttt{;} \\ outP\texttt{(}\$1\;\texttt{=>}\;\texttt{!}\mathbf{exit}\texttt{(}\$1\;\texttt{=>}\;x\texttt{))}\;\texttt{@}\mathbf{any} : \textsf{time}\;\mathbf{exit}\texttt{(}\$1\;\texttt{=>}\;\mathit{true}\texttt{)}\;\mathbf{start}\langle 0\rangle \end{array}$$

By using inference rule 7.29 we can deduce that

$$\frac{\begin{array}{c}\mathcal{C} \vdash inP\texttt{(}\$1\;\texttt{=>}\;\texttt{?}x : \textsf{int}\texttt{)}\;\ldots \implies \textsf{\textit{exit}}\,\llparenthesis\, x\;\texttt{=>}\;\textsf{int} \,\rrparenthesis \\ \mathcal{C}; x\;\texttt{=>}\;\textsf{int} \vdash outP\texttt{(}\$1\;\texttt{=>}\;\texttt{!}\mathbf{exit}\texttt{(}\$1\;\texttt{=>}\;x\texttt{))}\;\ldots \implies \textsf{\textit{guarded}}\,\llparenthesis\;\rrparenthesis\end{array}}{\mathcal{C} \vdash inP\texttt{(}\$1\;\texttt{=>}\;\texttt{?}x : \textsf{int}\texttt{)}\;\ldots\,;\; outP\texttt{(}\$1\;\texttt{=>}\;\texttt{!}\mathbf{exit}\texttt{(}\$1\;\texttt{=>}\;x\texttt{))}\;\ldots \implies \textsf{\textit{guarded}}\,\llparenthesis\, x\;\texttt{=>}\;\textsf{int} \,\rrparenthesis} \;(7.29)$$

(where $\ldots$ stands for $\texttt{@}\mathbf{any} : \textsf{time}\;\mathbf{exit}\texttt{(}\$1\;\texttt{=>}\;\mathit{true}\texttt{)}\;\mathbf{start}\langle 0\rangle$).

We can deduce the first premise from Example 7.2, by using the general rule 7.14:

$$\frac{\mathcal{C} \vdash inP\texttt{(}\$1\;\texttt{=>}\;\texttt{?}x : \textsf{int}\texttt{)}\;\ldots \implies \textsf{\textit{guarded}}\,\llparenthesis\, x\;\texttt{=>}\;\textsf{int} \,\rrparenthesis}{\mathcal{C} \vdash inP\texttt{(}\$1\;\texttt{=>}\;\texttt{?}x : \textsf{int}\texttt{)}\;\ldots \implies \textsf{\textit{exit}}\,\llparenthesis\, x\;\texttt{=>}\;\textsf{int} \,\rrparenthesis} \;(7.14)$$

The second premise has the following proof:

$$
\dfrac{\cdots \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{C}; x \;\texttt{=>}\; \mathsf{int} \vdash x \;\texttt{=>}\; \mathsf{int}}\,(7.23)}{\mathcal{C}; x \;\texttt{=>}\; \mathsf{int} \vdash x \Rightarrow^{\mathrm{T}} \mathsf{int}}\,(7.96)}{\mathcal{C}; x \;\texttt{=>}\; \mathsf{int} \vdash \$1 \;\texttt{=>}\; x \Rightarrow^{\mathrm{T}} \$1 \;\texttt{=>}\; \mathsf{int}}\,(7.100)}{\mathcal{C}; x \;\texttt{=>}\; \mathsf{int} \vdash \mathbf{exit}(\$1 \;\texttt{=>}\; x) \;\Longrightarrow\; \mathit{exit} \llparenthesis \$1 \;\texttt{=>}\; \mathsf{int} \rrparenthesis}\,(7.33)}{\mathcal{C}; x \;\texttt{=>}\; \mathsf{int} \vdash (\,!\mathbf{exit}(\$1 \;\texttt{=>}\; x) \;\Rightarrow^{\mathrm{PM}}\; \mathsf{int}) \;\mapsto\; \llparenthesis\ \rrparenthesis}\,(7.105)}{\mathcal{C}; x \;\texttt{=>}\; \mathsf{int} \vdash (\$1 \;\texttt{=>}\; !\mathbf{exit}(\$1 \;\texttt{=>}\; x) \;\Rightarrow^{\mathrm{PM}}\; \$1 \;\texttt{=>}\; \mathsf{int}) \;\mapsto\; \llparenthesis\ \rrparenthesis}\,(7.112)}{\mathcal{C}; x \;\texttt{=>}\; \mathsf{int} \vdash ((\$1 \;\texttt{=>}\; !\mathbf{exit}(\$1 \;\texttt{=>}\; x)) \;\Rightarrow^{\mathrm{PM}}\; (\$1 \;\texttt{=>}\; \mathsf{int})) \;\mapsto\; \llparenthesis\ \rrparenthesis}\,(7.107) \qquad \cdots}{\mathcal{C}; x \;\texttt{=>}\; \mathsf{int} \vdash \mathit{outP}(\$1 \;\texttt{=>}\; !\mathbf{exit}(\$1 \;\texttt{=>}\; x)) \;\ldots \;\Longrightarrow\; \mathit{guarded} \llparenthesis\ \rrparenthesis}\,(7.27)
$$

where absent premises are analogous to others we have already seen. Note that in order to use variable x as a value, the information x => int has to be present in the context.

7.4 Selection operator

Behaviours B_1 and B_2 composed in a selection like B_1 [] B_2 cannot finish without doing an action, that is, they have to be guarded. In this way, it is this action which decides the election (by synchronization with another process). Variables which are modified only by B_1 or B_2 must have been initialized before the selection, and the behaviour cannot change their types. Thus, it can be assured that independently of the behaviour executed, initialized variables after the selection are always the same. Variables which are modified by both behaviours may have been initialized previously or not, but both behaviours have to bind them to the same type. If this is the case, then selection B_1 [] B_2 returns bindings produced by both, and we know that it does an action before finishing.

The inference rule that formalizes these conditions is:

$$
\dfrac{\begin{array}{c}\mathcal{C}, RT_1, RT_2 \vdash B_1 \;\Longrightarrow\; \mathit{guarded} \llparenthesis RT_1, RT \rrparenthesis \\ \mathcal{C}, RT_1, RT_2 \vdash B_2 \;\Longrightarrow\; \mathit{guarded} \llparenthesis RT_2, RT \rrparenthesis\end{array}}{\mathcal{C}, RT_1, RT_2 \vdash B_1 \;\texttt{[]}\; B_2 \;\Longrightarrow\; \mathit{guarded} \llparenthesis RT_1, RT_2, RT \rrparenthesis} \qquad (7.31)
$$

where RT_1 consists of the initialized variables (present in the context) modified only by B_1, RT_2 consists of the initialized variables modified only by B_2, and RT consists of the variables modified by both B_1 and B_2, and they may be in the context or not, that is, they may be initialized or not. Note that RT_1, RT_2, and RT are pairwise disjoint.

Example 7.5 In order to show the application of this rule, let us see if the behaviour

```
 in2(?x2 : data); out1(!x1)
[]
 out1(!x1); in2(?x2 : data)
```

is correct in a context $\mathcal{C} = \mathcal{C}', x1$ => data (where, among other things, there is the information $x1$ => data, that is, variable $x1$ has a value of type data) and what bindings it produces. The behaviour is translated to

```
selection ≡
          in2($1 => ?x2 : data) ...; out1($1 => !exit($1 => x1)) ...
        []
          out1($1 => !exit($1 => x1)) ...; in2($1 => ?x2 : data) ...
```

where, everywhere, ... stands for **@any**:time **exit**($\$1$ => $true$) **start**$\langle 0 \rangle$.

First we use rule 7.29 twice to know if the sequential compositions are correct, and what they return. We can deduce that

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathit{in2}(\$1\ \mathsf{=>}\ ?x2 : \mathsf{data})\ \ldots \implies \mathit{exit}\,\llparenthesis\, x2\ \mathsf{=>}\ \mathsf{data}\,\rrparenthesis \\ \mathcal{C}; x2\ \mathsf{=>}\ \mathsf{data} \vdash \mathit{out1}(\$1\ \mathsf{=>}\ !\mathbf{exit}(\$1\ \mathsf{=>}\ x1))\ \ldots \implies \mathit{guarded}\,\llparenthesis\ \rrparenthesis\end{array}}{\begin{array}{l}\mathcal{C} \vdash \mathit{in2}(\$1\ \mathsf{=>}\ ?x2 : \mathsf{data})\ \ldots ;\ \mathit{out1}(\$1\ \mathsf{=>}\ !\mathbf{exit}(\$1\ \mathsf{=>}\ x1))\ \ldots \\ \qquad \implies \mathit{guarded}\,\llparenthesis\, x2\ \mathsf{=>}\ \mathsf{data}\,\rrparenthesis\end{array}} \quad (7.29)$$

We also deduce that

$$\frac{\begin{array}{c}\mathcal{C} \vdash \mathit{out1}(\$1\ \mathsf{=>}\ !\mathbf{exit}(\$1\ \mathsf{=>}\ x1))\ \ldots \implies \mathit{guarded}\,\llparenthesis\ \rrparenthesis \\ \mathcal{C} \vdash \mathit{in2}(\$1\ \mathsf{=>}\ ?x2 : \mathsf{data})\ \ldots \implies \mathit{exit}\,\llparenthesis\, x2\ \mathsf{=>}\ \mathsf{data}\,\rrparenthesis\end{array}}{\mathcal{C} \vdash \mathit{out1}(\$1\ \mathsf{=>}\ !\mathbf{exit}(\$1\ \mathsf{=>}\ x1))\ \ldots ;\ \mathit{in2}(\$1\ \mathsf{=>}\ ?x2 : \mathsf{data})\ \ldots \implies \mathit{guarded}\,\llparenthesis\, x2\ \mathsf{=>}\ \mathsf{data}\,\rrparenthesis} \quad (7.30)$$

Now, we can use rule 7.31 in order to deduce

$$\frac{\begin{array}{c}\mathcal{C} \vdash \mathit{in2}(\$1\ \mathsf{=>}\ ?x2 : \mathsf{data})\ \ldots ;\ \mathit{out1}(\$1\ \mathsf{=>}\ !\mathbf{exit}(\$1\ \mathsf{=>}\ x1))\ \ldots \implies \mathit{guarded}\,\llparenthesis\, x2\ \mathsf{=>}\ \mathsf{data}\,\rrparenthesis \\ \mathcal{C} \vdash \mathit{out1}(\$1\ \mathsf{=>}\ !\mathbf{exit}(\$1\ \mathsf{=>}\ x1))\ \ldots ;\ \mathit{in2}(\$1\ \mathsf{=>}\ ?x2 : \mathsf{data})\ \ldots \implies \mathit{guarded}\,\llparenthesis\, x2\ \mathsf{=>}\ \mathsf{data}\,\rrparenthesis\end{array}}{\mathcal{C} \vdash \mathit{selection} \implies \mathit{guarded}\,\llparenthesis\, x2\ \mathsf{=>}\ \mathsf{data}\,\rrparenthesis} \quad (7.31)$$

where we consider the application of the rule when RT_1 and RT_2 are empty and RT is equal to $x2\,\mathsf{=>}\,\mathsf{data}$.

Example 7.6 Let us see now an example where a variable which has already a value is modified only in one part of the selection. Let us consider the following behaviour

```
inP(?x : int);
( inP(?x : int)
[]
 outP(!x) )
```

which is translated to

```
selection2 ≡
    inP($1 => ?x : int) ...;
    ( inP($1 => ?x : int) ...
    []
     outP($1 => !exit($1 => x)) ...)
```

Note that we use parentheses () to make the abstract syntax behaviour clearer, although they have no meaning.

We have to use the sequential composition rule 7.29 to deduce

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathit{inP}(\$1\ \mathsf{=>}\ ?x : \mathsf{int})\ \ldots \implies \mathit{exit}\,\llparenthesis\, x\ \mathsf{=>}\ \mathsf{int}\,\rrparenthesis \\ \mathcal{C}; x\ \mathsf{=>}\ \mathsf{int} \vdash \mathit{inP}(\$1\ \mathsf{=>}\ ?x : \mathsf{int})\ \ldots\ \texttt{[]}\ \mathit{outP}(\$1\ \mathsf{=>}\ !\mathbf{exit}(\$1\ \mathsf{=>}\ x))\ \ldots \implies \mathit{guarded}\,\llparenthesis\, x\ \mathsf{=>}\ \mathsf{int}\,\rrparenthesis\end{array}}{\mathcal{C} \vdash \mathit{selection2} \implies \mathit{guarded}\,\llparenthesis\, x\ \mathsf{=>}\ \mathsf{int}\,\rrparenthesis} \quad (7.29)$$

where we can deduce the second premise by using rule 7.31 as follows:

$$\frac{\begin{array}{l}\mathcal{C}; x\ \mathsf{=>}\ \mathsf{int} \vdash \mathit{inP}(\$1\ \mathsf{=>}\ ?x : \mathsf{int})\ \ldots \implies \mathit{guarded}\,\llparenthesis\, x\ \mathsf{=>}\ \mathsf{int}\,\rrparenthesis \\ \mathcal{C}; x\ \mathsf{=>}\ \mathsf{int} \vdash \mathit{outP}(\$1\ \mathsf{=>}\ !\mathbf{exit}(\$1\ \mathsf{=>}\ x))\ \ldots \implies \mathit{guarded}\,\llparenthesis\ \rrparenthesis\end{array}}{\mathcal{C}; x\ \mathsf{=>}\ \mathsf{int} \vdash \mathit{inP}(\$1\ \mathsf{=>}\ ?x : \mathsf{int})\ \ldots\ \texttt{[]}\ \mathit{outP}(\$1\ \mathsf{=>}\ !\mathbf{exit}(\$1\ \mathsf{=>}\ x))\ \ldots \implies \mathit{guarded}\,\llparenthesis\, x\ \mathsf{=>}\ \mathsf{int}\,\rrparenthesis} \quad (7.31)$$

with $RT_1 = x\ \mathsf{=>}\ \mathsf{int}$, and RT_2 and RT both equal to the empty list of bindings.

Example 7.7 Let us see now that the behaviour (?x:=5 [] ?x:=2); **wait**(1) is not correct, by showing that we cannot deduce what kind of behaviour it is. It is translated to abstract syntax as

$$(?x{:=}\mathbf{exit}(\$1 \texttt{=>} 5)\ \texttt{[]}\ ?x{:=}\mathbf{exit}(\$1 \texttt{=>} 2));\ \mathbf{wait}(\mathbf{exit}(\$1 \texttt{=>} 1)).$$

First, we have to use one of the rules of the sequential composition.

$$\frac{\begin{array}{c} \mathcal{C} \vdash ?x{:=}\mathbf{exit}(\$1 \texttt{=>} 5)\ \texttt{[]}\ ?x{:=}\mathbf{exit}(\$1 \texttt{=>} 2) \implies \mathbf{?} \\ \mathcal{C}; RT \vdash \mathbf{wait}(\mathbf{exit}(\$1 \texttt{=>} 1)) \implies \mathit{exit} \llparenthesis\ \rrparenthesis \end{array}}{\mathcal{C} \vdash (?x{:=}\mathbf{exit}(\$1 \texttt{=>} 5)\ \texttt{[]}\ ?x{:=}\mathbf{exit}(\$1 \texttt{=>} 2));\ \mathbf{wait}(\mathbf{exit}(\$1 \texttt{=>} 1)) \implies \mathbf{?}}\ ()$$

In order to prove the first premise we have to use rule 7.31, but we cannot because the assignments are not guarded.

$$\frac{\begin{array}{c} \mathcal{C} \vdash ?x{:=}\mathbf{exit}(\$1 \texttt{=>} 5) \implies \mathit{exit} \llparenthesis x \texttt{=>} \mathsf{int} \rrparenthesis \\ \mathcal{C} \vdash ?x{:=}\mathbf{exit}(\$1 \texttt{=>} 2) \implies \mathit{exit} \llparenthesis x \texttt{=>} \mathsf{int} \rrparenthesis \end{array}}{\mathcal{C} \vdash ?x{:=}\mathbf{exit}(\$1 \texttt{=>} 5)\ \texttt{[]}\ ?x{:=}\mathbf{exit}(\$1 \texttt{=>} 2) \implies \mathbf{?}}\ ()$$

That ?x:=5 is an *exit* $\llparenthesis x \texttt{=>} \mathsf{int} \rrparenthesis$ behaviour is proved in Example 7.21 in Section 7.20.

This is the only way of trying to "check" this behaviour because we cannot apply any other inference rule. Thus, this behaviour does not pass the static semantics, that is, it has no meaning.

7.5 Internal action

The internal action **i** represents a behaviour that does not produce any binding, and that carries out an action before finishing. The following rule formalizes this:

$$\frac{}{\mathcal{C} \vdash \mathbf{i} \implies \mathit{guarded} \llparenthesis\ \rrparenthesis} \tag{7.32}$$

Example 7.8 As an example of the application of this rule, we can check if the behaviour

$\mathit{out2}$(!$x2$) [] **i**

which is translated to abstract syntax as

$\mathit{out2}(\$1 \texttt{=>}\ !\mathbf{exit}(\$1 \texttt{=>} x2))\ \ldots\ \texttt{[]}\ \mathbf{i}$

is semantically correct in a context where the information $x2$ => data is included. We have to do the following proof:

$$\frac{\begin{array}{c} \mathcal{C}, x2 \texttt{=>} \mathsf{data} \vdash \mathit{out2}(\$1 \texttt{=>}\ !\mathbf{exit}(\$1 \texttt{=>} x2))\ \ldots \implies \mathit{guarded} \llparenthesis\ \rrparenthesis \\ \mathcal{C}, x2 \texttt{=>} \mathsf{data} \vdash \mathbf{i} \implies \mathit{guarded} \llparenthesis\ \rrparenthesis \end{array}}{\mathcal{C}, x2 \texttt{=>} \mathsf{data} \vdash \mathit{out2}(\$1 \texttt{=>}\ !\mathbf{exit}(\$1 \texttt{=>} x2))\ \ldots\ \texttt{[]}\ \mathbf{i} \implies \mathit{guarded} \llparenthesis\ \rrparenthesis}\ (7.31)$$

where, the second premise is proved using the previous rule:

$$\frac{}{\mathcal{C}, x2 \texttt{=>} \mathsf{data} \vdash \mathbf{i} \implies \mathit{guarded} \llparenthesis\ \rrparenthesis}\ (7.32)$$

7.6 Successful termination with values

exit`(`RN`)` is a behaviour that finishes immediately, returning the values indicated by RN. If RN is of the form $V_1 \texttt{=>} N_1, \ldots, V_n \texttt{=>} N_n$, then **exit**`(`$RN$`)` produces the bindings $V_1 \texttt{=>} T_1, \ldots, V_n \texttt{=>} T_n$, where T_i is the type of the value N_i. The following rule explains what kind of behaviour **exit**`(`RN`)` is:

$$\frac{\mathcal{C} \vdash RN \Rightarrow^{\mathrm{T}} RT}{\mathcal{C} \vdash \textbf{exit}\texttt{(}RN\texttt{)} \implies \mathsf{exit}\,\llparenthesis\, RT \,\rrparenthesis} \tag{7.33}$$

Example 7.9 The value expression 3 is translated into the behaviour **exit**`(`$1`=>`3`)` which is a behaviour that *returns* an integer value (note the use of $1 to represent this), as stated in the following proof.

$$\dfrac{\dfrac{\dfrac{\mathcal{C} \vdash 3 \texttt{=>} \mathsf{int}}{\mathcal{C} \vdash 3 \Rightarrow^{\mathrm{T}} \mathsf{int}}\,(7.95)}{\mathcal{C} \vdash \$1 \texttt{=>} 3 \Rightarrow^{\mathrm{T}} \$1 \texttt{=>} \mathsf{int}}\,(7.100)}{\mathcal{C} \vdash \textbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{)} \implies \mathsf{exit}\,\llparenthesis\, \$1 \texttt{=>} \mathsf{int} \,\rrparenthesis}\,(7.33)$$

7.7 Inaction

stop is a behaviour that cannot do anything, and it does not finish. It is expressed by saying that **stop** returns a value of type **none**. Since **none** has no value, it says that **stop** does not finish.

$$\frac{}{\mathcal{C} \vdash \textbf{stop} \implies \mathsf{guarded}\,\llparenthesis\, \$1 \texttt{=>} \textbf{none} \,\rrparenthesis} \tag{7.34}$$

We think that it is not a good idea to represent the no termination of **stop** with the binding $1=>**none**. This makes that the behaviour G`(?`x`:int) [] `**stop** is considered semantically incorrect because **stop** produces the "binding" $1 `=>` **none**, which is not in the context, and which cannot be produced by G`(?`x`:int)`.

One solution is to make **stop** a *guarded* ⦇ ⦈ behaviour, and do not care about the no termination. Another solution is to modify the semantics of the selection operator and to add the following rule

$$\frac{\begin{array}{c}\mathcal{C} \vdash B_1 \implies \mathsf{guarded}\,\llparenthesis\, RT \,\rrparenthesis \\ \mathcal{C} \vdash B_2 \implies \mathsf{guarded}\,\llparenthesis\, \$1 \texttt{=>} \textbf{none} \,\rrparenthesis\end{array}}{\mathcal{C} \vdash B_1 \;\texttt{[]}\; B_2 \implies \mathsf{guarded}\,\llparenthesis\, RT \,\rrparenthesis}$$

and symmetrically. This modifies the meaning of $B \implies \mathsf{guarded}\,\llparenthesis\, RT \,\rrparenthesis$ since now it would mean that *if B finishes* then it produces bindings RT. Analogous rules should be included in the semantics of the others branching operators (`[>`, **trap**, etc.).

7.8 Time block

From the point of view of the static semantics, **stop** and **block** are equal, that is, behaviours which do not finish.

$$\frac{}{\mathcal{C} \vdash \textbf{block} \implies \mathsf{guarded}\,\llparenthesis\, \$1 \texttt{=>} \textbf{none} \,\rrparenthesis} \tag{7.35}$$

7.9 Parallel operator

The parallel composition of two behaviours $B_1 \;\texttt{|[}G_1\texttt{,}\ldots\texttt{,}G_n\texttt{]|}\; B_2$ $(n \geq 0)$ is correct in the context $\mathcal{C}$, from the point of view of the static semantics, if both behaviours are correct and they return disjoint bindings, and if the gate identifiers $G_1, \ldots, G_n$ are declared in the context $\mathcal{C}$. The bindings returned by the composition are those produced by both behaviours (which must be disjoint), and the parallel composition is guarded if at least one of them is. The inference rules are:

$$\frac{\begin{array}{c}\mathcal{C} \vdash B_1 \Longrightarrow \textit{exit} \llparenthesis RT_1 \rrparenthesis \qquad \mathcal{C} \vdash B_2 \Longrightarrow \textit{exit} \llparenthesis RT_2 \rrparenthesis \\ \mathcal{C} \vdash G_1 \Rightarrow \textit{gate}\langle\texttt{(}RT_1'\texttt{)}\rangle \;\cdots\; \mathcal{C} \vdash G_n \Rightarrow \textit{gate}\langle\texttt{(}RT_n'\texttt{)}\rangle\end{array}}{\mathcal{C} \vdash B_1 \;\texttt{|[}G_1\texttt{,}\ldots\texttt{,}G_n\texttt{]|}\; B_2 \Longrightarrow \textit{exit} \llparenthesis RT_1, RT_2 \rrparenthesis} \tag{7.36}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash B_1 \Longrightarrow \textit{exit} \llparenthesis RT_1 \rrparenthesis \qquad \mathcal{C} \vdash B_2 \Longrightarrow \textit{guarded} \llparenthesis RT_2 \rrparenthesis \\ \mathcal{C} \vdash G_1 \Rightarrow \textit{gate}\langle\texttt{(}RT_1'\texttt{)}\rangle \;\cdots\; \mathcal{C} \vdash G_n \Rightarrow \textit{gate}\langle\texttt{(}RT_n'\texttt{)}\rangle\end{array}}{\mathcal{C} \vdash B_1 \;\texttt{|[}G_1\texttt{,}\ldots\texttt{,}G_n\texttt{]|}\; B_2 \Longrightarrow \textit{guarded} \llparenthesis RT_1, RT_2 \rrparenthesis} \tag{7.37}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash B_1 \Longrightarrow \textit{guarded} \llparenthesis RT_1 \rrparenthesis \qquad \mathcal{C} \vdash B_2 \Longrightarrow \textit{exit} \llparenthesis RT_2 \rrparenthesis \\ \mathcal{C} \vdash G_1 \Rightarrow \textit{gate}\langle\texttt{(}RT_1'\texttt{)}\rangle \;\cdots\; \mathcal{C} \vdash G_n \Rightarrow \textit{gate}\langle\texttt{(}RT_n'\texttt{)}\rangle\end{array}}{\mathcal{C} \vdash B_1 \;\texttt{|[}G_1\texttt{,}\ldots\texttt{,}G_n\texttt{]|}\; B_2 \Longrightarrow \textit{guarded} \llparenthesis RT_1, RT_2 \rrparenthesis} \tag{7.38}$$

where RT_1 and RT_2 must be disjoint, that is, it is not allowed that both behaviours bind the same variable.

Note that the behaviour $B_1 \;\texttt{|[}G_1\texttt{,}\ldots\texttt{,}G_n\texttt{]|}\; B_2$ is *guarded* if at least one of them is[2]. This is because the whole behaviour finishes when both has finished, so if one of them does an action before finishing, the whole behaviour does it too.

Example 7.10 Let us see if the behaviour $\textit{in2}\texttt{(?}x2\texttt{:data)} \;\texttt{|||}\; \textit{out1}\texttt{(!}x1\texttt{)}$ is semantically correct in a context $\mathcal{C} = \mathcal{C}', x1 \;\texttt{=>}\; \mathsf{data}$. It is translated to

$$\textit{in2}\texttt{(}\$1 \texttt{=> ?}x2\texttt{:data)}\ldots \;\texttt{|[]|}\; \textit{out1}\texttt{(}\$1 \texttt{=> !}\mathbf{exit}\texttt{(}\$1 \texttt{=>} x1\texttt{))}\ldots$$

By using rule 7.38, we have

$$\frac{\begin{array}{l}\mathcal{C} \vdash \textit{in2}\texttt{(}\$1 \texttt{=> ?}x2\texttt{:data)}\ldots \Longrightarrow \textit{guarded} \llparenthesis x2 \texttt{=>} \mathsf{data} \rrparenthesis \\ \mathcal{C} \vdash \textit{out1}\texttt{(}\$1 \texttt{=> !}\mathbf{exit}\texttt{(}\$1 \texttt{=>} x1\texttt{))}\ldots \Longrightarrow \textit{exit} \llparenthesis\, \rrparenthesis\end{array}}{\mathcal{C} \vdash \textit{in2}\texttt{(}\$1 \texttt{=> ?}x2\texttt{:data)}\ldots \;\texttt{|[]|}\; \textit{out1}\texttt{(}\$1 \texttt{=> !}\mathbf{exit}\texttt{(}\$1 \texttt{=>} x1\texttt{))}\ldots \Longrightarrow \textit{guarded} \llparenthesis x2 \texttt{=>} \mathsf{data} \rrparenthesis} \tag{7.38}$$

Example 7.11 We can prove that

$\textit{pd}\texttt{(!}x1\texttt{);}\; \textit{sync} \;\texttt{|[}\, \textit{sync} \,\texttt{]|}\; \textit{sync}\texttt{;}\; \textit{pd}\texttt{(!}x2\texttt{)}$

is semantically correct in a context with at least the following information:

$$\mathcal{C} = \mathcal{C}', \textit{pd} \Rightarrow \textit{gate}\langle\texttt{(}\$1 \texttt{=>} \mathsf{data}\texttt{)}\rangle, \textit{sync} \Rightarrow \textit{gate}\langle\texttt{(}\$1 \texttt{=>} \mathbf{any}\texttt{)}\rangle, x1 \texttt{=>} \mathsf{data}, x2 \texttt{=>} \mathsf{data}.$$

The behaviour is translated into

[2] Note that we do not need a rule for the case in which both behaviours are guarded, because in this case we can use rule 7.14 first with one of them.

$$\begin{array}{l}
\mathit{parallel} \equiv \\
\qquad\qquad \mathit{pd}(\$1 \texttt{ => } \texttt{!}\mathbf{exit}(\$1 \texttt{ => } x1))\ldots;\ \mathit{sync}(\$1 \texttt{ => } ())\ldots \\
\qquad \texttt{|[}\ \mathit{sync}\ \texttt{]|} \\
\qquad\qquad \mathit{sync}(\$1 \texttt{ => } ())\ldots;\ \mathit{pd}(\$1 \texttt{ => } \texttt{!}\mathbf{exit}(\$1 \texttt{ => } x2))\ldots
\end{array}$$

We can use rule 7.38

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathit{pd}(\$1 \texttt{ => } \texttt{!}\mathbf{exit}(\$1 \texttt{ => } x1))\ldots;\ \mathit{sync}(\$1 \texttt{ => } ())\ldots \implies \mathsf{guarded}\,\llparenthesis\,\rrparenthesis \\ \mathcal{C} \vdash \mathit{sync}(\$1 \texttt{ => } ())\ldots;\ \mathit{pd}(\$1 \texttt{ => } \texttt{!}\mathbf{exit}(\$1 \texttt{ => } x2))\ldots \implies \mathsf{exit}\,\llparenthesis\,\rrparenthesis \\ \mathcal{C} \vdash \mathit{sync} \Rightarrow \mathsf{gate}\langle(\$1 \texttt{ => } \mathbf{any})\rangle\end{array}}{\mathcal{C} \vdash \mathit{parallel} \implies \mathsf{guarded}\,\llparenthesis\,\rrparenthesis} \tag{7.38}$$

where the first and second premises are easily proved, and the third premise is an application of axiom 7.23.

However, the behaviour

$$\begin{array}{l}
\qquad \mathit{pd}(\texttt{?}x\texttt{:}\mathsf{data});\ \mathit{sync} \\
\texttt{|[}\ \mathit{sync}\ \texttt{]|} \\
\qquad \mathit{sync};\ \mathit{pd}(\texttt{!}x)
\end{array}$$

is not correct in a context $\mathcal{C}$ where there is no information about x having a value because although

$$\mathcal{C} \vdash \mathit{pd}(\$1 \texttt{ => } \texttt{?}x\texttt{:}\mathsf{data})\ldots;\ \mathit{sync}(\$1 \texttt{ => } ())\ldots \implies \mathsf{guarded}\,\llparenthesis\, x \texttt{ => } \mathsf{data} \,\rrparenthesis$$

the binding $x \texttt{ => } \mathsf{data}$ is not propagated to $\mathit{sync};\ \mathit{pd}(\texttt{!}x)$. Thus

$$\mathit{sync}(\$1 \texttt{ => } ())\ldots;\ \mathit{pd}(\$1 \texttt{ => } \texttt{!}\mathbf{exit}(\$1 \texttt{ => } x))\ldots$$

is not semantically correct.

7.10 Synchronization operator

The static semantics rules for the synchronization operator are similar to those of the parallel operator.

$$\frac{\mathcal{C} \vdash B_1 \implies \mathsf{exit}\,\llparenthesis\, RT_1 \,\rrparenthesis \qquad \mathcal{C} \vdash B_2 \implies \mathsf{exit}\,\llparenthesis\, RT_2 \,\rrparenthesis}{\mathcal{C} \vdash B_1 \texttt{ || } B_2 \implies \mathsf{exit}\,\llparenthesis\, RT_1, RT_2 \,\rrparenthesis} \tag{7.39}$$

$$\frac{\mathcal{C} \vdash B_1 \implies \mathsf{exit}\,\llparenthesis\, RT_1 \,\rrparenthesis \qquad \mathcal{C} \vdash B_2 \implies \mathsf{guarded}\,\llparenthesis\, RT_2 \,\rrparenthesis}{\mathcal{C} \vdash B_1 \texttt{ || } B_2 \implies \mathsf{guarded}\,\llparenthesis\, RT_1, RT_2 \,\rrparenthesis} \tag{7.40}$$

$$\frac{\mathcal{C} \vdash B_1 \implies \mathsf{guarded}\,\llparenthesis\, RT_1 \,\rrparenthesis \qquad \mathcal{C} \vdash B_2 \implies \mathsf{exit}\,\llparenthesis\, RT_2 \,\rrparenthesis}{\mathcal{C} \vdash B_1 \texttt{ || } B_2 \implies \mathsf{guarded}\,\llparenthesis\, RT_1, RT_2 \,\rrparenthesis} \tag{7.41}$$

where RT_1 and RT_2 must be disjoint.

Like the parallel operator, $B_1 \texttt{ || } B_2$ is guarded if at least B_1 or B_2 is.

7.11 General parallel operator

We use the notation $\vec{G_i}$ to abbreviate the gate list $G_{i1}\texttt{,}\ldots\texttt{,}G_{ir_i}$.

A general parallel composition

$$\begin{array}{l} \textbf{par } G_1\#n_1\texttt{,} \ldots \texttt{,} G_p\#n_p \textbf{ in} \\ \quad \texttt{[} \vec{G_1} \texttt{]} \texttt{ -> } B_1 \\ \ldots \\ \texttt{||} \texttt{[} \vec{G_m} \texttt{]} \texttt{ -> } B_m \end{array}$$

is correct in context $\mathcal{C}$ if all the gate identifiers are declared in the context $\mathcal{C}$, all the B_i are correct behaviours, and they produce pairwise disjoint bindings. This is formalized with the following rule:

$$\frac{\begin{array}{l} \mathcal{C} \vdash G_1 \Rightarrow \textsf{gate}\langle(RT_1)\rangle \ \cdots\ \mathcal{C} \vdash G_p \Rightarrow \textsf{gate}\langle(RT_p)\rangle \\ \forall\, i\,.\,1 \leq i \leq m\,.\forall\, j\,.\,1 \leq j \leq r_i\,.\mathcal{C} \vdash G_{ij} \Rightarrow \textsf{gate}\langle(RT_{ij})\rangle \\ \mathcal{C} \vdash B_1 \Longrightarrow \textsf{exit}\llparenthesis RT'_1 \rrparenthesis \ \cdots\ \mathcal{C} \vdash B_m \Longrightarrow \textsf{exit}\llparenthesis RT'_m \rrparenthesis \end{array}}{\begin{array}{l} \mathcal{C} \vdash (\textbf{par } G_1\#n_1\texttt{,} \ldots \texttt{,} G_p\#n_p \textbf{ in} \\ \qquad \texttt{[} \vec{G_1} \texttt{]} \texttt{ -> } B_1 \\ \qquad \ldots \\ \qquad \texttt{||} \texttt{[} \vec{G_m} \texttt{]} \texttt{ -> } B_m) \Longrightarrow \textsf{exit}\llparenthesis RT'_1, \ldots, RT'_m \rrparenthesis \end{array}} \tag{7.42}$$

with the side condition that $RT'_1, \ldots, RT'_m$ must be pairwise disjoint. We have to ensure also that if gate G_i is in the list of degrees (G_1**#**n_1**,** $\ldots$ **,**G_p**#**n_p) with degree n_i, then there are at least n_i behaviours that have G_i in their synchronization list $\vec{G_j}$. So, we add the following side conditions

$$\begin{array}{c} n_1 \leq |\,\{\vec{G_j} \mid G_1 \in \vec{G_j}, 1 \leq j \leq m\}\,| \\ \vdots \\ n_p \leq |\,\{\vec{G_j} \mid G_p \in \vec{G_j}, 1 \leq j \leq m\}\,| \end{array}$$

where n_i's must be natural numbers greater than zero.

If at least one of the behaviours B_i is guarded then the general parallel composition is guarded:

$$\frac{\begin{array}{l} \mathcal{C} \vdash G_1 \Rightarrow \textsf{gate}\langle(RT_1)\rangle \ \cdots\ \mathcal{C} \vdash G_p \Rightarrow \textsf{gate}\langle(RT_p)\rangle \\ \mathcal{C} \vdash G_{ij} \Rightarrow \textsf{gate}\langle(RT_{ij})\rangle \quad \forall\, 1 \leq i \leq m \quad \forall\, 1 \leq j \leq r_i \\ \mathcal{C} \vdash B_1 \Longrightarrow \textsf{exit}\llparenthesis RT'_1 \rrparenthesis \\ \qquad \ldots \\ \mathcal{C} \vdash B_j \Longrightarrow \textsf{guarded}\llparenthesis RT'_j \rrparenthesis \\ \qquad \ldots \\ \mathcal{C} \vdash B_m \Longrightarrow \textsf{exit}\llparenthesis RT'_m \rrparenthesis \end{array}}{\begin{array}{l} \mathcal{C} \vdash (\textbf{par } G_1\#n_1\texttt{,} \ldots \texttt{,} G_p\#n_p \textbf{ in} \\ \qquad \texttt{[} \vec{G_1} \texttt{]} \texttt{ -> } B_1 \\ \qquad \ldots \\ \qquad \texttt{||} \texttt{[} \vec{G_m} \texttt{]} \texttt{ -> } B_m) \Longrightarrow \textsf{guarded}\llparenthesis RT'_1, \ldots, RT'_m \rrparenthesis \end{array}} \tag{7.43}$$

with the same side condition.

Example 7.12 Let us check that the general parallel behaviour

$$\begin{array}{l} \textit{generalPar} \ \equiv \\ \qquad \textbf{par } G\textbf{\#2 in} \ \ \texttt{[}G\texttt{,}G'\texttt{]} \texttt{ -> } P_1 \texttt{ || } \texttt{[}G\texttt{,}G'\texttt{]} \texttt{ -> } P_2 \texttt{ || } \texttt{[}G\texttt{,}G'\texttt{]} \texttt{ -> } P_3 \end{array}$$

is semantically correct assuming that behaviour P_i are correct $\textsf{guarded}\llparenthesis\ \rrparenthesis$ behaviours, and gates G and G' are declared in the context $\mathcal{C}$. We can use rule 7.43

$$\frac{\begin{array}{l}\mathcal{C} \vdash G \Rightarrow \textsf{\textit{gate}}\langle(\$1 \texttt{ => } \textsf{int})\rangle \mathcal{C} \vdash G' \Rightarrow \textsf{\textit{gate}}\langle(\$1 \texttt{ => } \textsf{int})\rangle \\ \mathcal{C} \vdash P_1 \implies \textsf{\textit{exit}} \llparenthesis\ \rrparenthesis \\ \mathcal{C} \vdash P_2 \implies \textsf{\textit{exit}} \llparenthesis\ \rrparenthesis \\ \mathcal{C} \vdash P_3 \implies \textsf{\textit{guarded}} \llparenthesis\ \rrparenthesis\end{array}}{\mathcal{C} \vdash \mathit{generalPar} \implies \textsf{\textit{guarded}} \llparenthesis\ \rrparenthesis} \qquad (7.43)$$

since $2 \leq 3$.

7.12 Parallel over values

A parallel over values behaviour **par** P **in** N `|||` B is semantically correct in the context $\mathcal{C}$ if the value N has type $\textsf{List}$ (the predefined type of lists of any type); the pattern P can be matched against the general type **any**; and the behaviour B is correct in the context $\mathcal{C}$ overridden with the bindings produced by the previous pattern-matching and it does not produce bindings. B is required to not produce bindings because the parallel over values represents the interleaving of a series of instantiations of B, one for each value of N. If B produce bindings, all the instantiations would produce the same bindings, and we would have an interleaving of behaviours which do not produce disjoint bindings as required in Section 7.9.

The whole behaviour **par** P **in** N `|||` B is guarded if B is guarded.

A behaviour can produce no bindings because it finishes and does not produce bindings (it is an $\textsf{\textit{exit}} \llparenthesis\ \rrparenthesis$ behaviour), or because it does not finish (it is an $\textsf{\textit{exit}} \llparenthesis \$1 \texttt{ => } \textbf{none} \rrparenthesis$ behaviour). The following rules covers all the cases:

$$\frac{\begin{array}{l}\mathcal{C} \vdash N \Rightarrow^{\mathbf{T}} \textsf{List} \\ \mathcal{C} \vdash (P \Rightarrow^{\mathbf{PM}} \textbf{any}) \mapsto \llparenthesis RT \rrparenthesis \\ \mathcal{C}; RT \vdash B \implies \textsf{\textit{guarded}} \llparenthesis \$1 \texttt{ => } \textbf{none} \rrparenthesis\end{array}}{\mathcal{C} \vdash \textbf{par } P \textbf{ in } N \texttt{ ||| } B \implies \textsf{\textit{guarded}} \llparenthesis \$1 \texttt{ => } \textbf{none} \rrparenthesis} \qquad (7.44)$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash N \Rightarrow^{\mathbf{T}} \textsf{List} \\ \mathcal{C} \vdash (P \Rightarrow^{\mathbf{PM}} \textbf{any}) \mapsto \llparenthesis RT \rrparenthesis \\ \mathcal{C}; RT \vdash B \implies \textsf{\textit{exit}} \llparenthesis \$1 \texttt{ => } \textbf{none} \rrparenthesis\end{array}}{\mathcal{C} \vdash \textbf{par } P \textbf{ in } N \texttt{ ||| } B \implies \textsf{\textit{exit}} \llparenthesis \$1 \texttt{ => } \textbf{none} \rrparenthesis} \qquad (7.45)$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash N \Rightarrow^{\mathbf{T}} \textsf{List} \\ \mathcal{C} \vdash (P \Rightarrow^{\mathbf{PM}} \textbf{any}) \mapsto \llparenthesis RT \rrparenthesis \\ \mathcal{C}; RT \vdash B \implies \textsf{\textit{guarded}} \llparenthesis\ \rrparenthesis\end{array}}{\mathcal{C} \vdash \textbf{par } P \textbf{ in } N \texttt{ ||| } B \implies \textsf{\textit{guarded}} \llparenthesis\ \rrparenthesis} \qquad (7.46)$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash N \Rightarrow^{\mathbf{T}} \textsf{List} \\ \mathcal{C} \vdash (P \Rightarrow^{\mathbf{PM}} \textbf{any}) \mapsto \llparenthesis RT \rrparenthesis \\ \mathcal{C}; RT \vdash B \implies \textsf{\textit{exit}} \llparenthesis\ \rrparenthesis\end{array}}{\mathcal{C} \vdash \textbf{par } P \textbf{ in } N \texttt{ ||| } B \implies \textsf{\textit{exit}} \llparenthesis\ \rrparenthesis} \qquad (7.47)$$

Example 7.13 Let us check that the behaviour

```
par ?x in [1,2,3] |||
  inP(!x)
endpar
```

which communicates the values 1, 2, and 3 through gate *inP* in any order, is semantically correct. It is translated into abstract syntax as:

$$\begin{array}{l}\textbf{par } \mathtt{?}x \textbf{ in } \mathit{cons}\mathtt{(}1\mathtt{,}\mathit{cons}\mathtt{(}2\mathtt{,}\mathit{cons}\mathtt{(}3\mathtt{,}\mathit{nil}\mathtt{()))) \; |||}\\ \quad \mathit{inP}\mathtt{(!}\textbf{exit}\mathtt{(}\$1 \mathtt{=>} x\mathtt{))} \; \mathtt{@}\textbf{any} : \mathsf{time}\ \textbf{exit}\mathtt{(}\$1 \mathtt{=>} \mathit{true}\mathtt{)} \ \textbf{start}\langle 0\rangle\end{array}$$

We can use rule 7.46 and deduce

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathit{cons}\mathtt{(}1\mathtt{,}\mathit{cons}\mathtt{(}2\mathtt{,}\mathit{cons}\mathtt{(}3\mathtt{,}\mathit{nil}\mathtt{())))} \Rightarrow^{\mathrm{T}} \mathsf{List}\\ \mathcal{C} \vdash (\mathtt{?}x \ \Rightarrow^{\mathrm{PM}} \ \textbf{any}) \ \mapsto \ \langle\!| \, x \mathtt{=>} \textbf{any} \, |\!\rangle\\ \mathcal{C}; x \mathtt{=>} \textbf{any} \vdash \mathit{inP}\mathtt{(!}\textbf{exit}\mathtt{(}\$1 \mathtt{=>} x\mathtt{))} \ldots \ \Longrightarrow \ \mathsf{guarded} \, \langle\!| \ |\!\rangle\end{array}}{\begin{array}{ll}\mathcal{C} \vdash & \textbf{par } \mathtt{?}x \textbf{ in } \mathit{cons}\mathtt{(}1\mathtt{,}\mathit{cons}\mathtt{(}2\mathtt{,}\mathit{cons}\mathtt{(}3\mathtt{,}\mathit{nil}\mathtt{())))} \ \mathtt{|||}\\ & \quad \mathit{inP}\mathtt{(!}\textbf{exit}\mathtt{(}\$1 \mathtt{=>} x\mathtt{))} \ \ldots \qquad\qquad \Longrightarrow \ \mathsf{guarded} \, \langle\!| \ |\!\rangle\end{array}}$$

The first premise is proved in Example 7.27 (in Section 7.29.3) for a shorter list, and the others are similar to others we have already seen.

7.13 Disabling operator

In order to keep internal action urgent and time deterministic, behaviour B_2 is required to be guarded in the disabling behaviour B_1 `[>` B_2. Let us imagine the following behaviour `(`**i** `[>` **exit**`())`;**wait**`(`1`)`;... Intuitively, it can perform an internal action or the behaviour **exit**`()` can finish immediately disabling the internal action and the whole behaviour can wait one unit of time. In fact, the real problem is that the termination of a behaviour is not directly observable when it is composed in sequence with another one. If we have the behaviour B;B' and B finishes, B' starts to be executed immediately, and the termination of B is not observed explicitly. Therefore, `(`**i** `[>` **exit**`())`;**wait**`(`1`)`;... can perform an internal action *or* wait one unit of time, and this is what we do not want, because it means that the internal action is not urgent. If B_2 is required to be guarded we forbid this problem, because it cannot finish immediately.

By the same reason, it seems that B_1 should be also guarded, in order to forbid, for example, the behaviour `(`**exit**`()` `[>` **i**`)`;**wait**`(`1`)`;... which can wait one unit of time if **exit**`()` finishes immediately before the internal action disables it, or the internal action can disable to the **exit**`()` behaviour, and the whole behaviour would perform an internal action. The problem is that requiring B_1 to be guarded does not avoid the problem, since when B_1 evolves (but does not finish) the disabling operator does not disappear. For example, G;**exit**`()` is guarded, but `(`G;**exit**`()` `[>` **i**`)`;**wait**`(`1`)`;... can perform an action on gate G and becomes `(`**exit**`()` `[>` **i**`)`;**wait**`(`1`)`;... which does not preserve the urgency of the internal action. The solution is given by means of the dynamic semantics of this operator. When B_1 finishes, the dynamic semantics rule of the disabling operator (see Section 8.13) states that the behaviour B_1 `[>` B_2 performs an internal action. Thus, the termination of B_1 is "observable" in the sense that something happens between the termination of B_1 and the starting of that behaviour composed in sequence after the disabling.

Regarding variables, the disabling operator behaves like the selection operator: initialized variables, that is, variables V such that V `=>` T appears in the context, may be modified only by B_1 or B_2, but non-initialized variables which are assigned (given a value) by one of them have to be assigned also by the another one. In that way the bindings produced by the disabling behaviour are always the same. Bindings produced by B_1 are forgotten if it is interrupted by B_2, so now we do not have to require that bindings produced by B_1 and B_2 be disjoint.

The rules for the disabling operator are:

$$\frac{\begin{array}{l} \mathcal{C}, RT_1, RT_2 \vdash B_1 \implies \textsf{\textit{guarded}}\,\llparenthesis\, RT_1, RT \,\rrparenthesis \\ \mathcal{C}, RT_1, RT_2 \vdash B_2 \implies \textsf{\textit{guarded}}\,\llparenthesis\, RT_2, RT \,\rrparenthesis \end{array}}{\mathcal{C}, RT_1, RT_2 \vdash B_1 \;\mathtt{[>}\; B_2 \implies \textsf{\textit{guarded}}\,\llparenthesis\, RT_1, RT_2, RT \,\rrparenthesis} \tag{7.48}$$

$$\frac{\begin{array}{l} \mathcal{C}, RT_1, RT_2 \vdash B_1 \implies \textsf{\textit{exit}}\,\llparenthesis\, RT_1, RT \,\rrparenthesis \\ \mathcal{C}, RT_1, RT_2 \vdash B_2 \implies \textsf{\textit{guarded}}\,\llparenthesis\, RT_2, RT \,\rrparenthesis \end{array}}{\mathcal{C}, RT_1, RT_2 \vdash B_1 \;\mathtt{[>}\; B_2 \implies \textsf{\textit{exit}}\,\llparenthesis\, RT_1, RT_2, RT \,\rrparenthesis} \tag{7.49}$$

Example 7.14 Let us see that the behaviour $\mathit{out2}\mathtt{(!}x2\mathtt{)\,[>\;(}\;\mathbf{wait}\mathtt{(}50\mathtt{)\,;}\;\mathbf{i}\;\mathtt{)}$ is semantically correct. It is translated into abstract syntax as

$$\mathit{out2}\mathtt{(}\$1 \mathtt{=>}\ \mathtt{!}\mathbf{exit}\mathtt{(}\$1 \mathtt{=>} x2\mathtt{))}\ldots\mathtt{[>}\;(\;\mathbf{wait}\mathtt{(}\mathbf{exit}\mathtt{(}\$1\mathtt{=>}50\mathtt{))\,;}\;\mathbf{i}\;)$$

We can use inference rule 7.48 with all RT_1, RT_2, and RT equal to the empty record type.

$$\frac{\begin{array}{l} \mathcal{C} \vdash \mathit{out2}\mathtt{(}\$1 \mathtt{=>}\ \mathtt{!}\mathbf{exit}\mathtt{(}\$1 \mathtt{=>} x2\mathtt{))}\ldots \implies \textsf{\textit{guarded}}\,\llparenthesis\;\rrparenthesis \\ \mathcal{C} \vdash \mathbf{wait}\mathtt{(}\mathbf{exit}\mathtt{(}\$1\mathtt{=>}50\mathtt{))\,;}\;\mathbf{i} \implies \textsf{\textit{guarded}}\,\llparenthesis\;\rrparenthesis \end{array}}{\mathcal{C} \vdash \mathit{out2}\mathtt{(}\$1 \mathtt{=>}\ \mathtt{!}\mathbf{exit}\mathtt{(}\$1 \mathtt{=>} x2\mathtt{))}\ldots\mathtt{[>}\;(\;\mathbf{wait}\mathtt{(}\mathbf{exit}\mathtt{(}\$1\mathtt{=>}50\mathtt{))\,;}\;\mathbf{i}\;) \implies \textsf{\textit{guarded}}\,\llparenthesis\;\rrparenthesis} \tag{7.48}$$

However, the behaviour $\mathit{in1}\mathtt{(?}x1\mathtt{:data)\,[>\;(}\;\mathbf{wait}\mathtt{(}50\mathtt{)\,;}\;\mathbf{i}\;\mathtt{)}$ is not correct in a context $\mathcal{C}$ where there is no information about $x1$, because

$$\mathcal{C} \vdash \mathit{in1}\mathtt{(}\$1 \mathtt{=>\ ?}x1\mathtt{:data)}\ldots \implies \textsf{\textit{guarded}}\,\llparenthesis\, x1 \mathtt{=>}\ \mathsf{data} \,\rrparenthesis$$

but

$$\mathcal{C} \vdash \mathbf{wait}\mathtt{(}\mathbf{exit}\mathtt{(}\$1\mathtt{=>}50\mathtt{))\,;}\;\mathbf{i} \implies \textsf{\textit{guarded}}\,\llparenthesis\;\rrparenthesis$$

and we cannot use rules 7.48 or 7.49, and there are not more rules that we can apply to the whole behaviour.

7.14 Suspend/Resume operator

This operator has a static semantics similar to the previous one. The only difference is that B_2 is checked in a context where the exception X is declared, so it can be used by B_2.

$$\frac{\begin{array}{l} \mathcal{C}, RT_1, RT_2 \vdash B_1 \implies \textsf{\textit{guarded}}\,\llparenthesis\, RT_1, RT \,\rrparenthesis \\ \mathcal{C}, RT_1, RT_2; X \Rightarrow \textsf{\textit{exn}}\langle\mathtt{(}\$1 \mathtt{=>\ ())}\rangle \vdash B_2 \implies \textsf{\textit{guarded}}\,\llparenthesis\, RT_2, RT \,\rrparenthesis \end{array}}{\mathcal{C}, RT_1, RT_2 \vdash B_1 \;\mathtt{[}X\mathtt{>}\; B_2 \implies \textsf{\textit{guarded}}\,\llparenthesis\, RT_1, RT_2, RT \,\rrparenthesis} \tag{7.50}$$

$$\frac{\begin{array}{l} \mathcal{C}, RT_1, RT_2 \vdash B_1 \implies \textsf{\textit{exit}}\,\llparenthesis\, RT_1, RT \,\rrparenthesis \\ \mathcal{C}, RT_1, RT_2; X \Rightarrow \textsf{\textit{exn}}\langle\mathtt{(}\$1 \mathtt{=>\ ())}\rangle \vdash B_2 \implies \textsf{\textit{guarded}}\,\llparenthesis\, RT_2, RT \,\rrparenthesis \end{array}}{\mathcal{C}, RT_1, RT_2 \vdash B_1 \;\mathtt{[}X\mathtt{>}\; B_2 \implies \textsf{\textit{exit}}\,\llparenthesis\, RT_1, RT_2, RT \,\rrparenthesis} \tag{7.51}$$

Example 7.15 Let us see now that the behaviour $\mathit{out2}\mathtt{(!}x2\mathtt{)\,[}\mathit{Ok}\mathtt{>\;(}\;\mathbf{wait}\mathtt{(}50\mathtt{)\,;}\;\mathbf{i}\;\mathtt{;}\;\mathbf{signal}\;\mathit{Ok}\mathtt{)}$ is semantically correct. It is translated into abstract syntax as

$$\mathit{out2}\mathtt{(}\$1 \mathtt{=>}\ \mathtt{!}\mathbf{exit}\mathtt{(}\$1 \mathtt{=>} x2\mathtt{))}\ldots\mathtt{[}\mathit{Ok}\mathtt{>}\;(\;\mathbf{wait}\mathtt{(}\mathbf{exit}\mathtt{(}\$1\mathtt{=>}50\mathtt{))\,;}\;\mathbf{i}\;\mathtt{;}\;\mathbf{signal}\;\mathit{Ok}\mathtt{(}\mathbf{exit}\mathtt{(}\$1\mathtt{=>\ ())))}$$

We can use inference rule 7.50 with all RT_1, RT_2, and RT equal to the empty record type.

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathit{out2}\,\texttt{(}\$1 \texttt{=>} \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} x2\texttt{))}\ldots \Longrightarrow \mathsf{guarded}\,\llparenthesis\,\rrparenthesis \\ \mathcal{C}; \mathit{Ok} \Rightarrow \mathsf{exn}\langle\texttt{(}\$1 \texttt{=> ())}\rangle \vdash \mathbf{wait}\texttt{(}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 50\texttt{))}\,\texttt{;}\ \mathbf{i}\ \texttt{;}\ \mathbf{signal}\ \mathit{Ok}\texttt{(}\mathbf{exit}\texttt{(}\$1 \texttt{=> ()))} \Longrightarrow \mathsf{guarded}\,\llparenthesis\,\rrparenthesis\end{array}}{\begin{array}{l}\mathcal{C} \vdash \mathit{out2}\,\texttt{(}\$1 \texttt{=>} \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} x2\texttt{))}\ldots \\ \qquad \texttt{[}\mathit{Ok}\texttt{>}\ \texttt{(}\ \mathbf{wait}\texttt{(}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 50\texttt{))}\,\texttt{;}\ \mathbf{i}\ \texttt{;}\ \mathbf{signal}\ \mathit{Ok}\texttt{(}\mathbf{exit}\texttt{(}\$1 \texttt{=> ())))} \Longrightarrow \mathsf{guarded}\,\llparenthesis\,\rrparenthesis\end{array}} \tag{7.48}$$

where the second premise is proved in Example 7.18 in Section 7.17.

7.15 Hiding operator

The behaviour $\mathbf{hide}\ G_1\texttt{:}T_1\texttt{,}\ldots\texttt{,}G_n\texttt{:}T_n\ \mathbf{in}\ B$ is correct in the context $\mathcal{C}$, if the T_i's are correct types in that context, and the behaviour B is correct in the context $\mathcal{C}$ overridden by the information that G_i's are gates of the corresponding type. Thus the **hide** operator declares new gates, that may be used in the behaviour B. The whole behaviour is guarded if the behaviour B is. The semantic rules are the following ones:

$$\frac{\begin{array}{l}\mathcal{C} \vdash T_1 \Rightarrow \mathsf{type}\ \cdots\ \mathcal{C} \vdash T_n \Rightarrow \mathsf{type} \\ \mathcal{C}; G_1 \Rightarrow \mathsf{gate}\langle\texttt{(}\$1 \texttt{=>} T_1\texttt{)}\rangle, \\ \qquad\qquad\qquad\qquad \ldots, \\ \qquad G_n \Rightarrow \mathsf{gate}\langle\texttt{(}\$1 \texttt{=>} T_n\texttt{)}\rangle \vdash B \Longrightarrow \mathsf{exit}\,\llparenthesis\, RT \,\rrparenthesis\end{array}}{\mathcal{C} \vdash \mathbf{hide}\ G_1\texttt{:}T_1\texttt{,}\ldots\texttt{,}G_n\texttt{:}T_n\ \mathbf{in}\ B \Longrightarrow \mathsf{exit}\,\llparenthesis\, RT \,\rrparenthesis} \tag{7.52}$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash T_1 \Rightarrow \mathsf{type}\ \cdots\ \mathcal{C} \vdash T_n \Rightarrow \mathsf{type} \\ \mathcal{C}; G_1 \Rightarrow \mathsf{gate}\langle\texttt{(}\$1 \texttt{=>} T_1\texttt{)}\rangle, \\ \qquad\qquad\qquad\qquad \ldots, \\ \qquad G_n \Rightarrow \mathsf{gate}\langle\texttt{(}\$1 \texttt{=>} T_n\texttt{)}\rangle \vdash B \Longrightarrow \mathsf{guarded}\,\llparenthesis\, RT \,\rrparenthesis\end{array}}{\mathcal{C} \vdash \mathbf{hide}\ G_1\texttt{:}T_1\texttt{,}\ldots\texttt{,}G_n\texttt{:}T_n\ \mathbf{in}\ B \Longrightarrow \mathsf{guarded}\,\llparenthesis\, RT \,\rrparenthesis} \tag{7.53}$$

Example 7.16 Let us check that the following behaviour is a $\mathsf{guarded}\,\llparenthesis\,\rrparenthesis$ behaviour:

```
hide sync in
      pd(!x1); sync
   |[ sync ]|
      sync; pd(!x2)
endhide
```

It is translated into abstract syntax as

$$\mathbf{hide}\ \mathit{sync}\texttt{:}\mathbf{any}\ \mathbf{in}\ \mathit{hideBody}$$

where

$$\begin{array}{l}\mathit{hideBody} \equiv \\ \qquad\qquad \mathit{pd}\texttt{(}\$1 \texttt{=>} \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} x1\texttt{))}\ldots\,\texttt{;}\ \mathit{sync}\texttt{(}\$1 \texttt{=> ())}\ldots \\ \qquad \texttt{|[}\ \mathit{sync}\ \texttt{]|} \\ \qquad\qquad \mathit{sync}\texttt{(}\$1 \texttt{=> ())}\ldots\,\texttt{;}\ \mathit{pd}\texttt{(}\$1 \texttt{=>} \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} x2\texttt{))}\ldots\end{array}$$

By using rule 7.53 we can infer

$$\frac{\begin{array}{l}\mathcal{C} \vdash \textbf{any} \Rightarrow \textsf{\textit{type}} \\ \mathcal{C}; \mathit{sync} \Rightarrow \textsf{\textit{gate}}\langle\,(\$1 \texttt{=>}\ \textbf{any})\,\rangle \vdash \mathit{hideBody} \implies \textsf{\textit{guarded}}\ \langle\!|\, \mathit{RT}\, |\!\rangle\end{array}}{\mathcal{C} \vdash \textbf{hide}\ \mathit{sync}\,\textbf{:any in}\ \mathit{hideBody} \implies \textsf{\textit{guarded}}\ \langle\!|\ |\!\rangle} \quad (7.53)$$

The first premise is rule 7.81, and the second premise was proved in Example 7.11.

7.16 Exception handling

A **trap** behaviour

> **trap**
>   **exception** X_1 ($1 => P_1) :T_1 **is** B_1
>   ...
>   **exception** X_n ($1 => P_n) :T_n **is** B_n
>   **exit** P_{n+1} **is** B_{n+1}
> **in** B

is semantically correct in a context $\mathcal{C}$ (we say below what conditions this context fulfills) if:

- The types given to the trapped (declared) exceptions are correct types declared in the context $\mathcal{C}$.
- The patterns catching the values communicated through the trapped exceptions can be matched against the corresponding types.
- The behaviours that handle the trapped exceptions are correct.
- The main behaviour B is semantically correct in the context $\mathcal{C}$ where the declared exceptions have been included.
- The pattern P_{n+1} can be pattern-matched against part of the bindings produced by B.
- The behaviour B_{n+1} is semantically correct.

The **trap** operator is a *branching* operator as the selection or disabling operators, in the sense that not all the parts of the behaviour are executed always. Hence, regarding variables, it behaves in a similar way. Initialized variables, that is, variables that has a value before the **trap** behaviour is started, can be modified only by some subcomponents; but non-initialized variables modified by one subcomponent have to be assigned by all the subcomponents. In the following rules the context $\mathcal{C}$ stands for a context $\mathcal{C}', RT'_i \odot \ldots \odot RT'_{n+2}$, where RT'_i represents the bindings produced by branch i related to initialized variables. Since an initialized variable can be modified by more than one branch, we use the $\odot$ operator.

If the behaviours handling the trapped exceptions, the main behaviour B, and the behaviour handling the termination of B are all exit behaviours, then the whole behaviour is also exit.

$$\frac{\begin{array}{l}
\mathcal{C} \vdash T_1 \Rightarrow \textit{type} \;\cdots\; \mathcal{C} \vdash T_n \Rightarrow \textit{type} \\
\forall i \,.\, 1 \leq i \leq n \,.\, \mathcal{C} \vdash ((\$1 \texttt{=>} P_i) \;\Rightarrow^{\mathbf{PM}}\; (\$1 \texttt{=>} T_i)) \;\mapsto\; \langle\!|\, RT_i \,|\!\rangle \\
\forall i \,.\, 1 \leq i \leq n \,.\, \mathcal{C}; RT_i \vdash B_i \;\Longrightarrow\; \textit{exit}\, \langle\!|\, RT'_i, RT \,|\!\rangle \\
\mathcal{C}; X_1 \Rightarrow \textit{exn}\langle (\$1 \texttt{=>} T_1) \rangle, \ldots, \\
\quad X_n \Rightarrow \textit{exn}\langle (\$1 \texttt{=>} T_n) \rangle \qquad \vdash B \;\Longrightarrow\; \textit{exit}\, \langle\!|\, RT'_{n+2}, RT' \,|\!\rangle \\
\mathcal{C} \vdash P_{n+1} \;\Rightarrow^{\mathbf{PM}}\; (RT') \;\mapsto\; \langle\!|\, RT'' \,|\!\rangle \\
\mathcal{C}; RT'' \vdash B_{n+1} \;\Longrightarrow\; \textit{exit}\, \langle\!|\, RT'_{n+1}, RT \,|\!\rangle
\end{array}}{\begin{array}{l}
\mathcal{C} \vdash \textbf{trap} \\
\qquad \textbf{exception } X_1 (\$1 \texttt{=>} P_1) : T_1 \textbf{ is } B_1 \\
\qquad\quad \ldots \\
\qquad \textbf{exception } X_n (\$1 \texttt{=>} P_n) : T_n \textbf{ is } B_n \\
\qquad \textbf{exit } P_{n+1} \textbf{ is } B_{n+1} \\
\quad \textbf{in } B \qquad\qquad\qquad\qquad\qquad\qquad \Longrightarrow\; \textit{exit}\, \langle\!|\, RT'_1 \odot \ldots \odot RT'_{n+2}, RT \,|\!\rangle
\end{array}} \tag{7.54}$$

If the behaviours handling the trapped exceptions and the main behaviour B are guarded behaviours, then the whole behaviour is guarded.

$$\frac{\begin{array}{l}
\mathcal{C} \vdash T_1 \Rightarrow \textit{type} \;\cdots\; \mathcal{C} \vdash T_n \Rightarrow \textit{type} \\
\forall i \,.\, 1 \leq i \leq n \,.\, \mathcal{C} \vdash ((\$1 \texttt{=>} P_i) \;\Rightarrow^{\mathbf{PM}}\; (\$1 \texttt{=>} T_i)) \;\mapsto\; \langle\!|\, RT_i \,|\!\rangle \\
\forall i \,.\, 1 \leq i \leq n \,.\, \mathcal{C}; RT_i \vdash B_i \;\Longrightarrow\; \textit{guarded}\, \langle\!|\, RT'_i, RT \,|\!\rangle \\
\mathcal{C}; X_1 \Rightarrow \textit{exn}\langle (\$1 \texttt{=>} T_1) \rangle, \ldots, \\
\quad X_n \Rightarrow \textit{exn}\langle (\$1 \texttt{=>} T_n) \rangle \qquad \vdash B \;\Longrightarrow\; \textit{guarded}\, \langle\!|\, RT'_{n+2}, RT' \,|\!\rangle \\
\mathcal{C} \vdash P_{n+1} \;\Rightarrow^{\mathbf{PM}}\; (RT') \;\mapsto\; \langle\!|\, RT'' \,|\!\rangle \\
\mathcal{C}; RT'' \vdash B_{n+1} \;\Longrightarrow\; \textit{exit}\, \langle\!|\, RT'_{n+1}, RT \,|\!\rangle
\end{array}}{\begin{array}{l}
\mathcal{C} \vdash \textbf{trap} \\
\qquad \textbf{exception } X_1 (\$1 \texttt{=>} P_1) : T_1 \textbf{ is } B_1 \\
\qquad\quad \ldots \\
\qquad \textbf{exception } X_n (\$1 \texttt{=>} P_n) : T_n \textbf{ is } B_n \\
\qquad \textbf{exit } P_{n+1} \textbf{ is } B_{n+1} \\
\quad \textbf{in } B \qquad\qquad\qquad\qquad\qquad\qquad \Longrightarrow\; \textit{guarded}\, \langle\!|\, RT'_1 \odot \ldots \odot RT'_{n+2}, RT \,|\!\rangle
\end{array}} \tag{7.55}$$

Finally, if the behaviours handling the trapped exceptions and the behaviour handling the termination of B are guarded, then the whole behaviour is also guarded.

$$\frac{\begin{array}{l}
\mathcal{C} \vdash T_1 \Rightarrow \textit{type} \;\cdots\; \mathcal{C} \vdash T_n \Rightarrow \textit{type} \\
\mathcal{C} \vdash ((\$1 \texttt{=>} P_i) \;\Rightarrow^{\mathbf{PM}}\; (\$1 \texttt{=>} T_i)) \;\mapsto\; \langle\!|\, RT_i \,|\!\rangle \qquad \forall 1 \leq i \leq n \\
\mathcal{C}; RT_i \vdash B_i \;\Longrightarrow\; \textit{guarded}\, \langle\!|\, RT'_i, RT \,|\!\rangle \qquad \forall 1 \leq i \leq n \\
\quad \mathcal{C}; X_1 \Rightarrow \textit{exn}\langle (\$1 \texttt{=>} T_1) \rangle, \\
\ldots, X_n \Rightarrow \textit{exn}\langle (\$1 \texttt{=>} T_n) \rangle \vdash B \;\Longrightarrow\; \textit{exit}\, \langle\!|\, RT'_{n+2}, RT' \,|\!\rangle \\
\mathcal{C} \vdash P_{n+1} \;\Rightarrow^{\mathbf{PM}}\; (RT') \;\mapsto\; \langle\!|\, RT'' \,|\!\rangle \\
\mathcal{C}; RT'' \vdash B_{n+1} \;\Longrightarrow\; \textit{guarded}\, \langle\!|\, RT'_{n+1}, RT \,|\!\rangle
\end{array}}{\begin{array}{l}
\mathcal{C} \vdash \textbf{trap} \\
\qquad \textbf{exception } X_1 (\$1 \texttt{=>} P_1) : T_1 \textbf{ is } B_1 \\
\qquad\quad \ldots \\
\qquad \textbf{exception } X_n (\$1 \texttt{=>} P_n) : T_n \textbf{ is } B_n \\
\qquad \textbf{exit } P_{n+1} \textbf{ is } B_{n+1} \\
\quad \textbf{in } B \qquad\qquad\qquad\qquad\qquad\qquad \Longrightarrow\; \textit{guarded}\, \langle\!|\, RT'_1 \odot \ldots \odot RT'_{n+2}, RT \,|\!\rangle
\end{array}} \tag{7.56}$$

Example 7.17 Let us see what kind of behaviour

```
trap
  exception Hd is outP(!0) endexn
in
  case ys:List is
        Nil -> signal Hd
      | Cons(?x,?xs) -> outP(!x)
  endcase
endtrap
```

is, supposing that ys has a value of type List. It is translated into abstract syntax as follows:

```
exnHandling  ≡
    trap
      exception Hd($1 => ()) is outP($1 => !exit($1 => 0)) ...
      exit () is exit()
    in
      case exit($1 => ys):List is BM1 | (BM2 | BM3)
```

where BM_1, BM_2, and BM_3 are defined in Example 7.25 in Section 7.23 where this **case** behaviour is checked.

We have to use rule 7.55:

$$\frac{\begin{array}{l}\mathcal{C} \vdash \texttt{()} \Rightarrow \mathit{type} \\ \mathcal{C} \vdash (\texttt{(\$1 => ())} \ \Rightarrow^{\mathbf{PM}} \ \texttt{(\$1 => ())}) \ \mapsto \ \{\!|\ |\!\} \\ \mathcal{C} \vdash \mathit{outP}\texttt{(!}\mathbf{exit}\texttt{(\$1 => 0))}\ldots \ \Longrightarrow \ \mathit{guarded}\ \{\!|\ |\!\} \\ \mathcal{C}; \mathit{Hd} \Rightarrow \mathit{exn}\langle\texttt{(\$1 => ())}\rangle \vdash \mathbf{case}\ \mathbf{exit}\texttt{(\$1 => } ys\texttt{)}\texttt{:List}\ \mathbf{is}\ BM_1 \ \texttt{|}\ (BM_2 \ \texttt{|}\ BM_3) \ \Longrightarrow \ \mathit{guarded}\ \{\!|\ |\!\} \\ \mathcal{C} \vdash \texttt{()} \ \Rightarrow^{\mathbf{PM}} \ \texttt{()} \ \mapsto \ \{\!|\ |\!\} \\ \mathcal{C} \vdash \mathbf{exit}\texttt{()} \ \Longrightarrow \ \mathit{exit}\ \{\!|\ |\!\}\end{array}}{\mathcal{C} \vdash \mathit{exnHandling} \ \Longrightarrow \ \mathit{guarded}\ \{\!|\ |\!\}} \tag{7.55}$$

where the premise dealing with the **case** behaviour is checked in Example 7.25 and the rest are easy to prove.

7.17 Exception signaling

The abstract syntax of the **signal** instruction is

$$\mathbf{signal}\ X\texttt{(}E\texttt{)}$$

where the expression E is **exit**(\$1 => ()) by default. That is, when no expression is given in the specification, the value () is sent together with the exception.

The behaviour **signal** X(E) is semantically correct in the context $\mathcal{C}$ if X is an exception in that context, and if when X has type $\mathit{exn}\langle\texttt{(\$1 => }T\texttt{)}\rangle$ then expression E returns one value of type T.

$$\frac{\begin{array}{l}\mathcal{C} \vdash X \Rightarrow \mathit{exn}\langle\texttt{(\$1 => }T\texttt{)}\rangle \\ \mathcal{C} \vdash E \ \Longrightarrow \ \mathit{exit}\ \{\!|\ \texttt{\$1 => }T\ |\!\}\end{array}}{\mathcal{C} \vdash \mathbf{signal}\ X\texttt{(}E\texttt{)} \ \Longrightarrow \ \mathit{guarded}\ \{\!|\ |\!\}} \tag{7.57}$$

Example 7.18 Let us check that the behaviour **wait**(50) **; i ; signal** Ok is semantically correct in a context $\mathcal{C} = \mathcal{C}', \mathit{Ok} \Rightarrow \mathit{exn}\langle\texttt{(\$1 => ())}\rangle$ where exception Ok is declared. The behaviour is translated to abstract syntax as follows:

$$\textbf{wait}\texttt{(}\textbf{exit}\texttt{(}\$1 \texttt{=>} 50\texttt{))}\texttt{;}\ \texttt{(}\mathbf{i}\ \texttt{;}\ \textbf{signal}\ Ok\texttt{(}\textbf{exit}\texttt{(}\$1 \texttt{=>} \texttt{()}\texttt{)))}$$

We have to use rule 7.29 as follows:

$$\dfrac{\mathcal{C} \vdash \textbf{wait}\texttt{(}\textbf{exit}\texttt{(}\$1 \texttt{=>} 50\texttt{))} \Longrightarrow \mathit{exit}\llparenthesis\ \rrparenthesis \quad \dfrac{\begin{array}{l}\mathcal{C} \vdash \mathbf{i} \Longrightarrow \mathit{guarded}\llparenthesis\ \rrparenthesis \\ \mathcal{C} \vdash \textbf{signal}\ Ok\texttt{(}\textbf{exit}\texttt{(}\$1 \texttt{=>} \texttt{()}\texttt{))} \Longrightarrow \mathit{guarded}\llparenthesis\ \rrparenthesis\end{array}}{\mathcal{C} \vdash \mathbf{i}\ \texttt{;}\ \textbf{signal}\ Ok\texttt{(}\textbf{exit}\texttt{(}\$1 \texttt{=>} \texttt{()}\texttt{))} \Longrightarrow \mathit{guarded}\llparenthesis\ \rrparenthesis}\,(7.30)}{\mathcal{C} \vdash \textbf{wait}\texttt{(}\textbf{exit}\texttt{(}\$1 \texttt{=>} 50\texttt{))}\texttt{;}\ \texttt{(}\mathbf{i}\ \texttt{;}\ \textbf{signal}\ Ok\texttt{(}\textbf{exit}\texttt{(}\$1 \texttt{=>} \texttt{()}\texttt{)))} \Longrightarrow \mathit{guarded}\llparenthesis\ \rrparenthesis}\,(7.29)$$

where the premise regarding the exception signaling is proved as follows:

$$\dfrac{\begin{array}{l}\mathcal{C} \vdash Ok \Rightarrow \mathit{exn}\langle\texttt{(}\$1 \texttt{=>} \texttt{()}\texttt{)}\rangle \\ \mathcal{C} \vdash \textbf{exit}\texttt{(}\$1 \texttt{=>} \texttt{()}\texttt{)} \Longrightarrow \mathit{exit}\llparenthesis\,\$1 \texttt{=>} \texttt{()}\,\rrparenthesis\end{array}}{\mathcal{C} \vdash \textbf{signal}\ Ok\texttt{(}\textbf{exit}\texttt{(}\$1 \texttt{=>} \texttt{()}\texttt{))} \Longrightarrow \mathit{guarded}\llparenthesis\ \rrparenthesis}\,(7.57)$$

7.18 Delay instruction

If E is an expression that returns only a value of type time then the behaviour $\textbf{wait}\texttt{(}E\texttt{)}$ is correct, it does no action before finishing, and produces no bindings.

$$\dfrac{\mathcal{C} \vdash E \Longrightarrow \mathit{exit}\llparenthesis\,\$1 \texttt{=>} \mathsf{time}\,\rrparenthesis}{\mathcal{C} \vdash \textbf{wait}\texttt{(}E\texttt{)} \Longrightarrow \mathit{exit}\llparenthesis\ \rrparenthesis} \tag{7.58}$$

Example 7.19 We can see that $\textbf{wait}\texttt{(1)}$ is a correct behaviour, assuming that data type time is a synonymous of nat. It is translated to $\textbf{wait}\texttt{(}\textbf{exit}\texttt{(}\$1 \texttt{=>} 1\texttt{))}$. The proof is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{C} \vdash 1 \texttt{=>} \mathsf{time}}{\mathcal{C} \vdash 1 \Rightarrow^{\mathrm{T}} \mathsf{time}}\,(7.95)}{\mathcal{C} \vdash \$1 \texttt{=>} 1 \Rightarrow^{\mathrm{T}} \$1 \texttt{=>} \mathsf{time}}\,(7.100)}{\mathcal{C} \vdash \textbf{exit}\texttt{(}\$1 \texttt{=>} 1\texttt{)} \Longrightarrow \mathit{exit}\llparenthesis\,\$1 \texttt{=>} \mathsf{time}\,\rrparenthesis}\,(7.33)}{\mathcal{C} \vdash \textbf{wait}\texttt{(}\textbf{exit}\texttt{(}\$1 \texttt{=>} 1\texttt{))} \Longrightarrow \mathit{exit}\llparenthesis\ \rrparenthesis}\,(7.58)$$

7.19 Renaming operator

A renaming behaviour

$$\begin{array}{l}
\textbf{rename} \\
\quad \textbf{gate}\ G_1\texttt{(}\$1 \texttt{=>} P_1\texttt{)}\texttt{:}T_1\ \textbf{is}\ G_1'\texttt{(}\$1 \texttt{=>} P_1'\texttt{)} \\
\qquad \ldots \\
\quad \textbf{gate}\ G_m\texttt{(}\$1 \texttt{=>} P_m\texttt{)}\texttt{:}T_m\ \textbf{is}\ G_m'\texttt{(}\$1 \texttt{=>} P_m'\texttt{)} \\
\quad \textbf{signal}\ X_1\texttt{(}\$1 \texttt{=>} P_1''\texttt{)}\texttt{:}T_1'\ \textbf{is}\ X_1'\texttt{(}E_1\texttt{)} \\
\qquad \ldots \\
\quad \textbf{signal}\ X_n\texttt{(}\$1 \texttt{=>} P_n''\texttt{)}\texttt{:}T_n'\ \textbf{is}\ X_n'\texttt{(}E_n\texttt{)} \\
\textbf{in}\ B
\end{array}$$

is semantically correct in the context $\mathcal{C}$ if all the following conditions are fulfilled:

- The types given to the declared gates are correct types declared in the context $\mathcal{C}$,

$$\mathcal{C} \vdash T_1 \Rightarrow \textit{\textsf{type}} \ \cdots \ \mathcal{C} \vdash T_m \Rightarrow \textit{\textsf{type}}.$$

- The declared patterns that catch the values communicated through the gates can be matched against the corresponding types:

$$\forall\, i\,.\, 1 \le i \le m\,.\, \mathcal{C} \vdash (\,(\texttt{\$1 =>}\ P_i) \ \ \Rightarrow^{\mathbf{PM}} \ \ (\texttt{\$1 =>}\ T_i)\,) \ \mapsto \ \llparenthesis\, RT_i \,\rrparenthesis.$$

- The actions performed instead of the renamed gates are correct actions in the context $\mathcal{C}$,

$$\forall\, i\,.\, 1 \le i \le m\,.\, \mathcal{C}; RT_i \vdash G'_i \ (\texttt{\$1 =>}\ P'_i)\ \texttt{@}\mathbf{any} : \textsf{time}\ \mathbf{exit}(\texttt{\$1 =>}\ \mathit{true})\ \mathbf{start}\langle 0\rangle \ \Longrightarrow \ \textit{\textsf{guarded}}\, \llparenthesis\ \rrparenthesis.$$

- The types given to the declared exceptions are correct,

$$\mathcal{C} \vdash T'_1 \Rightarrow \textit{\textsf{type}} \ \cdots \ \mathcal{C} \vdash T'_m \Rightarrow \textit{\textsf{type}}.$$

- The patterns used to catch the value raised together with the declared exceptions can be matched against the corresponding types,

$$\forall\, i\,.\, 1 \le i \le n\,.\, \mathcal{C} \vdash (\,(\texttt{\$1 =>}\ P''_i) \ \ \Rightarrow^{\mathbf{PM}} \ \ (\texttt{\$1 =>}\ T'_i)\,) \ \mapsto \ \llparenthesis\, RT'_i \,\rrparenthesis.$$

- The behaviours that raise the exceptions used instead of the renamed exceptions are correct

$$\forall\, i\,.\, 1 \le i \le n\,.\, \mathcal{C}; RT'_i \vdash \mathbf{signal}\ X'_i(E_i) \ \Longrightarrow \ \textit{\textsf{guarded}}\, \llparenthesis\ \rrparenthesis,$$

  and

- The behaviour B is correct in a context where the declared gates and exceptions have been included,

$$\begin{array}{l} \mathcal{C}; G_1 \Rightarrow \textit{\textsf{gate}}\langle(\texttt{\$1 =>}\ T_1)\rangle \odot \\ \quad \ldots \odot G_m \Rightarrow \textit{\textsf{gate}}\langle(\texttt{\$1 =>}\ T_m)\rangle \\ \quad \odot\, X_1 \Rightarrow \textit{\textsf{exn}}\langle(\texttt{\$1 =>}\ T'_1)\rangle \odot \\ \quad \ldots \odot X_m \Rightarrow \textit{\textsf{exn}}\langle(\texttt{\$1 =>}\ T'_n)\rangle \ \ \vdash B \ \Longrightarrow \ \textit{\textsf{exit}}\, \llparenthesis\, RT \,\rrparenthesis \end{array}$$

  where the matching union operator $\odot$ is used because, if the same gate G is used in several renaming clauses, the declared type must be the same.

$$\frac{\begin{array}{l} \mathcal{C} \vdash T_1 \Rightarrow \textit{\textsf{type}} \ \cdots \ \mathcal{C} \vdash T_m \Rightarrow \textit{\textsf{type}} \\ \forall\, i\,.\, 1 \le i \le m\,.\, \mathcal{C} \vdash (\,(\texttt{\$1 =>}\ P_i) \ \ \Rightarrow^{\mathbf{PM}} \ \ (\texttt{\$1 =>}\ T_i)\,) \ \mapsto \ \llparenthesis\, RT_i \,\rrparenthesis \\ \forall\, i\,.\, 1 \le i \le m\,.\, \mathcal{C}; RT_i \vdash G'_i \ (\texttt{\$1 =>}\ P'_i)\ \texttt{@}\mathbf{any} : \textsf{time}\ \mathbf{exit}(\texttt{\$1 =>}\ \mathit{true})\ \mathbf{start}\langle 0\rangle \ \Longrightarrow \ \textit{\textsf{guarded}}\, \llparenthesis\ \rrparenthesis \\ \mathcal{C} \vdash T'_1 \Rightarrow \textit{\textsf{type}} \ \cdots \ \mathcal{C} \vdash T'_m \Rightarrow \textit{\textsf{type}} \\ \forall\, i\,.\, 1 \le i \le n\,.\, \mathcal{C} \vdash (\,(\texttt{\$1 =>}\ P''_i) \ \ \Rightarrow^{\mathbf{PM}} \ \ (\texttt{\$1 =>}\ T'_i)\,) \ \mapsto \ \llparenthesis\, RT'_i \,\rrparenthesis \\ \forall\, i\,.\, 1 \le i \le n\,.\, \mathcal{C}; RT'_i \vdash \mathbf{signal}\ X'_i(E_i) \ \Longrightarrow \ \textit{\textsf{guarded}}\, \llparenthesis\ \rrparenthesis \\ \mathcal{C}; G_1 \Rightarrow \textit{\textsf{gate}}\langle(\texttt{\$1 =>}\ T_1)\rangle \odot \\ \quad \ldots \odot G_m \Rightarrow \textit{\textsf{gate}}\langle(\texttt{\$1 =>}\ T_m)\rangle \\ \quad \odot\, X_1 \Rightarrow \textit{\textsf{exn}}\langle(\texttt{\$1 =>}\ T'_1)\rangle \odot \\ \quad \ldots \odot X_m \Rightarrow \textit{\textsf{exn}}\langle(\texttt{\$1 =>}\ T'_n)\rangle \ \ \vdash B \ \Longrightarrow \ \textit{\textsf{exit}}\, \llparenthesis\, RT \,\rrparenthesis \end{array}}{\begin{array}{l} \mathcal{C} \vdash \mathbf{rename} \\ \qquad \mathbf{gate}\ G_1(\texttt{\$1 =>}\ P_1) : T_1\ \mathbf{is}\ G'_1(\texttt{\$1 =>}\ P'_1) \\ \qquad\quad \ldots \\ \qquad \mathbf{gate}\ G_m(\texttt{\$1 =>}\ P_m) : T_m\ \mathbf{is}\ G'_m(\texttt{\$1 =>}\ P'_m) \\ \qquad \mathbf{signal}\ X_1(\texttt{\$1 =>}\ P''_1) : T'_1\ \mathbf{is}\ X'_1(E_1) \\ \qquad\quad \ldots \\ \qquad \mathbf{signal}\ X_m(\texttt{\$1 =>}\ P''_n) : T'_n\ \mathbf{is}\ X'_n(E_n) \\ \quad \mathbf{in}\ B \qquad\qquad\qquad\qquad\qquad\qquad \Longrightarrow \ \textit{\textsf{exit}}\, \llparenthesis\, RT \,\rrparenthesis \end{array}} \tag{7.59}$$

$$\dfrac{\begin{array}{l}
\mathcal{C} \vdash T_1 \Rightarrow \textit{type} \ \cdots \ \mathcal{C} \vdash T_m \Rightarrow \textit{type} \\
\forall\, i\,.\, 1 \le i \le m\,.\, \mathcal{C} \vdash (\,(\$1 \texttt{=>} P_i) \ \Rightarrow^{\mathbf{PM}} \ (\$1 \texttt{=>} T_i)\,) \ \mapsto \ \llparenthesis\, RT_i \,\rrparenthesis \\
\forall\, i\,.\, 1 \le i \le m\,.\, \mathcal{C}; RT_i \vdash G'_i \ (\$1 \texttt{=>} P'_i) \ \texttt{@}\mathbf{any} : \mathsf{time}\ \mathbf{exit}(\$1 \texttt{=>} \mathit{true}) \ \mathbf{start}\langle 0\rangle \ \Longrightarrow \ \textit{guarded}\ \llparenthesis\ \rrparenthesis \\
\mathcal{C} \vdash T'_1 \Rightarrow \textit{type} \ \cdots \ \mathcal{C} \vdash T'_m \Rightarrow \textit{type} \\
\forall\, i\,.\, 1 \le i \le n\,.\, \mathcal{C} \vdash (\,(\$1 \texttt{=>} P''_i) \ \Rightarrow^{\mathbf{PM}} \ (\$1 \texttt{=>} T'_i)\,) \ \mapsto \ \llparenthesis\, RT'_i \,\rrparenthesis \\
\forall\, i\,.\, 1 \le i \le n\,.\, \mathcal{C}; RT'_i \vdash \mathbf{signal}\ X'_i(E_i) \ \Longrightarrow \ \textit{guarded}\ \llparenthesis\ \rrparenthesis \\
\mathcal{C}; G_1 \Rightarrow \textit{gate}\langle(\$1 \texttt{=>} T_1)\rangle \odot \\
\quad \ldots \odot G_m \Rightarrow \textit{gate}\langle(\$1 \texttt{=>} T_m)\rangle \\
\quad \odot X_1 \Rightarrow \textit{exn}\langle(\$1 \texttt{=>} T'_1)\rangle \odot \\
\quad \ldots \odot X_m \Rightarrow \textit{exn}\langle(\$1 \texttt{=>} T'_n)\rangle \ \ \vdash B \ \Longrightarrow \ \textit{guarded}\ \llparenthesis\, RT \,\rrparenthesis
\end{array}}{\begin{array}{l}
\mathcal{C} \vdash \mathbf{rename} \\
\qquad \mathbf{gate}\ G_1(\$1 \texttt{=>} P_1) : T_1\ \mathbf{is}\ G'_1(\$1 \texttt{=>} P'_1) \\
\qquad\quad \ldots \\
\qquad \mathbf{gate}\ G_m(\$1 \texttt{=>} P_m) : T_m\ \mathbf{is}\ G'_m(\$1 \texttt{=>} P'_m) \\
\qquad \mathbf{signal}\ X_1(\$1 \texttt{=>} P''_1) : T'_1\ \mathbf{is}\ X'_1(E_1) \\
\qquad\quad \ldots \\
\qquad \mathbf{signal}\ X_m(\$1 \texttt{=>} P''_n) : T'_n\ \mathbf{is}\ X'_n(E_n) \\
\quad \mathbf{in}\ B \qquad\qquad\qquad\qquad \Longrightarrow \ \textit{guarded}\ \llparenthesis\, RT \,\rrparenthesis
\end{array}} \quad (7.60)$$

Example 7.20 Let us check that the renaming behaviour

```
rename
  gate inP (?a):int is intro (!a)
in
  inP(!3)
endren
```

is a semantically correct *guarded* $\llparenthesis\ \rrparenthesis$ behaviour.

It is translated into abstract syntax as follows:

```
renaming ≡
     rename
       gate inP ($1 => ?a):int is intro ($1 => !exit($1 => a))
     in
       inP(!exit($1 => 3)) @any : time exit($1 => true) start⟨0⟩
```

We use rule 7.60 as follows:

$$\dfrac{\begin{array}{l}
\mathcal{C} \vdash \mathsf{int} \Rightarrow \textit{type} \\
\mathcal{C} \vdash (\,(\$1 \texttt{=>} \texttt{?}a) \ \Rightarrow^{\mathbf{PM}} \ (\$1 \texttt{=>} \mathsf{int})\,) \ \mapsto \ \llparenthesis\, a \texttt{=>} \mathsf{int} \,\rrparenthesis \\
\mathcal{C}; a \texttt{=>} \mathsf{int} \vdash \mathit{intro}\ (\$1 \texttt{=>} \texttt{!}\mathbf{exit}(\$1 \texttt{=>} a))\ \texttt{@}\mathbf{any} : \mathsf{time}\ \mathbf{exit}(\$1 \texttt{=>} \mathit{true})\ \mathbf{start}\langle 0\rangle \ \Longrightarrow \ \textit{guarded}\ \llparenthesis\ \rrparenthesis \\
\mathcal{C}; \mathit{inP} \Rightarrow \textit{gate}\langle(\$1 \texttt{=>} \mathbf{any})\rangle \vdash \mathit{inP}(\texttt{!}\mathbf{exit}(\$1 \texttt{=>} 3))\ \texttt{@}\mathbf{any} : \mathsf{time}\ \mathbf{exit}(\$1 \texttt{=>} \mathit{true})\ \mathbf{start}\langle 0\rangle \ \Longrightarrow \ \textit{guarded}\ \llparenthesis\ \rrparenthesis
\end{array}}{\mathcal{C} \vdash \mathit{renaming} \ \Longrightarrow \ \textit{guarded}\ \llparenthesis\ \rrparenthesis} \quad (7.60)$$

where all the premises are easily proved.

7.20 Assignment

In order to ensure that the assignment P`:=`E is semantically correct, we have to show that E is a correct expression that returns only one value, say with type T; and that we can match the pattern P against type T, producing some bindings RT. If this is fulfilled, then P`:=`E finishes immediately and returns the bindings RT. This is formally expressed in the following rule:

$$\dfrac{\begin{array}{l}\mathcal{C} \vdash E \Longrightarrow \mathit{exit} \llparenthesis \$1 \texttt{=>} T \rrparenthesis \\ \mathcal{C} \vdash (P \Rightarrow^{\mathbf{PM}} T) \mapsto \llparenthesis RT \rrparenthesis\end{array}}{\mathcal{C} \vdash P\texttt{:=}E \Longrightarrow \mathit{exit} \llparenthesis RT \rrparenthesis} \qquad (7.61)$$

Example 7.21 Let us see what kind of behaviour `?`x `:=` 5 is. It is translated to `?`x `:=` **exit**`($1 => 5)`. We use rule 7.61 to do the following proof:

$$\dfrac{\begin{array}{l}\mathcal{C} \vdash \mathbf{exit}(\$1 \texttt{=>} 5) \Longrightarrow \mathit{exit} \llparenthesis \$1 \texttt{=>} \mathsf{int} \rrparenthesis \\ \mathcal{C} \vdash (?x \Rightarrow^{\mathbf{PM}} \mathsf{int}) \mapsto \llparenthesis x \texttt{=>} \mathsf{int} \rrparenthesis\end{array}}{\mathcal{C} \vdash ?x \texttt{ := } \mathbf{exit}(\$1 \texttt{=>} 5) \Longrightarrow \mathit{exit} \llparenthesis x \texttt{=>} \mathsf{int} \rrparenthesis} \ (7.61)$$

where the first premise was proved in Example 7.9, and the second one is proved by using rule 7.102 assuming that variable x is declared in context $\mathcal{C}$ of type int. Example 7.23 in Section 7.22 shows that this assignment is not correct in a context where variable x is restricted to have type bool.

Example 7.22 We can write multiple assignments by means of records of patterns. Let us see that the behaviour `(?`x`,?`y`) := (5,`$true$`)` is semantically correct. It is translated to

$$(\$1 \texttt{=>} ?x, \$2 \texttt{=>} ?y) \texttt{ := } \mathbf{exit}(\$1 \texttt{=>} (\$1 \texttt{=>} 5, \$2 \texttt{=>} \mathit{true}))$$

We have to use rule 7.61 to conclude that the behaviour is correct and it binds both variables, x and y.

$$\dfrac{\begin{array}{l}\mathcal{C} \vdash \mathbf{exit}(\$1 \texttt{=>} (\$1 \texttt{=>} 5, \$2 \texttt{=>} \mathit{true})) \Longrightarrow \mathit{exit} \llparenthesis \$1 \texttt{=>} (\$1 \texttt{=>} \mathsf{int}, \$2 \texttt{=>} \mathsf{bool}) \rrparenthesis \\ \mathcal{C} \vdash ((\$1 \texttt{=>} ?x, \$2 \texttt{=>} ?y) \Rightarrow^{\mathbf{PM}} (\$1 \texttt{=>} \mathsf{int}, \$2 \texttt{=>} \mathsf{bool})) \mapsto \llparenthesis x \texttt{=>} \mathsf{int}, y \texttt{=>} \mathsf{bool} \rrparenthesis\end{array}}{\mathcal{C} \vdash (\$1 \texttt{=>} ?x, \$2 \texttt{=>} ?y) \texttt{ :=} \mathbf{exit}(\$1 \texttt{=>} (\$1 \texttt{=>} 5, \$2 \texttt{=>} \mathit{true})) \Longrightarrow \mathit{exit} \llparenthesis x \texttt{=>} \mathsf{int}, y \texttt{=>} \mathsf{bool} \rrparenthesis} \ (7.61)$$

where the premises are fulfilled thanks to the following proofs:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{C} \vdash 5 \texttt{=>} \mathsf{int}}{\mathcal{C} \vdash \$1 \texttt{=>} 5 \Rightarrow^{\mathbf{T}} \$1 \texttt{=>} \mathsf{int}}\ (7.100) \quad \dfrac{\mathcal{C} \vdash \mathit{true} \texttt{=>} \mathsf{bool}}{\mathcal{C} \vdash \$2 \texttt{=>} \mathit{true} \Rightarrow^{\mathbf{T}} \$2 \texttt{=>} \mathsf{bool}}\ (7.100)}{\mathcal{C} \vdash \$1 \texttt{=>} 5, \$2 \texttt{=>} \mathit{true} \Rightarrow^{\mathbf{T}} \$1 \texttt{=>} \mathsf{int}, \$2 \texttt{=>} \mathsf{bool}}\ (7.101)}{\mathcal{C} \vdash (\$1 \texttt{=>} 5, \$2 \texttt{=>} \mathit{true}) \Rightarrow^{\mathbf{T}} (\$1 \texttt{=>} \mathsf{int}, \$2 \texttt{=>} \mathsf{bool})}\ (7.98)}{\mathcal{C} \vdash (\$1 \texttt{=>} (\$1 \texttt{=>} 5, \$2 \texttt{=>} \mathit{true})) \Rightarrow^{\mathbf{T}} (\$1 \texttt{=>} (\$1 \texttt{=>} \mathsf{int}, \$2 \texttt{=>} \mathsf{bool}))}\ (7.100)}{\mathcal{C} \vdash \mathbf{exit}(\$1 \texttt{=>} (\$1 \texttt{=>} 5, \$2 \texttt{=>} \mathit{true})) \Longrightarrow \mathit{exit} \llparenthesis \$1 \texttt{=>} (\$1 \texttt{=>} \mathsf{int}, \$2 \texttt{=>} \mathsf{bool}) \rrparenthesis}\ (7.33)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{C} \vdash ?x \Rightarrow^{\mathbf{PM}} \mathsf{int} \mapsto \llparenthesis x \texttt{=>} \mathsf{int} \rrparenthesis}\ (7.103)}{\mathcal{C} \vdash \$1 \texttt{=>} ?x \Rightarrow^{\mathbf{PM}} \$1 \texttt{=>} \mathsf{int} \mapsto \llparenthesis x \texttt{=>} \mathsf{int} \rrparenthesis}\ (7.112) \quad \dfrac{\dfrac{}{\mathcal{C} \vdash ?y \Rightarrow^{\mathbf{PM}} \mathsf{bool} \mapsto \llparenthesis y \texttt{=>} \mathsf{bool} \rrparenthesis}\ (7.103)}{\mathcal{C} \vdash \$2 \texttt{=>} ?y \Rightarrow^{\mathbf{PM}} \$2 \texttt{=>} \mathsf{bool} \mapsto \llparenthesis y \texttt{=>} \mathsf{bool} \rrparenthesis}\ (7.112)}{\mathcal{C} \vdash \$1 \texttt{=>} ?x, \$2 \texttt{=>} ?y \Rightarrow^{\mathbf{PM}} \$1 \texttt{=>} \mathsf{int}, \$2 \texttt{=>} \mathsf{bool} \mapsto \llparenthesis x \texttt{=>} \mathsf{int}, y \texttt{=>} \mathsf{bool} \rrparenthesis}\ (7.113)}{\mathcal{C} \vdash ((\$1 \texttt{=>} ?x, \$2 \texttt{=>} ?y) \Rightarrow^{\mathbf{PM}} (\$1 \texttt{=>} \mathsf{int}, \$2 \texttt{=>} \mathsf{bool})) \mapsto \llparenthesis x \texttt{=>} \mathsf{int}, y \texttt{=>} \mathsf{bool} \rrparenthesis}\ (7.107)$$

7.21 Nondeterministic assignment

The nondeterministic assignment P `:=` **any** T `[`E`]` is correct in the context $\mathcal{C}$ if T is a type declared in the context, the pattern P can be matched against the type T by producing bindings RT, and in the context $\mathcal{C}; RT$ where the information in RT has been included, the *boolean* expression E is correct. This kind of assignment does an internal action **i** before finishing, indicating that a value of type T that fulfills E has been chosen in an internal way. Thus, the nondeterministic assignment is guarded and it produces bindings RT.

$$\frac{\begin{array}{l} \mathcal{C} \vdash T \Rightarrow \textit{type} \\ \mathcal{C} \vdash (P \ \Rightarrow^{\mathbf{PM}}\ T) \ \mapsto\ \langle\!|\, RT \,|\!\rangle \\ \mathcal{C}; RT \vdash E \ \Longrightarrow\ \textit{exit}\ \langle\!|\, \$1 \texttt{ => } \mathsf{bool} \,|\!\rangle \end{array}}{\mathcal{C} \vdash P \ \texttt{:=}\ \mathbf{any}\ T\ \texttt{[}\,E\,\texttt{]} \ \Longrightarrow\ \textit{guarded}\ \langle\!|\, RT \,|\!\rangle} \tag{7.62}$$

7.22 Variable declaration

In order to check if a variable declaration

$$\mathbf{var}\ V_1 \,\texttt{:}\, T_1[\texttt{:=}E_1], \ldots, V_n \,\texttt{:}\, T_n[\texttt{:=}E_n]\ \mathbf{in}\ B$$

is semantically correct in the context $\mathcal{C}$, the following conditions must be fulfilled:

1. The assignments to the initialized variables $V_k \,\texttt{:}\, T_k \,\texttt{:=} E_k$ must be correct behaviours in the context $\mathcal{C}$. If some expression E_k refers to a variable V with the same name than one in $\{V_1, \ldots, V_n\}$, it refers to a global variable with the same name V.
2. The behaviour B must be correct in the context $\mathcal{C}$ where the information that the declared variables have their type restricted ($V_i {\Rightarrow} ?T_i$) and the bindings produced by the assignments have been added. The behaviour B may produce bindings RT, RT'', where RT represents (global) variables not in $\{V_1, \ldots, V_n\}$ and RT'' represents declared variables modified by B.

The whole **var** behaviour is an exit or guarded behaviour depending on what B is, and it produces bindings RT in any case. The rules are:

$$\frac{\begin{array}{l} \forall\, j_i \in \{k \mid V_k \,\texttt{:}\, T_k \,\texttt{:=} E_k\} \,.\, \mathcal{C} \vdash ?V_{j_i} \,\texttt{:=} E_{j_i} \ \Longrightarrow\ \textit{exit}\ \langle\!|\, RT_{j_i} \,|\!\rangle \\ \mathcal{C}; (V_1 {\Rightarrow} ?T_1); \\ \quad \vdots \\ \quad (V_n {\Rightarrow} ?T_n); RT' \vdash B \ \Longrightarrow\ \textit{exit}\ \langle\!|\, RT, RT'' \,|\!\rangle \end{array}}{\mathcal{C} \vdash \mathbf{var}\ V_1 \,\texttt{:}\, T_1[\texttt{:=}E_1], \ldots, V_n \,\texttt{:}\, T_n[\texttt{:=}E_n]\ \mathbf{in}\ B \ \Longrightarrow\ \textit{exit}\ \langle\!|\, RT \,|\!\rangle} \tag{7.63}$$

with the side conditions that

- RT_{j_i}'s are pairwise disjoint, that is, variables $\{V_1, \ldots, V_n\}$ have different names,
- $RT' = RT_{j_1}, \ldots, RT_{j_m}$ where $\{j_1, \ldots, j_m\} \ = \ \{k \mid V_k \,\texttt{:}\, T_k \,\texttt{:=} E_k\}$,
- RT is disjoint with $V_1 \texttt{ => } T_1, \ldots, V_n \texttt{ => } T_n$, and
- $RT'' \subseteq V_1 \texttt{ => } T_1, \ldots, V_n \texttt{ => } T_n$.

$$\frac{\begin{array}{l}\forall\, j_i \in \{k \mid V_k\!:\!T_k\texttt{:=}E_k\}\,.\,\mathcal{C} \vdash \texttt{?}V_{j_i}\texttt{:=}E_{j_i} \implies \textit{exit}\,\langle\!| RT_{j_i} |\!\rangle \\ \mathcal{C};(V_1{\Rightarrow}\texttt{?}T_1); \\ \quad\vdots \\ \quad(V_n{\Rightarrow}\texttt{?}T_n); RT' \vdash B \implies \textit{guarded}\,\langle\!| RT, RT'' |\!\rangle\end{array}}{\mathcal{C} \vdash \textbf{var}\ V_1\!:\!T_1[\texttt{:=}E_1],\ldots,V_n\!:\!T_n[\texttt{:=}E_n]\ \textbf{in}\ B \implies \textit{guarded}\,\langle\!| RT |\!\rangle} \tag{7.64}$$

with the same side conditions.

Example 7.23 Let us check that the behaviour **var** x:bool **in** ?x:=5 **endvar** is not correct. It is translated to **var** x:bool **in** ?x:=**exit**(\$1 => 5).

We have to use one of the rules above (7.63 or 7.64) with only the last condition because there are no initialized variables:

$$\frac{\mathcal{C};\ x{\Rightarrow}\texttt{?bool} \vdash \texttt{?}x\texttt{:=}\textbf{exit}(\$1 \texttt{=>} 5) \implies \textbf{?}}{\mathcal{C} \vdash \textbf{var}\ x\!:\!\texttt{bool}\ \textbf{in}\ \texttt{?}x\texttt{:=}\textbf{exit}(\$1 \texttt{=>} 5) \implies \textbf{?}}$$

In order to prove the second premise, we have to use rule 7.61

$$\frac{\begin{array}{l}\mathcal{C};\ x{\Rightarrow}\texttt{?bool} \vdash \textbf{exit}(\$1 \texttt{=>} 5) \implies \textit{exit}\,\langle\!| \$1 \texttt{=>}\ \texttt{int} |\!\rangle \\ \mathcal{C};\ x{\Rightarrow}\texttt{?bool} \vdash (\texttt{?}x\ \ \Rightarrow^{\mathbf{PM}}\ \texttt{int}) \mapsto \textbf{?}\end{array}}{\mathcal{C};\ x{\Rightarrow}\texttt{?bool} \vdash \texttt{?}x\texttt{:=}\textbf{exit}(\$1 \texttt{=>} 5) \implies \textbf{?}}$$

Now, we have to use rule 7.102 (in Section 7.31.1) because there is a restriction about variable x in the context, but we cannot because int $\not\sqsubseteq$ bool.

Example 7.24 Let us check that the behaviour

$$\textbf{var}\ x\texttt{:int := 0 ,}y\texttt{:bool}\ \textbf{in}\ G\texttt{(!}x\texttt{);}G'\texttt{(?}y\texttt{)}\ \textbf{endvar}$$

is semantically correct. It is translated to

$$\begin{array}{l} \mathit{varDecl} \quad \equiv \\ \quad \textbf{var}\ x\texttt{:int := }\textbf{exit}(\$1 \texttt{=>} 0)\ \texttt{,}y\texttt{:bool}\ \textbf{in}\ G(\$1 \texttt{=> !}\textbf{exit}(\$1 \texttt{=>} x))\ldots\texttt{;}G'(\$1 \texttt{=> ?}y)\ldots \end{array}$$

We use rule 7.64 as follows:

$$\frac{\begin{array}{l}\mathcal{C} \vdash \texttt{?}x\ \texttt{:=}\ \textbf{exit}(\$1 \texttt{=>} 0) \implies \textit{exit}\,\langle\!| x \texttt{=>}\ \texttt{int} |\!\rangle \\ \mathcal{C}; x{\Rightarrow}\texttt{?int}; \\ y{\Rightarrow}\texttt{?bool}; x \texttt{=>}\ \texttt{int} \vdash G(\$1 \texttt{=> !}\textbf{exit}(\$1 \texttt{=>} x))\ldots\texttt{;}G'(\$1 \texttt{=> ?}y)\ldots \implies \textit{guarded}\,\langle\!| y \texttt{=>}\ \texttt{bool} |\!\rangle\end{array}}{\mathcal{C} \vdash \mathit{varDecl} \implies \textit{guarded}\,\langle\!|\ |\!\rangle} \tag{7.64}$$

with $RT'' = y$ => bool and RT equal to the empty list of bindings.

7.23 Case operator

The behaviour **case** E:T **is** BM (where BM represents the different branches of the **case** statement) is semantically correct in a context $\mathcal{C}$ if expression E returns a value of type T (in that context) and BM can be "behaviour pattern-matched" against type T by producing bindings RT. Behaviour pattern-matching is explained in the next section. If these premises are fulfilled, then the **case** behaviour returns bindings RT, and it is guarded only when the behaviour pattern-matching is guarded.

$$\frac{\begin{array}{l}\mathcal{C} \vdash T \Rightarrow \mathsf{type} \\ \mathcal{C} \vdash E \implies \mathsf{exit} \llparenthesis \$1 \texttt{=>} T \rrparenthesis \\ \mathcal{C} \vdash (BM \ \overset{\mathrm{PM}}{\Rightarrow} \ T) \implies \mathsf{exit} \llparenthesis RT \rrparenthesis\end{array}}{\mathcal{C} \vdash \textbf{case } E\!:\!T \textbf{ is } BM \implies \mathsf{exit} \llparenthesis RT \rrparenthesis} \tag{7.65}$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash E \implies \mathsf{exit} \llparenthesis \$1 \texttt{=>} T \rrparenthesis \\ \mathcal{C} \vdash (BM \ \overset{\mathrm{PM}}{\Rightarrow} \ T) \implies \mathsf{guarded} \llparenthesis RT \rrparenthesis\end{array}}{\mathcal{C} \vdash \textbf{case } E\!:\!T \textbf{ is } BM \implies \mathsf{guarded} \llparenthesis RT \rrparenthesis} \tag{7.66}$$

If more than one expression is used in the **case**, then each expression must be correct, and the branches BM must be behaviour pattern-matched against the record type built from the types of these expressions.

$$\frac{\begin{array}{l}\mathcal{C} \vdash T_1 \Rightarrow \mathsf{type} \ \cdots \ \mathcal{C} \vdash T_n \Rightarrow \mathsf{type} \\ \mathcal{C} \vdash E_1 \implies \textbf{exit}(\$1 \texttt{=>} T_1) \quad \cdots \quad \mathcal{C} \vdash E_n \implies \textbf{exit}(\$1 \texttt{=>} T_n) \\ \mathcal{C} \vdash (BM \ \overset{\mathrm{PM}}{\Rightarrow} \ (\$1 \texttt{=>} T_1, \ldots, \$n \texttt{=>} T_n)) \implies \mathsf{exit} \llparenthesis RT' \rrparenthesis\end{array}}{\mathcal{C} \vdash \textbf{case } (E_1\!:\!T_1, \ldots, E_n\!:\!T_n) \textbf{ is } BM \implies \mathsf{exit} \llparenthesis RT' \rrparenthesis} \tag{7.67}$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash T_1 \Rightarrow \mathsf{type} \ \cdots \ \mathcal{C} \vdash T_n \Rightarrow \mathsf{type} \\ \mathcal{C} \vdash E_1 \implies \textbf{exit}(\$1 \texttt{=>} T_1) \quad \cdots \quad \mathcal{C} \vdash E_n \implies \textbf{exit}(\$1 \texttt{=>} T_n) \\ \mathcal{C} \vdash (BM \ \overset{\mathrm{PM}}{\Rightarrow} \ (\$1 \texttt{=>} T_1, \ldots, \$n \texttt{=>} T_n)) \implies \mathsf{guarded} \llparenthesis RT' \rrparenthesis\end{array}}{\mathcal{C} \vdash \textbf{case } (E_1\!:\!T_1, \ldots, E_n\!:\!T_n) \textbf{ is } BM \implies \mathsf{guarded} \llparenthesis RT' \rrparenthesis} \tag{7.68}$$

Example 7.25 The behaviour

```
case ys:List is
        nil -> signal Hd
      | cons(?x,?xs) -> outP(!x)
endcase
```

communicates the head of the list ys through gate $outP$ if it is not empty. If ys is empty, the exception Hd is raised. Let us check that it is correct in the context $\mathcal{C} = \mathcal{C}', ys \texttt{=>} \mathsf{List}, Hd \Rightarrow \mathsf{exn}\langle(\$1 \texttt{=>} ())\rangle, outP \Rightarrow \mathsf{gate}\langle(\$1 \texttt{=>} \textbf{any})\rangle$.

It is translated to abstract syntax as:

$$\textbf{case exit}(\$1 \texttt{=>} ys)\!:\!\mathsf{List} \textbf{ is } BM_1 \mid (BM_2 \mid BM_3)$$

where

$$BM_1 \equiv Nil()\ [\textbf{exit}(\$1 \texttt{=>} true)]\texttt{->} \textbf{signal } Hd(\textbf{exit}(\$1 \texttt{=>} ()))$$

$$BM_2 \equiv Cons(\$1 \texttt{=>} ?x, \$2 \texttt{=>} ?xs)\ [\textbf{exit}(\$1 \texttt{=>} true)]\texttt{->} outP(\$1 \texttt{=>} !\textbf{exit}(\$1 \texttt{=>} x)) \ldots$$

$$BM_3 \equiv \textbf{any}\!:\!\mathsf{List}\ [\textbf{exit}(\$1 \texttt{=>} true)] \texttt{->} \textbf{signal } Match(\textbf{exit}(\$1 \texttt{=>} ()))$$

We have to use rule 7.66

$$\frac{\begin{array}{c}\mathcal{C} \vdash \mathbf{exit}(\$1 \texttt{=>}\ ys) \ \Longrightarrow\ \textsf{\textit{exit}}\ \llparenthesis\, \$1 \texttt{=>} \textsf{List} \,\rrparenthesis \\ \mathcal{C} \vdash (BM_1 \ \texttt{|}\ (BM_2 \ \texttt{|}\ BM_3) \ \Rightarrow^{\mathbf{PM}} \ \textsf{List}) \ \Longrightarrow\ \textsf{\textit{guarded}}\ \llparenthesis\ \rrparenthesis\end{array}}{\mathcal{C} \vdash \mathbf{case}\ \mathbf{exit}(\$1 \texttt{=>}\ ys)\texttt{:}\textsf{List}\ \mathbf{is}\ BM_1 \ \texttt{|}\ (BM_2 \ \texttt{|}\ BM_3) \ \Longrightarrow\ \textsf{\textit{guarded}}\ \llparenthesis\ \rrparenthesis} \ (7.66)$$

where the second premise is proved using rule 7.72 twice:

$$\frac{\mathcal{C} \vdash (BM_1 \ \Rightarrow^{\mathbf{PM}} \ \textsf{List}) \ \Longrightarrow\ \textsf{\textit{guarded}}\ \llparenthesis\ \rrparenthesis \quad \dfrac{\begin{array}{c}\mathcal{C} \vdash (BM_2 \ \Rightarrow^{\mathbf{PM}} \ \textsf{List}) \ \Longrightarrow\ \textsf{\textit{guarded}}\ \llparenthesis\ \rrparenthesis \\ \mathcal{C} \vdash (BM_3 \ \Rightarrow^{\mathbf{PM}} \ \textsf{List}) \ \Longrightarrow\ \textsf{\textit{guarded}}\ \llparenthesis\ \rrparenthesis\end{array}}{\mathcal{C} \vdash (BM_2 \ \texttt{|}\ BM_3 \ \Rightarrow^{\mathbf{PM}} \ \textsf{List}) \ \Longrightarrow\ \textsf{\textit{guarded}}\ \llparenthesis\ \rrparenthesis} \ (7.72)}{\mathcal{C} \vdash (BM_1 \ \texttt{|}\ (BM_2 \ \texttt{|}\ BM_3) \ \Rightarrow^{\mathbf{PM}} \ \textsf{List}) \ \Longrightarrow\ \textsf{\textit{guarded}}\ \llparenthesis\ \rrparenthesis} \ (7.72)$$

where all the premises are fulfilled due to the following proofs:

$$\frac{\begin{array}{l}\mathcal{C} \vdash (\mathit{Nil}() \ \Rightarrow^{\mathbf{PM}} \ \textsf{List}) \ \mapsto \ \llparenthesis\ \rrparenthesis \\ \mathcal{C} \vdash \mathbf{exit}(\$1 \texttt{=>}\ \mathit{true}) \ \Longrightarrow\ \textsf{\textit{exit}}\ \llparenthesis\, \$1 \texttt{=>} \textsf{bool} \,\rrparenthesis \\ \mathcal{C} \vdash \mathbf{signal}\ \mathit{Hd}(\mathbf{exit}(\$1 \texttt{=>}\ \texttt{()})) \ \Longrightarrow\ \textsf{\textit{guarded}}\ \llparenthesis\ \rrparenthesis\end{array}}{\mathcal{C} \vdash (BM_1 \ \Rightarrow^{\mathbf{PM}} \ \textsf{List}) \ \Longrightarrow\ \textsf{\textit{guarded}}\ \llparenthesis\ \rrparenthesis} \ (7.70)$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash (\mathit{Cons}(\$1 \texttt{=>}\ ?x\texttt{,}\$2 \texttt{=>}\ ?xs) \ \Rightarrow^{\mathbf{PM}} \ \textsf{List}) \ \mapsto \ \llparenthesis\, x \texttt{=>} \mathbf{any}, xs \texttt{=>} \textsf{List} \,\rrparenthesis \\ \mathcal{C}; x \texttt{=>} \mathbf{any}, xs \texttt{=>} \textsf{List} \vdash \mathbf{exit}(\$1 \texttt{=>}\ \mathit{true}) \ \Longrightarrow\ \textsf{\textit{exit}}\ \llparenthesis\, \$1 \texttt{=>} \textsf{bool} \,\rrparenthesis \\ \mathcal{C}; x \texttt{=>} \mathbf{any}, xs \texttt{=>} \textsf{List} \vdash \mathit{outP}(\$1 \texttt{=>}\ \texttt{!}\mathbf{exit}(\$1 \texttt{=>}\ x))\ \ldots \ \Longrightarrow\ \textsf{\textit{guarded}}\ \llparenthesis\ \rrparenthesis\end{array}}{\mathcal{C} \vdash (BM_2 \ \Rightarrow^{\mathbf{PM}} \ \textsf{List}) \ \Longrightarrow\ \textsf{\textit{guarded}}\ \llparenthesis\ \rrparenthesis} \ (7.70)$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash (\mathbf{any}\texttt{:}\textsf{List} \ \Rightarrow^{\mathbf{PM}} \ \textsf{List}) \ \mapsto \ \llparenthesis\ \rrparenthesis \\ \mathcal{C} \vdash \mathbf{exit}(\$1 \texttt{=>}\ \mathit{true}) \ \Longrightarrow\ \textsf{\textit{exit}}\ \llparenthesis\, \$1 \texttt{=>} \textsf{bool} \,\rrparenthesis \\ \mathcal{C} \vdash \mathbf{signal}\ \mathit{Match}(\mathbf{exit}(\$1 \texttt{=>}\ \texttt{()})) \ \Longrightarrow\ \textsf{\textit{guarded}}\ \llparenthesis\ \rrparenthesis\end{array}}{\mathcal{C} \vdash (BM_3 \ \Rightarrow^{\mathbf{PM}} \ \textsf{List}) \ \Longrightarrow\ \textsf{\textit{guarded}}\ \llparenthesis\ \rrparenthesis} \ (7.70)$$

7.24 Behaviour pattern-matching

In this section we explain how the different possibilities (or branches) of a **case** behaviour, BM, are matched against a type T.

7.24.1 Single match

The clause P `[`E`] ->` B can be behaviour pattern-matched against type T in the context $\mathcal{C}$ if:

- pattern P can be matched against type T in the context $\mathcal{C}$, by producing bindings RT,
- expression E returns a boolean value in the context $\mathcal{C}$ overridden by bindings RT, and
- behaviour B is correct in the context $\mathcal{C}$ overridden by bindings RT.

Then behaviour pattern-matching returns bindings produced by behaviour B, and it is guarded if B is.

$$\frac{\begin{array}{l}\mathcal{C} \vdash (P \ \Rightarrow^{\mathbf{PM}}\ T) \ \mapsto\ \llparenthesis\, RT \,\rrparenthesis \\ \mathcal{C}; RT \vdash E \ \Longrightarrow\ \textsf{\textit{exit}}\,\llparenthesis\, \$1 \texttt{ => } \textsf{bool} \,\rrparenthesis \\ \mathcal{C}; RT \vdash B \ \Longrightarrow\ \textsf{\textit{exit}}\,\llparenthesis\, RT' \,\rrparenthesis\end{array}}{\mathcal{C} \vdash ((P \texttt{ [}E\texttt{] -> } B) \ \Rightarrow^{\mathbf{PM}}\ T) \ \Longrightarrow\ \textsf{\textit{exit}}\,\llparenthesis\, RT' \,\rrparenthesis} \tag{7.69}$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash (P \ \Rightarrow^{\mathbf{PM}}\ T) \ \mapsto\ \llparenthesis\, RT \,\rrparenthesis \\ \mathcal{C}; RT \vdash E \ \Longrightarrow\ \textsf{\textit{exit}}\,\llparenthesis\, \$1 \texttt{ => } \textsf{bool} \,\rrparenthesis \\ \mathcal{C}; RT \vdash B \ \Longrightarrow\ \textsf{\textit{guarded}}\,\llparenthesis\, RT' \,\rrparenthesis\end{array}}{\mathcal{C} \vdash ((P \texttt{ [}E\texttt{] -> } B) \ \Rightarrow^{\mathbf{PM}}\ T) \ \Longrightarrow\ \textsf{\textit{guarded}}\,\llparenthesis\, RT' \,\rrparenthesis} \tag{7.70}$$

7.24.2 Multiple match

When there is more than one branch, BM_1 | BM_2, BM_1 and BM_2 are behaviour pattern-matched against type T. Regarding variables, it is like in the case of the selection operator, that is, initialized variables may be modified only by BM_1 or BM_2, but non-initialized variables (modified by BM_1 or BM_2) may be assigned by both. The multiple behaviour pattern-matching is guarded if all the branches are guarded.

$$\frac{\begin{array}{c}\mathcal{C}, RT_1, RT_2 \vdash (BM_1 \ \Rightarrow^{\mathbf{PM}}\ T) \ \Longrightarrow\ \textsf{\textit{exit}}\,\llparenthesis\, RT_1, RT \,\rrparenthesis \\ \mathcal{C}, RT_1, RT_2 \vdash (BM_2 \ \Rightarrow^{\mathbf{PM}}\ T) \ \Longrightarrow\ \textsf{\textit{exit}}\,\llparenthesis\, RT_2, RT \,\rrparenthesis\end{array}}{\mathcal{C}, RT_1, RT_2 \vdash (BM_1 \texttt{ | } BM_2 \ \Rightarrow^{\mathbf{PM}}\ T) \ \Longrightarrow\ \textsf{\textit{exit}}\,\llparenthesis\, RT_1, RT_2, RT \,\rrparenthesis} \tag{7.71}$$

$$\frac{\begin{array}{c}\mathcal{C}, RT_1, RT_2 \vdash (BM_1 \ \Rightarrow^{\mathbf{PM}}\ T) \ \Longrightarrow\ \textsf{\textit{guarded}}\,\llparenthesis\, RT_1, RT \,\rrparenthesis \\ \mathcal{C}, RT_1, RT_2 \vdash (BM_2 \ \Rightarrow^{\mathbf{PM}}\ T) \ \Longrightarrow\ \textsf{\textit{guarded}}\,\llparenthesis\, RT_2, RT \,\rrparenthesis\end{array}}{\mathcal{C}, RT_1, RT_2 \vdash (BM_1 \texttt{ | } BM_2 \ \Rightarrow^{\mathbf{PM}}\ T) \ \Longrightarrow\ \textsf{\textit{guarded}}\,\llparenthesis\, RT_1, RT_2, RT \,\rrparenthesis} \tag{7.72}$$

7.25 Iteration loop

The infinite loop **loop** B is semantically correct if behaviour B is, and it does not finish. It is indicated as in the case of the **stop** behaviour with binding \$1 => **none**.

$$\frac{\mathcal{C} \vdash B \ \Longrightarrow\ \textsf{\textit{exit}}\,\llparenthesis\, RT \,\rrparenthesis}{\mathcal{C} \vdash \mathbf{loop}\ B \ \Longrightarrow\ \textsf{\textit{guarded}}\,\llparenthesis\, \$1 \texttt{ => } \mathbf{none} \,\rrparenthesis} \tag{7.73}$$

7.26 Process instantiation

An abstract syntax process instantiation

$$\Pi \texttt{ [} G_1 \texttt{,} \ldots \texttt{,} G_m \texttt{] (} E_1 \texttt{,} \ldots \texttt{,} E_p \texttt{) [} X_1 \texttt{,} \ldots \texttt{,} X_n \texttt{]}$$

is semantically correct in the context $\mathcal{C}$ if all the following conditions are fulfilled:

- The identifier Π is declared in the context $\mathcal{C}$,

$$\begin{array}{ll}\mathcal{C} \vdash \Pi \Rightarrow & \texttt{[} \textsf{\textit{gate}}\langle \texttt{(}\$1 \texttt{ => } T_1\texttt{)} \rangle \texttt{,} \ldots \texttt{,} \textsf{\textit{gate}}\langle \texttt{(}\$1 \texttt{ => } T_m\texttt{)} \rangle \texttt{]} \\ & \texttt{(}V_1 \texttt{:} T'_1, \ldots, V_p \texttt{:} T'_p\texttt{)} \\ & \texttt{[} \textsf{\textit{exn}}\langle \texttt{(}\$1 \texttt{ => } T''_1\texttt{)} \rangle \texttt{,} \ldots \texttt{,} \textsf{\textit{exn}}\langle \texttt{(}\$1 \texttt{ => } T''_n\texttt{)} \rangle \texttt{]} \rightarrow \textsf{\textit{exit}}\,\llparenthesis\, RT \,\rrparenthesis.\end{array}$$

- The *actual* gates, that is, the gate identifiers in the instantiation, are declared in the context $\mathcal{C}$, and their types correspond with the types in the process declaration,

$$\mathcal{C} \vdash G_1 \Rightarrow \mathit{gate}\langle(\$1 \texttt{=>} T_1)\rangle \ \cdots \ \mathcal{C} \vdash G_m \Rightarrow \mathit{gate}\langle(\$1 \texttt{=>} T_m)\rangle.$$

- The actual (input) parameters are expressions that return values of the corresponding types,

$$\mathcal{C} \vdash E_1 \implies \mathit{exit} \llparenthesis \$1 \texttt{=>} T_1' \rrparenthesis \ \cdots \ \mathcal{C} \vdash E_p \implies \mathit{exit} \llparenthesis \$1 \texttt{=>} T_p' \rrparenthesis.$$

- The actual exceptions are declared exceptions that are raised together with types that correspond with the types of the formal exceptions

$$\mathcal{C} \vdash X_1 \Rightarrow \mathit{exn}\langle(\$1 \texttt{=>} T_1'')\rangle \ \cdots \ \mathcal{C} \vdash X_m \Rightarrow \mathit{exn}\langle(\$1 \texttt{=>} T_n'')\rangle.$$

If these conditions are fulfilled, then the instantiation is an $\mathit{exit} \llparenthesis RT \rrparenthesis$ behaviour, as stated by the following rule:

$$\frac{\begin{array}{l} \mathcal{C} \vdash \Pi \Rightarrow \texttt{[}\,\mathit{gate}\langle(\$1 \texttt{=>} T_1)\rangle, \ldots, \mathit{gate}\langle(\$1 \texttt{=>} T_m)\rangle\,\texttt{]} \\ \qquad\qquad (V_1 : T_1', \ldots, V_p : T_p') \\ \qquad\qquad \texttt{[}\,\mathit{exn}\langle(\$1 \texttt{=>} T_1'')\rangle, \ldots, \mathit{exn}\langle(\$1 \texttt{=>} T_n'')\rangle\,\texttt{]} \rightarrow \mathit{exit} \llparenthesis RT \rrparenthesis \\ \mathcal{C} \vdash G_1 \Rightarrow \mathit{gate}\langle(\$1 \texttt{=>} T_1)\rangle \ \cdots \ \mathcal{C} \vdash G_m \Rightarrow \mathit{gate}\langle(\$1 \texttt{=>} T_m)\rangle \\ \mathcal{C} \vdash E_1 \implies \mathit{exit} \llparenthesis \$1 \texttt{=>} T_1' \rrparenthesis \ \cdots \ \mathcal{C} \vdash E_p \implies \mathit{exit} \llparenthesis \$1 \texttt{=>} T_p' \rrparenthesis \\ \mathcal{C} \vdash X_1 \Rightarrow \mathit{exn}\langle(\$1 \texttt{=>} T_1'')\rangle \ \cdots \ \mathcal{C} \vdash X_m \Rightarrow \mathit{exn}\langle(\$1 \texttt{=>} T_n'')\rangle \end{array}}{\mathcal{C} \vdash \Pi\ \texttt{[}G_1, \ldots, G_m\texttt{]}\ (E_1, \ldots, E_p)\ \texttt{[}X_1, \ldots, X_n\texttt{]} \implies \mathit{exit} \llparenthesis RT \rrparenthesis} \tag{7.74}$$

If the process declaration in the context says that Π is a $\mathit{guarded} \llparenthesis RT \rrparenthesis$ behaviour and the same conditions as above are fulfilled, then the process instantiation is of type $\mathit{guarded} \llparenthesis RT \rrparenthesis$.

$$\frac{\begin{array}{l} \mathcal{C} \vdash \Pi \Rightarrow \texttt{[}\,\mathit{gate}\langle(\$1 \texttt{=>} T_1)\rangle, \ldots, \mathit{gate}\langle(\$1 \texttt{=>} T_m)\rangle\,\texttt{]} \\ \qquad\qquad (V_1 : T_1', \ldots, V_p : T_p') \\ \qquad\qquad \texttt{[}\,\mathit{exn}\langle(\$1 \texttt{=>} T_1'')\rangle, \ldots, \mathit{exn}\langle(\$1 \texttt{=>} T_n'')\rangle\,\texttt{]} \rightarrow \mathit{guarded} \llparenthesis RT \rrparenthesis \\ \mathcal{C} \vdash G_1 \Rightarrow \mathit{gate}\langle(\$1 \texttt{=>} T_1)\rangle \ \cdots \ \mathcal{C} \vdash G_m \Rightarrow \mathit{gate}\langle(\$1 \texttt{=>} T_m)\rangle \\ \mathcal{C} \vdash E_1 \implies \mathit{exit} \llparenthesis \$1 \texttt{=>} T_1' \rrparenthesis \ \cdots \ \mathcal{C} \vdash E_p \implies \mathit{exit} \llparenthesis \$1 \texttt{=>} T_p' \rrparenthesis \\ \mathcal{C} \vdash X_1 \Rightarrow \mathit{exn}\langle(\$1 \texttt{=>} T_1'')\rangle \ \cdots \ \mathcal{C} \vdash X_m \Rightarrow \mathit{exn}\langle(\$1 \texttt{=>} T_n'')\rangle \end{array}}{\mathcal{C} \vdash \Pi\ \texttt{[}G_1, \ldots, G_m\texttt{]}\ (E_1, \ldots, E_p)\ \texttt{[}X_1, \ldots, X_n\texttt{]} \implies \mathit{guarded} \llparenthesis RT \rrparenthesis} \tag{7.75}$$

Example 7.26 Let us check that the process instantiation

$$\mathsf{User2}\ \texttt{[}acc\texttt{,}abd\texttt{]}\,\texttt{(}myid\texttt{)}$$

is correct in a context $\mathcal{C}$ like

$$\begin{array}{rl} \mathcal{C} = & \mathcal{C}', acc \Rightarrow \mathit{gate}\langle(\$1 \texttt{=>} \mathsf{id})\rangle, abd \Rightarrow \mathit{gate}\langle(\$1 \texttt{=>} \mathsf{id})\rangle, myid \texttt{=>} \mathsf{id} \\ & \mathsf{User2} \Rightarrow \texttt{[}\,\mathit{gate}\langle(\$1 \texttt{=>} \mathsf{id})\rangle, \mathit{gate}\langle(\$1 \texttt{=>} \mathsf{id})\rangle\,\texttt{]}\,(myid : \mathsf{id})\ \texttt{[ ]} \rightarrow \mathit{guarded} \llparenthesis\ \rrparenthesis \end{array}$$

It is translated to

$$\mathsf{User2}\ \texttt{[}acc\texttt{,}abd\texttt{]}\,\texttt{(}\mathbf{exit}\texttt{(}\$1 \texttt{=>} myid\texttt{))}\ \texttt{[ ]}$$

and is easily checked using rule 7.75

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathrm{User2} \Rightarrow \texttt{[}\,\textsf{\textit{gate}}\langle\texttt{(\$1 => }\mathsf{id}\texttt{)}\rangle\texttt{,}\textsf{\textit{gate}}\langle\texttt{(\$1 => }\mathsf{id}\texttt{)}\rangle\,\texttt{]}\,\texttt{(}\mathit{myid}\texttt{:}\mathsf{id}\texttt{)}\,\texttt{[ ]} \rightarrow \textsf{\textit{guarded}}\ \llparenthesis\ \rrparenthesis \\ \mathcal{C} \vdash \mathit{acc} \Rightarrow \textsf{\textit{gate}}\langle\texttt{(\$1 => }\mathsf{id}\texttt{)}\rangle \\ \mathcal{C} \vdash \mathit{abd} \Rightarrow \textsf{\textit{gate}}\langle\texttt{(\$1 => }\mathsf{id}\texttt{)}\rangle \\ \mathcal{C} \vdash \mathbf{exit}\texttt{(\$1 => }\mathit{myid}\texttt{)} \implies \textsf{\textit{exit}}\ \llparenthesis\ \texttt{\$1 => }\mathsf{id}\ \rrparenthesis\end{array}}{\mathcal{C} \vdash \mathrm{User2}\ \texttt{[}\mathit{acc}\texttt{,}\mathit{abd}\texttt{]}\,\texttt{(}\mathbf{exit}\texttt{(\$1 => }\mathit{myid}\texttt{))}\,\texttt{[ ]} \implies \textsf{\textit{guarded}}\ \llparenthesis\ RT\ \rrparenthesis} \tag{7.75}$$

This example is related with Example 9.1 in Section 9.1.4.

7.27 Type expressions

In the abstract syntax, we have four different kinds of type expressions: an identifier, the empty type **none**, the universal type **any**, and a record type, $\texttt{(}RT\texttt{)}$.

7.27.1 Type identifier

The only information we can deduce about a type identifier S in a context $\mathcal{C}$ is what is included in the context. We can deduce that a type S is a declared type in the context $\mathcal{C}$,

$$\frac{}{\mathcal{C}, S \Rightarrow \textsf{\textit{type}} \vdash S \Rightarrow \textsf{\textit{type}}} \tag{7.76}$$

or that the type S is equivalent to another type T, that is, type T is the definition of the type identifier S,

$$\frac{}{\mathcal{C}, S \equiv T \vdash S \equiv T} \tag{7.77}$$

Note that these two rules are not new, but instantiations of the general axiom 7.23 in Section 7.1.

7.27.2 Empty type

The type **none**, which has no values, is a predefined type, so we can deduce that it is a type in any context $\mathcal{C}$,

$$\frac{}{\mathcal{C} \vdash \mathbf{none} \Rightarrow \textsf{\textit{type}}} \tag{7.78}$$

If we have a value of type **none**, then this value may be of *any* type because **none** has no values. Therefore, **none** is subtype of any type T.

$$\frac{}{\mathcal{C} \vdash \mathbf{none} \sqsubseteq T} \tag{7.79}$$

Finally, if we have a record type with a field V of type **none**, then the whole record type is equivalent to **none**, that is, since the field V can have no value, then there is no value of the whole record type,

$$\frac{}{\mathcal{C} \vdash \mathbf{none} \equiv \texttt{(}V \texttt{ => } \mathbf{none}\texttt{,}RT\texttt{)}} \tag{7.80}$$

7.27.3 Universal type

The type **any**, which includes all values of any type, is a predefined type,

$$\frac{}{\mathcal{C} \vdash \textbf{any} \Rightarrow \textit{type}} \tag{7.81}$$

Every type T is a subtype of **any**,

$$\frac{}{\mathcal{C} \vdash T \sqsubseteq \textbf{any}} \tag{7.82}$$

7.27.4 Record type

A record type `(`RT`)` is correct in the context $\mathcal{C}$ if we can deduce from $\mathcal{C}$ that RT is a correct record type expression (see next section),

$$\frac{\mathcal{C} \vdash RT \Rightarrow \textit{record}}{\mathcal{C} \vdash \texttt{(}RT\texttt{)} \Rightarrow \textit{type}} \tag{7.83}$$

A record type `(`RT`)` is a subtype of another record type `(`RT'`)` whenever RT is a subtype of RT' (see next section).

$$\frac{\mathcal{C} \vdash RT \sqsubseteq RT'}{\mathcal{C} \vdash \texttt{(}RT\texttt{)} \sqsubseteq \texttt{(}RT'\texttt{)}} \tag{7.84}$$

7.28 Record type expressions

A record type expression is what we can write enclosed between parentheses `()` to build a record type. They are lists (possibly empty) of types named with field names, V `=>` T, and they can include at the end the reserved word **etc** that represents any record type expression.

7.28.1 Empty record

The empty record type expression (which has no syntax, and is represented here by "()") is considered correct in the static semantics.

$$\frac{}{\mathcal{C} \vdash () \Rightarrow \textit{record}} \tag{7.85}$$

7.28.2 Singleton record

A singleton record type expression, V `=>` T, is correct in the context $\mathcal{C}$ if T is a type in that context.

$$\frac{\mathcal{C} \vdash T \Rightarrow \textit{type}}{\mathcal{C} \vdash (V \texttt{ => } T) \Rightarrow \textit{record}} \tag{7.86}$$

Do not confuse the parentheses (), that we use to group elements and has no meaning, with the parentheses `()` used to build record types.

Regarding the subtyping relation, if T is a subtype of T', then the record type expression V `=>` T is a subtype of V `=>` T'. Note that both record type expressions have the same field name.

$$\frac{\mathcal{C} \vdash T \sqsubseteq T'}{\mathcal{C} \vdash (V \texttt{=>} T) \sqsubseteq (V \texttt{=>} T')} \tag{7.87}$$

7.28.3 Record disjoint union

A record type expression with more than one field $RT_1\texttt{,}RT_2$ is a well-formed record type expression if both RT_1 and RT_2 are correct and they have disjoint fields.

$$\frac{\mathcal{C} \vdash RT_1 \Rightarrow \textsf{\textit{record}} \qquad \mathcal{C} \vdash RT_2 \Rightarrow \textsf{\textit{record}}}{\mathcal{C} \vdash RT_1\texttt{,}RT_2 \Rightarrow \textsf{\textit{record}}} \tag{7.88}$$

with the side condition that RT_1 and RT_2 have disjoint fields.

With respect to subtyping, the disjoint union $RT_1\texttt{,}RT_2$ is a subtype of $RT_1'\texttt{,}RT_2'$ if it can be proved that $RT_1 \sqsubseteq RT_1'$ and $RT_2 \sqsubseteq RT_2'$.

$$\frac{\mathcal{C} \vdash RT_1 \sqsubseteq RT_1' \qquad \mathcal{C} \vdash RT_2 \sqsubseteq RT_2'}{\mathcal{C} \vdash RT_1\texttt{,}RT_2 \sqsubseteq RT_1'\texttt{,}RT_2'} \tag{7.89}$$

The binary operator "," between records is commutative, associative, and it has the empty record as identity.

$$\frac{}{\mathcal{C} \vdash RT_1\texttt{,}RT_2 \equiv RT_2\texttt{,}RT_1} \tag{7.90}$$

$$\frac{}{\mathcal{C} \vdash (RT_1\texttt{,}RT_2)\texttt{,}RT_3 \equiv RT_1\texttt{,}(RT_2\texttt{,}RT_3)} \tag{7.91}$$

$$\frac{}{\mathcal{C} \vdash ()\texttt{,}RT \equiv RT} \tag{7.92}$$

7.28.4 Universal record

Intuitively, **etc** represents the fields that have no interest at certain level. It is a record type expression in any context

$$\frac{}{\mathcal{C} \vdash \textbf{etc} \Rightarrow \textsf{\textit{record}}} \tag{7.93}$$

and it is a supertype of any record type expression RT

$$\frac{}{\mathcal{C} \vdash RT \sqsubseteq \textbf{etc}} \tag{7.94}$$

7.29 Value Expressions

7.29.1 Primitive constants

A primitive constant K is a constant defined in the predefined library of E-LOTOS. We can deduce that a constant K has type T if this information is in the context. The initial context must include information about all constants in the predefined library of E-LOTOS.

$$\frac{\mathcal{C} \vdash K \Rightarrow T}{\mathcal{C} \vdash K \Rightarrow^{\mathbf{T}} T} \tag{7.95}$$

7.29.2 Variables

Variable V is a correct value expression of type T if it has been assigned a value of that type.

$$\frac{\mathcal{C} \vdash V \texttt{=>} T}{\mathcal{C} \vdash V \Rightarrow^{\mathbf{T}} T} \tag{7.96}$$

7.29.3 Constructor application

The constructor application $C\ N$ is correct and it has type S if the constructor C is declared of type $\texttt{(}RT\texttt{)} \to S$, and value N has type $\texttt{(}RT\texttt{)}$.

$$\frac{\begin{array}{l}\mathcal{C} \vdash C \Rightarrow (\texttt{(}RT\texttt{)} \to S) \\ \mathcal{C} \vdash N \Rightarrow^{\mathbf{T}} \texttt{(}RT\texttt{)}\end{array}}{\mathcal{C} \vdash C\ N \Rightarrow^{\mathbf{T}} S} \tag{7.97}$$

Example 7.27 Let us see the type of the value $\mathit{cons}\texttt{(}3\texttt{,}\mathit{nil}\texttt{())}$ in a context $\mathcal{C} = (\mathcal{C}', \mathit{cons} \Rightarrow \texttt{(}\mathbf{any}\texttt{,}\mathsf{List}\texttt{)} \to \mathsf{List}, \mathit{nil} \Rightarrow \texttt{()} \to \mathsf{List})$, where the constructors *cons* and *nil* are the predefined constructors. The proof is as follows:

$$\dfrac{\mathcal{C} \vdash \mathit{cons} \Rightarrow \texttt{(}\mathbf{any}\texttt{,}\mathsf{List}\texttt{)} \to \mathsf{List} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\mathcal{C} \vdash 3 \texttt{=>} \mathsf{int}}{\mathcal{C} \vdash 3 \Rightarrow^{\mathbf{T}} \mathsf{int}}\,(7.95) \quad \dfrac{}{\mathcal{C} \vdash \mathsf{int} \sqsubseteq \mathbf{any}}\,(7.82)}{\mathcal{C} \vdash 3 \Rightarrow^{\mathbf{T}} \mathbf{any}}\,(7.17) \quad \dfrac{\mathcal{C} \vdash \mathit{nil} \Rightarrow \texttt{()} \to \mathsf{List} \quad \dfrac{\dfrac{}{\mathcal{C} \vdash () \Rightarrow^{\mathbf{T}} ()}\,(7.99)}{\mathcal{C} \vdash \texttt{()} \Rightarrow^{\mathbf{T}} \texttt{()}}\,(7.98)}{\mathcal{C} \vdash \mathit{nil}\texttt{()} \Rightarrow^{\mathbf{T}} \mathsf{List}}\,(7.97)}{\mathcal{C} \vdash 3\texttt{,}\mathit{nil}\texttt{()} \Rightarrow^{\mathbf{T}} \mathbf{any}\texttt{,}\mathsf{List}}\,(7.101)}{\mathcal{C} \vdash \texttt{(}3\texttt{,}\mathit{nil}\texttt{())} \Rightarrow^{\mathbf{T}} \texttt{(}\mathbf{any}\texttt{,}\mathsf{List}\texttt{)}}\,(7.98)}{\mathcal{C} \vdash \mathit{cons}\texttt{(}3\texttt{,}\mathit{nil}\texttt{())} \Rightarrow^{\mathbf{T}} \mathsf{List}}\,(7.97)$$

7.29.4 Record values

We can deduce that the record value $\texttt{(}RN\texttt{)}$ has type $\texttt{(}RT\texttt{)}$ if the record value expression RN has type RT (see Section 7.30).

$$\frac{\mathcal{C} \vdash RN \Rightarrow^{\mathbf{T}} RT}{\mathcal{C} \vdash \texttt{(}RN\texttt{)} \Rightarrow^{\mathbf{T}} \texttt{(}RT\texttt{)}} \tag{7.98}$$

7.30 Record value expressions

A record value expression is a list (possibly empty) of values where each value has a field name.

7.30.1 Empty record

The empty record value expression (which has no syntax, and is represented here by ()) has empty record type (which is also represented by ()):

$$\frac{}{\mathcal{C} \vdash () \Rightarrow^{\mathbf{T}} ()} \tag{7.99}$$

7.30.2 Singleton record

If the value N has type T, then the record value expression V `=>` N has type V `=>` T.

$$\frac{\mathcal{C} \vdash N \Rightarrow^{\mathbf{T}} T}{\mathcal{C} \vdash (V \texttt{=>} N) \Rightarrow^{\mathbf{T}} (V \texttt{=>} T)} \tag{7.100}$$

7.30.3 Record disjoint union

The type of a record value expression with more than one field is built from the types of each field, in a recursive way.

$$\frac{\mathcal{C} \vdash RN_1 \Rightarrow^{\mathbf{T}} RT_1 \qquad \mathcal{C} \vdash RN_2 \Rightarrow^{\mathbf{T}} RT_2}{\mathcal{C} \vdash RN_1\texttt{,}\ RN_2 \Rightarrow^{\mathbf{T}} RT_1\texttt{,}\ RT_2} \tag{7.101}$$

with the side condition that RN_1 and RN_2 have disjoint fields.

7.31 Patterns

We are going to see now the meaning of the different patterns used in E-LOTOS. The following rules define how pattern-matching is performed. In the static semantics, patterns are matched against types. The pattern-matching produces bindings between these variables and types, meaning that variables have been given a value. As we will see in Chapter 8, patterns are matched against values in the dynamic semantics.

7.31.1 Variable pattern

If pattern `?`V is matched against type T in context $\mathcal{C}$, there are two cases:

- if variable V is declared with a **var** operator, then it is restricted, and something like V`=>?`T' will be in the context $\mathcal{C}$. In this case, the pattern `?`V can be matched against type T provided that T is a subtype of T'. The binding produced binds V with the type T:

$$\frac{\mathcal{C} \vdash T \sqsubseteq T'}{\mathcal{C} \vdash (\texttt{?}V \ \Rightarrow^{\mathbf{PM}}\ T) \ \mapsto\ \langle\!|\, V \texttt{=>} T \,|\!\rangle} \ [V \Rightarrow ?T' \in \mathcal{C}] \tag{7.102}$$

- if variable V is not declared, then `?`V can be matched against any type T, and the binding produced binds V with the type T:

$$\frac{}{\mathcal{C} \vdash (\texttt{?}V \ \Rightarrow^{\mathbf{PM}}\ T) \ \mapsto\ \langle\!|\, V \texttt{=>} T \,|\!\rangle} \ [\nexists V \Rightarrow ?T' \in \mathcal{C}] \tag{7.103}$$

These are the rules in [Que98] because there variables *do not need* to be declared. But we have discovered that a process declaration is semantically correct (Section 9.1.4) if the body of the process only produces those bindings related to **out** parameters. This implies that (local) variables used in the body have to be in the scope of a **var** declaration because bindings related to these variables cannot go outside the process body. And, since all the behaviours are declared inside a process declaration, this implies that variables *have to be declared*, as we said at the beginning of the tutorial. Thus, rule 7.103 will never be used.

Another problem (present in [Que98]) is that when a variable V is bound to a type T and the binding V => T is included in the context, it overrides the information $V \Rightarrow ?T'$, so it is lost the information about the type T' used in the declaration of the variable V. Therefore, rule 7.103 could be used and the variable V could be assigned a value of any type. For example, the behaviour

```
var x:bool in
  ?x := true ; ?x := 5
endvar
```

would be semantically correct! We have propose to remove rule 7.103 and to modify rule 7.102 as follows:

$$\frac{\mathcal{C} \vdash T \sqsubseteq T'}{\mathcal{C} \vdash (?V \ \Rightarrow^{\mathbf{PM}} \ T) \ \mapsto \ \langle\!| \, V \texttt{=>} T' \,|\!\rangle} \ [V \Rightarrow ?T' \in \mathcal{C} \lor V \texttt{=>} T' \in \mathcal{C}] \tag{7.104}$$

That is, the *static* type T' used in the declaration of variable V is always kept together with the variable. So the behaviour

```
var x:any in
  ?x := true ; ?x := 5
endvar
```

is semantically correct, but

```
var x:bool in
  ?x := true ; ?x := 5
endvar
```

is not.

7.31.2 Expression pattern

Pattern !E can be matched against type T if expression E returns a value of type T.

$$\frac{\mathcal{C} \vdash E \implies \mathit{exit} \,\langle\!| \, \$1 \texttt{=>} T \,|\!\rangle}{\mathcal{C} \vdash (!E \ \Rightarrow^{\mathbf{PM}} \ T) \ \mapsto \ \langle\!| \ |\!\rangle} \tag{7.105}$$

Note that although the syntax of E-LOTOS allows to write things like !(?x:=3), it has no meaning, that is, it cannot be matched against any type. In a pattern !E, we can only write an expression E that returns on value (which may be a record), so we can write !3, or !**if** x>0 **then** 3 **else** 5 **endif**.

7.31.3 Wildcard pattern

In [Que98] it is said that the wildcard pattern **any**:T is well formed in the context $\mathcal{C}$ if T is a type in that context, and it can be matched only against type T.

$$\frac{\mathcal{C} \vdash T \Rightarrow \textsf{\textit{type}}}{\mathcal{C} \vdash (\textbf{any}\texttt{:}T \ \Rightarrow^{\mathbf{PM}} \ T) \ \mapsto \ \llparenthesis \ \rrparenthesis}$$

But then we have that $\texttt{!3:}\mathsf{nat} \ \Rightarrow^{\mathbf{PM}} \ \mathsf{int}$ (assuming $\mathsf{nat} \sqsubseteq \mathsf{int}$) but $\textbf{any}\texttt{:}\mathsf{nat} \ \not\Rightarrow^{\mathbf{PM}} \ \mathsf{int}$.

We have propose to modify this rule and to say that the pattern $\textbf{any}\texttt{:}T$ can be matched against type T' if T is a subtype of T'.

$$\frac{\begin{array}{c}\mathcal{C} \vdash T \Rightarrow \textsf{\textit{type}} \\ \mathcal{C} \vdash T \sqsubseteq T'\end{array}}{\mathcal{C} \vdash (\textbf{any}\texttt{:}T \ \Rightarrow^{\mathbf{PM}} \ T') \ \mapsto \ \llparenthesis \ \rrparenthesis} \tag{7.106}$$

7.31.4 Record pattern

The record pattern $\texttt{(}RP\texttt{)}$ can be matched against the record type $\texttt{(}RT\texttt{)}$ provided that the record of patterns RP can be matched against RT (see Section 7.32).

$$\frac{\mathcal{C} \vdash (RP \ \Rightarrow^{\mathbf{PM}} \ RT') \ \mapsto \ \llparenthesis\, RT \,\rrparenthesis}{\mathcal{C} \vdash (\texttt{(}RP\texttt{)} \ \Rightarrow^{\mathbf{PM}} \ \texttt{(}RT'\texttt{)}) \ \mapsto \ \llparenthesis\, RT \,\rrparenthesis} \tag{7.107}$$

$\texttt{(}RP\texttt{)}$ can also be matched against the universal type **any**, provided that RP can be matched against the record type which has the same fields as RP and all its fields have type **any**:

$$\frac{\mathcal{C} \vdash (V_1 \texttt{=>} P_1\texttt{,} \ldots \texttt{,} V_n \texttt{=>} P_n \ \Rightarrow^{\mathbf{PM}} \ V_1 \texttt{=>} \textbf{any}\texttt{,} \ldots \texttt{,} V_n \texttt{=>} \textbf{any}) \ \mapsto \ \llparenthesis\, RT \,\rrparenthesis}{\mathcal{C} \vdash (\texttt{(}V_1 \texttt{=>} P_1\texttt{,} \ldots \texttt{,} V_n \texttt{=>} P_n\texttt{)} \ \Rightarrow^{\mathbf{PM}} \ \textbf{any}) \ \mapsto \ \llparenthesis\, RT \,\rrparenthesis} \tag{7.108}$$

Example 7.28 The following proof shows how the pattern $\texttt{(}\$1\texttt{=>!}\textbf{exit}\texttt{(}\$1\texttt{=>}3\texttt{))}$ can be matched against the type $\texttt{(}\$1 \texttt{=>} \mathsf{int}\texttt{)}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{C} \vdash 3 \texttt{=>} \mathsf{int}}{\mathcal{C} \vdash 3 \Rightarrow^{\mathbf{T}} \mathsf{int}}\,(7.95)}{\mathcal{C} \vdash \$1 \texttt{=>} 3 \Rightarrow^{\mathbf{T}} \$1 \texttt{=>} \mathsf{int}}\,(7.100)}{\mathcal{C} \vdash \textbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{)} \ \Longrightarrow \ \textsf{\textit{exit}}\,\llparenthesis\, \$1 \texttt{=>} \mathsf{int} \,\rrparenthesis}\,(7.33)}{\mathcal{C} \vdash (\texttt{!}\textbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{)} \ \Rightarrow^{\mathbf{PM}} \ \mathsf{int}) \ \mapsto \ \llparenthesis \ \rrparenthesis}\,(7.105)}{\mathcal{C} \vdash (\$1 \texttt{=>} \texttt{!}\textbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{)} \ \Rightarrow^{\mathbf{PM}} \ \$1 \texttt{=>} \mathsf{int}) \ \mapsto \ \llparenthesis \ \rrparenthesis}\,(7.112)}{\mathcal{C} \vdash (\texttt{(}\$1 \texttt{=>} \texttt{!}\textbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{))} \ \Rightarrow^{\mathbf{PM}} \ \texttt{(}\$1 \texttt{=>} \mathsf{int}\texttt{)}) \ \mapsto \ \llparenthesis \ \rrparenthesis}\,(7.107)$$

Example 7.29 Let us see how the pattern $\texttt{(!(}x \texttt{=>} 2\texttt{,}y \texttt{=>} \textit{true}\texttt{))}$, which is translated to

$$\texttt{(}\$1 \texttt{=>} \texttt{!}\textbf{exit}\texttt{(}\$1 \texttt{=>} \texttt{(}x \texttt{=>} 2\texttt{,}y \texttt{=>} \textit{true}\texttt{)))},$$

can be patter-matched against type **any**.

The proof is as follows:

$$\dfrac{\dfrac{T1 \qquad\qquad T2}{\mathcal{C} \vdash (x \texttt{=>} \texttt{!}\textbf{exit}\texttt{(}\$1 \texttt{=>} 2\texttt{)}\texttt{,}y \texttt{=>} \texttt{!}\textbf{exit}\texttt{(}\$1 \texttt{=>} \textit{true}) \ \Rightarrow^{\mathbf{PM}} \ x \texttt{=>} \textbf{any},y \texttt{=>} \textbf{any}) \ \mapsto \ \llparenthesis \ \rrparenthesis}\,(7.113)}{\mathcal{C} \vdash (\texttt{(}x \texttt{=>} \texttt{!}\textbf{exit}\texttt{(}\$1 \texttt{=>} 2\texttt{)}\texttt{,}y \texttt{=>} \texttt{!}\textbf{exit}\texttt{(}\$1 \texttt{=>} \textit{true}\texttt{))} \ \Rightarrow^{\mathbf{PM}} \ \textbf{any}) \ \mapsto \ \llparenthesis \ \rrparenthesis}\,(7.108)$$

where the proof trees $T1$ and $T2$ are

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{C} \vdash 2 \Rightarrow \mathsf{int}}{\mathcal{C} \vdash 2 \Rightarrow^{\mathrm{T}} \mathsf{int}}\,(7.95)}{\mathcal{C} \vdash \$1 \texttt{=>} 2 \Rightarrow^{\mathrm{T}} \$1 \texttt{=>} \mathsf{int}}\,(7.100)}{\mathcal{C} \vdash \mathbf{exit}(\$1 \texttt{=>} 2) \Longrightarrow \mathit{exit}\llparenthesis \$1 \texttt{=>} \mathsf{int} \rrparenthesis}\,(7.33) \quad \dfrac{\dfrac{}{\mathcal{C} \vdash \mathsf{int} \sqsubseteq \mathbf{any}}\,(7.82)}{\mathcal{C} \vdash \$1 \texttt{=>} \mathsf{int} \sqsubseteq \$1 \texttt{=>} \mathbf{any}}\,(7.87)}{\mathcal{C} \vdash \mathbf{exit}(\$1 \texttt{=>} 2) \Longrightarrow \mathit{exit}\llparenthesis \$1 \texttt{=>} \mathbf{any} \rrparenthesis}\,(7.15)}{\mathcal{C} \vdash (\texttt{!}\mathbf{exit}(\$1 \texttt{=>} 2) \ \Rightarrow^{\mathrm{PM}} \ \mathbf{any}) \ \mapsto \ \llparenthesis\ \rrparenthesis}\,(7.105)}{\mathcal{C} \vdash (x \texttt{=>}\ \texttt{!}\mathbf{exit}(\$1 \texttt{=>} 2) \ \Rightarrow^{\mathrm{PM}} \ x \texttt{=>} \mathbf{any}) \ \mapsto \ \llparenthesis\ \rrparenthesis}\,(7.112)$$

and

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{C} \vdash \mathit{true} \Rightarrow \mathsf{bool}}{\mathcal{C} \vdash \mathit{true} \Rightarrow^{\mathrm{T}} \mathsf{bool}}\,(7.95)}{\mathcal{C} \vdash \$1 \texttt{=>} \mathit{true} \Rightarrow^{\mathrm{T}} \$1 \texttt{=>} \mathsf{bool}}\,(7.100)}{\mathcal{C} \vdash \mathbf{exit}(\$1 \texttt{=>} \mathit{true}) \Longrightarrow \mathit{exit}\llparenthesis \$1 \texttt{=>} \mathsf{bool} \rrparenthesis}\,(7.33) \quad \dfrac{\dfrac{}{\mathcal{C} \vdash \mathsf{bool} \sqsubseteq \mathbf{any}}\,(7.82)}{\mathcal{C} \vdash \$1 \texttt{=>} \mathsf{bool} \sqsubseteq \$1 \texttt{=>} \mathbf{any}}\,(7.87)}{\mathcal{C} \vdash \mathbf{exit}(\$1 \texttt{=>} \mathit{true}) \Longrightarrow \mathit{exit}\llparenthesis \$1 \texttt{=>} \mathbf{any} \rrparenthesis}\,(7.15)}{\mathcal{C} \vdash (\texttt{!}\mathbf{exit}(\$1 \texttt{=>} \mathit{true}) \ \Rightarrow^{\mathrm{PM}} \ \mathbf{any}) \ \mapsto \ \llparenthesis\ \rrparenthesis}\,(7.105)}{\mathcal{C} \vdash (y \texttt{=>}\ \texttt{!}\mathbf{exit}(\$1 \texttt{=>} \mathit{true}) \ \Rightarrow^{\mathrm{PM}} \ y \texttt{=>} \mathbf{any}) \ \mapsto \ \llparenthesis\ \rrparenthesis}\,(7.112)$$

7.31.5 Constructor pattern

Pattern $C\ P$ can be matched against type T in the context $\mathcal{C}$ if C is a constructor declared in $\mathcal{C}$, $C \Rightarrow (RT) \rightarrow S$, S is a subtype of T (in this case T may be only S or **any**), and pattern P can be matched against record type (RT). Bindings returned by this last pattern-matching are also returned by the first one.

$$\frac{\begin{array}{l}\mathcal{C} \vdash C \Rightarrow (RT) \rightarrow S \\ \mathcal{C} \vdash S \sqsubseteq T \\ \mathcal{C} \vdash (P \ \Rightarrow^{\mathrm{PM}} \ (RT)) \ \mapsto \ \llparenthesis RT' \rrparenthesis\end{array}}{\mathcal{C} \vdash (C\ P \ \Rightarrow^{\mathrm{PM}} \ T') \ \mapsto \ \llparenthesis RT' \rrparenthesis} \tag{7.109}$$

Example 7.30 Pattern $\mathit{cons}(?x, ?xs)$ can be matched against type $\mathsf{intlist}$ in a context $\mathcal{C} = \mathcal{C}', \mathit{cons} \Rightarrow (\mathsf{int}, \mathsf{intlist}) \rightarrow \mathsf{intlist}$, and it produces bindings $x \texttt{=>} \mathsf{int}, xs \texttt{=>} \mathsf{intlist}$. The proof is as follows:

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathit{cons} \Rightarrow (\mathsf{int}, \mathsf{intlist}) \rightarrow \mathsf{intlist} \\ \mathcal{C} \vdash \mathsf{intlist} \sqsubseteq \mathsf{intlist} \\ \mathcal{C} \vdash ((?x, ?xs) \ \Rightarrow^{\mathrm{PM}} \ (\mathsf{int}, \mathsf{intlist})) \ \mapsto \ \llparenthesis x \texttt{=>} \mathsf{int}, xs \texttt{=>} \mathsf{intlist} \rrparenthesis\end{array}}{\mathcal{C} \vdash (\mathit{cons}(?x, ?xs) \ \Rightarrow^{\mathrm{PM}} \ \mathsf{intlist}) \ \mapsto \ \llparenthesis x \texttt{=>} \mathsf{int}, xs \texttt{=>} \mathsf{intlist} \rrparenthesis}\,(7.109)$$

7.31.6 Explicit typing pattern

The explicit typing pattern $P{:}T$ is correct in the context $\mathcal{C}$ provided that T is a type in that context, and it can be matched against type T' whenever P can be matched against type T (producing bindings RT) and T is a subtype of T'. The bindings RT are produced.

$$\frac{\mathcal{C} \vdash T \Rightarrow \textsf{\textit{type}} \qquad \mathcal{C} \vdash T \sqsubseteq T' \qquad \mathcal{C} \vdash (P \ \overset{\mathbf{PM}}{\Rightarrow}\ T) \ \mapsto\ \langle\!| RT |\!\rangle}{\mathcal{C} \vdash (P\!:\!T \ \overset{\mathbf{PM}}{\Rightarrow}\ T') \ \mapsto\ \langle\!| RT |\!\rangle} \tag{7.110}$$

7.32 Record of patterns

A record of patterns, RP, is a list (possibly empty) of patterns separated by commas, where each element is preceded by a field name, V_1 `=>` P_1 `,` ... `,`V_n `=>` P_n.

7.32.1 Empty record pattern

The empty record of patterns (which has no syntax, and is represented here by "()") can be only matched against the empty record type (also represented by "()").

$$\frac{}{\mathcal{C} \vdash (() \ \overset{\mathbf{PM}}{\Rightarrow}\ ()) \ \mapsto\ \langle\!| \ |\!\rangle} \tag{7.111}$$

7.32.2 Singleton record pattern

A singleton record of patterns V `=>` P can be matched against type V `=>` T (note that the same field name is used), provided that pattern P can be matched against type T.

$$\frac{\mathcal{C} \vdash (P \ \overset{\mathbf{PM}}{\Rightarrow}\ T) \ \mapsto\ \langle\!| RT |\!\rangle}{\mathcal{C} \vdash ((V \texttt{ => } P) \ \overset{\mathbf{PM}}{\Rightarrow}\ (V \texttt{ => } T)) \ \mapsto\ \langle\!| RT |\!\rangle} \tag{7.112}$$

7.32.3 Record disjoint union

The record of patterns RP_1 `,` RP_2 can be matched against the record type RT_1 `,` RT_2, provided that RP_1 can be matched against RT_1, RP_2 can be matched against RT_2, and these pattern matchings produce disjoint bindings.

$$\frac{\mathcal{C} \vdash (RP_1 \ \overset{\mathbf{PM}}{\Rightarrow}\ RT_1) \ \mapsto\ \langle\!| RT_1' |\!\rangle \qquad \mathcal{C} \vdash (RP_2 \ \overset{\mathbf{PM}}{\Rightarrow}\ RT_2) \ \mapsto\ \langle\!| RT_2' |\!\rangle}{\mathcal{C} \vdash (RP_1\texttt{,}\ RP_2 \ \overset{\mathbf{PM}}{\Rightarrow}\ RT_1\texttt{,}\ RT_2) \ \mapsto\ \langle\!| RT_1', RT_2' |\!\rangle} \tag{7.113}$$

with the side condition that RT_1' and RT_2' must have disjoint fields.

7.32.4 Wildcard record pattern

The pattern **etc** can be used to be matched against any record type, RT.

$$\frac{}{\mathcal{C} \vdash (\mathbf{etc} \ \overset{\mathbf{PM}}{\Rightarrow}\ RT) \ \mapsto\ \langle\!| \ |\!\rangle} \tag{7.114}$$

8 Dynamic Semantics of Base Language

In this chapter we introduce the *dynamic operational semantics* of E-LOTOS. This semantics allows to deduce the transitions that a behaviour in the abstract syntax which has passed the static semantics can perform.

8.1 Introduction

The transitions that a behaviour B can perform are:

- $B \xrightarrow{G\texttt{(}RN\texttt{)}} B'$: behaviour B can perform an action on gate G, communicating the value $\texttt{(}RN\texttt{)}$, and becoming B' in doing so. As we saw in the static semantics, a gate can communicate only values which are records with one field, named \$1, so $\texttt{(}RN\texttt{)}$ will have always the form $\texttt{(}\$1 \texttt{=>} N\texttt{)}$. When we want to distinguish the value N, we will write $B \xrightarrow{G\texttt{(}\$1\texttt{=>}N\texttt{)}} B'$.
- $B \xrightarrow{X\texttt{(}RN\texttt{)}} B'$: B can raise the exception X together with the value $\texttt{(}RN\texttt{)}$, which is a record with one field called \$1, and becomes B' in doing so.
- $B \xrightarrow{\mathbf{i}} B'$: behaviour B can perform an internal action **i**, becoming B'.
- $B \xrightarrow{\delta\texttt{(}RN\texttt{)}} B'$: behaviour B can finish producing bindings RN between variables in patterns of B and values. These bindings are not saved in a memory or state, but they are transferred to the following behaviours, that is, behaviours composed in sequence with B, by means of the substitution operator (see below). δ is a special gate used by the dynamic semantics to propagate the bindings produced by a behaviour. For example, the abstract syntax behaviour $\texttt{?}x\ \texttt{:=}\ \textbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{)}$ binds variable x with the value 3, and this binding is propagated by means of gate δ,

$$\texttt{?}x\ \texttt{:=}\ \textbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{)} \xrightarrow{\delta\texttt{(}x\texttt{=>}3\texttt{)}} \textbf{block}$$

  as we will see in Example 8.8. Behaviour B' is never used in the inference rules, in the sense that when it appears in a premise of an inference rule, it is never used in the conclusion. We write it in order to write always transitions ending in a behaviour, and it will be always the behaviour **block**.
- $B \xrightarrow{\epsilon\langle d\rangle} B'$: behaviour B can idle d units of time, becoming B'.

Sometimes we will use these abbreviations:

- $B \xrightarrow{a\texttt{(}RN\texttt{)}} B'$, meaning $B \xrightarrow{G\texttt{(}RN\texttt{)}} B'$, $B \xrightarrow{X\texttt{(}RN\texttt{)}} B'$, or $B \xrightarrow{\mathbf{i}} B'$; and
- $B \xrightarrow{\mu\texttt{(}RN\texttt{)}} B'$, meaning $B \xrightarrow{a\texttt{(}RN\texttt{)}} B'$ or $B \xrightarrow{\delta\texttt{(}RN\texttt{)}} B'$.

We will also write $B \xrightarrow{\mu(RN)@d} B'$ when either:

- $B \xrightarrow{\mu(RN)} B'$ and $d = 0$, or
- $B \xrightarrow{\epsilon\langle d\rangle} B''$ and $B'' \xrightarrow{\mu(RN)} B'$.

Contexts

The main judgements of the dynamic semantics are

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

$$\mathcal{E} \vdash B \xrightarrow{\epsilon\langle d\rangle} B'$$

which mean that in context $\mathcal{E}$ the given transitions are possible, where context $\mathcal{E}$ gives the needed information related to identifiers in B. These contexts are built with the following grammar:

$$\begin{array}{rcllr}
\mathcal{E} & ::= & K \Rightarrow T & \textit{predefined constant} & (\mathcal{E}1) \\
& \mid & S \equiv T & \textit{type equivalence} & (\mathcal{E}2) \\
& \mid & T \sqsubseteq T' & \textit{subtype} & (\mathcal{E}3) \\
& \mid & C \Rightarrow (RT) \rightarrow S & \textit{constructor} & (\mathcal{E}4) \\
& \mid & \Pi \Rightarrow \lambda\,[\,G_1 : T_1, \ldots, G_m : T_m\,] & \textit{process identifier} & (\mathcal{E}5) \\
& & \quad (V_1 : T'_1, \ldots, V_p : T'_p) & & \\
& & \quad [\,X_1 : T''_1, \ldots, X_n : T''_n\,] \rightarrow B & & \\
& \mid & & \textit{trivial} & (\mathcal{E}6) \\
& \mid & \mathcal{E}, \mathcal{E} & \textit{disjoint union} & (\mathcal{E}7)
\end{array}$$

The information saved in the contexts of the dynamic semantics consists of

- a subset of the information saved in the static semantics contexts that needed to be able to infer the type of a value N (and which has the same meaning as in the static semantics); and
- the information related to a process declaration needed to execute an instantiation, that is, the name and type of the formal gates, parameters, and exceptions, and the body of the process.

Judgements

As in the static semantics, there are more judgements not dealing with transitions. There are judgements to know a value's type, as follows

$$\mathcal{E} \vdash N \;\;\rightarrow^{\mathrm{T}}\;\; T$$

meaning that in context $\mathcal{E}$ value N has type T. Similarly for records of values,

$$\mathcal{E} \vdash RN \;\;\rightarrow^{\mathrm{T}}\;\; RT.$$

There are also judgements dealing with subtyping,

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

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

and inference rules stating that the subtyping relation is a preorder. We omit them here because they are as in the static semantics.

We have also judgements relative to pattern matching between patterns and values:

$$\mathcal{E} \vdash (P \ \rightarrow^{\text{PM}}\ N) \ \mapsto\ \langle\!|\, RN \,|\!\rangle$$

$$\mathcal{E} \vdash (RP \ \rightarrow^{\text{PM}}\ RN) \ \mapsto\ \langle\!|\, RN' \,|\!\rangle$$

and a new kind of judgements

$$\mathcal{E} \vdash (P \ \rightarrow^{\text{PM}}\ N) \ \mapsto\ \mathit{fail}$$

$$\mathcal{E} \vdash (RP \ \rightarrow^{\text{PM}}\ RN) \ \mapsto\ \mathit{fail}$$

meaning that it is not possible to match the pattern P (respectively RP) against value N (respectively RN). This information is needed when the different branches of a **case** statement are checked (see Section 8.24).

Substitution operator

The substitution operator, $B\llbracket RN \rrbracket$ is used to propagate bindings RN produced by a behaviour to the following one B. For example, when the behaviour $B_1\texttt{;}B_2$ is executed and B_1 finishes producing bindings RN_1, then B_2 is executed after substituting variables bound in RN_1 (see inference rule 8.11 below). For example, if $\texttt{?}x\texttt{:=}1\texttt{;}G\texttt{(!}x\texttt{)}$ is executed, $\texttt{?}x\texttt{:=}1$ finishes producing binding $x \texttt{=>} 1$, and $G\texttt{(!}1\texttt{)}$ is then executed. But if $\texttt{?}x\texttt{:=}1\texttt{;?}x\texttt{:=}x\texttt{+}1\texttt{;}G\texttt{(!}x\texttt{)}$ is executed and the binding $x \texttt{=>} 1$ is produced, it can only affect to the expression in the second assignment but not to the action on gate G, because the second assignment binds variable x again. Therefore, the substitution operator is defined as syntactic substitution of free variables for values with the following exceptions:

- **Actions**

  $$\begin{array}{ll} & (G\ \texttt{(}\$1 \texttt{=>} P_1\texttt{)}\ \texttt{@}P_2\ E\ \textbf{start}\langle d\rangle)\llbracket RN \rrbracket \\ \overset{\text{def}}{=} & G\ \texttt{(}\$1 \texttt{=>} P_1\llbracket RN \rrbracket\texttt{)}\ \texttt{@}P_2\llbracket RN \rrbracket\ E\llbracket RN' \rrbracket\ \textbf{start}\langle d\rangle \end{array}$$

  where $RN' = RN - (\mathit{vars}(P_1) \cup \mathit{vars}(P_2))$, and $\mathit{vars}(P)$ is the set of variables bound by pattern P (and it is easy to define).

- **Sequential composition**

  $$(B_1\texttt{;}B_2)\llbracket RN \rrbracket \ \overset{\text{def}}{=}\ B_1\llbracket RN \rrbracket\texttt{;}B_2\llbracket RN - \mathit{vars}(B_1) \rrbracket$$

  where $\mathit{vars}(B)$ represents the set of variables bound by behaviour B, and it is easy and cumbersome to define[1].

- **Selection operator** In the branching operators (those composed by several subbehaviours where not all of them have to be executed) not all the branches have to assign the same variables. If the behaviour $\texttt{?}x\texttt{:=}1\texttt{;(?}x\texttt{:=}x\texttt{+}1\texttt{;}G_1\ \texttt{[]}\ G_2\texttt{);?}y\texttt{:=}x$ begins to be executed and binding $x\texttt{=>}1$ is produced, it should not affect the last assignment, since if the first branch of the selection is chosen, the value of variable x will change. Therefore, branching operators stops the substitution of variables bound in any branch and remembers the bindings introducing them in the behaviours:

  $$(B_1\ \texttt{[]}\ B_2)\llbracket RN \rrbracket \ \overset{\text{def}}{=}\ \mathit{beh}(RN)\texttt{;}B_1\ \texttt{[]}\ \mathit{beh}(RN)\texttt{;}B_2$$

  where $\mathit{beh}(V_1 \texttt{=>} N_1, \ldots, V_n \texttt{=>} N_n) \ \overset{\text{def}}{=}\ \texttt{?}V_1\texttt{:=}\textbf{exit}\texttt{(}\$1 \texttt{=>} N_1\texttt{)}\texttt{;}\ldots\texttt{;?}V_n\texttt{:=}\textbf{exit}\texttt{(}\$1 \texttt{=>} N_n\texttt{)}$.

[1] In fact, if $\mathcal{C} \vdash B \Longrightarrow \mathit{exit}\,\langle\!|\, V_1 \texttt{=>} T_1, \ldots, V_n \texttt{=>} T_n \,|\!\rangle$ then $\mathit{vars}(B) = \{V_1, \ldots, V_n\}$. $\mathcal{C}$ is the context of the static semantics.

- **Parallel over values operator**

$$(\textbf{par } P \textbf{ in } N \texttt{ ||| } B)\llbracket RN\rrbracket \quad \overset{\text{def}}{=} \quad \textbf{par } P \textbf{ in } N \texttt{ ||| } B\llbracket RN - \mathit{vars}(P)\rrbracket$$

- **Disabling operator** This is also a branching operator.

$$(B_1 \texttt{ [> } B_2)\llbracket RN\rrbracket \quad \overset{\text{def}}{=} \quad \mathit{beh}(RN)\texttt{;}B_1 \texttt{ [> } \mathit{beh}(RN)\texttt{;}B_2$$

- **Suspend/Resume operator** This is a branching operator.

$$(B_1 \texttt{ [}X\texttt{> } B_2)\llbracket RN\rrbracket \quad \overset{\text{def}}{=} \quad \mathit{beh}(RN)\texttt{;}B_1 \texttt{ [}X\texttt{> } \mathit{beh}(RN)\texttt{;}B_2$$

- **Exception handling** This is a branching operator.

$$\begin{array}{lcl}
(\textbf{trap} & \overset{\text{def}}{=} & \textbf{trap} \\
\quad \textbf{exception } X_1\,\texttt{(\$1 => }P_1\texttt{)}\texttt{:}T_1 \textbf{ is } B_1 & & \quad \textbf{exception } X_1\,\texttt{(\$1 => }P_1\texttt{)}\texttt{:}T_1 \textbf{ is } B_1\llbracket RN - \mathit{vars}(P_1)\rrbracket \\
\qquad \ldots & & \qquad \ldots \\
\quad \textbf{exception } X_n\,\texttt{(\$1 => }P_n\texttt{)}\texttt{:}T_n \textbf{ is } B_n & & \quad \textbf{exception } X_n\,\texttt{(\$1 => }P_n\texttt{)}\texttt{:}T_n \textbf{ is } B_n\llbracket RN - \mathit{vars}(P_n)\rrbracket \\
\quad \textbf{exit } P_{n+1} \textbf{ is } B_{n+1} & & \quad \textbf{exit } P_{n+1} \textbf{ is } B_{n+1}\llbracket RN - \mathit{vars}(P_{n+1})\rrbracket \\
\textbf{in } B\ )\llbracket RN\rrbracket & & \textbf{in } B\llbracket RN\rrbracket
\end{array}$$

- **Renaming operator**

$$\begin{array}{lcl}
(\textbf{rename} & \overset{\text{def}}{=} & \textbf{rename} \\
\quad \textbf{gate } G_1\texttt{(\$1 => }P_1\texttt{)}\texttt{:}T_1 \textbf{ is } G_1'\texttt{(\$1 => }P_1'\texttt{)} & & \quad \textbf{gate } G_1\texttt{(\$1 => }P_1\texttt{)}\texttt{:}T_1 \textbf{ is } G_1'\texttt{(\$1 => }P_1'\llbracket RN - \mathit{vars}(P_1)\rrbracket\texttt{)} \\
\qquad \ldots & & \qquad \ldots \\
\quad \textbf{gate } G_m\texttt{(\$1 => }P_m\texttt{)}\texttt{:}T_m \textbf{ is } G_m'\texttt{(\$1 => }P_m'\texttt{)} & & \quad \textbf{gate } G_m\texttt{(\$1 => }P_m\texttt{)}\texttt{:}T_m \textbf{ is } G_m'\texttt{(\$1 => }P_m'\llbracket RN - \mathit{vars}(P_m)\rrbracket\texttt{)} \\
\quad \textbf{signal } X_1\texttt{(\$1 => }P_1''\texttt{)}\texttt{:}T_1' \textbf{ is } X_1'\texttt{(}E_1\texttt{)} & & \quad \textbf{signal } X_1\texttt{(\$1 => }P_1''\texttt{)}\texttt{:}T_1' \textbf{ is } X_1'\texttt{(}E_1\llbracket RN - \mathit{vars}(P_1'')\rrbracket\texttt{)} \\
\qquad \ldots & & \qquad \ldots \\
\quad \textbf{signal } X_m\texttt{(\$1 => }P_n''\texttt{)}\texttt{:}T_n' \textbf{ is } X_n'\texttt{(}E_n\texttt{)} & & \quad \textbf{signal } X_m\texttt{(\$1 => }P_n''\texttt{)}\texttt{:}T_n' \textbf{ is } X_n'\texttt{(}E_n\llbracket RN - \mathit{vars}(P_n'')\rrbracket\texttt{)} \\
\textbf{in } B\ )\llbracket RN\rrbracket & & \textbf{in } B\llbracket RN\rrbracket
\end{array}$$

- **Variable declaration**

$$\begin{array}{ll}
& (\textbf{var } V_1\texttt{:}T_1[\texttt{:=}E_1],\ldots,V_n\texttt{:}T_n[\texttt{:=}E_n] \textbf{ in } B)\llbracket RN\rrbracket \\
\overset{\text{def}}{=} & \textbf{var } V_1\texttt{:}T_1[\texttt{:=}E_1\llbracket RN\rrbracket],\ldots,V_n\texttt{:}T_n[\texttt{:=}E_n\llbracket RN\rrbracket] \textbf{ in } B\llbracket RN\rrbracket
\end{array}$$

- **Case operator** This is also a branching operator.

$$\begin{array}{ll}
& (\textbf{case } E\texttt{:}T \textbf{ is } P_1\ \texttt{[}E_1\texttt{]} \texttt{ -> } B_1 \texttt{ | } \ldots \texttt{ | } P_n\ \texttt{[}E_n\texttt{]} \texttt{ -> } B_n)\llbracket RN\rrbracket \\
\overset{\text{def}}{=} & \textbf{case } E\texttt{:}T \textbf{ is } P_1\ \texttt{[}E_1\llbracket RN - \mathit{vars}(P_1)\rrbracket\texttt{]} \texttt{ -> } B_1\llbracket RN - \mathit{vars}(P_1)\rrbracket \\
& \qquad \texttt{| } \ldots \texttt{ | } P_n\ \texttt{[}E_n\llbracket RN - \mathit{vars}(P_n)\rrbracket\texttt{]} \texttt{ -> } B_n\llbracket RN - \mathit{vars}(P_n)\rrbracket
\end{array}$$

- **Iteration loop**

$$(\textbf{loop } B)\llbracket RN\rrbracket \quad \overset{\text{def}}{=} \quad B\llbracket RN\rrbracket\texttt{;}\textbf{loop } B\llbracket RN - \mathit{vars}(B)\rrbracket$$

8.2 Actions

The action $G\ \texttt{(}\$1\texttt{=>}P_1\texttt{)}\ \texttt{@}P_2\ E\ \mathbf{start}\langle d\rangle$ has the transition labelled $G\texttt{(}\$1 \texttt{=>} N\texttt{)}$, in the context $\mathcal{E}$, provided that the following conditions are fulfilled:

- The pattern $\texttt{(}\$1 \texttt{=>} P_1\texttt{)}$ can be matched (in the context $\mathcal{E}$) against the value $\texttt{(}\$1 \texttt{=>} N\texttt{)}$, producing bindings RN_1,

$$\mathcal{E} \vdash (\texttt{(}\$1 \texttt{=>} P_1\texttt{)} \ \rightharpoonup^{\mathbf{PM}}\ \texttt{(}\$1 \texttt{=>} N\texttt{)}) \ \mapsto\ \langle\!|\, RN_1 \,|\!\rangle$$

- The pattern P_2 can be matched against the time value d, producing bindings RN_2,

$$\mathcal{E} \vdash (P_2 \ \rightharpoonup^{\mathbf{PM}}\ d) \ \mapsto\ \langle\!|\, RN_2 \,|\!\rangle$$

- The expression E after substituting the bindings RN_1, RN_2 returns *true*,

$$\mathcal{E} \vdash E\,[\![\, RN_1, RN_2 \,]\!] \xrightarrow{\delta\texttt{(}\$1\texttt{=>}\mathit{true}\texttt{)}} E'$$

The following inference rule summarizes these conditions:

$$\frac{\begin{array}{l}\mathcal{E} \vdash (\texttt{(}\$1 \texttt{=>} P_1\texttt{)} \ \rightharpoonup^{\mathbf{PM}}\ \texttt{(}\$1 \texttt{=>} N\texttt{)}) \ \mapsto\ \langle\!|\, RN_1 \,|\!\rangle \\ \mathcal{E} \vdash (P_2 \ \rightharpoonup^{\mathbf{PM}}\ d) \ \mapsto\ \langle\!|\, RN_2 \,|\!\rangle \\ \mathcal{E} \vdash E\,[\![\, RN_1, RN_2 \,]\!] \xrightarrow{\delta\texttt{(}\$1\texttt{=>}\mathit{true}\texttt{)}} E'\end{array}}{\mathcal{E} \vdash G\ \texttt{(}\$1 \texttt{=>} P_1\texttt{)}\ \texttt{@}P_2\ E\ \mathbf{start}\langle d\rangle \xrightarrow{G\texttt{(}\$1\texttt{=>}N\texttt{)}} \mathbf{exit}\texttt{(}RN_1\texttt{,}RN_2\texttt{)}} \tag{8.8}$$

The behaviour $\mathbf{exit}\texttt{(}RN_1\texttt{,}RN_2\texttt{)}$ is used in order to

- indicate that the action has finished and if there is a behaviour B composed in sequence with the action, B can start to be executed; and to
- *propagate* the bindings produced by the action to any behaviour which is composed in sequence with this action (see Section 8.3 and Example 8.7).

When we were studying the inference rules in [Que98] we though that rule 8.8 should check that N is a correct value for gate G, that is, if gate G can communicate values of type T, N must have type T. These type checking is not needed because when this rule is used, that is, when we want to know if an action on gate G has the transition $\xrightarrow{G\texttt{(}\$1\texttt{=>}N\texttt{)}}$, N has the type of values which G can communicate. This rule is used in the inference rules of the **hide** (Section 8.15) and **rename** (Section 8.19) operators and, there, the value N is *invented* (we say that a value is invented in an inference rule if it appears in a premise of the rule but not in the conclusion) of the corresponding type.

⏱ Regarding time, an action can idle any period of time. Hence, we have the following transition

$$\frac{}{\mathcal{E} \vdash G\ \texttt{(}\$1 \texttt{=>} P_1\texttt{)}\ \texttt{@}P_2\ E\ \mathbf{start}\langle d\rangle \xrightarrow{\epsilon\langle d'\rangle} G\ \texttt{(}\$1 \texttt{=>} P_1\texttt{)}\ \texttt{@}P_2\ E\ \mathbf{start}\langle d + d'\rangle}\ [d' > 0] \tag{8.9}$$

for any time delay $d' > 0$. Although pattern P_2 is used to restrict *when* the action can be performed, that is not taken into account in this rule, but in the previous one. Thus, an action can idle any time, even if it cannot be performed then due to the restrictions in P_2.

Example 8.1 Let us see how the behaviour $\mathit{outP}\texttt{(!}3\texttt{)}$ can evolve. It is translated to abstract syntax as

$$\mathit{outP}\texttt{(}\$1 \texttt{=> !}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{))}\ \texttt{@}\mathbf{any} : \mathsf{time}\ \mathbf{exit}\texttt{(}\$1 \texttt{=>} \mathit{true}\texttt{)}\ \mathbf{start}\langle 0\rangle.$$

By using rule 8.8 we can deduce

$$\frac{\begin{array}{l} \mathcal{E}\vdash ((\$1 \texttt{=>} \texttt{!}\mathbf{exit}(\$1 \texttt{=>} 3)) \;\;\rightarrow^{\mathrm{PM}}\;\; (\$1 \texttt{=>} 3)) \;\mapsto\; \langle\!|\;|\!\rangle \\ \mathcal{E}\vdash (\mathbf{any} : \mathsf{time} \;\;\rightarrow^{\mathrm{PM}}\;\; 0) \;\mapsto\; \langle\!|\;|\!\rangle \\ \mathcal{E}\vdash \mathbf{exit}(\$1 \texttt{=>} \mathit{true}) \xrightarrow{\delta(\$1\texttt{=>}\mathit{true})} \mathbf{block} \end{array}}{\mathcal{E}\vdash \mathit{outP}(\$1 \texttt{=>} \texttt{!}\mathbf{exit}(\$1 \texttt{=>} 3))\; \texttt{@}\mathbf{any} : \mathsf{time}\; \mathbf{exit}(\$1 \texttt{=>} \mathit{true})\; \mathbf{start}\langle 0\rangle \xrightarrow{\mathit{outP}(\$1\texttt{=>}3)} \mathbf{exit}()}\;(8.8)$$

The first premise is proved as follows:

$$\dfrac{\dfrac{\dfrac{\mathcal{E}\vdash \mathbf{exit}(\$1 \texttt{=>} 3) \xrightarrow{\delta(\$1\texttt{=>}3)} \mathbf{block}}{\mathcal{E}\vdash (\texttt{!}\mathbf{exit}(\$1 \texttt{=>} 3) \;\;\rightarrow^{\mathrm{PM}}\;\; 3) \;\mapsto\; \langle\!|\;|\!\rangle}\;(8.150)}{\mathcal{E}\vdash (\$1 \texttt{=>} \texttt{!}\mathbf{exit}(\$1 \texttt{=>} 3) \;\;\rightarrow^{\mathrm{PM}}\;\; \$1 \texttt{=>} 3) \;\mapsto\; \langle\!|\;|\!\rangle}\;(8.140)}{\mathcal{E}\vdash ((\$1 \texttt{=>} \texttt{!}\mathbf{exit}(\$1 \texttt{=>} 3)) \;\;\rightarrow^{\mathrm{PM}}\;\; (\$1 \texttt{=>} 3)) \;\mapsto\; \langle\!|\;|\!\rangle}\;(8.134)$$

The second and third premises have the following proofs

$$\frac{\mathcal{E}\vdash 0 \;\;\rightarrow^{\mathrm{T}}\;\; \mathsf{time}}{\mathcal{E}\vdash (\mathbf{any} : \mathsf{time} \;\;\rightarrow^{\mathrm{PM}}\;\; 0) \;\mapsto\; \langle\!|\;|\!\rangle}\;(8.137) \qquad\qquad \frac{}{\mathcal{E}\vdash \mathbf{exit}(\$1 \texttt{=>} \mathit{true}) \xrightarrow{\delta(\$1\texttt{=>}\mathit{true})} \mathbf{block}}\;(8.23)$$

We can also use rule 8.9 to show that the given action can idle, for example, two units of time:

$$\frac{}{\mathcal{E}\vdash \mathit{outP}(\$1 \texttt{=>} \texttt{!}\mathbf{exit}(\$1 \texttt{=>} 3))\; \ldots\; \mathbf{start}\langle 0\rangle \xrightarrow{\epsilon\langle 2\rangle} \mathit{outP}(\$1 \texttt{=>} \texttt{!}\mathbf{exit}(\$1 \texttt{=>} 3))\; \ldots\; \mathbf{start}\langle 2\rangle}\;(8.9)$$

where … stands for $\texttt{@}\mathbf{any} : \mathsf{time}\; \mathbf{exit}(\$1 \texttt{=>} \mathit{true})$.

And, by combining these results, we can show that the evolution of the action $\mathit{outP}(\texttt{!3})$ might be:

$$\begin{array}{ll} \mathit{outP}(\$1 \texttt{=>} \texttt{!}\mathbf{exit}(\$1 \texttt{=>} 3))\; \texttt{@}\mathbf{any} : \mathsf{time}\; \mathbf{exit}(\$1 \texttt{=>} \mathit{true})\; \mathbf{start}\langle 0\rangle & \xrightarrow{\epsilon\langle 2\rangle} \\ \mathit{outP}(\$1 \texttt{=>} \texttt{!}\mathbf{exit}(\$1 \texttt{=>} 3))\; \texttt{@}\mathbf{any} : \mathsf{time}\; \mathbf{exit}(\$1 \texttt{=>} \mathit{true})\; \mathbf{start}\langle 2\rangle & \xrightarrow{\mathit{outP}(\$1\texttt{=>}3)} \\ \mathbf{exit}() & \end{array}$$

where, in order to prove the second step, we have to change 0 to 2 in the first proof.

Therefore, the action $\mathit{outP}(\texttt{!3})$ may perform a communication on gate *outP* *immediately*, or it can delay any time before doing this communication. The next example shows how we can control the time instant when the communication occurs.

Example 8.2 Let us suppose that the value 3 has to be communicated through the gate *outP* exactly after one unit of time has passed since the action was enabled. That is, the action is specified as $\mathit{outP}(\texttt{!3})\;\texttt{@!1}$, translated to abstract syntax as

$$\mathit{outP}(\$1 \texttt{=>} \texttt{!}\mathbf{exit}(\$1 \texttt{=>} 3))\; \texttt{@!}\mathbf{exit}(\$1 \texttt{=>} 1)\; \mathbf{exit}(\$1 \texttt{=>} \mathit{true})\; \mathbf{start}\langle 0\rangle.$$

If we try to use rule 8.8 directly, we cannot prove anything, because the pattern `!1` cannot be matched against the value 0. So, we can only use the rule 8.9, and prove the following[2]

$$\frac{}{\begin{array}{l} \mathcal{E}\vdash \mathit{outP}(\$1 \texttt{=>} \texttt{!}\mathbf{exit}(\$1 \texttt{=>} 3))\; \texttt{@!}\mathbf{exit}(\$1 \texttt{=>} 1)\; \mathbf{exit}(\$1 \texttt{=>} \mathit{true})\; \mathbf{start}\langle 0\rangle \\ \quad\xrightarrow{\epsilon\langle 1\rangle} \mathit{outP}(\$1 \texttt{=>} \texttt{!}\mathbf{exit}(\$1 \texttt{=>} 3))\; \texttt{@!}\mathbf{exit}(\$1 \texttt{=>} 1)\; \mathbf{exit}(\$1 \texttt{=>} \mathit{true})\; \mathbf{start}\langle 1\rangle \end{array}}\;(8.9)$$

[2] In fact, we can use this rule to prove that the action $\mathit{outP}(\texttt{!3})\;\texttt{@!1}$ can idle any time, but only when exactly one unit of time is delayed, the resulting behaviour can perform the communication.

Now, we can use rule 8.8 to show the following:

$$\frac{\begin{array}{l}\mathcal{E} \vdash ((\$1 \texttt{=>}\ \texttt{!}\mathbf{exit}(\$1 \texttt{=>} 3)) \ \rightharpoonup^{\mathbf{PM}}\ (\$1 \texttt{=>} 3)) \ \mapsto\ \llparenthesis\ \rrparenthesis \\ \mathcal{E} \vdash (\texttt{!}\mathbf{exit}(\$1 \texttt{=>} 1) \ \rightharpoonup^{\mathbf{PM}}\ 1) \ \mapsto\ \llparenthesis\ \rrparenthesis \\ \mathcal{E} \vdash \mathbf{exit}(\$1 \texttt{=>}\ \mathit{true}) \xrightarrow{\delta(\$1\texttt{=>}\mathit{true})} \mathbf{block}\end{array}}{\mathcal{E} \vdash \mathit{outP}(\$1 \texttt{=>}\ \texttt{!}\mathbf{exit}(\$1 \texttt{=>} 3))\ \texttt{@!}\mathbf{exit}(\$1 \texttt{=>} 1)\ \mathbf{exit}(\$1 \texttt{=>}\ \mathit{true})\ \mathbf{start}\langle 1\rangle \xrightarrow{\mathit{outP}(\$1\texttt{=>}3)} \mathbf{exit}()} \tag{8.8}$$

where the premises are easily proved.

Example 8.3 Let us see now how the behaviour $\mathit{inP}(?x : \mathsf{int})$, which is translated to

$$\mathit{inP}(\$1 \texttt{=>}\ ?x : \mathsf{int})\ \texttt{@}\mathbf{any} : \mathsf{time}\ \mathbf{exit}(\$1 \texttt{=>}\ \mathit{true})\ \mathbf{start}\langle 0\rangle,$$

can evolve.

By applying rule 8.8, we can deduce

$$\frac{\begin{array}{l}\mathcal{E} \vdash ((\$1 \texttt{=>}\ ?x : \mathsf{int}) \ \rightharpoonup^{\mathbf{PM}}\ (\$1 \texttt{=>}\ n)) \ \mapsto\ \llparenthesis\, x \texttt{=>}\ n\, \rrparenthesis \\ \mathcal{E} \vdash (\mathbf{any} : \mathsf{time} \ \rightharpoonup^{\mathbf{PM}}\ 0) \ \mapsto\ \llparenthesis\ \rrparenthesis \\ \mathcal{E} \vdash \mathbf{exit}(\$1 \texttt{=>}\ \mathit{true}) \xrightarrow{\delta(\$1\texttt{=>}\mathit{true})} \mathbf{block}\end{array}}{\mathcal{E} \vdash \mathit{inP}(\$1 \texttt{=>}\ ?x : \mathsf{int})\ \texttt{@}\mathbf{any} : \mathsf{time}\ \mathbf{exit}(\$1 \texttt{=>}\ \mathit{true})\ \mathbf{start}\langle 0\rangle \xrightarrow{\mathit{inP}(\$1\texttt{=>}n)} \mathbf{exit}(x \texttt{=>}\ n)} \tag{8.8}$$

where n may be any constant of type int. That is, this action can offer a communication of *any* integer through the gate *inP*, binding the variable x with this value.

The first premise is proved in Example 8.16 in Section 8.31.3 with $n = 8$.

8.3 Sequential composition

We are going to see now how a sequential composition $B_1\,;B_2$ can evolve. Its evolution depends, obviously, on the evolutions of both B_1 and B_2. If B_1 can offer a communication, perform an internal action, or raise an exception (all of them represented by a), then, the sequential composition can do the same.

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{a(RN)} B_1'}{\mathcal{E} \vdash B_1\,;B_2 \xrightarrow{a(RN)} B_1'\,;B_2} \tag{8.10}$$

If the first behaviour finishes producing bindings RN_1, and the second behaviour modified by these bindings can perform an a, then, we have

$$\frac{\begin{array}{l}\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN_1)} B_1' \\ \mathcal{E} \vdash B_2\,[\![\, RN_1 \,]\!] \xrightarrow{a(RN_2)} B_2'\end{array}}{\mathcal{E} \vdash B_1\,;B_2 \xrightarrow{a(RN_2)} \mathbf{exit}(RN_1)\,;B_2'} \tag{8.11}$$

Note how the bindings produced by B_1 are transfered to B_2 via the substitution operator.

We use the behaviour $\mathbf{exit}(RN_1)$ to annotate the bindings produced by B_1. These bindings will be used later, when B_2 also finishes. The following rule shows how when B_1 can finish, and B_2 modified by the bindings produced by B_1 can also finish, the whole sequential composition finishes and it produces the bindings of B_1 overridden by the bindings of B_2:

$$\frac{\begin{array}{c}\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN_1)} B_1' \\ \mathcal{E} \vdash B_2 [\![RN_1]\!] \xrightarrow{\delta(RN_2)} B_2'\end{array}}{\mathcal{E} \vdash B_1 ; B_2 \xrightarrow{\delta(RN_1;RN_2)} \mathbf{block}} \tag{8.12}$$

The sequential composition $B_1 ; B_2$ can idle a period of time d if B_1 can do so:

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\epsilon\langle d\rangle} B_1'}{\mathcal{E} \vdash B_1 ; B_2 \xrightarrow{\epsilon\langle d\rangle} B_1' ; B_2} \tag{8.13}$$

And, if B_1 can finish after d_1 units of time, and B_2 (modified by the bindings produced by B_1) can idle a period of time d_2, then the sequential composition can delay $d_1 + d_2$:

$$\frac{\begin{array}{c}\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN_1)@d_1} B_1' \\ \mathcal{E} \vdash B_2 [\![RN_1]\!] \xrightarrow{\epsilon\langle d_2\rangle} B_2'\end{array}}{\mathcal{E} \vdash B_1 ; B_2 \xrightarrow{\epsilon\langle d_1+d_2\rangle} \mathbf{exit}(RN_1) ; B_2'} \tag{8.14}$$

Example 8.4 Let us see the (possible) evolution of the behaviour $\mathit{inP}(?x : \mathsf{int});\ \mathit{outP}(!x)$ which is translated to

$$\begin{array}{l}\mathit{inP}(\$1 \Rightarrow ?x : \mathsf{int})\ @\mathbf{any} : \mathsf{time}\ \mathbf{exit}(\$1 \Rightarrow \mathit{true})\ \mathbf{start}\langle 0\rangle\, ; \\ \mathit{outP}(\$1 \Rightarrow !\mathbf{exit}(\$1 \Rightarrow x))\ @\mathbf{any} : \mathsf{time}\ \mathbf{exit}(\$1 \Rightarrow \mathit{true})\ \mathbf{start}\langle 0\rangle\end{array} .$$

First, we can use rule 8.10 to prove that

$$\frac{\mathcal{E} \vdash \mathit{inP}(\$1 \Rightarrow ?x : \mathsf{int})\ \ldots \xrightarrow{\mathit{inP}(\$1\Rightarrow n)} \mathbf{exit}(x \Rightarrow n)}{\mathcal{E} \vdash \mathit{inP}(\$1 \Rightarrow ?x : \mathsf{int})\ \ldots ;\ \mathit{outP}(\$1 \Rightarrow !\mathbf{exit}(\$1 \Rightarrow x))\ \ldots \xrightarrow{\mathit{inP}(\$1\Rightarrow n)} \mathbf{exit}(x \Rightarrow n) ;\ \mathit{outP}(!\mathbf{exit}(\$1 \Rightarrow x))\ \ldots} \tag{8.10}$$

where n stands for any integer constant. Then, we can use rule 8.11 to prove that

$$\frac{\begin{array}{c}\mathcal{E} \vdash \mathbf{exit}(x \Rightarrow n) \xrightarrow{\delta(x\Rightarrow n)} \mathbf{block} \\ \mathcal{E} \vdash \mathit{outP}(\$1 \Rightarrow !\mathbf{exit}(\$1 \Rightarrow n))\ \ldots \xrightarrow{\mathit{outP}(\$1\Rightarrow n)} \mathbf{exit}()\end{array}}{\mathcal{E} \vdash \mathbf{exit}(x \Rightarrow n) ;\ \mathit{outP}(\$1 \Rightarrow !\mathbf{exit}(\$1 \Rightarrow x))\ \ldots \xrightarrow{\mathit{outP}(\$1\Rightarrow n)} \mathbf{exit}(x \Rightarrow n) ;\ \mathbf{exit}()} \tag{8.11}$$

Note how value n has been substituted for variable x in the second premise.

Finally, we use rule 8.12 to conclude that

$$\frac{\begin{array}{c}\mathcal{E} \vdash \mathbf{exit}(x \Rightarrow n) \xrightarrow{\delta(x\Rightarrow n)} \mathbf{block} \\ \mathcal{E} \vdash \mathbf{exit}() \xrightarrow{\delta()} \mathbf{block}\end{array}}{\mathcal{E} \vdash \mathbf{exit}(x \Rightarrow n) ;\ \mathbf{exit}() \xrightarrow{\delta(x\Rightarrow n)} \mathbf{block}} \tag{8.12}$$

8.4 Selection operator

A selection B_1 `[]` B_2 can offer a communication on a gate if B_1 or B_2 does, and this offer constitutes a decision, in the sense that only one branch will continue.

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{G(RN)} B_1'}{\mathcal{E} \vdash B_1 \texttt{ [] } B_2 \xrightarrow{G(RN)} B_1'} \tag{8.15}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{G(RN)} B_2'}{\mathcal{E} \vdash B_1 \texttt{ [] } B_2 \xrightarrow{G(RN)} B_2'} \tag{8.16}$$

The same occurs if one of the behaviours performs an internal action:

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\mathbf{i}} B_1'}{\mathcal{E} \vdash B_1 \texttt{ [] } B_2 \xrightarrow{\mathbf{i}} B_1'} \tag{8.17}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{\mathbf{i}} B_2'}{\mathcal{E} \vdash B_1 \texttt{ [] } B_2 \xrightarrow{\mathbf{i}} B_2'} \tag{8.18}$$

However, things are not the same when B_1 or B_2 raises an exception. If we said that in this case the whole behaviour raises the same exception, we would not preserve the urgency of internal actions. For example, the behaviour

```
trap
  exception X is wait(1) endexn
in ( i [] signal X )
endtrap
```

could perform an internal action (the selection can do an internal action, and then the **trap** behaviour can too), or it could wait one unit of time (the selection would raise exception X and the exception would be trapped and handled with the behaviour **wait**(1)), and this is what we do not want. In order to avoid this problem, the dynamic semantics rules of the selection operator do not allow a selection behaviour to raise an exception. If B_1 (or B_2) raises an exception becoming B_1' (respectively B_2'), B_1 [] B_2 performs an internal action (that breaks the selection) becoming a behaviour that raise the exception raised by B_1 and follows as B_1' (respectively B_2'). Example 8.12 in Section 8.16 shows the possible transitions of the above behaviour.

The rules are as follows:

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{X\texttt{(\$1=>}N\texttt{)}} B_1'}{\mathcal{E} \vdash B_1 \texttt{ [] } B_2 \xrightarrow{\mathbf{i}} \textbf{signal } X\texttt{(}\textbf{exit}\texttt{(\$1 => }N\texttt{))};B_1'} \tag{8.19}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{X\texttt{(\$1=>}N\texttt{)}} B_2'}{\mathcal{E} \vdash B_1 \texttt{ [] } B_2 \xrightarrow{\mathbf{i}} \textbf{signal } X\texttt{(}\textbf{exit}\texttt{(\$1 => }N\texttt{))};B_2'} \tag{8.20}$$

where the exception raised is annotated in order to be raised again.

We have to note that there are no rules dealing with the case where B_1 or B_2 finishes (by offering a communication on gate δ). It is because the behaviour B_1 [] B_2 has passed the static semantics, and, therefore, both behaviours are guarded, that is, they cannot finish without performing an action or raising an exception.

The following rule establishes that the behaviour B_1 [] B_2 can idle a period of time d, if *both* B_1 and B_2 can idle the same period d.

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\epsilon\langle d\rangle} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\epsilon\langle d\rangle} B_2'}{\mathcal{E} \vdash B_1 \texttt{ [] } B_2 \xrightarrow{\epsilon\langle d\rangle} B_1' \texttt{ [] } B_2'} \tag{8.21}$$

Example 8.5 Let us see which transitions the behaviour

$$\begin{array}{l} \textit{in2}\texttt{(}?x2 : \textsf{data}\texttt{)}\texttt{;}\ \textit{out1}\texttt{(!}5\texttt{)} \\ \texttt{[]} \\ \textit{out1}\texttt{(!}5\texttt{)}\texttt{;}\ \textit{in2}\texttt{(}?x2 : \textsf{data}\texttt{)} \end{array}$$

can make, assuming that data is a synonymous of int. It is translated to

$$\textit{selection1} \texttt{ [] } \textit{selection2}$$

where

$$\begin{array}{ll} \textit{selection1} \equiv & \textit{in2}\texttt{(}\$1 \texttt{=>} ?x2 : \textsf{data}\texttt{)}\ \ldots\texttt{;}\ \textit{out1}\texttt{(}\$1 \texttt{=>} \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 5\texttt{))}\ \ldots \\ \textit{selection2} \equiv & \textit{out1}\texttt{(}\$1 \texttt{=>} \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 5\texttt{))}\ \ldots\texttt{;}\ \textit{in2}\texttt{(}\$1 \texttt{=>} ?x2 : \textsf{data}\texttt{)}\ \ldots \end{array}$$

The whole behaviour can offer a communication on gate *in2* of any value of type data (any integer).

$$\frac{\mathcal{E} \vdash \textit{selection1} \xrightarrow{\textit{in2}\texttt{(\$1=>}n\texttt{)}} \mathbf{exit}\texttt{(}x2 \texttt{=>} n\texttt{)}\texttt{;}\ \textit{out1}\texttt{(}\$1 \texttt{=>} \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 5\texttt{))}\ \ldots}{\mathcal{E} \vdash \textit{selection1} \texttt{ [] } \textit{selection2} \xrightarrow{\textit{in2}\texttt{(\$1=>}n\texttt{)}} \mathbf{exit}\texttt{(}x2 \texttt{=>} n\texttt{)}\texttt{;}\ \textit{out1}\texttt{(}\$1 \texttt{=>} \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 5\texttt{))}\ \ldots} \tag{8.15}$$

where n is a constant of type data.

It can also offer a communication on gate *out1* of the value 5.

$$\frac{\mathcal{E} \vdash \textit{selection2} \xrightarrow{\textit{out1}\texttt{(\$1=>5)}} \mathbf{exit}\texttt{()}\texttt{;}\ \textit{in2}\texttt{(}\$1 \texttt{=>} ?x2 : \textsf{data}\texttt{)}\ \ldots}{\mathcal{E} \vdash \textit{selection1} \texttt{ [] } \textit{selection2} \xrightarrow{\textit{out1}\texttt{(\$1=>5)}} \mathbf{exit}\texttt{()}\texttt{;}\ \textit{in2}\texttt{(}\$1 \texttt{=>} ?x2 : \textsf{data}\texttt{)}\ \ldots} \tag{8.16}$$

Finally, *selection1* [] *selection2* can idle any time d (assuming that it is not in the scope of a **hide** operator that hides *out1* or *in2*).

$$\frac{\begin{array}{l} \mathcal{E} \vdash \textit{selection1} \xrightarrow{\epsilon\langle d\rangle} \textit{in2}\texttt{(}\$1 \texttt{=>} ?x2 : \textsf{data}\texttt{)}\ \ldots\mathbf{start}\langle d\rangle\texttt{;}\ \textit{out1}\texttt{(}\$1 \texttt{=>} \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 5\texttt{))}\ \ldots \\ \mathcal{E} \vdash \textit{selection2} \xrightarrow{\epsilon\langle d\rangle} \textit{out1}\texttt{(}\$1 \texttt{=>} \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 5\texttt{))}\ \ldots\mathbf{start}\langle d\rangle\texttt{;}\ \textit{in2}\texttt{(}\$1 \texttt{=>} ?x2 : \textsf{data}\texttt{)}\ \ldots \end{array}}{\begin{array}{ll} \mathcal{E} \vdash \textit{selection1} \texttt{ [] } \textit{selection2} \xrightarrow{\epsilon\langle d\rangle} & \textit{in2}\texttt{(}\$1 \texttt{=>} ?x2 : \textsf{data}\texttt{)}\ \ldots\mathbf{start}\langle d\rangle\texttt{;}\ \textit{out1}\texttt{(}\$1 \texttt{=>} \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 5\texttt{))}\ \ldots \\ & \texttt{[]}\ \textit{out1}\texttt{(}\$1 \texttt{=>} \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 5\texttt{))}\ \ldots\mathbf{start}\langle d\rangle\texttt{;}\ \textit{in2}\texttt{(}\$1 \texttt{=>} ?x2 : \textsf{data}\texttt{)}\ \ldots \end{array}} \tag{8.21}$$

Note how the time d in annotated in both parts of the selection.

8.5 Internal action

The internal action **i** has only the transition $\xrightarrow{\mathbf{i}}$ and immediately finishes.

$$\frac{}{\mathcal{E} \vdash \mathbf{i} \xrightarrow{\mathbf{i}} \mathbf{exit}\texttt{()}} \tag{8.22}$$

Since we want the internal action to be urgent, it cannot delay any time, so there is no rule dealing with time transitions.

Example 8.6 Let us see what transitions the behaviour *out2*`(!5) [] i` can perform. It is translated to $out2\texttt{(}\$1 \texttt{=>}\ \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 5\texttt{))}\ \texttt{@}\mathbf{any} : \mathsf{time}\ \mathbf{exit}\texttt{(}\$1 \texttt{=>}\ true\texttt{)}\ \mathbf{start}\langle 0\rangle\ \texttt{[]}\ \mathbf{i}$.

First, it can offer a communication on gate *out2*, due to the following proof:

$$\dfrac{\dfrac{\begin{array}{l}\mathcal{E} \vdash (\texttt{(}\$1 \texttt{=>}\ \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 5\texttt{))} \ \rightarrow^{\mathbf{PM}}\ \texttt{(}\$1 \texttt{=>} 5\texttt{))} \ \mapsto\ \langle\!|\ |\!\rangle \\ \mathcal{E} \vdash (\mathbf{any} : \mathsf{time} \ \rightarrow^{\mathbf{PM}}\ 0) \ \mapsto\ \langle\!|\ |\!\rangle \\ \mathcal{E} \vdash \mathbf{exit}\texttt{(}\$1 \texttt{=>}\ true\texttt{)} \xrightarrow{\delta\texttt{(}\$1\texttt{=>}true\texttt{)}} \mathbf{block}\end{array}}{\mathcal{E} \vdash out2\texttt{(}\$1 \texttt{=>}\ \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 5\texttt{))}\ \texttt{@}\mathbf{any} : \mathsf{time}\ \mathbf{exit}\texttt{(}\$1 \texttt{=>}\ true\texttt{)}\ \mathbf{start}\langle 0\rangle \xrightarrow{out2\texttt{(}\$1\texttt{=>}5\texttt{)}} \mathbf{exit}\texttt{()}}\ (8.8)}{\mathcal{E} \vdash out2\texttt{(}\$1 \texttt{=>}\ \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 5\texttt{))}\ \texttt{@}\mathbf{any} : \mathsf{time}\ \mathbf{exit}\texttt{(}\$1 \texttt{=>}\ true\texttt{)}\ \mathbf{start}\langle 0\rangle\ \texttt{[]}\ \mathbf{i} \xrightarrow{out2\texttt{(}\$1\texttt{=>}5\texttt{)}} \mathbf{exit}\texttt{()}}\ (8.15)$$

Second, it can perform an internal action:

$$\dfrac{\mathcal{E} \vdash \mathbf{i} \xrightarrow{\mathbf{i}} \mathbf{exit}\texttt{()}}{\mathcal{E} \vdash out2\texttt{(}\$1 \texttt{=>}\ \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 5\texttt{))}\ \texttt{@}\mathbf{any} : \mathsf{time}\ \mathbf{exit}\texttt{(}\$1 \texttt{=>}\ true\texttt{)}\ \mathbf{start}\langle 0\rangle\ \texttt{[]}\ \mathbf{i} \xrightarrow{\mathbf{i}} \mathbf{exit}\texttt{()}}\ (8.18)$$

There are no more transitions for this behaviour. So, *out2*`(!5) [] i` cannot delay, because of the urgency of the internal action. Here, we see that the urgency of the internal action does not mean that internal action *has to be performed*, but it means that if the internal action is performed, it has to be *without delay*.

8.6 Successful termination with values

$\mathbf{exit}\texttt{(}RN\texttt{)}$ is a behaviour that finishes immediately (it has no time transition) and communicates bindings RN through the special gate δ.

$$\dfrac{}{\mathcal{E} \vdash \mathbf{exit}\texttt{(}RN\texttt{)} \xrightarrow{\delta\texttt{(}RN\texttt{)}} \mathbf{block}} \qquad (8.23)$$

Example 8.7 The behaviour $\mathbf{exit}\texttt{(}RN\texttt{)}$ is used to propagate (through gate δ) bindings produced by a behaviour. Let us see how these bindings are propagated when the behaviour is $inP\texttt{(?}x : \mathsf{int}\texttt{)}$, which is translated to

$$inP\texttt{(}\$1 \texttt{=>}\ \texttt{?}x : \mathsf{int}\texttt{)}\ \texttt{@}\mathbf{any} : \mathsf{time}\ \mathbf{exit}\texttt{(}\$1 \texttt{=>}\ true\texttt{)}\ \mathbf{start}\langle 0\rangle.$$

As we saw in Example 8.3, this behaviour has the following transition

$$inP\texttt{(}\$1 \texttt{=>}\ \texttt{?}x : \mathsf{int}\texttt{)}\ \ldots \xrightarrow{inP\texttt{(}\$1\texttt{=>}n\texttt{)}} \mathbf{exit}\texttt{(}x \texttt{=>}\ n\texttt{)}$$

where n may be any constant of type int. Now the binding $x \texttt{=>}\ n$ is propagated because $\mathbf{exit}\texttt{(}x \texttt{=>}\ n\texttt{)}$ has the following transition

$$\dfrac{}{\mathcal{E} \vdash \mathbf{exit}\texttt{(}x \texttt{=>}\ n\texttt{)} \xrightarrow{\delta\texttt{(}x\texttt{=>}n\texttt{)}} \mathbf{block}}\ (8.23)$$

Example 8.8 The behaviour $\mathbf{exit}\texttt{(}RN\texttt{)}$ is also used to propagate values returned by an expression. For example, the value expression 3 is translated into $\mathbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{)}$ which *returns* the integer value 3.

$$\dfrac{}{\mathcal{E} \vdash \mathbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{)} \xrightarrow{\delta\texttt{(}\$1\texttt{=>}3\texttt{)}} \mathbf{block}}\ (8.23)$$

8.7 Inaction

The behaviour **stop** can do nothing, but wait. Therefore, the only transitions it has are time transitions.

$$\frac{}{\mathcal{E} \vdash \textbf{stop} \xrightarrow{\epsilon\langle d\rangle} \textbf{stop}} \tag{8.24}$$

8.8 Time block

The behaviour **block** cannot do anything, neither delay. Hence, there is no inference rule dealing with this behaviour.

This is a quite strange behaviour because, as we have said, time is considered *foreign* to the process. For example, the behaviour

$$\textbf{block} \texttt{ ||| } \textbf{wait}\texttt{(}1\texttt{);}G$$

cannot do anything because time is blocked, which does not seem reasonable.

8.9 Parallel operator

In a parallel composition $B_1 \texttt{ |[}G_1\texttt{,} \dots \texttt{,}G_n\texttt{]| } B_2$, if B_1 or B_2 offers a communication on a gate G not in the list $G_1\texttt{,} \dots \texttt{,}G_n$, then the parallel composition offers that communication:

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{G\texttt{(}RN\texttt{)}} B_1'}{\mathcal{E} \vdash B_1 \texttt{ |[}G_1\texttt{,} \dots \texttt{,}G_n\texttt{]| } B_2 \xrightarrow{G\texttt{(}RN\texttt{)}} B_1' \texttt{ |[}G_1\texttt{,} \dots \texttt{,}G_n\texttt{]| } B_2} \; [G \notin \{G_1, \dots, G_n\}] \tag{8.25}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{G\texttt{(}RN\texttt{)}} B_2'}{\mathcal{E} \vdash B_1 \texttt{ |[}G_1\texttt{,} \dots \texttt{,}G_n\texttt{]| } B_2 \xrightarrow{G\texttt{(}RN\texttt{)}} B_1 \texttt{ |[}G_1\texttt{,} \dots \texttt{,}G_n\texttt{]| } B_2'} \; [G \notin \{G_1, \dots, G_n\}] \tag{8.26}$$

However, in order to offer a communication on a gate G_i in the given synchronization list $G_1\texttt{,} \dots \texttt{,}G_n$, both B_1 and B_2 have to offer the same communication on that gate:

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{G_i\texttt{(\$1=>}N\texttt{)}} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{G_i\texttt{(\$1=>}N\texttt{)}} B_2'}{\mathcal{E} \vdash B_1 \texttt{ |[}G_1\texttt{,} \dots \texttt{,}G_n\texttt{]| } B_2 \xrightarrow{G_i\texttt{(\$1=>}N\texttt{)}} B_1' \texttt{ |[}G_1\texttt{,} \dots \texttt{,}G_n\texttt{]| } B_2'} \tag{8.27}$$

Internal actions can be made by both B_1 or B_2 without the knowledge of the other one. That is, $B_1 \texttt{ |[}G_1\texttt{,} \dots \texttt{,}G_n\texttt{]| } B_2$ may perform the following transitions:

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\textbf{i}} B_1'}{\mathcal{E} \vdash B_1 \texttt{ |[}G_1\texttt{,} \dots \texttt{,}G_n\texttt{]| } B_2 \xrightarrow{\textbf{i}} B_1' \texttt{ |[}G_1\texttt{,} \dots \texttt{,}G_n\texttt{]| } B_2} \tag{8.28}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{\textbf{i}} B_2'}{\mathcal{E} \vdash B_1 \texttt{ |[}G_1\texttt{,} \dots \texttt{,}G_n\texttt{]| } B_2 \xrightarrow{\textbf{i}} B_1 \texttt{ |[}G_1\texttt{,} \dots \texttt{,}G_n\texttt{]| } B_2'} \tag{8.29}$$

In order to preserve the urgency of internal actions, when B_1 or B_2 raises an exception, the parallel behaviour performs an internal action, and the exception raised is annotated outside the parallel behaviour in order to be raised again.

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{X\texttt{(}RN\texttt{)}} B_1'}{\mathcal{E} \vdash B_1\ \texttt{|[}G_1\texttt{,}\ldots\texttt{,}G_n\texttt{]|}\ B_2 \xrightarrow{\mathbf{i}} \mathbf{signal}\ X\texttt{(}\mathbf{exit}\texttt{(}RN\texttt{))}\texttt{;(}B_1'\ \texttt{|[}G_1\texttt{,}\ldots\texttt{,}G_n\texttt{]|}\ B_2\texttt{)}} \tag{8.30}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{X\texttt{(}RN\texttt{)}} B_2'}{\mathcal{E} \vdash B_1\ \texttt{|[}G_1\texttt{,}\ldots\texttt{,}G_n\texttt{]|}\ B_2 \xrightarrow{\mathbf{i}} \mathbf{signal}\ X\texttt{(}\mathbf{exit}\texttt{(}RN\texttt{))}\texttt{;(}B_1\ \texttt{|[}G_1\texttt{,}\ldots\texttt{,}G_n\texttt{]|}\ B_2'\texttt{)}} \tag{8.31}$$

The parallel composition $B_1\ \texttt{|[}G_1\texttt{,}\ldots\texttt{,}G_n\texttt{]|}\ B_2$ finishes when B_1 and B_2 have finished, and it produces the bindings produced by both, which have to be disjoint bindings, as required by the static semantics.

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta\texttt{(}RN_1\texttt{)}} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\delta\texttt{(}RN_2\texttt{)}} B_2'}{\mathcal{E} \vdash B_1\ \texttt{|[}G_1\texttt{,}\ldots\texttt{,}G_n\texttt{]|}\ B_2 \xrightarrow{\delta\texttt{(}RN_1\texttt{,}RN_2\texttt{)}} \mathbf{block}} \tag{8.32}$$

If both B_1 and B_2 can idle a period of time d, then the parallel composition can also idle d.

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\epsilon\langle d\rangle} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\epsilon\langle d\rangle} B_2'}{\mathcal{E} \vdash B_1\ \texttt{|[}G_1\texttt{,}\ldots\texttt{,}G_n\texttt{]|}\ B_2 \xrightarrow{\epsilon\langle d\rangle} B_1'\ \texttt{|[}G_1\texttt{,}\ldots\texttt{,}G_n\texttt{]|}\ B_2'} \tag{8.33}$$

But if one of them, say B_1, can finish after d units of time, and B_2 can idle a time greater than d, then the parallel composition can idle this greater time. It is said that termination is urgent except when it is in the scope of a parallel operator, where both behaviours have to finish at the same time.

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta\texttt{(}RN\texttt{)@}d} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\epsilon\langle d+d'\rangle} B_2'}{\mathcal{E} \vdash B_1\ \texttt{|[}G_1\texttt{,}\ldots\texttt{,}G_n\texttt{]|}\ B_2 \xrightarrow{\epsilon\langle d+d'\rangle} \mathbf{exit}\texttt{(}RN\texttt{)}\ \texttt{|[}G_1\texttt{,}\ldots\texttt{,}G_n\texttt{]|}\ B_2'}\ [0 < d'] \tag{8.34}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\epsilon\langle d+d'\rangle} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\delta\texttt{(}RN\texttt{)@}d} B_2'}{\mathcal{E} \vdash B_1\ \texttt{|[}G_1\texttt{,}\ldots\texttt{,}G_n\texttt{]|}\ B_2 \xrightarrow{\epsilon\langle d+d'\rangle} B_1'\ \texttt{|[}G_1\texttt{,}\ldots\texttt{,}G_n\texttt{]|}\ \mathbf{exit}\texttt{(}RN\texttt{)}}\ [0 < d'] \tag{8.35}$$

Here $\mathbf{exit}\texttt{(}RN\texttt{)}$ is used to annotate the bindings produced by the behaviour which finishes first.

Example 8.9 Let us see that the behaviour $\mathit{in2}\texttt{(?}x2\texttt{:data)}\ \texttt{|||}\ \mathit{out1}\texttt{(!3)}$, which is translated to

$$\mathit{in2}\texttt{(}\$1 \texttt{=>} \texttt{?}x2\texttt{:data)}\ldots\ \texttt{|[]|}\ \mathit{out1}\texttt{(}\$1 \texttt{=>}\ \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{))}\ldots$$

has the following evolution

$$\begin{array}{lll}
\mathit{in2}\texttt{(}\$1 \texttt{=>} \texttt{?}x2\texttt{:data)}\ldots\ \texttt{|[]|}\ \mathit{out1}\texttt{(}\$1 \texttt{=>}\ \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{))}\ldots & \xrightarrow{\mathit{in2}\texttt{(\$1=>8)}} & \\
\mathbf{exit}\texttt{(}x2 \texttt{=>} 8\texttt{)}\ \texttt{|[]|}\ \mathit{out1}\texttt{(}\$1 \texttt{=>}\ \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{))}\ldots & \xrightarrow{\epsilon\langle 1\rangle} & \\
\mathbf{exit}\texttt{(}x2 \texttt{=>} 8\texttt{)}\ \texttt{|[]|}\ \mathit{out1}\texttt{(}\$1 \texttt{=>}\ \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{))}\ldots\mathbf{start}\langle 1\rangle & \xrightarrow{\mathit{out1}\texttt{(\$1=>3)}} & \\
\mathbf{exit}\texttt{(}x2 \texttt{=>} 8\texttt{)}\ \texttt{|[]|}\ \mathbf{exit}\texttt{()} & \xrightarrow{\delta\texttt{(}x2\texttt{=>8)}} & \mathbf{block}
\end{array}$$

The first transition is proved as follows:

$$\frac{\mathcal{E} \vdash \mathit{in2}\texttt{(}\$1 \texttt{=>} \texttt{?}x2\texttt{:data)}\ldots \xrightarrow{\mathit{in2}\texttt{(\$1=>8)}} \mathbf{exit}\texttt{(}x2 \texttt{=>} 8\texttt{)}}{\begin{array}{l}\mathcal{E} \vdash\ \ \mathit{in2}\texttt{(}\$1 \texttt{=>} \texttt{?}x2\texttt{:data)}\ldots\ \texttt{|[]|}\ \mathit{out1}\texttt{(}\$1 \texttt{=>}\ \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{))}\ldots \\ \qquad \xrightarrow{\mathit{in2}\texttt{(\$1=>8)}} \mathbf{exit}\texttt{(}x2 \texttt{=>} 8\texttt{)}\ \texttt{|[]|}\ \mathit{out1}\texttt{(}\$1 \texttt{=>}\ \texttt{!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{))}\ldots\end{array}} \tag{8.25}$$

since gate *in2* $\notin$ `[ ]`.

The transition $\xrightarrow{\epsilon\langle 1\rangle}$ is proved by using rule 8.34

$$\frac{\begin{array}{l}\mathcal{E} \vdash \mathbf{exit}\texttt{(}x2 \texttt{=>} 8\texttt{)} \xrightarrow{\delta\texttt{(}x2\texttt{=>}8\texttt{)@}0} \mathbf{block}\\ \mathcal{E} \vdash \mathit{out1}\texttt{(}\$1 \texttt{=> !}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{))}\ldots \xrightarrow{\epsilon\langle 1\rangle} \mathit{out1}\texttt{(}\$1 \texttt{=> !}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{))}\ldots\mathbf{start}\langle 1\rangle\end{array}}{\begin{array}{l}\mathcal{E} \vdash \ \ \mathbf{exit}\texttt{(}x2 \texttt{=>} 8\texttt{)}\ \texttt{|[]|}\ \mathit{out1}\texttt{(}\$1 \texttt{=> !}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{))}\ldots\\ \qquad\qquad \xrightarrow{\epsilon\langle 1\rangle} \mathbf{exit}\texttt{(}x2 \texttt{=>} 8\texttt{)}\ \texttt{|[]|}\ \mathit{out1}\texttt{(}\$1 \texttt{=> !}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{))}\ldots\mathbf{start}\langle 1\rangle\end{array}} \tag{8.34}$$

The third transition is correct due to the following rule:

$$\frac{\mathcal{E} \vdash \mathit{out1}\texttt{(}\$1 \texttt{=> !}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{))}\ldots\mathbf{start}\langle 1\rangle \xrightarrow{\mathit{out1}\texttt{(\$1=>3)}} \mathbf{exit}\texttt{()}}{\mathcal{E} \vdash \mathbf{exit}\texttt{(}x2 \texttt{=>} 8\texttt{)}\ \texttt{|[]|}\ \mathit{out1}\texttt{(}\$1 \texttt{=> !}\mathbf{exit}\texttt{(}\$1 \texttt{=>} 3\texttt{))}\ldots\mathbf{start}\langle 1\rangle \xrightarrow{\mathit{out1}\texttt{(\$1=>3)}} \mathbf{exit}\texttt{(}x2 \texttt{=>} 8\texttt{)}\ \texttt{|[]|}\ \mathbf{exit}\texttt{()}} \tag{8.26}$$

And, finally, the fourth transition is proved as follows:

$$\frac{\begin{array}{l}\mathcal{E} \vdash \mathbf{exit}\texttt{(}x2 \texttt{=>} 8\texttt{)} \xrightarrow{\delta\texttt{(}x2\texttt{=>}8\texttt{)}} \mathbf{block}\\ \mathcal{E} \vdash \mathbf{exit}\texttt{()} \xrightarrow{\delta\texttt{()}} \mathbf{block}\end{array}}{\mathcal{E} \vdash \mathbf{exit}\texttt{(}x2 \texttt{=>} 8\texttt{)}\ \texttt{|[]|}\ \mathbf{exit}\texttt{()} \xrightarrow{\delta\texttt{(}x2\texttt{=>}8\texttt{)}} \mathbf{block}} \tag{8.32}$$

8.10 Synchronization operator

The synchronization operator is similar to the previous one. Communication offers on any gate G have to be offered by both B_1 and B_2:

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{G\texttt{(\$1=>}N\texttt{)}} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{G\texttt{(\$1=>}N\texttt{)}} B_2'}{\mathcal{E} \vdash B_1\ \texttt{||}\ B_2 \xrightarrow{G\texttt{(\$1=>}N\texttt{)}} B_1'\ \texttt{||}\ B_2'} \tag{8.36}$$

and internal actions can be made separately by B_1 and B_2:

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\mathbf{i}} B_1'}{\mathcal{E} \vdash B_1\ \texttt{||}\ B_2 \xrightarrow{\mathbf{i}} B_1'\ \texttt{||}\ B_2} \tag{8.37}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{\mathbf{i}} B_2'}{\mathcal{E} \vdash B_1\ \texttt{||}\ B_2 \xrightarrow{\mathbf{i}} B_1\ \texttt{||}\ B_2'} \tag{8.38}$$

If an exception is raised by one of the components, an internal action is performed, and the exception is raised again:

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{X\texttt{(\$1=>}N\texttt{)}} B_1'}{\mathcal{E} \vdash B_1\ \texttt{||}\ B_2 \xrightarrow{\mathbf{i}} \mathbf{signal}\ X\texttt{(}\mathbf{exit}\texttt{(}\$1 \texttt{=>} N\texttt{))};\texttt{(}B_1'\ \texttt{||}\ B_2\texttt{)}} \tag{8.39}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{X\texttt{(\$1=>}N\texttt{)}} B_2'}{\mathcal{E} \vdash B_1\ \texttt{||}\ B_2 \xrightarrow{\mathbf{i}} \mathbf{signal}\ X\texttt{(}\mathbf{exit}\texttt{(}\$1 \texttt{=>} N\texttt{))};\texttt{(}B_1\ \texttt{||}\ B_2'\texttt{)}} \tag{8.40}$$

$B_1 \;\texttt{||}\; B_2$ finishes when its two components finish, and it propagates bindings produced by them, which are disjoint because $B_1 \;\texttt{||}\; B_2$ has passed the static semantics:

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta\texttt{(}RN_1\texttt{)}} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\delta\texttt{(}RN_2\texttt{)}} B_2'}{\mathcal{E} \vdash B_1 \;\texttt{||}\; B_2 \xrightarrow{\delta\texttt{(}RN_1\texttt{,}RN_2\texttt{)}} \mathbf{block}} \tag{8.41}$$

Regarding time, the synchronization operator behaves as the parallel one. If both components can idle a period d then the whole behaviour idles this time:

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\epsilon\langle d\rangle} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\epsilon\langle d\rangle} B_2'}{\mathcal{E} \vdash B_1 \;\texttt{||}\; B_2 \xrightarrow{\epsilon\langle d\rangle} B_1' \;\texttt{||}\; B_2'} \tag{8.42}$$

If one can finish after d units of time, and the other can idle a greater time, the whole behaviour idles that time:

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta\texttt{(}RN\texttt{)@}d} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\epsilon\langle d+d'\rangle} B_2'}{\mathcal{E} \vdash B_1 \;\texttt{||}\; B_2 \xrightarrow{\epsilon\langle d+d'\rangle} \mathbf{exit}\texttt{(}RN\texttt{)} \;\texttt{||}\; B_2'} \; [0 < d'] \tag{8.43}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\epsilon\langle d+d'\rangle} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\delta\texttt{(}RN\texttt{)@}d} B_2'}{\mathcal{E} \vdash B_1 \;\texttt{||}\; B_2 \xrightarrow{\epsilon\langle d+d'\rangle} B_1' \;\texttt{||}\; \mathbf{exit}\texttt{(}RN\texttt{)}} \; [0 < d'] \tag{8.44}$$

8.11 General parallel operator

As in the static semantics, we use the notation $\vec{G_i}$ to abbreviate the gate list $G_{i1}\texttt{,} \ldots \texttt{,}G_{ir_i}$.

A general parallel behaviour can offer a communication on a gate G_j which is in the list of degrees $(G_1\texttt{\#}n_1\texttt{,} \ldots \texttt{,}G_p\texttt{\#}n_p)$ with degree n_j, if n_j subcomponents B_i, having G_j in their synchronization list, can offer the same communication.

$$\frac{\forall\, i \in \Sigma \,.\, \mathcal{E} \vdash B_i \xrightarrow{G_j\texttt{(}RN\texttt{)}} B_i'}{\begin{array}{l} \mathcal{E} \vdash (\mathbf{par}\ G_1\texttt{\#}n_1\texttt{,} \ldots \texttt{,}G_p\texttt{\#}n_p\ \mathbf{in} \\ \qquad \texttt{[}\vec{G_1}\texttt{]} \to B_1 \ \cdots \ \texttt{||[}\vec{G_m}\texttt{]} \to B_m) \\ \qquad \xrightarrow{G_j\texttt{(}RN\texttt{)}} (\mathbf{par}\ G_1\texttt{\#}n_1\texttt{,} \ldots \texttt{,}G_p\texttt{\#}n_p\ \mathbf{in} \\ \qquad\qquad \texttt{[}\vec{G_1}\texttt{]} \to B_1' \ \cdots \ \texttt{||[}\vec{G_m}\texttt{]} \to B_m') \end{array}} \; [G_j \in \{G_1, \ldots, G_p\}] \tag{8.45}$$

with the side conditions that Σ represents a subset of n_j subcomponents having G_j in their synchronization list, that is, $\Sigma \subseteq \{1, \ldots, m\} \wedge |\Sigma| = n_j \wedge G_j \in \bigcap_{i\in\Sigma} \vec{G_i}$; and that the behaviours that do not participate in the communication, do not change, $\forall\, i \notin \Sigma \,.\, B_i = B_i'$.

A general parallel composition can also offer a communication on a gate G which is not in the list of degrees, if all the subcomponents having G in their synchronization list offer the same communication.

$$\frac{\forall i \in \Sigma.\mathcal{E} \vdash B_i \xrightarrow{G\texttt{(}RN\texttt{)}} B_i'}{\begin{array}{l} \mathcal{E} \vdash (\mathbf{par}\ G_1\texttt{\#}n_1\texttt{,} \ldots \texttt{,}G_p\texttt{\#}n_p\ \mathbf{in} \\ \qquad \texttt{[}\vec{G_1}\texttt{]} \to B_1 \ \cdots \ \texttt{||[}\vec{G_m}\texttt{]} \to B_m) \\ \qquad \xrightarrow{G\texttt{(}RN\texttt{)}} (\mathbf{par}\ G_1\texttt{\#}n_1\texttt{,} \ldots \texttt{,}G_p\texttt{\#}n_p\ \mathbf{in} \\ \qquad\qquad \texttt{[}\vec{G_1}\texttt{]} \to B_1' \ \cdots \ \texttt{||[}\vec{G_m}\texttt{]} \to B_m') \end{array}} \; [G \notin \{G_1, \ldots, G_p\}] \tag{8.46}$$

here Σ is the nonempty subset of behaviours having G in their synchronization list, $\Sigma = \{i \mid G \in \vec{G_i}, 1 \leq i \leq m\} \wedge \Sigma \neq \emptyset$; and $\forall i \notin \Sigma . B_i = B'_i$.

Any subcomponent B_j can offer a communication on a gate which is not in its synchronization list, and this communication is offered by the whole general parallel behaviour.

$$\frac{\mathcal{E} \vdash B_j \xrightarrow{G(RN)} B'_j}{\begin{array}{l}\mathcal{E} \vdash (\textbf{par } G_1\#n_1, \ldots, G_p\#n_p \textbf{ in} \\ \qquad [\vec{G_1}] \to B_1 \ \cdots \ || [\vec{G_j}] \to B_j \ \cdots \ || [\vec{G_m}] \to B_m) \\ \qquad \xrightarrow{G(RN)} (\textbf{par } G_1\#N_1, \ldots, G_p\#N_p \textbf{ in} \\ \qquad\qquad [\vec{G_1}] \to B_1 \ \cdots \ || [\vec{G_j}] \to B'_j \ \cdots \ || [\vec{G_m}] \to B_m)\end{array}} \left[G \notin \vec{G_j}\right] \tag{8.47}$$

Internal actions can be performed by any of the subcomponents.

$$\frac{\mathcal{E} \vdash B_j \xrightarrow{\mathbf{i}} B'_j}{\begin{array}{l}\mathcal{E} \vdash (\textbf{par } G_1\#n_1, \ldots, G_p\#n_p \textbf{ in} \\ \qquad [\vec{G_1}] \to B_1 \ \cdots \ || [\vec{G_j}] \to B_j \ \cdots \ || [\vec{G_m}] \to B_m) \\ \qquad \xrightarrow{\mathbf{i}} \quad (\textbf{par } G_1\#N_1, \ldots, G_p\#N_p \textbf{ in} \\ \qquad\qquad [\vec{G_1}] \to B_1 \ \cdots \ || [\vec{G_j}] \to B'_j \ \cdots \ || [\vec{G_m}] \to B_m)\end{array}} \tag{8.48}$$

Exceptions can also be performed by any of the subcomponents, by performing an internal action first.

$$\frac{\mathcal{E} \vdash B_j \xrightarrow{X(\$1 \texttt{=>} N)} B'_j}{\begin{array}{l}\mathcal{E} \vdash (\textbf{par } G_1\#n_1, \ldots, G_p\#n_p \textbf{ in} \\ \qquad [\vec{G_1}] \to B_1 \ \cdots \ || [\vec{G_j}] \to B_j \ \cdots \ || [\vec{G_m}] \to B_m) \\ \qquad \xrightarrow{\mathbf{i}} \textbf{signal } X(\textbf{exit}(\$1 \texttt{=>} N)) ; (\textbf{par } G_1\#n_1, \ldots, G_p\#n_p \textbf{ in} \\ \qquad\qquad || [\vec{G_1}] \to B_1 \ \cdots \ || [\vec{G_j}] \to B'_j \ \cdots \ || [\vec{G_m}] \to B_m)\end{array}} \tag{8.49}$$

The general parallel composition finishes when all its subcomponent behaviours finish, and it produces all disjoint bindings produced by them.

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN_1)} B'_1 \ \cdots \ \mathcal{E} \vdash B_m \xrightarrow{\delta(RN_m)} B'_m}{\mathcal{E} \vdash (\textbf{par } G_1\#n_1, \ldots, G_p\#n_p \textbf{ in } [\vec{G_1}] \to B_1 \ \cdots \ || [\vec{G_m}] \to B_m) \xrightarrow{\delta(RN_1, \ldots, RN_m)} \textbf{block}} \tag{8.50}$$

If all the subcomponents can idle a period of time d, then the whole general parallel behaviour can idle that time.

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\epsilon\langle d \rangle} B'_1 \ \cdots \ \mathcal{E} \vdash B_m \xrightarrow{\epsilon\langle d \rangle} B'_m}{\begin{array}{l}\mathcal{E} \vdash (\textbf{par } G_1\#n_1, \ldots, G_p\#n_p \textbf{ in } [\vec{G_1}] \to B_1 \ \cdots \ || [\vec{G_m}] \to B_m) \\ \qquad \xrightarrow{\epsilon\langle d \rangle} (\textbf{par } G_1\#n_1, \ldots, G_p\#n_p \textbf{ in } [\vec{G_1}] \to B'_1 \ \cdots \ || [\vec{G_m}] \to B'_m)\end{array}} \tag{8.51}$$

In [Que98] there is one more rule dealing with the case when one subcomponent has finished and the others can idle a time d. But that rule is not enough general. We need a rule dealing with the case when some subcomponents (but not all) have finished (possibly at different times) before time d and the others can idle a time d. In this case the whole behaviour can idle d.

$$\frac{\begin{array}{c}\forall\, i \in \Sigma_1 \,.\, \mathcal{E} \vdash B_i \xrightarrow{\epsilon\langle d\rangle} B_i' \\ \forall\, i \in \Sigma_2 \,.\, \mathcal{E} \vdash B_i \xrightarrow{\delta(RN)@d_i} B_i'\end{array}}{\begin{array}{l}\mathcal{E} \vdash (\mathbf{par}\ G_1\texttt{\#}n_1\texttt{,}\ldots\texttt{,}G_p\texttt{\#}n_p\ \mathbf{in} \\ \qquad \texttt{[}\vec{G_1}\,\texttt{]} \rightarrow B_1\ \cdots\ \texttt{||[}\vec{G_j}\,\texttt{]} \rightarrow B_j\ \cdots\ \texttt{||[}\vec{G_m}\,\texttt{]} \rightarrow B_m) \\ \quad \xrightarrow{\epsilon\langle d\rangle} (\mathbf{par}\ G_1\texttt{\#}n_1\texttt{,}\ldots\texttt{,}G_p\texttt{\#}n_p\ \mathbf{in} \\ \qquad\qquad \texttt{[}\vec{G_1}\,\texttt{]} \rightarrow B_1\ \cdots\ \texttt{||[}\vec{G_j}\,\texttt{]} \rightarrow \mathbf{exit}(RN)\ \cdots\ \texttt{||[}\vec{G_m}\,\texttt{]} \rightarrow B_m)\end{array}} \tag{8.52}$$

with the side conditions that $\Sigma_1 \cap \Sigma_2 = \emptyset$, $\Sigma_1 \neq \emptyset$, $\Sigma_1 \cup \Sigma_2 = \{1, 2, ..., m\}$, and $\forall\, i \in \Sigma_2 \,.\, d_i < d$.

8.12 Parallel over values

A parallel over values behaviour **par** P **in** N `|||` B has the same transitions that the interleaving of a series of instantiations of B, one for each value of the list N. If the list N has elements, then we have

$$\frac{\begin{array}{l}\mathcal{E} \vdash (P \ \xrightarrow{}^{\mathbf{PM}}\ N_1) \ \mapsto\ \langle\!|\, RN \,|\!\rangle \\ \mathcal{E} \vdash B\,[\![\, RN \,]\!]\ \texttt{|[]|}\ \mathbf{par}\ P\ \mathbf{in}\ N_2\ \texttt{|||}\ B \xrightarrow{\mu(RN')} B'\end{array}}{\mathcal{E} \vdash \mathbf{par}\ P\ \mathbf{in}\ \mathit{cons}(N_1\texttt{,}N_2)\ \texttt{|||}\ B \xrightarrow{\mu(RN')} B'} \tag{8.53}$$

where, in this case, $\mu(RN')$ abbreviates $G(RN')$, **i**, or $\delta(RN')$.

In [Que98] there is also a rule dealing with the case when the interleaving of the instantiations of B raises an exception,

$$\frac{\begin{array}{l}\mathcal{E} \vdash (P \ \xrightarrow{}^{\mathbf{PM}}\ N_1) \ \mapsto\ \langle\!|\, RN \,|\!\rangle \\ \mathcal{E} \vdash B\,[\![\, RN \,]\!]\ \texttt{|[]|}\ \mathbf{par}\ P\ \mathbf{in}\ N_2\ \texttt{|||}\ B \xrightarrow{X(RN')} B'\end{array}}{\mathcal{E} \vdash \mathbf{par}\ P\ \mathbf{in}\ \mathit{cons}(N_1\texttt{,}N_2)\ \texttt{|||}\ B \xrightarrow{\mathbf{i}} \mathbf{signal}\ X(\mathbf{exit}(RN'))\texttt{;}B'}$$

but this case will never occur because an interleaving behaviour cannot raise exceptions itself (see Section 8.9).

There are no more rules dealing with transitions other than time transitions in [Que98], because when the list N is empty, then the parallel over values behaviour does not do anything. But this is an error, which makes that the previous unfolding does not finish. The behaviour

$$\mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{cons}(1\texttt{,}\mathit{cons}(2\texttt{,}\mathit{nil}))\ \texttt{|||}\ B$$

is unfolded as

$$B\,[\![\, x\ \texttt{=>}\ 1 \,]\!]\ \texttt{|[]|}\ B\,[\![\, x\ \texttt{=>}\ 2 \,]\!]\ \texttt{|[]|}\ \mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{nil}\ \texttt{|||}\ B$$

and the parallel operator rules states that this behaviour finishes if *all* the subbehaviours finish. So, we need a rule which says that **par** ?x **in** *nil* `|||` B can finish:

$$\frac{}{\mathcal{E} \vdash \mathbf{par}\ P\ \mathbf{in}\ \mathit{nil}\ \texttt{|||}\ B \xrightarrow{\delta()} \mathbf{block}} \tag{8.54}$$

Regarding time, the behaviour **par** P **in** N `|||` B can idle d units of time if its unfolding can idle the same time.

$$\frac{\begin{array}{l}\mathcal{E} \vdash (P \ \xrightarrow{}^{\mathbf{PM}}\ N_1) \ \mapsto\ \langle\!|\, RN \,|\!\rangle \\ \mathcal{E} \vdash B\,[\![\, RN \,]\!]\ \texttt{|[]|}\ \mathbf{par}\ P\ \mathbf{in}\ N_2\ \texttt{|||}\ B \xrightarrow{\epsilon\langle d\rangle} B'\end{array}}{\mathcal{E} \vdash \mathbf{par}\ P\ \mathbf{in}\ \mathit{cons}(N_1\texttt{,}N_2)\ \texttt{|||}\ B \xrightarrow{\epsilon\langle d\rangle} B'} \tag{8.55}$$

Example 8.10 Let us see how the behaviour

$$\begin{array}{l}\mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \texttt{[1,2,3] |||}\\ \quad \mathit{inP}\texttt{(!}x\texttt{)}\\ \mathbf{endpar}\end{array}$$

can communicate the values in the list `[1,2,3]` through gate *inP*. It is translated to

$$\begin{array}{l}\mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{cons}\texttt{(1,}\mathit{cons}\texttt{(2,}\mathit{cons}\texttt{(3,}\mathit{nil}\texttt{()))) |||}\\ \quad \mathit{inP}\texttt{(!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} x\texttt{)) @}\mathbf{any} : \mathsf{time}\ \mathbf{exit}\texttt{(}\$1 \texttt{=>} \mathit{true}\texttt{)}\ \mathbf{start}\langle 0\rangle\end{array}$$

and a possible execution would be:

$$\begin{array}{ll}\mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{cons}\texttt{(1,}\mathit{cons}\texttt{(2,}\mathit{cons}\texttt{(3,}\mathit{nil}\texttt{()))) ||| }\mathit{inP}\texttt{(!}x\texttt{)} & \xrightarrow{\mathit{inP}\texttt{(\$1=>1)}}\\ \mathbf{exit}\texttt{() |[]| }\mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{cons}\texttt{(2,}\mathit{cons}\texttt{(3,}\mathit{nil}\texttt{())) ||| }\mathit{inP}\texttt{(!}x\texttt{)} & \xrightarrow{\mathit{inP}\texttt{(\$1=>3)}}\\ \mathbf{exit}\texttt{() |[]| }\mathit{inP}\texttt{(!2) |[]| }\mathbf{exit}\texttt{() |[]| }\mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{nil}\texttt{() ||| }\mathit{inP}\texttt{(!}x\texttt{)} & \xrightarrow{\mathit{inP}\texttt{(\$1=>2)}}\\ \mathbf{exit}\texttt{() |[]| }\mathbf{exit}\texttt{() |[]| }\mathbf{exit}\texttt{() |[]| }\mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{nil}\texttt{() ||| }\mathit{inP}\texttt{(!}x\texttt{)} & \xrightarrow{\delta\texttt{()}}\\ \mathbf{block} & \end{array}$$

where we write $\mathit{inP}\texttt{(!}x\texttt{)}$ instead of $\mathit{inP}\texttt{(!}\mathbf{exit}\texttt{(}\$1 \texttt{=>} x\texttt{)) @}\mathbf{any} : \mathsf{time}\ \mathbf{exit}\texttt{(}\$1 \texttt{=>} \mathit{true}\texttt{)}\ \mathbf{start}\langle 0\rangle$, in order to do the behaviours shorter.

The first transition is possible due to the following proof:

$$\dfrac{\begin{array}{l}\mathcal{E} \vdash (\texttt{?}x \ \ \rightarrow^{\mathbf{PM}}\ \ 1) \ \mapsto\ \langle\!| x \texttt{=>} 1 |\!\rangle\\ \mathcal{E} \vdash \mathit{inP}\texttt{(!1) |[]| }\mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{cons}\texttt{(2,}\mathit{cons}\texttt{(3,}\mathit{nil}\texttt{()))|||}\mathit{inP}\texttt{(!}x\texttt{)}\\ \quad \xrightarrow{\mathit{inP}\texttt{(\$1=>1)}} \mathbf{exit}\texttt{() |[]| }\mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{cons}\texttt{(2,}\mathit{cons}\texttt{(3,}\mathit{nil}\texttt{()))|||}\mathit{inP}\texttt{(!}x\texttt{)}\end{array}}{\begin{array}{l}\mathcal{E} \vdash \mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{cons}\texttt{(1,}\mathit{cons}\texttt{(2,}\mathit{cons}\texttt{(3,}\mathit{nil}\texttt{())))|||}\mathit{inP}\texttt{(!}x\texttt{)}\\ \quad \xrightarrow{\mathit{inP}\texttt{(\$1=>1)}} \mathbf{exit}\texttt{() |[]| }\mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{cons}\texttt{(2,}\mathit{cons}\texttt{(3,}\mathit{nil}\texttt{()))|||}\mathit{inP}\texttt{(!}x\texttt{)}\end{array}}\ (8.53)$$

where, in order to proof second premise, we have to use rule 8.25.

Second transition is possible due to the following transition:

$$\dfrac{\mathcal{E} \vdash (\texttt{?}x \ \ \rightarrow^{\mathbf{PM}}\ \ 2) \ \mapsto\ \langle\!| x \texttt{=>} 2 |\!\rangle \qquad \dfrac{\dfrac{\mathcal{E} \vdash (\texttt{?}x \ \ \rightarrow^{\mathbf{PM}}\ \ 3) \ \mapsto\ \langle\!| x \texttt{=>} 3 |\!\rangle \qquad \dfrac{\mathcal{E} \vdash \mathit{inP}\texttt{(!3)} \xrightarrow{\mathit{inP}\texttt{(\$1=>3)}} \mathbf{exit}\texttt{()}}{\begin{array}{l}\mathcal{E} \vdash \mathit{inP}\texttt{(!3) |[]| }\mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{nil}\texttt{()|||}\mathit{inP}\texttt{(!}x\texttt{)}\\ \quad \xrightarrow{\mathit{inP}\texttt{(\$1=>3)}} \mathbf{exit}\texttt{()|[]| }\mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{nil}\texttt{()|||}\mathit{inP}\texttt{(!}x\texttt{)}\end{array}}\ (8.25)}{\begin{array}{l}\mathcal{E} \vdash \mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{cons}\texttt{(3,}\mathit{nil}\texttt{())|||}\mathit{inP}\texttt{(!}x\texttt{)}\\ \quad \xrightarrow{\mathit{inP}\texttt{(\$1=>3)}} \mathbf{exit}\texttt{() |[]| }\mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{nil}\texttt{()|||}\mathit{inP}\texttt{(!}x\texttt{)}\end{array}}\ (8.53)}{\begin{array}{l}\mathcal{E} \vdash \mathit{inP}\texttt{(!2) |[]| }\mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{cons}\texttt{(3,}\mathit{nil}\texttt{())|||}\mathit{inP}\texttt{(!}x\texttt{)}\\ \quad \xrightarrow{\mathit{inP}\texttt{(\$1=>3)}} \mathit{inP}\texttt{(!2) |[]| }\mathbf{exit}\texttt{() |[]| }\mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{nil}\texttt{()|||}\mathit{inP}\texttt{(!}x\texttt{)}\end{array}}\ (8.26)}{\dfrac{\begin{array}{l}\mathcal{E} \vdash \mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{cons}\texttt{(2,}\mathit{cons}\texttt{(3,}\mathit{nil}\texttt{()))|||}\mathit{inP}\texttt{(!}x\texttt{)}\\ \quad \xrightarrow{\mathit{inP}\texttt{(\$1=>3)}} \mathit{inP}\texttt{(!2) |[]| }\mathbf{exit}\texttt{() |[]| }\mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{nil}\texttt{()|||}\mathit{inP}\texttt{(!}x\texttt{)}\end{array}}{\begin{array}{l}\mathcal{E} \vdash \mathbf{exit}\texttt{() |[]| }\mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{cons}\texttt{(2,}\mathit{cons}\texttt{(3,}\mathit{nil}\texttt{()))|||}\mathit{inP}\texttt{(!}x\texttt{)}\\ \quad \xrightarrow{\mathit{inP}\texttt{(\$1=>3)}} \mathbf{exit}\texttt{() |[]| }\mathit{inP}\texttt{(!2) |[]| }\mathbf{exit}\texttt{() |[]| }\mathbf{par}\ \texttt{?}x\ \mathbf{in}\ \mathit{nil}\texttt{()|||}\mathit{inP}\texttt{(!}x\texttt{)}\end{array}}\ (8.26)}\ (8.53)$$

The third transition is easily proved by using rules 8.26 and 8.25. The fourth transition is proved by using rules 8.23, 8.32, and 8.54.

8.13 Disabling operator

The behaviour B_1 `[>` B_2 offers a communication on a gate if B_1 offers it:

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{G\texttt{(}RN\texttt{)}} B_1'}{\mathcal{E} \vdash B_1 \;\texttt{[>}\; B_2 \xrightarrow{G\texttt{(}RN\texttt{)}} B_1' \;\texttt{[>}\; B_2} \tag{8.56}$$

Note that the disabling operator continues in the resulting behaviour. The same occurs with internal actions made by B_1:

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\mathbf{i}} B_1'}{\mathcal{E} \vdash B_1 \;\texttt{[>}\; B_2 \xrightarrow{\mathbf{i}} B_1' \;\texttt{[>}\; B_2} \tag{8.57}$$

If B_1 finishes and produces bindings RN, then the behaviour B_1 `[>` B_2 performs an internal action, and these bindings are noted in the behaviour **exit** `(`RN`)`, which will propagate them.

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta\texttt{(}RN\texttt{)}} B_1'}{\mathcal{E} \vdash B_1 \;\texttt{[>}\; B_2 \xrightarrow{\mathbf{i}} \mathbf{exit}\texttt{(}RN\texttt{)}} \tag{8.58}$$

In order to preserve the urgency of internal actions, when B_1 raises an exception, an internal action is performed before the exception is raised again.

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{X\texttt{(}RN\texttt{)}} B_1'}{\mathcal{E} \vdash B_1 \;\texttt{[>}\; B_2 \xrightarrow{\mathbf{i}} \mathbf{signal}\ X\texttt{(}\mathbf{exit}\texttt{(}RN\texttt{))}\,\texttt{;}\texttt{(}B_1' \;\texttt{[>}\; B_2\texttt{)}} \tag{8.59}$$

If B_2 offers a communication (or performs an internal action), then B_1 `[>` B_2 offers the same communication (or performs an internal action), and the disabling operator disappears.

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{G\texttt{(}RN\texttt{)}} B_2'}{\mathcal{E} \vdash B_1 \;\texttt{[>}\; B_2 \xrightarrow{G\texttt{(}RN\texttt{)}} B_2'} \tag{8.60}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{\mathbf{i}} B_2'}{\mathcal{E} \vdash B_1 \;\texttt{[>}\; B_2 \xrightarrow{\mathbf{i}} B_2'} \tag{8.61}$$

When B_2 raises an exception, an internal action is performed and the exception will be raised again.

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{X\texttt{(}RN\texttt{)}} B_2'}{\mathcal{E} \vdash B_1 \;\texttt{[>}\; B_2 \xrightarrow{\mathbf{i}} \mathbf{signal}\ X\texttt{(}\mathbf{exit}\texttt{(}RN\texttt{))}\,\texttt{;}B_2'} \tag{8.62}$$

Note that there is no rule dealing with the termination of B_2 because B_2 is a guarded behaviour.

B_1 `[>` B_2 can idle a period of time d if B_1 and B_2 do.

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\epsilon\langle d\rangle} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\epsilon\langle d\rangle} B_2'}{\mathcal{E} \vdash B_1 \;\texttt{[>}\; B_2 \xrightarrow{\epsilon\langle d\rangle} B_1' \;\texttt{[>}\; B_2'} \tag{8.63}$$

8.14 Suspend/Resume operator

The suspend/resume operator is similar to the previous one when the first behaviour, B_1, performs an action or raises an exception.

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{G\texttt{(}RN\texttt{)}} B_1'}{\mathcal{E} \vdash B_1 \texttt{ [}X\texttt{> } B_2 \xrightarrow{G\texttt{(}RN\texttt{)}} B_1' \texttt{ [}X\texttt{> } B_2} \tag{8.64}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\mathbf{i}} B_1'}{\mathcal{E} \vdash B_1 \texttt{ [}X\texttt{> } B_2 \xrightarrow{\mathbf{i}} B_1' \texttt{ [}X\texttt{> } B_2} \tag{8.65}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta\texttt{(}RN\texttt{)}} B_1'}{\mathcal{E} \vdash B_1 \texttt{ [}X\texttt{> } B_2 \xrightarrow{\mathbf{i}} \mathbf{exit}\texttt{(}RN\texttt{)}} \tag{8.66}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{X'\texttt{(}RN\texttt{)}} B_1'}{\mathcal{E} \vdash B_1 \texttt{ [}X\texttt{> } B_2 \xrightarrow{\mathbf{i}} \mathbf{signal}\ X'\ \texttt{(}RN\texttt{);(}B_1' \texttt{ [}X\texttt{> } B_2\texttt{)}} \tag{8.67}$$

Things are different when the second (disabling) behaviour, B_2, performs an action or raises an exception. In these cases we have to remember the disabled behaviour B_1 because when B_2 raises the exception X, B_1 has to be resumed. This is done by means of a **trap** behaviour that catches exception X, and handles it with B_1 `[`X`>` B_2.

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{G\texttt{(}RN\texttt{)}} B_2'}{\mathcal{E} \vdash B_1 \texttt{ [}X\texttt{> } B_2 \xrightarrow{G\texttt{(}RN\texttt{)}} \begin{array}{l}\mathbf{trap\ exception}\ X\texttt{(\$1 => ()):()}\ \mathbf{is}\ B_1 \texttt{ [}X\texttt{> } B_2 \\ \mathbf{in}\ B_2'\end{array}} \tag{8.68}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{\mathbf{i}} B_2'}{\mathcal{E} \vdash B_1 \texttt{ [}X\texttt{> } B_2 \xrightarrow{\mathbf{i}} \begin{array}{l}\mathbf{trap\ exception}\ X\texttt{(\$1 => ()):()}\ \mathbf{is}\ B_1 \texttt{ [}X\texttt{> } B_2 \\ \mathbf{in}\ B_2'\end{array}} \tag{8.69}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{X'\texttt{(\$1=>}N\texttt{)}} B_2'}{\mathcal{E} \vdash B_1 \texttt{ [}X\texttt{> } B_2 \xrightarrow{\mathbf{i}} \begin{array}{l}\mathbf{signal}\ X'\ \texttt{(\$1 => }N\texttt{);} \\ \mathbf{trap\ exception}\ X\texttt{(\$1 => ()):()}\ \mathbf{is}\ B_1 \texttt{ [}X\texttt{> } B_2 \\ \mathbf{in}\ B_2'\end{array}}\ [X' \neq X] \tag{8.70}$$

Regarding time, the suspend/resume operator behaves like the disabling operator: the behavior B_1 `[`X`>` B_2 can idle time d if both behaviours can idle that time.

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\epsilon\langle d\rangle} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\epsilon\langle d\rangle} B_2'}{\mathcal{E} \vdash B_1 \texttt{ [}X\texttt{> } B_2 \xrightarrow{\epsilon\langle d\rangle} B_1' \texttt{ [}X\texttt{> } B_2'} \tag{8.71}$$

8.15 Hiding operator

The hiding operator transforms actions on hidden gates in internal actions. Thus, if the behaviour B performs an action not in $\{G_1, \ldots, G_n\}$ (including the internal action), raises an exception, or finishes, then **hide** $G_1\texttt{:}T_1\texttt{,} \ldots \texttt{,}G_n\texttt{:}T_n$ **in** B behaves in the same way.

$$\frac{\mathcal{E} \vdash B \xrightarrow{\mu\texttt{(}RN\texttt{)}} B'}{\mathcal{E} \vdash \textbf{hide } G_1 \texttt{:} T_1 \texttt{,} \dots \texttt{,} G_n \texttt{:} T_n \textbf{ in } B \xrightarrow{\mu\texttt{(}RN\texttt{)}} \textbf{hide } G_1 \texttt{:} T_1 \texttt{,} \dots \texttt{,} G_n \texttt{:} T_n \textbf{ in } B'} \quad [\mu \notin \{G_1, \dots, G_n\}] \tag{8.72}$$

But if B performs a communication on a hidden gate, then the whole behaviour performs an internal action.

$$\frac{\begin{array}{l} \mathcal{E} \vdash \texttt{(}\$1 \texttt{=>} N\texttt{)} \ \to^{\mathrm{T}} \ \texttt{(}\$1 \texttt{=>} T_i\texttt{)} \\ \mathcal{E} \vdash B \xrightarrow{G_i \texttt{(}\$1\texttt{=>}N\texttt{)}} B' \end{array}}{\mathcal{E} \vdash \textbf{hide } G_1 \texttt{:} T_1 \texttt{,} \dots \texttt{,} G_n \texttt{:} T_n \textbf{ in } B \xrightarrow{\mathbf{i}} \textbf{hide } G_1 \texttt{:} T_1 \texttt{,} \dots \texttt{,} G_n \texttt{:} T_n \textbf{ in } B'} \tag{8.73}$$

Note how in this rule the value N is *invented* in the sense that it appears in the premises but not in the conclusion. Although the value N is invented, it is checked that it has the correct type, so it is not needed to be checked when the second premise is proved (see Section 8.2).

Since the hiding operator transforms hidden actions in internal actions, and these ones are urgent, the behaviour $\textbf{hide } G_1 \texttt{:} T_1 \texttt{,} \dots \texttt{,} G_n \texttt{:} T_n \textbf{ in } B$ can idle a period of time d only if B cannot perform an action on a hidden gate before d. This is expressed in the following rule by means of the predicate **refusing**, which is defined in the following way:

$$\mathcal{E} \vdash B \textbf{ refusing } (G_1 \texttt{:} T_1 \texttt{,} \dots \texttt{,} G_n \texttt{:} T_n, d)$$

if there is no N and $d' < d$ such that $\mathcal{E} \vdash \texttt{(}\$1 \texttt{=>} N\texttt{)} \ \to^{\mathrm{T}} \ \texttt{(}\$1 \texttt{=>} T_i\texttt{)}$ and $\mathcal{E} \vdash B \xrightarrow{G_i \texttt{(}RN\texttt{)@}d'} B'$.

$$\frac{\begin{array}{l} \mathcal{E} \vdash B \xrightarrow{\epsilon\langle d \rangle} B' \\ \mathcal{E} \vdash B \textbf{ refusing } (G_1 \texttt{:} T_1 \texttt{,} \dots \texttt{,} G_n \texttt{:} T_n, d) \end{array}}{\mathcal{E} \vdash \textbf{hide } G_1 \texttt{:} T_1 \texttt{,} \dots \texttt{,} G_n \texttt{:} T_n \textbf{ in } B \xrightarrow{\epsilon\langle d \rangle} \textbf{hide } G_1 \texttt{:} T_1 \texttt{,} \dots \texttt{,} G_n \texttt{:} T_n \textbf{ in } B'} \tag{8.74}$$

Note that we use a negative premise to define the timed dynamic semantics. This represents no problem because the behaviour in the premise is simpler that the behaviour in the conclusion of the inference rule, so a *stratification* [Gro93] may be defined in order to prove that the semantics is well defined.

Example 8.11 Let us see which transitions the behaviour

hide *sync* **in**
  pd(!1); *sync* |[*sync*]| *sync*; *pd*(!2)
endhide

can perform. It is translated into abstract syntax as

hidding ≡
  hide *sync*:**any** **in**
    pd($1 => !1) . . . ; *sync*($1 => ()) . . . |[*sync*]| *sync*($1 => ()) . . . ; *pd*($1 => !2)

First, it can idle (because *pd* is not urgent in this context and it does not belong to the list of gates of the parallel operator) or it can perform a communication on gate *pd*. Thus, the first possible transitions are

$$\begin{array}{ll} \mathit{hidding} \xrightarrow{\epsilon\langle d\rangle} & \textbf{hide}\ \mathit{sync}\texttt{:}\textbf{any in} \\ & \quad \mathit{pd}\texttt{(\$1 => !1)}\ldots\textbf{start}\langle d\rangle\texttt{;}\ \mathit{sync}\texttt{(\$1 => ())}\ldots \\ & \texttt{|[}\ \mathit{sync}\ \texttt{]|} \\ & \quad \mathit{sync}\texttt{(\$1 => ())}\ldots\textbf{start}\langle d\rangle\texttt{;}\ \mathit{pd}\texttt{(\$1 => !2)}\ldots \end{array}$$

or

$$\begin{array}{ll} \mathit{hidding} \xrightarrow{\mathit{pd}\texttt{(\$1=>1)}} & \textbf{hide}\ \mathit{sync}\texttt{:}\textbf{any in} \\ & \quad \textbf{exit}\texttt{()}\texttt{;}\ \mathit{sync}\texttt{(\$1 => ())}\ldots\ \texttt{|[}\ \mathit{sync}\ \texttt{]|}\ \mathit{sync}\texttt{(\$1 => ())}\ldots\texttt{;}\ \mathit{pd}\texttt{(\$1 => !2)}\ldots \end{array}$$

In this latter case, no time transition is possible, because the negative premise in rule 8.74 is not fulfilled, the body of the **hide** can perform a communication on a hidden gate, so the **hide** behaviour performs an internal action.

$$\begin{array}{l} \textbf{hide}\ \mathit{sync}\texttt{:}\textbf{any in exit}\texttt{()}\texttt{;}\ \mathit{sync}\texttt{(\$1 => ())}\ldots\ \texttt{|[}\ \mathit{sync}\ \texttt{]|}\ \mathit{sync}\texttt{(\$1 => ())}\ldots\texttt{;}\ \mathit{pd}\texttt{(\$1 => !2)} \\ \quad \xrightarrow{\texttt{i}} \textbf{hide}\ \mathit{sync}\texttt{:}\textbf{any in exit}\texttt{()}\texttt{;}\ \textbf{exit}\texttt{()}\ \texttt{|[}\ \mathit{sync}\ \texttt{]|}\ \textbf{exit}\texttt{()}\texttt{;}\ \mathit{pd}\texttt{(\$1 => !2)} \end{array}$$

Now, a time transition is possible

$$\begin{array}{l} \textbf{hide}\ \mathit{sync}\texttt{:}\textbf{any in exit}\texttt{()}\texttt{;}\ \textbf{exit}\texttt{()}\ \texttt{|[}\ \mathit{sync}\ \texttt{]|}\ \textbf{exit}\texttt{()}\texttt{;}\ \mathit{pd}\texttt{(\$1 => !2)}\ldots \\ \quad \xrightarrow{\epsilon\langle d'\rangle} \textbf{hide}\ \mathit{sync}\texttt{:}\textbf{any in exit}\texttt{()}\texttt{;}\ \textbf{exit}\texttt{()}\ \texttt{|[}\ \mathit{sync}\ \texttt{]|}\ \textbf{exit}\texttt{()}\texttt{;}\ \mathit{pd}\texttt{(\$1 => !2)}\ldots\textbf{start}\langle d'\rangle \end{array}$$

and also a communication on gate pd.

$$\begin{array}{l} \textbf{hide}\ \mathit{sync}\texttt{:}\textbf{any in exit}\texttt{()}\texttt{;}\ \textbf{exit}\texttt{()}\ \texttt{|[}\ \mathit{sync}\ \texttt{]|}\ \textbf{exit}\texttt{()}\texttt{;}\ \mathit{pd}\texttt{(\$1 => !2)}\ldots \\ \quad \xrightarrow{\mathit{pd}\texttt{(\$1=>2)}} \textbf{hide}\ \mathit{sync}\texttt{:}\textbf{any in exit}\texttt{()}\texttt{;}\ \textbf{exit}\texttt{()}\ \texttt{|[}\ \mathit{sync}\ \texttt{]|}\ \textbf{exit}\texttt{()}\texttt{;}\ \textbf{exit}\texttt{()} \end{array}$$

Finally, in this latter case, the behaviour can only finish.

8.16 Exception handling

A **trap** behaviour has the transitions of its body if they are not related with trapped exceptions.

$$\frac{\mathcal{E} \vdash B \xrightarrow{\mu(RN)} B'}{\begin{array}{lll} \mathcal{E} \vdash \textbf{trap} & \xrightarrow{\mu(RN)} & \textbf{trap} \\ \quad \textbf{exception}\ X_1\texttt{(\$1 => }P_1\texttt{)}\texttt{:}T_1\ \textbf{is}\ B_1 & & \quad \textbf{exception}\ X_1\texttt{(\$1 => }P_1\texttt{)}\texttt{:}T_1\ \textbf{is}\ B_1 \\ \quad\quad \ldots & & \quad\quad \ldots \\ \quad \textbf{exception}\ X_n\texttt{(\$1 => }P_n\texttt{)}\texttt{:}T_n\ \textbf{is}\ B_n & & \quad \textbf{exception}\ X_n\texttt{(\$1 => }P_n\texttt{)}\texttt{:}T_n\ \textbf{is}\ B_n \\ \quad \textbf{exit}\ P_{n+1}\ \textbf{is}\ B_{n+1} & & \quad \textbf{exit}\ P_{n+1}\ \textbf{is}\ B_{n+1} \\ \textbf{in}\ B & & \textbf{in}\ B' \end{array}} \tag{8.75}$$

with the side condition $\mu \notin \{X_1, \ldots, X_n\}$.

However, if the body B raises a trapped exception X_i together with value N, this value can be matched against the corresponding pattern P_i and has type T_i, and the handler of the exception X_i, B_i, has the transition $\xrightarrow{\mu(RN')}$ becoming B'_i, then the **trap** behaviour has this transition becoming also B'_i.

$$\frac{\begin{array}{l}\mathcal{E} \vdash B \xrightarrow{X_i\,\texttt{(}\$1\texttt{=>}N\texttt{)}} B' \\ \mathcal{E} \vdash \texttt{((}\$1 \texttt{=>} P_i \texttt{:} T_i\texttt{)} \ \rightharpoonup^{\mathbf{PM}} \ \texttt{(}\$1 \texttt{=>} N\texttt{))} \ \mapsto \ \langle\!| \, RN \, |\!\rangle \\ \mathcal{E} \vdash B_i \, [\![\, RN \,]\!] \xrightarrow{\mu\texttt{(}RN'\texttt{)}} B_i'\end{array}}{\begin{array}{l} \mathcal{E} \vdash \textbf{trap} \\ \qquad \textbf{exception } X_1\texttt{(}\$1 \texttt{=>} P_1\texttt{)}\texttt{:}T_1 \textbf{ is } B_1 \\ \qquad \quad \ldots \\ \qquad \textbf{exception } X_n\texttt{(}\$1 \texttt{=>} P_n\texttt{)}\texttt{:}T_n \textbf{ is } B_n \\ \qquad \textbf{exit } P_{n+1} \textbf{ is } B_{n+1} \\ \quad \textbf{in } B \qquad\qquad\qquad\qquad \xrightarrow{\mu\texttt{(}RN'\texttt{)}} B_i' \end{array}} \ [X_i \in \{X_1, \ldots, X_n\}] \tag{8.76}$$

Finally, if behaviour B finishes returning bindings RN_1, RN_2, pattern P_{n+1} can be matched against part of the binding returned, RN_1, and the handler B_{n+1} has the transition $\xrightarrow{\mu\texttt{(}RN''\texttt{)}}$ becoming B'_{n+1}, then the **trap** behaviour has this transition, becoming $\textbf{exit}\texttt{(}RN_2\texttt{)}\,\texttt{;}B'_{n+1}$. Note how $\textbf{exit}\texttt{(}RN_2\texttt{)}$ is used to propagate those bindings produced by B not trapped by P_{n+1}.

$$\frac{\begin{array}{l}\mathcal{E} \vdash B \xrightarrow{\delta\texttt{(}RN_1\texttt{,}RN_2\texttt{)}} B' \\ \mathcal{E} \vdash \texttt{(}P_{n+1} \ \rightharpoonup^{\mathbf{PM}} \ \texttt{(}RN_1\texttt{))} \ \mapsto \ \langle\!| \, RN' \, |\!\rangle \\ \mathcal{E} \vdash B_{n+1} \, [\![\, RN' \,]\!] \xrightarrow{\mu\texttt{(}RN''\texttt{)}} B'_{n+1}\end{array}}{\begin{array}{l} \mathcal{E} \vdash \textbf{trap} \\ \qquad \textbf{exception } X_1\texttt{(}\$1 \texttt{=>} P_1\texttt{)}\texttt{:}T_1 \textbf{ is } B_1 \\ \qquad \quad \ldots \\ \qquad \textbf{exception } X_n\texttt{(}\$1 \texttt{=>} P_n\texttt{)}\texttt{:}T_n \textbf{ is } B_n \\ \qquad \textbf{exit } P_{n+1} \textbf{ is } B_{n+1} \\ \quad \textbf{in } B \qquad\qquad\qquad\qquad \xrightarrow{\mu\texttt{(}RN''\texttt{)}} \textbf{exit}\texttt{(}RN_2\texttt{)}\,\texttt{;}B'_{n+1} \end{array}} \tag{8.77}$$

Regarding time, a **trap** behaviour has the same time transitions than its main behaviour.

$$\frac{\mathcal{E} \vdash B \xrightarrow{\epsilon\langle d\rangle} B'}{\begin{array}{l} \mathcal{E} \vdash \textbf{trap} \\ \qquad \textbf{exception } X_1\texttt{(}\$1 \texttt{=>} P_1\texttt{)}\texttt{:}T_1 \textbf{ is } B_1 \\ \qquad \quad \ldots \\ \qquad \textbf{exception } X_n\texttt{(}\$1 \texttt{=>} P_n\texttt{)}\texttt{:}T_n \textbf{ is } B_n \\ \qquad \textbf{exit } P_{n+1} \textbf{ is } B_{n+1} \\ \quad \textbf{in } B \end{array} \xrightarrow{\epsilon\langle d\rangle} \begin{array}{l} \textbf{trap} \\ \qquad \textbf{exception } X_1\texttt{(}\$1 \texttt{=>} P_1\texttt{)}\texttt{:}T_1 \textbf{ is } B_1 \\ \qquad \quad \ldots \\ \qquad \textbf{exception } X_n\texttt{(}\$1 \texttt{=>} P_n\texttt{)}\texttt{:}T_n \textbf{ is } B_n \\ \qquad \textbf{exit } P_{n+1} \textbf{ is } B_{n+1} \\ \textbf{in } B' \end{array}} \tag{8.78}$$

Example 8.12 Let us check now that the behaviour

```
trap
  exception X is wait(1) endexn
in ( i [] signal X )
endtrap
```

cannot do time transitions, that is, that the internal action is urgent. It is translated to

$$\begin{array}{l} \textit{exnHandling} \quad \equiv \\ \qquad \textbf{trap exception } X\texttt{(}\$1 \texttt{=> ())}\texttt{:()} \textbf{ is wait}\texttt{(}\textbf{exit}\texttt{(}\$1 \texttt{=>} 1\texttt{))} \\ \qquad \textbf{in i } \texttt{[]} \textbf{ signal } X\texttt{(}\$1 \texttt{=> ())} \end{array}$$

If we try to use rule 8.78 we cannot because the selection behaviour has no time transitions, since internal action and exception signaling are urgent. Another possibility is to try rule 8.76 in order to reach the **wait** behaviour, but we cannot because the selection cannot raise the exception directly. Therefore, the only possible transitions are to perform an internal action.

$$\begin{array}{ll} \mathit{exnHandling} \xrightarrow{\mathbf{i}} & \mathbf{trap\ exception}\ X\texttt{(\$1 => ()):()}\ \mathbf{is\ wait}\texttt{(}\mathbf{exit}\texttt{(\$1 => 1))} \\ & \mathbf{in\ exit}\texttt{()} \end{array}$$

(using rules 8.17 and 8.75), or

$$\begin{array}{ll} \mathit{exnHandling} \xrightarrow{\mathbf{i}} & \mathbf{trap\ exception}\ X\texttt{(\$1 => ()):()}\ \mathbf{is\ wait}\texttt{(}\mathbf{exit}\texttt{(\$1 => 1))} \\ & \mathbf{in\ signal}\texttt{(}\mathbf{exit}\texttt{(\$1 => ()));}\mathbf{exit}\texttt{()} \end{array}$$

(using rules 8.20 and 8.75).

8.17 Exception signaling

The behaviour **signal** X(*E*) raises exception X together with bindings RN if expression E finishes producing these bindings.

$$\frac{\mathcal{E} \vdash E \xrightarrow{\delta\texttt{(\$1=>}N\texttt{)}} E'}{\mathcal{E} \vdash \mathbf{signal}\ X\texttt{(}E\texttt{)} \xrightarrow{X\texttt{(\$1=>}N\texttt{)}} \mathbf{exit}\texttt{()}} \tag{8.79}$$

If the evaluation of expression E raises an exception X', then the behaviour **signal** X(*E*) raises this exception.

$$\frac{\mathcal{E} \vdash E \xrightarrow{X'\texttt{(}RN\texttt{)}} E'}{\mathcal{E} \vdash \mathbf{signal}\ X\texttt{(}E\texttt{)} \xrightarrow{X'\texttt{(}RN\texttt{)}} \mathbf{signal}\ X\texttt{(}E'\texttt{)}} \tag{8.80}$$

Exception signaling is urgent, that is, exceptions are raised as soon as possible. Thus, the behaviour **signal** X(*E*) cannot idle any time.

8.18 Delay instruction

The behaviour **wait**(*E*) finishes when expression E evaluates to 0 (returns 0), and informs of its termination with a communication on gate δ.

$$\frac{\mathcal{E} \vdash E \xrightarrow{\delta\texttt{(\$1=>0)}} E'}{\mathcal{E} \vdash \mathbf{wait}\texttt{(}E\texttt{)} \xrightarrow{\delta\texttt{()}} \mathbf{block}} \tag{8.81}$$

If the evaluation of **wait**(*E*) raises an exception, then **wait**(*E*) raises the same exception.

$$\frac{\mathcal{E} \vdash E \xrightarrow{X\texttt{(}RN\texttt{)}} E'}{\mathcal{E} \vdash \mathbf{wait}\texttt{(}E\texttt{)} \xrightarrow{X\texttt{(}RN\texttt{)}} \mathbf{wait}\ \texttt{(}E'\texttt{)}} \tag{8.82}$$

The behaviour **wait**(*E*) can idle any time $d > 0$ less than or equal to the value $d + d'$ to which E evaluates, and becomes a behaviour that can idle the remaining time, **wait**(d').

$$\frac{\mathcal{E} \vdash E \xrightarrow{\delta(\$1\texttt{=>}d+d')} E'}{\mathcal{E} \vdash \textbf{wait}(E) \xrightarrow{\epsilon\langle d\rangle} \textbf{wait}(\textbf{exit}(\$1 \texttt{=>} d'))}\ [0 < d] \tag{8.83}$$

We have to write $\textbf{wait}(\textbf{exit}(\$1 \texttt{=>} d'))$ instead of $\textbf{wait}(d')$ because $\textbf{exit}(\$1 \texttt{=>} d')$ is an abstract syntax expression but d' is not.

8.19 Renaming operator

A renaming behaviour has the transition $\xrightarrow{\mu(RN)}$ if the renamed behaviour B has the same transition and μ is not a renamed gate or exception.

$$\frac{\mathcal{E} \vdash B \xrightarrow{\mu(RN)} B'}{\begin{array}{lll}
\mathcal{E} \vdash \textbf{rename} & \xrightarrow{\mu(RN)} & \textbf{rename} \\
\quad \textbf{gate}\ G_1(\$1 \texttt{=>} P_1)\texttt{:}T_1\ \textbf{is}\ G'_1(\$1 \texttt{=>} P'_1) & & \quad \textbf{gate}\ G_1(\$1 \texttt{=>} P_1)\texttt{:}T_1\ \textbf{is}\ G'_1(\$1 \texttt{=>} P'_1) \\
\quad\quad \ldots & & \quad\quad \ldots \\
\quad \textbf{gate}\ G_m(\$1 \texttt{=>} P_m)\texttt{:}T_m\ \textbf{is}\ G'_m(\$1 \texttt{=>} P'_m) & & \quad \textbf{gate}\ G_m(\$1 \texttt{=>} P_m)\texttt{:}T_m\ \textbf{is}\ G'_m(\$1 \texttt{=>} P'_m) \\
\quad \textbf{signal}\ X_1(\$1 \texttt{=>} P''_1)\texttt{:}T'_1\ \textbf{is}\ X'_1(E_1) & & \quad \textbf{signal}\ X_1(\$1 \texttt{=>} P''_1)\texttt{:}T'_1\ \textbf{is}\ X'_1(E_1) \\
\quad\quad \ldots & & \quad\quad \ldots \\
\quad \textbf{signal}\ X_n(\$1 \texttt{=>} P''_n)\texttt{:}T'_n\ \textbf{is}\ X'_n(E_n) & & \quad \textbf{signal}\ X_n(\$1 \texttt{=>} P''_n)\texttt{:}T'_n\ \textbf{is}\ X'_n(E_n) \\
\textbf{in}\ B & & \textbf{in}\ B'
\end{array}} \tag{8.84}$$

with the side condition $\mu \notin \{G_1, \ldots, G_m, X_1, \ldots, X_m\}$.

If B can perform an action on a renamed gate G_i, the value communicated through gate G_i has the correct type and can be pattern-matched against the pattern defining the capture of G_i (note that both conditions are checked by the first premise, see Section 8.31.6), and the action performed instead of G_i has the transition $\xrightarrow{G'_i(RN')}$, then the renaming behaviour has the same transition $\xrightarrow{G'_i(RN')}$.

$$\frac{\begin{array}{l}
\mathcal{E} \vdash ((\$1 \texttt{=>} P_i\texttt{:}T_i) \ \rightarrow^{\mathrm{PM}}\ (\$1 \texttt{=>} N)) \ \mapsto\ \langle\!|\, RN \,|\!\rangle \\
\mathcal{E} \vdash B \xrightarrow{G_i(\$1\texttt{=>}N)} B' \\
\mathcal{E} \vdash (G'_i(\$1 \texttt{=>} P'_1)\ \texttt{@}\textbf{any}\ \texttt{:}\ \mathsf{time}\ \textbf{exit}(\$1 \texttt{=>} \mathit{true})\ \textbf{start}\langle 0\rangle)\, [\![\, RN \,]\!] \xrightarrow{G'_i(RN')} B'_i
\end{array}}{\begin{array}{lll}
\mathcal{E} \vdash \textbf{rename} & \xrightarrow{G'_i(RN')} & \textbf{rename} \\
\quad \textbf{gate}\ G_1(\$1 \texttt{=>} P_1)\texttt{:}T_1\ \textbf{is}\ G'_1(\$1 \texttt{=>} P'_1) & & \quad \textbf{gate}\ G_1(\$1 \texttt{=>} P_1)\texttt{:}T_1\ \textbf{is}\ G'_1(\$1 \texttt{=>} P'_1) \\
\quad\quad \ldots & & \quad\quad \ldots \\
\quad \textbf{gate}\ G_m(\$1 \texttt{=>} P_m)\texttt{:}T_m\ \textbf{is}\ G'_m(\$1 \texttt{=>} P'_m) & & \quad \textbf{gate}\ G_m(\$1 \texttt{=>} P_m)\texttt{:}T_m\ \textbf{is}\ G'_m(\$1 \texttt{=>} P'_m) \\
\quad \textbf{signal}\ X_1(\$1 \texttt{=>} P''_1)\texttt{:}T'_1\ \textbf{is}\ X'_1(E_1) & & \quad \textbf{signal}\ X_1(\$1 \texttt{=>} P''_1)\texttt{:}T'_1\ \textbf{is}\ X'_1(E_1) \\
\quad\quad \ldots & & \quad\quad \ldots \\
\quad \textbf{signal}\ X_n(\$1 \texttt{=>} P''_n)\texttt{:}T'_n\ \textbf{is}\ X'_n(E_n) & & \quad \textbf{signal}\ X_n(\$1 \texttt{=>} P''_n)\texttt{:}T'_n\ \textbf{is}\ X'_n(E_n) \\
\textbf{in}\ B & & \textbf{in}\ B'
\end{array}} \tag{8.85}$$

Finally, if B can raise a renamed exception X_i, the value raised together with the exception can be pattern-matched against the pattern defining the capture of X_i, and the behaviour which raises the exception renaming X'_i has the transition $\xrightarrow{X'_i(RN')}$, then the renaming behaviour has the same transition $\xrightarrow{X'_i(RN')}$.

$$\frac{\begin{array}{l}\mathcal{E} \vdash B \xrightarrow{X_i\,(\$1 \texttt{=>} N)} B' \\ \mathcal{E} \vdash ((\$1 \texttt{=>} P''_i) \ \rightarrow^{\mathbf{PM}} \ (\$1 \texttt{=>} N)) \ \mapsto \ \langle\!|\, RN \,|\!\rangle \\ \mathcal{E} \vdash (\mathbf{signal}\ X'_i\,(E_i))\,[\![\, RN \,]\!] \xrightarrow{X'_i\,(RN')} B'_i\end{array}}{\begin{array}{l}\mathcal{E} \vdash \mathbf{rename} \\ \quad \mathbf{gate}\ G_1\,(\$1 \texttt{=>} P_1)\,\texttt{:}T_1\ \mathbf{is}\ G'_1\,(\$1 \texttt{=>} P'_1) \\ \qquad \ldots \\ \quad \mathbf{gate}\ G_m\,(\$1 \texttt{=>} P_m)\,\texttt{:}T_m\ \mathbf{is}\ G'_m\,(\$1 \texttt{=>} P'_m) \\ \quad \mathbf{signal}\ X_1\,(\$1 \texttt{=>} P''_1)\,\texttt{:}T'_1\ \mathbf{is}\ X'_1\,(E_1) \\ \qquad \ldots \\ \quad \mathbf{signal}\ X_n\,(\$1 \texttt{=>} P''_n)\,\texttt{:}T'_n\ \mathbf{is}\ X'_n\,(E_n) \\ \mathbf{in}\ B\end{array} \xrightarrow{X'_i\,(RN')} \begin{array}{l}\mathbf{rename} \\ \quad \mathbf{gate}\ G_1\,(\$1 \texttt{=>} P_1)\,\texttt{:}T_1\ \mathbf{is}\ G'_1\,(\$1 \texttt{=>} P'_1) \\ \qquad \ldots \\ \quad \mathbf{gate}\ G_m\,(\$1 \texttt{=>} P_m)\,\texttt{:}T_m\ \mathbf{is}\ G'_m\,(\$1 \texttt{=>} P'_m) \\ \quad \mathbf{signal}\ X_1\,(\$1 \texttt{=>} P''_1)\,\texttt{:}T'_1\ \mathbf{is}\ X'_1\,(E_1) \\ \qquad \ldots \\ \quad \mathbf{signal}\ X_n\,(\$1 \texttt{=>} P''_n)\,\texttt{:}T'_n\ \mathbf{is}\ X'_n\,(E_n) \\ \mathbf{in}\ B'\end{array}} \tag{8.86}$$

Regarding time, a renaming behaviour has the same time transitions than the renamed behaviour.

$$\frac{\mathcal{E} \vdash B \xrightarrow{\epsilon\langle d\rangle} B'}{\begin{array}{l}\mathcal{E} \vdash (\mathbf{rename} \\ \quad \mathbf{gate}\ G_1\,(\$1 \texttt{=>} P_1)\,\texttt{:}T_1\ \mathbf{is}\ G'_1\,(\$1 \texttt{=>} P'_1) \\ \qquad \ldots \\ \quad \mathbf{gate}\ G_m\,(\$1 \texttt{=>} P_m)\,\texttt{:}T_m\ \mathbf{is}\ G'_m\,(\$1 \texttt{=>} P'_m) \\ \quad \mathbf{signal}\ X_1\,(\$1 \texttt{=>} P''_1)\,\texttt{:}T'_1\ \mathbf{is}\ X'_1\,(E_1) \\ \qquad \ldots \\ \quad \mathbf{signal}\ X_n\,(\$1 \texttt{=>} P''_n)\,\texttt{:}T'_n\ \mathbf{is}\ X'_n\,(E_n) \\ \mathbf{in}\ B\end{array} \xrightarrow{\epsilon\langle d\rangle} \begin{array}{l}(\mathbf{rename} \\ \quad \mathbf{gate}\ G_1\,(\$1 \texttt{=>} P_1)\,\texttt{:}T_1\ \mathbf{is}\ G'_1\,(\$1 \texttt{=>} P'_1) \\ \qquad \ldots \\ \quad \mathbf{gate}\ G_m\,(\$1 \texttt{=>} P_m)\,\texttt{:}T_m\ \mathbf{is}\ G'_m\,(\$1 \texttt{=>} P'_m) \\ \quad \mathbf{signal}\ X_1\,(\$1 \texttt{=>} P''_1)\,\texttt{:}T'_1\ \mathbf{is}\ X'_1\,(E_1) \\ \qquad \ldots \\ \quad \mathbf{signal}\ X_n\,(\$1 \texttt{=>} P''_n)\,\texttt{:}T'_n\ \mathbf{is}\ X'_n\,(E_n) \\ \mathbf{in}\ B\end{array}} \tag{8.87}$$

Example 8.13 Let us see how the behaviour

```
rename
  gate inP (?a):int is intro (!a)
in
  inP(!3)
endren
```

can perform a communication on gate *intro*.

It is translated to:

```
renaming ≡
    rename
      gate inP ($1 => ?a):int is intro ($1 => !exit($1 => a))
    in
      inP(!exit($1 => 3)) ...
```

We can use rule 8.85 and deduce the following:

$$\frac{\begin{array}{l}\mathcal{E} \vdash (\,(\$1 \mathtt{=>}\ ?a\!:\!\mathsf{int})\ \ \overset{\mathbf{PM}}{\rightarrow}\ \ (\$1 \mathtt{=>} 3)\,)\ \mapsto\ \langle\!|\, a \mathtt{=>} 3 \,|\!\rangle \\ \mathcal{E} \vdash \mathit{inP}(\,!\mathbf{exit}(\$1 \mathtt{=>} 3)\,)\ \ldots \xrightarrow{\mathit{inP}(\$1\mathtt{=>}3)} \mathbf{exit}() \\ \mathcal{E} \vdash \mathit{intro}\ \ (\$1 \mathtt{=>}\ !\mathbf{exit}(\$1 \mathtt{=>} 3))\ \mathtt{@}\mathbf{any} : \mathsf{time}\ \mathbf{exit}(\$1 \mathtt{=>} \mathit{true})\ \mathbf{start}\langle 0\rangle \xrightarrow{\mathit{intro}(\$1\mathtt{=>}3)} \mathbf{exit}()\end{array}}{\mathcal{E} \vdash \mathit{renaming} \xrightarrow{\mathit{intro}(\$1\mathtt{=>}3)} \begin{array}{l}\mathbf{rename\ gate}\ \mathit{inP}\ \ (\$1 \mathtt{=>}\ ?a)\!:\!\mathsf{int}\ \mathbf{is}\ \mathit{intro}\ \ (\$1 \mathtt{=>}\ !\mathbf{exit}(\$1 \mathtt{=>} a)) \\ \mathbf{in\ exit}()\end{array}}$$

8.20 Assignment

The assignment P `:=` E finishes immediately if expression E returns the value N and pattern P can be matched against N by producing bindings RN. These bindings are returned by the assignment.

$$\frac{\begin{array}{l}\mathcal{E} \vdash E \xrightarrow{\delta(\$1\mathtt{=>}N)} E' \\ \mathcal{E} \vdash (P\ \ \overset{\mathbf{PM}}{\rightarrow}\ \ N)\ \mapsto\ \langle\!|\, RN \,|\!\rangle\end{array}}{\mathcal{E} \vdash P\ \mathtt{:=}\ E \xrightarrow{\delta(RN)} \mathbf{block}} \tag{8.88}$$

If expression E raises an exception when it is being evaluated, then the assignment raises this exception.

$$\frac{\mathcal{E} \vdash E \xrightarrow{X\langle RN\rangle} E'}{\mathcal{E} \vdash P\mathtt{:=}E \xrightarrow{X\langle RN\rangle} P\mathtt{:=}E'} \tag{8.89}$$

There are no rules dealing with time transitions, that is, assignments are urgent.

Example 8.14 Let us see which bindings the assignment $(?x,?y)\ \mathtt{:=}\ (5,\mathit{true})$ produces, and how they are propagated through gate δ. It is translated to

$$(\$1 \mathtt{=>}\ ?x, \$2 \mathtt{=>}\ ?y)\ \mathtt{:=}\ \mathbf{exit}(\$1 \mathtt{=>} (\$1 \mathtt{=>} 5, \$2 \mathtt{=>} \mathit{true}))$$

and we can use rule 8.88 to deduce

$$\frac{\begin{array}{l}\mathcal{E} \vdash \mathbf{exit}(\$1 \mathtt{=>} (\$1 \mathtt{=>} 5, \$2 \mathtt{=>} \mathit{true})) \xrightarrow{\delta(\$1\mathtt{=>}(\$1\mathtt{=>}5,\$2\mathtt{=>}\mathit{true}))} \mathbf{block} \\ \mathcal{E} \vdash (\,(\$1 \mathtt{=>}\ ?x, \$2 \mathtt{=>}\ ?y)\ \ \overset{\mathbf{PM}}{\rightarrow}\ \ (\$1 \mathtt{=>} 5, \$2 \mathtt{=>} \mathit{true})\,)\ \mapsto\ \langle\!|\, x \mathtt{=>} 5, y \mathtt{=>} \mathit{true} \,|\!\rangle\end{array}}{\mathcal{E} \vdash (\$1 \mathtt{=>}\ ?x, \$2 \mathtt{=>}\ ?y)\ \mathtt{:=}\ \mathbf{exit}(\$1 \mathtt{=>} (\$1 \mathtt{=>} 5, \$2 \mathtt{=>} \mathit{true})) \xrightarrow{\delta(x\mathtt{=>}5,y\mathtt{=>}\mathit{true})} \mathbf{block}} \tag{8.88}$$

8.21 Nondeterministic assignment

The nondeterministic assignment P `:=` **any** T `[` E `]` performs an internal action if there is a value N of type T that can be pattern-matched against pattern P by producing bindings RN, and the expression E with these bindings substituted, returns *true*. The nondeterministic assignment becomes **exit**`(`RN`)` that will return the produced bindings.

$$\frac{\begin{array}{l}\mathcal{E} \vdash N\ \ \overset{\mathbf{T}}{\rightarrow}\ \ T \\ \mathcal{E} \vdash (P\ \ \overset{\mathbf{PM}}{\rightarrow}\ \ N)\ \mapsto\ \langle\!|\, RN \,|\!\rangle \\ \mathcal{E} \vdash E[\![RN]\!] \xrightarrow{\delta(\$1\mathtt{=>}\mathit{true})} E'\end{array}}{\mathcal{E} \vdash P\ \mathtt{:=}\ \mathbf{any}\ T\ \mathtt{[}\ E\ \mathtt{]} \xrightarrow{\mathbf{i}} \mathbf{exit}(RN)} \tag{8.90}$$

Nondeterministic assignment is also urgent, so there is no rule defining time transitions.

8.22 Variable declaration

The variable declaration **var** $V_1 \mathbin{:} T_1[\texttt{:=}E_1], \ldots, V_n \mathbin{:} T_n[\texttt{:=}E_n]$ **in** B has the transition $\xrightarrow{a(RN')}$ if all the assignments to the initialized variables finish and behaviour B after substituting the bindings produced by the assignments has the same transition.

$$\frac{\begin{array}{l}\forall\, j_i \in \{k \mid V_k \mathbin{:} T_k \texttt{:=} E_k\} \,.\, \mathcal{E} \vdash ?V_{j_i} \texttt{:=} E_{j_i} \xrightarrow{\delta(RN_{j_i})} \textbf{block} \\ \mathcal{E} \vdash B \llbracket\, RN_{j_1}, \ldots, RN_{j_m} \,\rrbracket \xrightarrow{a(RN')} B'\end{array}}{\mathcal{E} \vdash \textbf{var}\ V_1 \mathbin{:} T_1[\texttt{:=}E_1], \ldots, V_n \mathbin{:} T_n[\texttt{:=}E_n]\ \textbf{in}\ B \xrightarrow{a(RN')} \textbf{var}\ V_1 \mathbin{:} T_1, \ldots, V_n \mathbin{:} T_n\ \textbf{in}\ B'} \tag{8.91}$$

with the side condition that $\{j_1, \ldots, j_m\} = \{k \mid V_k \mathbin{:} T_k \texttt{:=} E_k\}$. Note that the expressions assigned to initialized variables are removed from the resulting behaviour **var** $V_1 \mathbin{:} T_1, \ldots, V_n \mathbin{:} T_n$ **in** B', so they are only evaluated once.

If the assignments to the initialized variables finish and the behaviour B after substituting the bindings produced by the assignments finishes producing bindings RN, then the variable declaration finishes producing bindings $RN - \{V_1, \ldots, V_n\}$, that is, all the bindings produced by B minus bindings related to declared variables.

$$\frac{\begin{array}{l}\forall\, j_i \in \{k \mid V_k \mathbin{:} T_k \texttt{:=} E_k\} \,.\, \mathcal{E} \vdash ?V_{j_i} \texttt{:=} E_{j_i} \xrightarrow{\delta(RN_{j_i})} \textbf{block} \\ \mathcal{E} \vdash B \llbracket\, RN_{j_1}, \ldots, RN_{j_m} \,\rrbracket \xrightarrow{\delta(RN')} B'\end{array}}{\mathcal{E} \vdash \textbf{var}\ V_1 \mathbin{:} T_1[\texttt{:=}E_1], \ldots, V_n \mathbin{:} T_n[\texttt{:=}E_n]\ \textbf{in}\ B \xrightarrow{\delta(RN' - \{V_1, \ldots, V_n\})} \textbf{block}} \tag{8.92}$$

with the same side condition as above.

The variable declaration **var** $V_1 \mathbin{:} T_1[\texttt{:=}E_1], \ldots, V_n \mathbin{:} T_n[\texttt{:=}E_n]$ **in** B has the same time transitions than B.

$$\frac{\mathcal{E} \vdash B \xrightarrow{\epsilon\langle d\rangle} B'}{\mathcal{E} \vdash \textbf{var}\ V_1 \mathbin{:} T_1, \ldots, V_n \mathbin{:} T_n\ \textbf{in}\ B \xrightarrow{\epsilon\langle d\rangle} \textbf{var}\ V_1 \mathbin{:} T_1, \ldots, V_n \mathbin{:} T_n\ \textbf{in}\ B'} \tag{8.93}$$

8.23 Case operator

The behaviour **case** $E \mathbin{:} T$ **is** BM has the transition $\xrightarrow{\mu(RN)}$ if expression E finishes and returns a value N, and BM can be behaviour pattern-matched against this value producing transition $\xrightarrow{\mu(RN)}$.

$$\frac{\begin{array}{l}\mathcal{E} \vdash E \xrightarrow{\delta(\texttt{\$1=>}N)} E' \\ \mathcal{E} \vdash (BM \ \xrightarrow{}^{\mathrm{PM}}\ N) \xrightarrow{\mu(RN)} B\end{array}}{\mathcal{E} \vdash \textbf{case}\ E \mathbin{:} T\ \textbf{is}\ BM \xrightarrow{\mu(RN)} B} \tag{8.94}$$

If expression E raises an exception X when it is being evaluated, then the whole **case** behaviour raises the same exception X.

$$\frac{\mathcal{E} \vdash E \xrightarrow{X\texttt{(}RN\texttt{)}} E'}{\mathcal{E} \vdash \textbf{case}\ E{:}T\ \textbf{is}\ BM \xrightarrow{X\texttt{(}RN\texttt{)}} \textbf{case}\ E'{:}T\ \textbf{is}\ BM} \tag{8.95}$$

Where there is more than one expression, then all the expressions have to be evaluated and BM has to be behaviour pattern-matched against the record built from the values returned by the expressions.

$$\frac{\begin{array}{c}\mathcal{E} \vdash E_1 \xrightarrow{\delta\texttt{(\$1=>}N_1\texttt{)}} E_1' \quad \cdots \quad \mathcal{E} \vdash E_n \xrightarrow{\delta\texttt{(\$1=>}N_n\texttt{)}} E_n' \\ \mathcal{E} \vdash (BM \ \xrightarrow{}^{\mathbf{PM}} \ \texttt{(\$1 =>}\ N_1\texttt{,}\ \ldots\ \texttt{,\$}n\ \texttt{=>}\ N_n\texttt{))} \xrightarrow{\mu\texttt{(}RN'\texttt{)}} B\end{array}}{\mathcal{E} \vdash \textbf{case}\ \texttt{(}E_1{:}T_1, \ldots, E_n{:}T_n\texttt{)}\ \textbf{is}\ BM \xrightarrow{\mu\texttt{(}RN'\texttt{)}} B} \tag{8.96}$$

If some expression E_j raises an exception, then the whole behaviour raises the same exception.

$$\frac{\mathcal{E} \vdash E_j \xrightarrow{X\texttt{(}RN\texttt{)}} E_j'}{\begin{array}{ll}\mathcal{E} \vdash \textbf{case}\ \texttt{(}E_1{:}T_1, \ldots, E_j : T_j, \ldots, E_n{:}T_n\texttt{)} & \xrightarrow{X\texttt{(}RN\texttt{)}} \textbf{case}\ \texttt{(}E_1{:}T_1, \ldots, E_j' : T_j, \ldots, E_n{:}T_n\texttt{)} \\ \quad \textbf{is}\ BM & \quad\quad \textbf{is}\ BM\end{array}} \tag{8.97}$$

⏱ If the expression E finishes (respectively $E_1, \ldots, E_n$ finish) and the behaviour pattern-matching can idle a time d, then the **case** behaviour can idle that time.

$$\frac{\begin{array}{c}\mathcal{E} \vdash E \xrightarrow{\delta\texttt{(\$1=>}N\texttt{)}} E' \\ \mathcal{E} \vdash (BM \ \xrightarrow{}^{\mathbf{PM}} \ N) \xrightarrow{\epsilon\langle d\rangle} B\end{array}}{\mathcal{E} \vdash \textbf{case}\ E{:}T\ \textbf{is}\ BM \xrightarrow{\epsilon\langle d\rangle} B} \tag{8.98}$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash E_1 \xrightarrow{\delta\texttt{(\$1=>}N_1\texttt{)}} E_1' \quad \cdots \quad \mathcal{E} \vdash E_n \xrightarrow{\delta\texttt{(\$1=>}N_n\texttt{)}} E_n' \\ \mathcal{E} \vdash (BM \ \xrightarrow{}^{\mathbf{PM}} \ \texttt{(\$1 =>}\ N_1\texttt{,}\ \ldots\ \texttt{,\$}n\ \texttt{=>}\ N_n\texttt{))} \xrightarrow{\epsilon\langle d\rangle} B\end{array}}{\mathcal{E} \vdash \textbf{case}\ (E_1{:}T_1, \ldots, E_n{:}T_n)\ \textbf{is}\ BM \xrightarrow{\epsilon\langle d\rangle} B} \tag{8.99}$$

Example 8.15 Let us see how the behaviour

```
case ys:List is
        nil -> signal Hd
      | cons(?x,?xs) -> outP(!x)
endcase
```

communicates through gate *outP* the value 8 when $ys = cons(8, nil)$. It is translated to

$\textbf{case exit}\texttt{(\$1 =>}\ cons\texttt{(\$1 =>}\ 8, \texttt{\$2 =>}\ nil\texttt{()))}\texttt{:List}\ \textbf{is}\ BM_1\ \texttt{|}\ (BM_2\ \texttt{|}\ BM_3)$

where

$BM_1 \equiv Nil\texttt{()}\ \texttt{[}\textbf{exit}\texttt{(\$1 =>}\ true\texttt{)]->}\ \textbf{signal}\ Hd\texttt{(}\textbf{exit}\texttt{(\$1 => ()))}$

$BM_2 \equiv Cons\texttt{(\$1 => ?}x\texttt{,\$2 => ?}xs\texttt{)}\ \texttt{[}\textbf{exit}\texttt{(\$1 =>}\ true\texttt{)]->}\ outP\texttt{(\$1 => !}\textbf{exit}\texttt{(\$1 =>}\ x\texttt{))}\ldots$

$BM_3 \equiv \textbf{any}\texttt{:List}\ \texttt{[}\textbf{exit}\texttt{(\$1 =>}\ true\texttt{)]}\ \texttt{->}\ \textbf{signal}\ Match\texttt{(}\textbf{exit}\texttt{(\$1 => ()))}$

By using rule 8.94 we deduce

$$\frac{\begin{array}{l}\mathcal{E} \vdash \mathbf{exit}\texttt{(}\$1 \texttt{ => } \mathit{cons}\texttt{(}\$1 \texttt{ => } 8, \$2 \texttt{ => } \mathit{nil}\texttt{()))} \xrightarrow{\delta\texttt{(\$1=>}\mathit{cons}\texttt{(\$1=>8,\$2=>}\mathit{nil}\texttt{()))}} \mathbf{block} \\ \mathcal{E} \vdash (BM_1 \ \texttt{|}\ (BM_2 \ \texttt{|}\ BM_3) \ \rightarrow^{\mathbf{PM}} \ \mathit{cons}\texttt{(}\$1 \texttt{ => } 8, \$2 \texttt{ => } \mathit{nil}\texttt{()))} \xrightarrow{\mathit{outP}\texttt{(\$1=>8)}} \mathbf{exit}\texttt{()}\end{array}}{\mathcal{E} \vdash \mathbf{case\ exit}\texttt{(}\$1 \texttt{ => } \mathit{cons}\texttt{(}\$1 \texttt{ => } 8, \$2 \texttt{ => } \mathit{nil}\texttt{())):}\mathsf{List}\ \mathbf{is}\ BM_1 \ \texttt{|}\ (BM_2 \ \texttt{|}\ BM_3) \xrightarrow{\mathit{outP}\texttt{(\$1=>8)}} \mathbf{exit}\texttt{()}} \quad (8.94)$$

where the second premise is proved as follows:

$$\frac{\begin{array}{l}\mathcal{E} \vdash (BM_1 \ \rightarrow^{\mathbf{PM}} \ \mathit{cons}\texttt{(}\$1 \texttt{ => } 8, \$2 \texttt{ => } \mathit{nil}\texttt{()))} \mapsto \ \textsf{\textit{fail}} \\ \mathcal{E} \vdash (BM_2 \ \texttt{|}\ BM_3 \ \rightarrow^{\mathbf{PM}} \ \mathit{cons}\texttt{(}\$1 \texttt{ => } 8, \$2 \texttt{ => } \mathit{nil}\texttt{()))} \xrightarrow{\mathit{outP}\texttt{(\$1=>8)}} \mathbf{exit}\texttt{()}\end{array}}{\mathcal{E} \vdash (BM_1 \ \texttt{|}\ (BM_2 \ \texttt{|}\ BM_3) \ \rightarrow^{\mathbf{PM}} \ \mathit{cons}\texttt{(}\$1 \texttt{ => } 8, \$2 \texttt{ => } \mathit{nil}\texttt{()))} \xrightarrow{\mathit{outP}\texttt{(\$1=>8)}} \mathbf{exit}\texttt{()}} \quad (8.106)$$

where now, the first premise is proved using rule 8.100:

$$\frac{\dfrac{}{\mathcal{E} \vdash (\mathit{nil}\texttt{()} \ \rightarrow^{\mathbf{PM}} \ \mathit{cons}\texttt{(}\$1 \texttt{ => } 8, \$2 \texttt{ => } \mathit{nil}\texttt{()))} \mapsto \ \textsf{\textit{fail}}}\ (8.145)}{\mathcal{E} \vdash (BM_1 \ \rightarrow^{\mathbf{PM}} \ \mathit{cons}\texttt{(}\$1 \texttt{ => } 8, \$2 \texttt{ => } \mathit{nil}\texttt{()))} \mapsto \ \textsf{\textit{fail}}}\ (8.100)$$

and second premise is proved using rule 8.105:

$$\frac{\dfrac{\begin{array}{l}\mathcal{E} \vdash (\mathit{cons}\texttt{(}\$1 \texttt{ => ?}x\texttt{,}\$2 \texttt{ => ?}xs) \ \rightarrow^{\mathbf{PM}} \ \mathit{cons}\texttt{(}\$1 \texttt{ => } 8, \$2 \texttt{ => } \mathit{nil}\texttt{()))} \mapsto \ \langle\!|\, x \texttt{ => } 8, xs \texttt{ => } \mathit{nil}\texttt{()} \,|\!\rangle \\ \mathcal{E} \vdash \mathbf{exit}\texttt{(}\$1 \texttt{ => } \mathit{true}) \xrightarrow{\delta\texttt{(\$1=>}\mathit{true}\texttt{)}} \mathbf{block} \\ \mathcal{E} \vdash \mathit{outP}\texttt{(}\$1 \texttt{ => !}\mathbf{exit}\texttt{(}\$1 \texttt{ => } 8\texttt{))} \ldots \xrightarrow{\mathit{outP}\texttt{(\$1=>8)}} \mathbf{exit}\texttt{()}\end{array}}{\mathcal{E} \vdash (BM_2 \ \rightarrow^{\mathbf{PM}} \ \mathit{cons}\texttt{(}\$1 \texttt{ => } 8, \$2 \texttt{ => } \mathit{nil}\texttt{()))} \xrightarrow{\mathit{outP}\texttt{(\$1=>8)}} \mathbf{exit}\texttt{()}}\ (8.103)}{\mathcal{E} \vdash (BM_2 \texttt{|} BM_3 \ \rightarrow^{\mathbf{PM}} \ \mathit{cons}\texttt{(}\$1 \texttt{ => } 8, \$2 \texttt{ => } \mathit{nil}\texttt{()))} \xrightarrow{\mathit{outP}\texttt{(\$1=>8)}} \mathbf{exit}\texttt{()}}\ (8.105)$$

8.24 Behaviour pattern-matching

We define now how the different branches of a **case** behaviour, BM, are matched against a value expression N. A branch of the **case** can *fail* either because the pattern does not match against the value, or because the boolean expression is evaluated to *false*. The different possibilities are proved in sequence, that is, a branch is checked if all the previous ones have failed.

8.24.1 Single match

First, we define how the clause $P\ \texttt{[}\,E\,\texttt{]}\ \texttt{->}\ B$ is behaviour pattern-matched against value N. If the pattern P cannot be matched against value N, then the clause fails.

$$\frac{\mathcal{E} \vdash (P \ \rightarrow^{\mathbf{PM}} \ N) \mapsto \ \textsf{\textit{fail}}}{\mathcal{E} \vdash ((P\ \texttt{[}\,E\,\texttt{]}\ \texttt{->}\ B) \ \rightarrow^{\mathbf{PM}} \ N) \mapsto \ \textsf{\textit{fail}}} \quad (8.100)$$

If the pattern P can be matched against value N by producing bindings RN, but expression E with the bindings RN substituted returns *false*, then the clause $P\ \texttt{[}\,E\,\texttt{]}\ \texttt{->}\ B$ fails.

$$\frac{\begin{array}{l}\mathcal{E} \vdash (P \ \rightarrow^{\mathbf{PM}} \ N) \mapsto \ \langle\!|\, RN \,|\!\rangle \\ \mathcal{E} \vdash E\,[\![\,RN\,]\!] \xrightarrow{\delta\texttt{(\$1=>}\mathit{false}\texttt{)}} E'\end{array}}{\mathcal{E} \vdash ((P\ \texttt{[}\,E\,\texttt{]}\ \texttt{->}\ B) \ \rightarrow^{\mathbf{PM}} \ N) \mapsto \ \textsf{\textit{fail}}} \quad (8.101)$$

The clause also fails if the expression raises an exception when it is being evaluated. Note that the exception makes that the behaviour pattern-matching fails, but it is not propagated, that is, it is lost.

$$\frac{\begin{array}{c}\mathcal{E} \vdash (P \ \rightarrow^{\mathbf{PM}} \ N) \ \mapsto \ \llparenthesis RN \rrparenthesis \\ \mathcal{E} \vdash E\llbracket RN \rrbracket \xrightarrow{X(RN)} E'\end{array}}{\mathcal{E} \vdash ((P\ \texttt{[}E\texttt{]}\ \texttt{->}\ B) \ \rightarrow^{\mathbf{PM}} \ N) \ \mapsto \ \textsf{\textit{fail}}} \tag{8.102}$$

Finally, if the pattern P can be matched against value N by producing bindings RN, the expression E returns *true*, and the behaviour B where the bindings RN have been substituted can evolve, then the clause $P\ \texttt{[}E\texttt{]}\ \texttt{->}\ B$ evolves as B does.

$$\frac{\begin{array}{c}\mathcal{E} \vdash (P \ \rightarrow^{\mathbf{PM}} \ N) \ \mapsto \ \llparenthesis RN \rrparenthesis \\ \mathcal{E} \vdash E\llbracket RN \rrbracket \xrightarrow{\delta(\texttt{\$1=>}true)} E' \\ \mathcal{E} \vdash B\llbracket RN \rrbracket \xrightarrow{\mu(RN')} B'\end{array}}{\mathcal{E} \vdash ((P\ \texttt{[}E\texttt{]}\ \texttt{->}\ B) \ \rightarrow^{\mathbf{PM}} \ N) \xrightarrow{\mu(RN')} B'} \tag{8.103}$$

Regarding time transitions, the clause $P\ \texttt{[}E\texttt{]}\ \texttt{->}\ B$ can idle d units of time if it does not fail, and B can idle d.

$$\frac{\begin{array}{c}\mathcal{E} \vdash (P \ \rightarrow^{\mathbf{PM}} \ N) \ \mapsto \ \llparenthesis RN \rrparenthesis \\ \mathcal{E} \vdash E\llbracket RN \rrbracket \xrightarrow{\delta(\texttt{\$1=>}true)} E' \\ \mathcal{E} \vdash B\llbracket RN \rrbracket \xrightarrow{\epsilon\langle d\rangle} B'\end{array}}{\mathcal{E} \vdash ((P\ \texttt{[}E\texttt{]}\ \texttt{->}\ B) \ \rightarrow^{\mathbf{PM}} \ N) \xrightarrow{\epsilon\langle d\rangle} B'} \tag{8.104}$$

8.24.2 Multiple match

When there are several branches $BM_1\ \texttt{|}\ BM_2$ they are behaviour pattern-matched in sequence. If the first branch BM_1 can evolve, then $BM_1\ \texttt{|}\ BM_2$ evolves in the same way.

$$\frac{\mathcal{E} \vdash (BM_1 \ \rightarrow^{\mathbf{PM}} \ N) \xrightarrow{\mu(RN)} B}{\mathcal{E} \vdash ((BM_1\ \texttt{|}\ BM_2) \ \rightarrow^{\mathbf{PM}} \ N) \xrightarrow{\mu(RN)} B} \tag{8.105}$$

If the first part BM_1 fails, but the second part BM_2 can evolve, then $BM_1\ \texttt{|}\ BM_2$ evolves in the same way as BM_2 does.

$$\frac{\begin{array}{c}\mathcal{E} \vdash (BM_1 \ \rightarrow^{\mathbf{PM}} \ N) \ \mapsto \ \textsf{\textit{fail}} \\ \mathcal{E} \vdash (BM_2 \ \rightarrow^{\mathbf{PM}} \ N) \xrightarrow{\mu(RN)} B\end{array}}{\mathcal{E} \vdash ((BM_1\ \texttt{|}\ BM_2) \ \rightarrow^{\mathbf{PM}} \ N) \xrightarrow{\mu(RN)} B} \tag{8.106}$$

Finally, if both parts fail, then $BM_1\ \texttt{|}\ BM_2$ also fails.

$$\frac{\begin{array}{c}\mathcal{E} \vdash (BM_1 \ \rightarrow^{\mathbf{PM}} \ N) \ \mapsto \ \textsf{\textit{fail}} \\ \mathcal{E} \vdash (BM_2 \ \rightarrow^{\mathbf{PM}} \ N) \ \mapsto \ \textsf{\textit{fail}}\end{array}}{\mathcal{E} \vdash ((BM_1\ \texttt{|}\ BM_2) \ \rightarrow^{\mathbf{PM}} \ N) \ \mapsto \ \textsf{\textit{fail}}} \tag{8.107}$$

The composition $BM_1\ \texttt{|}\ BM_2$ can idle d units of time if the first branch that does not fail can idle that time.

$$\frac{\mathcal{E} \vdash (BM_1 \ \rightarrow^{\mathbf{PM}}\ N) \xrightarrow{\epsilon\langle d\rangle} B}{\mathcal{E} \vdash ((BM_1 \mid BM_2)\ \rightarrow^{\mathbf{PM}}\ N) \xrightarrow{\epsilon\langle d\rangle} B} \tag{8.108}$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash (BM_1 \ \rightarrow^{\mathbf{PM}}\ N) \mapsto \textsf{\textit{fail}} \\ \mathcal{E} \vdash (BM_2 \ \rightarrow^{\mathbf{PM}}\ N) \xrightarrow{\epsilon\langle d\rangle} B\end{array}}{\mathcal{E} \vdash ((BM_1 \mid BM_2) \Rightarrow N) \xrightarrow{\epsilon\langle d\rangle} B} \tag{8.109}$$

8.25 Iteration loop

The infinite loop **loop** B can evolve if its unfolding B;**loop** B evolves, and **loop** B evolves in the same way as its unfolding.

$$\frac{\mathcal{E} \vdash B\,;\mathbf{loop}\ B \xrightarrow{\mu(RN)} B'}{\mathcal{E} \vdash \mathbf{loop}\ B \xrightarrow{\mu(RN)} B'} \tag{8.110}$$

The same occurs with time transitions. The behaviour **loop** B has the same time transitions as its unfolding B;**loop** B.

$$\frac{\mathcal{E} \vdash B;\ \mathbf{loop}\ B \xrightarrow{\epsilon\langle d\rangle} B'}{\mathcal{E} \vdash \mathbf{loop}\ B \xrightarrow{\epsilon\langle d\rangle} B'} \tag{8.111}$$

8.26 Process instantiation

A process instantiation $\Pi\,[\,G_1, \ldots, G_m\,]\,(E_1, \ldots, E_p)\,[\,X_1, \ldots, X_n\,]$ can evolve if the body of the process Π where actual gates and exceptions have been substituted for the formal ones, and values returned by the actual parameters have been substituted for the corresponding input parameter variables, can evolve. The evaluation of the parameters and how they are "introduced" in the body is expressed in the following rule by means of a **case** behaviour that pattern-matches the input variables against the values returned by the expressions. Gates and exceptions are syntactically substituted: $B[G_1/G'_1, \ldots, G_m/G'_m, X_1/X'_1, \ldots, X_n/X'_n]$ represents the behaviour B where gates G_i and exceptions X_i have been simultaneously substituted for G'_i and X'_i. The inference rule for process instantiation is as follows:

$$\frac{\begin{array}{l}\mathcal{E} \vdash \Pi \Rightarrow \lambda\,[\,G'_1 : T_1, \ldots, G'_m : T_m\,]\,(V_1 : T'_1, \ldots, V_p : T'_p)\,[\,X'_1 : T''_1, \ldots, X'_n : T''_n\,] \rightarrow B \\ \mathcal{E} \vdash (\mathbf{case}\ (E_1 : T'_1, \ldots, E_p : T'_p)\ \mathtt{is} \\ \qquad (\$1 \texttt{ => } ?V_1, \ldots, \$p \texttt{ => } ?V_p)\ \texttt{->} \\ \qquad\quad B[G_1/G'_1, \ldots, G_m/G'_m, X_1/X'_1, \ldots, X_n/X'_n]) \xrightarrow{\mu(RN)} B'\end{array}}{\mathcal{E} \vdash \Pi\,[\,G_1, \ldots, G_m\,]\,(E_1, \ldots, E_p)\,[\,X_1, \ldots, X_n\,] \xrightarrow{\mu(RN)} B'} \tag{8.112}$$

And there is a analogous rule dealing with time transitions:

$$\frac{\begin{array}{l}\mathcal{E} \vdash \Pi \Rightarrow \lambda\mathtt{[}\,G'_1\!:\!T_1,\ldots,G'_m\!:\!T_m\,\mathtt{]}\,\mathtt{(}V_1\!:\!T'_1,\ldots,V_p\!:\!T'_p\mathtt{)}\,\mathtt{[}\,X'_1\!:\!T''_1,\ldots,X'_n\!:\!T''_n\,\mathtt{]} \rightarrow B\\ \mathcal{E} \vdash (\textbf{case}\ \mathtt{(}E_1\!:\!T'_1,\ldots,E_p\!:\!T'_p\mathtt{)}\ \textbf{is}\\ \qquad \mathtt{(}\$1\ \mathtt{=>}\ \mathtt{?}V_1,\ldots,\$p\ \mathtt{=>}\ \mathtt{?}V_p\mathtt{)}\ \mathtt{->}\\ \qquad B[G_1/G'_1,\ldots,G_m/G'_m,X_1/X'_1,\ldots,X_n/X'_n]) \xrightarrow{\epsilon\langle d\rangle} B'\end{array}}{\mathcal{E} \vdash \Pi\mathtt{[}\,G_1\mathtt{,}\ldots\mathtt{,}G_m\,\mathtt{]}\,\mathtt{(}E_1\mathtt{,}\ldots\mathtt{,}E_p\mathtt{)}\,\mathtt{[}\,X_1\mathtt{,}\ldots\mathtt{,}X_n\,\mathtt{]} \xrightarrow{\epsilon\langle d\rangle} B'} \tag{8.113}$$

8.27 Type expressions

Regarding type expressions, in the dynamic semantics we need inference rules that establish the subtyping relationship, because they are used in the general rule 8.125.

8.27.1 Type identifier

The information we can deduce about a type identifier S is what is included in the context.

$$\frac{}{\mathcal{E}, S \equiv T \vdash S \equiv T} \tag{8.114}$$

8.27.2 Empty type

The type **none** is subtype of any type T,

$$\frac{}{\mathcal{E} \vdash \textbf{none} \sqsubseteq T} \tag{8.115}$$

and it is equivalent to any record type with a field V of type **none**,

$$\frac{}{\mathcal{E} \vdash \textbf{none} \equiv \mathtt{(}V\ \mathtt{=>}\ \textbf{none}\mathtt{,}RT\mathtt{)}} \tag{8.116}$$

8.27.3 Universal type

The type **any** is a supertype of any type T,

$$\frac{}{\mathcal{E} \vdash T \sqsubseteq \textbf{any}} \tag{8.117}$$

8.27.4 Record type

A record type $\mathtt{(}RT\mathtt{)}$ is a subtype of another record type $\mathtt{(}RT'\mathtt{)}$ whenever RT is a subtype of RT' (see next section).

$$\frac{\mathcal{E} \vdash RT \sqsubseteq RT'}{\mathcal{E} \vdash \mathtt{(}RT\mathtt{)} \sqsubseteq \mathtt{(}RT'\mathtt{)}} \tag{8.118}$$

8.28 Record type expressions

8.28.1 Singleton record

If T is a subtype of T', then the record type expression V `=>` T is a subtype of V `=>` T'.

$$\frac{\mathcal{E} \vdash T \sqsubseteq T'}{\mathcal{E} \vdash V \texttt{=>} T \sqsubseteq V \texttt{=>} T'} \tag{8.119}$$

8.28.2 Record disjoint union

The disjoint union $RT_1 \texttt{,} RT_2$ is a subtype of $RT'_1 \texttt{,} RT'_2$ if it can be proved that $RT_1 \sqsubseteq RT'_1$ and $RT_2 \sqsubseteq RT'_2$.

$$\frac{\mathcal{E} \vdash RT_1 \sqsubseteq RT'_1 \qquad \mathcal{E} \vdash RT_2 \sqsubseteq RT'_2}{\mathcal{E} \vdash RT_1 \texttt{,} RT_2 \sqsubseteq RT'_1 \texttt{,} RT'_2} \tag{8.120}$$

The binary operator "`,`" between records is commutative, associative, and it has the empty record as identity.

$$\frac{}{\mathcal{E} \vdash RT_1 \texttt{,} RT_2 \equiv RT_2 \texttt{,} RT_1} \tag{8.121}$$

$$\frac{}{\mathcal{E} \vdash (RT_1 \texttt{,} RT_2) \texttt{,} RT_3 \equiv RT_1 \texttt{,} (RT_2 \texttt{,} RT_3)} \tag{8.122}$$

$$\frac{}{\mathcal{E} \vdash () \texttt{,} RT \equiv RT} \tag{8.123}$$

8.28.3 Universal record

etc is a supertype of any record type expression RT

$$\frac{}{\mathcal{E} \vdash RT \sqsubseteq \textbf{etc}} \tag{8.124}$$

8.29 Value Expressions

The information about which is the type T of a value N, $N \ \xrightarrow{\mathbf{T}}\ T$, is needed in the dynamic semantics in the inference rules that *invent* values, in the sense that these values appear in the premises of the rule but not in the conclusion. The sole places where this occurs are in the dynamic semantics of the **hide** (Section 8.15) and **rename** (Section 8.19) operators (where values of the type of the corresponding declared gate are invented), and in the dynamic semantics of the nondeterministic assignment, P `:=` **any** T `[` E `]` (Section 8.21) (where a value of type T is invented).

As in the static semantics, we have the following general rule:

$$\frac{\begin{array}{l}\mathcal{E} \vdash N \ \xrightarrow{\mathbf{T}}\ T \\ \mathcal{E} \vdash T \sqsubseteq T'\end{array}}{\mathcal{E} \vdash N \ \xrightarrow{\mathbf{T}}\ T'} \tag{8.125}$$

The inference rules are as in the static semantics, so we do not explain it again.

8.29.1 Primitive constants

$$\frac{\mathcal{E} \vdash K \Rightarrow T}{\mathcal{E} \vdash K \ \rightarrow^{\mathrm{T}} \ T} \tag{8.126}$$

8.29.2 Variables

We do not need to know the type of a variable because when values are invented variables are not used.

8.29.3 Constructor application

$$\frac{\begin{array}{l}\mathcal{E} \vdash C \Rightarrow ((RT) \rightarrow S) \\ \mathcal{E} \vdash N \ \rightarrow^{\mathrm{T}} \ (RT)\end{array}}{\mathcal{E} \vdash C\ N \ \rightarrow^{\mathrm{T}} \ S} \tag{8.127}$$

8.29.4 Record values

$$\frac{\mathcal{E} \vdash RN \ \rightarrow^{\mathrm{T}} \ RT}{\mathcal{E} \vdash (RN) \ \rightarrow^{\mathrm{T}} \ (RT)} \tag{8.128}$$

8.30 Record value expressions

As in the static semantics, we have the following general rule:

$$\frac{\begin{array}{l}\mathcal{E} \vdash RN \ \rightarrow^{\mathrm{T}} \ RT \\ \mathcal{E} \vdash RT \sqsubseteq RT'\end{array}}{\mathcal{E} \vdash RN \ \rightarrow^{\mathrm{T}} \ RT'} \tag{8.129}$$

8.30.1 Empty record

$$\frac{}{\mathcal{E} \vdash () \ \rightarrow^{\mathrm{T}} \ ()} \tag{8.130}$$

8.30.2 Singleton record

$$\frac{\mathcal{E} \vdash N \ \rightarrow^{\mathrm{T}} \ T}{\mathcal{E} \vdash (V \texttt{=>} N) \ \rightarrow^{\mathrm{T}} \ (V \texttt{=>} T)} \tag{8.131}$$

8.30.3 Record disjoint union

$$\frac{\mathcal{E} \vdash RN_1 \ \rightarrow^{\mathrm{T}} \ RT_1 \qquad \mathcal{E} \vdash RN_2 \ \rightarrow^{\mathrm{T}} \ RT_2}{\mathcal{E} \vdash RN_1\texttt{,}\ RN_2 \ \rightarrow^{\mathrm{T}} \ RT_1\texttt{,}\ RT_2} \tag{8.132}$$

with the side condition that RN_1 and RN_2 have disjoint fields.

8.31 Patterns

We define in the following sections how the different patterns of E-LOTOS can be matched against values, and what bindings this pattern-matching produces. In the static semantics, patterns are matched against types, and the rules define only the successful pattern-matching, that is, a pattern cannot be matched against a type if there is not a proof which proves that the pattern-matching can be done. In the dynamic semantics it is important to have rules for proving that a pattern cannot be matched against a value. This is useful when a **case** behaviour is being executed and a value is matched against the different branches of the **case**, as we saw in Section 8.24.

In [Que98] type checking, $\xrightarrow{\mathbf{T}}$, is used in the pattern-matching inference rules in some places where we think that it is not needed. Intuitively, when the pattern-matching judgement

$$\mathcal{E} \vdash (P \ \xrightarrow{\mathbf{PM}} \ N) \ \mapsto \ \langle\!|\, RN \,|\!\rangle$$

is used as a premise in an inference rule $\mathcal{IR}$, first the corresponding static semantics rules had proved

$$\mathcal{C} \vdash (P \ \xrightarrow{\mathbf{PM}} \ T) \ \mapsto \ \langle\!|\, RT \,|\!\rangle$$

and $\mathcal{IR}$ checks that $N \ \xrightarrow{\mathbf{T}} \ T$ or it is true due to the following general result: if $E \implies \textit{exit}\,\langle\!|\, \$1\texttt{=>}T, RT \,|\!\rangle$ and $E \xrightarrow{\delta(\$1\texttt{=>}N,RN)} E'$, then $N \ \xrightarrow{\mathbf{T}} \ T$; and when RT is the empty record type, RN is the empty record value (which can be easily proved by induction on the length of the proofs). Thus, we do not need to check it again in the inference rule of the pattern-matching. The sole place where type checking is needed is in those patterns with an explicit type, wildcard pattern and explicit typing pattern, as we see below.

8.31.1 Variable pattern

Pattern $\texttt{?}V$ can be always pattern-matched against value N.

$$\frac{}{\mathcal{E} \vdash (\texttt{?}V \ \xrightarrow{\mathbf{PM}} \ N) \ \mapsto \ \langle\!|\, V \Rightarrow N \,|\!\rangle} \tag{8.133}$$

8.31.2 Expression pattern

The pattern $\texttt{!}E$ can be matched against value N if expression E returns the value N, that is, E communicates through δ the bindings $\$1 \texttt{=>} N$.

$$\frac{\mathcal{E} \vdash E \xrightarrow{\delta(\$1\texttt{=>}N)} E'}{\mathcal{E} \vdash (\texttt{!}E \ \xrightarrow{\mathbf{PM}} \ N) \ \mapsto \ \langle\!|\ |\!\rangle} \tag{8.134}$$

If expression E returns a value N' different to N, then the pattern $\texttt{!}E$ cannot be matched against value N.

$$\frac{\mathcal{E} \vdash E \xrightarrow{\delta\langle\$1\texttt{=>}N'\rangle} E'}{\mathcal{E} \vdash (\texttt{!}E \ \xrightarrow{\mathbf{PM}} \ N) \ \mapsto \ \textit{fail}} \ [N \neq N'] \tag{8.135}$$

And if the expression E raises an exception when it is being executed, the pattern-matching also fails.

$$\frac{\mathcal{E} \vdash E \xrightarrow{X(RN)} E'}{\mathcal{E} \vdash (\texttt{!}E \ \xrightarrow{\mathbf{PM}} \ N) \ \mapsto \ \textit{fail}} \tag{8.136}$$

8.31.3 Wildcard pattern

As we saw in Section 7.31.3, we have modified the static semantics of the wildcard pattern **any**$:T$. Therefore, we have to modify also the dynamic semantics. When we try to pattern-match the pattern **any**$:T$ against value N, we have to check the type of the value N because although it has passed the static semantics, perhaps there it was pattern-matched against a supertype of T. For example, the behaviour

```
case E:int is
        any:nat -> B_1
      | any:int -> B_2
endcase
```

is semantically correct (assuming $\mathsf{nat} \sqsubseteq \mathsf{int}$), but we want that if E evaluates to -2, the first branch fails.

Thus, the pattern **any**$:T$ can be matched against value N if N has type T.

$$\frac{\mathcal{E} \vdash N \ \rightarrow^{\mathbf{T}} \ T}{\mathcal{E} \vdash (\mathbf{any}{:}T \ \rightarrow^{\mathbf{PM}} \ N) \ \mapsto \ \llparenthesis \ \rrparenthesis} \tag{8.137}$$

and if N has not type T, the pattern **any**$:T$ fails

$$\frac{\mathcal{E} \vdash N \ \not\rightarrow^{\mathbf{T}} \ T}{\mathcal{E} \vdash (\mathbf{any}{:}T \ \rightarrow^{\mathbf{PM}} \ N) \ \mapsto \ \mathit{fail}} \tag{8.138}$$

If we follow the notation used in [Que98] in the explicit typing pattern, we should write

$$\frac{\begin{array}{l}\mathcal{E} \vdash N \ \rightarrow^{\mathbf{T}} \ T' \\ \mathcal{E} \vdash T \sqcap T' \Rightarrow \mathbf{none}\end{array}}{\mathcal{E} \vdash (\mathbf{any}{:}T \ \rightarrow^{\mathbf{PM}} \ N) \ \mapsto \ \mathit{fail}} \tag{8.139}$$

where $T \sqcap T'$ represents the *greatest lower bound* of T and T' with respect to the subtyping relation $\sqsubseteq$. $T \sqcap T' \Rightarrow \mathbf{none}$ implies that $T \not\sqsubseteq T'$ and $T' \not\sqsubseteq T$, and, therefore, $N \ \rightarrow^{\mathbf{T}} \ T'$ and $T \sqcap T' \Rightarrow \mathbf{none}$ implies $N \ \not\rightarrow^{\mathbf{T}} \ T$.

8.31.4 Record pattern

The record pattern (RP) can be matched against record value (RN) if RP can be matched against RN (see next section) and it produces the same bindings.

$$\frac{\mathcal{E} \vdash (RP \ \rightarrow^{\mathbf{PM}} \ RN') \ \mapsto \ \llparenthesis\, RN \,\rrparenthesis}{\mathcal{E} \vdash ((RP) \ \rightarrow^{\mathbf{PM}} \ (RN')) \ \mapsto \ \llparenthesis\, RN \,\rrparenthesis} \tag{8.140}$$

But it the pattern-matching between RP and RN fails, then the pattern-matching between (RP) and (RN) also fails.

$$\frac{\mathcal{E} \vdash (RP \ \rightarrow^{\mathbf{PM}} \ RN') \ \mapsto \ \mathit{fail}}{\mathcal{E} \vdash ((RP) \ \rightarrow^{\mathbf{PM}} \ (RN')) \ \mapsto \ \mathit{fail}} \tag{8.141}$$

Finally, if (RP) is pattern-matched against a value N which is not a record value (RN') then the pattern-matching fails.

$$\frac{}{\mathcal{E} \vdash ((RP) \ \rightarrow^{\mathbf{PM}} \ N) \Rightarrow \mathit{fail}} \ [\nexists\, RN' \mid N = (RN')] \tag{8.142}$$

Example 8.16 Let us see how the pattern $(\$1 \texttt{=>} ?x : \mathsf{int})$ can be matched against the value $(\$1 \texttt{=>} 8)$.

$$\dfrac{\dfrac{\dfrac{\mathcal{E} \vdash 8 \;\rightharpoonup^{\mathbf{T}}\; \mathsf{int} \qquad \dfrac{}{\mathcal{E} \vdash (?x \;\rightharpoonup^{\mathbf{PM}}\; 8) \;\mapsto\; \langle\!|\, x \texttt{=>} 8 \,|\!\rangle}\;(8.133)}{E \vdash (?x : \mathsf{int} \;\rightharpoonup^{\mathbf{PM}}\; 8) \;\mapsto\; \langle\!|\, x \texttt{=>} 8 \,|\!\rangle}\;(8.146)}{\mathcal{E} \vdash (\$1 \texttt{=>} ?x : \mathsf{int} \;\rightharpoonup^{\mathbf{PM}}\; \$1 \texttt{=>} 8) \;\mapsto\; \langle\!|\, x \texttt{=>} 8 \,|\!\rangle}\;(8.150)}{\mathcal{E} \vdash ((\$1 \texttt{=>} ?x : \mathsf{int}) \;\rightharpoonup^{\mathbf{PM}}\; (\$1 \texttt{=>} 8)) \;\mapsto\; \langle\!|\, x \texttt{=>} 8 \,|\!\rangle}\;(8.140)$$

8.31.5 Constructor pattern

The pattern $C\ P$ can be pattern-matched against value $C(RN)$ if pattern P can be pattern-matched against value (RN), producing the same bindings.

$$\dfrac{\mathcal{E} \vdash (P \;\rightharpoonup^{\mathbf{PM}}\; (RN)) \;\mapsto\; \langle\!|\, RN' \,|\!\rangle}{\mathcal{E} \vdash (C\ P \;\rightharpoonup^{\mathbf{PM}}\; C(RN)) \;\mapsto\; \langle\!|\, RN' \,|\!\rangle} \tag{8.143}$$

If P cannot be pattern-matched against (RN) then the pattern-matching between $C\ P$ and $C(RN)$ fails.

$$\dfrac{\mathcal{E} \vdash (P \;\rightharpoonup^{\mathbf{PM}}\; (RN)) \;\mapsto\; \mathit{fail}}{\mathcal{E} \vdash (C\ P \;\rightharpoonup^{\mathbf{PM}}\; C(RN)) \;\mapsto\; \mathit{fail}} \tag{8.144}$$

The pattern $C\ P$ cannot be pattern-matched against value N if N is not of the form $C\ N'$, that is, if N is not a value beginning with the constructor C.

$$\dfrac{}{\mathcal{E} \vdash (C\ P \;\rightharpoonup^{\mathbf{PM}}\; N) \;\mapsto\; \mathit{fail}}\ [\nexists\, C\ N' \mid N = C\ N'] \tag{8.145}$$

8.31.6 Explicit typing pattern

As in the case of the wilcard patter, when we try to pattern-match the explicit typing pattern $P{:}T$ against value N, we have to check the type of the value N. For example, the behaviour

```
case E:int is
        ?x:nat -> B_1
    | any:int -> B_2
endcase
```

is semantically correct (assuming $\mathsf{nat} \sqsubseteq \mathsf{int}$), but, as before, we want that if E evaluates to -2, the first branch fails.

Hence, the pattern $P{:}T$ can be pattern-matched against value N if N has type T and pattern P can be pattern-matched against value N.

$$\dfrac{\begin{array}{l}\mathcal{E} \vdash N \;\rightharpoonup^{\mathbf{T}}\; T \\ \mathcal{E} \vdash (P \;\rightharpoonup^{\mathbf{PM}}\; N) \;\mapsto\; \langle\!|\, RN \,|\!\rangle\end{array}}{\mathcal{E} \vdash (P{:}T \;\rightharpoonup^{\mathbf{PM}}\; N) \;\mapsto\; \langle\!|\, RN \,|\!\rangle} \tag{8.146}$$

If the pattern-matching between P and N fails, then the pattern $P{:}T$ cannot be pattern-matched against value N.

$$\frac{\begin{array}{l}\mathcal{E} \vdash N \ \rightarrow^{\mathbf{T}}\ T \\ \mathcal{E} \vdash (P \ \rightarrow^{\mathbf{PM}}\ N) \ \mapsto\ \textsf{\textit{fail}}\end{array}}{\mathcal{E} \vdash (P\!:\!T \ \rightarrow^{\mathbf{PM}}\ N) \ \mapsto\ \textsf{\textit{fail}}} \tag{8.147}$$

Finally, if N has not type T then the pattern-matching between $P\!:\!T$ and N fails.

$$\frac{\begin{array}{l}\mathcal{E} \vdash N \ \rightarrow^{\mathbf{T}}\ T' \\ \mathcal{E} \vdash T \sqcap T' \Rightarrow \mathbf{none}\end{array}}{\mathcal{E} \vdash (P\!:\!T \ \rightarrow^{\mathbf{PM}}\ N) \ \mapsto\ \textsf{\textit{fail}}} \tag{8.148}$$

8.32 Record of patterns

8.32.1 Empty record pattern

The empty record of patterns (which has no syntax, and is represented here by "()") can be only pattern-matched against the empty record value (also represented by "()") producing no bindings.

$$\frac{}{\mathcal{E} \vdash (() \ \rightarrow^{\mathbf{PM}}\ ()) \ \mapsto\ \llparenthesis\ \rrparenthesis} \tag{8.149}$$

8.32.2 Singleton record pattern

The singleton record pattern V => P can be pattern-matched against the record value V => N if P can be pattern-matched against N.

$$\frac{\mathcal{E} \vdash (P \ \rightarrow^{\mathbf{PM}}\ N) \ \mapsto\ \llparenthesis\, RN \,\rrparenthesis}{\mathcal{E} \vdash (V \texttt{ => } P \ \rightarrow^{\mathbf{PM}}\ V \texttt{ => } N) \ \mapsto\ \llparenthesis\, RN \,\rrparenthesis} \tag{8.150}$$

And if the pattern-matching between P and N fails then V => P cannot be pattern-matched against the record value V => N.

$$\frac{\mathcal{E} \vdash (P \ \rightarrow^{\mathbf{PM}}\ N) \ \mapsto\ \textsf{\textit{fail}}}{\mathcal{E} \vdash (V \texttt{ => } P \ \rightarrow^{\mathbf{PM}}\ V \texttt{ => } N) \ \mapsto\ \textsf{\textit{fail}}} \tag{8.151}$$

8.32.3 Record disjoint union

The record of patterns RP_1 , RP_2 can be pattern-matched against value RN_1 , RN_2 if RP_1 can be pattern-matched against value RN_1 and RP_2 can be pattern-matched against value RN_2.

$$\frac{\mathcal{E} \vdash (RP_1 \ \rightarrow^{\mathbf{PM}}\ RN_1) \ \mapsto\ \llparenthesis\, RN'_1 \,\rrparenthesis \qquad \mathcal{E} \vdash (RP_2 \ \rightarrow^{\mathbf{PM}}\ RN_2) \ \mapsto\ \llparenthesis\, RN'_2 \,\rrparenthesis}{\mathcal{E} \vdash (RP_1\,\texttt{,}\ RP_2 \ \rightarrow^{\mathbf{PM}}\ RN_1\,\texttt{,}\ RN_2) \ \mapsto\ \llparenthesis\, RN'_1, RN'_2 \,\rrparenthesis} \tag{8.152}$$

If RP_1 or RP_2 cannot be pattern-matched against the corresponding pattern, then the record patter-matching fails.

$$\frac{\mathcal{E} \vdash (RP_1 \ \rightarrow^{\mathbf{PM}}\ RN_1) \ \mapsto\ \textsf{\textit{fail}}}{\mathcal{E} \vdash (RP_1\,\texttt{,}\ RP_2 \ \rightarrow^{\mathbf{PM}}\ RN_1\,\texttt{,}\ RN_2) \ \mapsto\ \textsf{\textit{fail}}} \tag{8.153}$$

$$\frac{\mathcal{E} \vdash (RP_2 \ \xrightarrow{}^{\mathbf{PM}} \ RN_2) \ \mapsto \ \textsf{\textit{fail}}}{\mathcal{E} \vdash (RP_1\,,\ RP_2 \ \xrightarrow{}^{\mathbf{PM}} \ RN_1\,,\ RN_2) \ \mapsto \ \textsf{\textit{fail}}} \tag{8.154}$$

8.32.4 Wildcard record pattern

The wildcard record pattern **etc** can be pattern-matched against any record of values producing no bindings.

$$\frac{}{\mathcal{E} \vdash (\mathbf{etc} \ \xrightarrow{}^{\mathbf{PM}} \ RN) \ \mapsto \ \langle\!|\ |\!\rangle} \tag{8.155}$$

9 Semantics of Module Language

We are going to define in this chapter the semantics of a very simple (but complete) E-LOTOS specification which will help us to understand how the context used in the previous chapters are built. The kind of specification we describe is like the following:

specification $\equiv$
module mod-id **is**
m-body
endmod
specification Σ **import** mod-id **is**
gates $G_1 : T_1$ **,** . . . **,** $G_m : T_m$
exceptions $X_1 : T'_1$ **,** . . . **,** $X_n : T'_n$
behaviour B
endspec

where *m-body* is a sequence of type (Section 3.1) and process (Section 2.20) declarations.

9.1 Static Semantics

A specification is semantically correct if we can prove the following judgement

$$\vdash \textit{specification} \implies \mathsf{ok}$$

using the following rule

$$\frac{\begin{array}{l} \mathcal{C}_0 \vdash \textit{m-body} \implies \mathcal{C} \\ \mathcal{C}_0, \mathcal{C}, \\ \quad G_1 \Rightarrow \mathsf{gate}\langle \texttt{(\$1 =>} \ T_1\texttt{)} \rangle, \ldots, G_m \Rightarrow \mathsf{gate}\langle \texttt{(\$1 =>} \ T_m\texttt{)} \rangle, \\ \quad X_1 \Rightarrow \mathsf{exn}\langle \texttt{(\$1 =>} \ T'_1\texttt{)} \rangle, \ldots, X_n \Rightarrow \mathsf{exn}\langle \texttt{(\$1 =>} \ T'_n\texttt{)} \rangle \qquad \vdash B \implies \mathsf{exit}\, \langle\!| \, RT \, |\!\rangle \end{array}}{\vdash \textit{specification} \implies \mathsf{ok}} \tag{9.1}$$

where $\mathcal{C}_0$ is the initial context that includes the constants, types, and functions defined in the predefined library of E-LOTOS.

The first premise states that the module body *m-body* can be semantically checked (with the inference rules we will see in the following sections) returning a context $\mathcal{C}$ which represents the *type* of the module body.

The second premise states that the behaviour B is semantically correct (and we have to use the inference rules in Chapter 7 in order to prove it) in a context which includes $\mathcal{C}_0$ (the initial context), all the declarations from *m-body*, $\mathcal{C}$, and the declared gates and exceptions that B can use.

In the following sections we define how the different declarations in *m-body* are semantically checked, and what contexts each one returns. The judgements are of the form

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

meaning that in context $\mathcal{C}$ declaration D is semantically correct and it produces context $\mathcal{C}'$.

9.1.1 Synonymous type

The synonymous type declaration **type** S **renames** T is semantically correct in the context $\mathcal{C}$ if T is a declared type in this context.

$$\frac{\mathcal{C} \vdash T \Rightarrow \textsf{\textit{type}}}{\mathcal{C} \vdash \textbf{type}\ S\ \textbf{renames}\ T \implies (S \Rightarrow \textsf{\textit{type}}, S \equiv T)} \tag{9.2}$$

9.1.2 Named record type

The named record type declaration **type** S **is** (RT) is semantically correct in the context $\mathcal{C}$ if the record type (RT) is correct in that context.

$$\frac{\mathcal{C} \vdash (RT) \Rightarrow \textsf{\textit{type}}}{\mathcal{C} \vdash \textbf{type}\ S\ \textbf{is}\ (RT) \implies (S \Rightarrow \textsf{\textit{type}}, S \equiv (RT))} \tag{9.3}$$

9.1.3 New data type

The user defined new data type

type S **is**
$C_1(RT_1)$ | ... | $C_n(RT_n)$

is semantically correct in the context $\mathcal{C}$ if the type of the arguments given to the constructors are correct in the context $\mathcal{C}$ where the type S which is being declared is included. This is done to allow recursive types, that is, type S can be used in the argument types RT_i.

$$\frac{\forall i\,.\,1 \le i \le n\,.\,\mathcal{C}, S \Rightarrow \textsf{\textit{type}} \vdash (RT_i) \Rightarrow \textsf{\textbf{type}}}{\begin{array}{l}\mathcal{C} \vdash \textbf{type}\ S\ \textbf{is}\ C_1(RT_1)\ \mid\ \ldots\ \mid\ C_n(RT_n)\\ \qquad \implies (S \Rightarrow \textsf{\textit{type}}, C_1 \Rightarrow (RT_1) \rightarrow S, \ldots, C_n \Rightarrow (RT_n) \rightarrow S)\end{array}} \tag{9.4}$$

9.1.4 Process declaration

A process declaration

processDecl $\equiv$
process Π [G_1:T_1, ..., G_n:T_n] (V_1:T_1', ..., V_p:T_p') :(RT)
raises [X_1:T_1'', ..., X_n:T_n'']
is B

is semantically correct if the types given to gates, parameters, and exceptions are correct types declared in the context; and the body of the process B is semantically correct in a context where we have include the declared gates, parameters, exceptions, and the information related to the process declaration, in order to allow recursive instantiations in B.

If the body B is a guarded behaviour the inference rule is as follows:

$$\frac{\begin{array}{l}
\forall i\,.\,1 \le i \le m\,.\,\mathcal{C} \vdash T_i \Rightarrow \mathsf{type} \\
\forall i\,.\,1 \le i \le p\,.\,\mathcal{C} \vdash T_i' \Rightarrow \mathsf{type} \\
\forall i\,.\,1 \le i \le n\,.\,\mathcal{C} \vdash T_i'' \Rightarrow \mathsf{type} \\
\mathcal{C} \vdash \texttt{(}RT\texttt{)} \Rightarrow \mathbf{type} \\
\mathcal{C}, G_1 \Rightarrow \mathsf{gate}\langle\texttt{(}\$1 \texttt{=>} T_1\texttt{)}\rangle, \ldots, G_m \Rightarrow \mathsf{gate}\langle\texttt{(}\$1 \texttt{=>} T_m\texttt{)}\rangle, \\
\quad V_1 \texttt{=>} T_1', \ldots, V_p \texttt{=>} T_p', \\
\quad X_1 \Rightarrow \mathsf{gate}\langle\texttt{(}\$1 \texttt{=>} T_1''\texttt{)}\rangle, \ldots, X_n \Rightarrow \mathsf{gate}\langle\texttt{(}\$1 \texttt{=>} T_n''\texttt{)}\rangle, \\
\quad proc \qquad\qquad\qquad\qquad\qquad\qquad \vdash B \implies \mathsf{guarded} \llparenthesis\, RT \,\rrparenthesis
\end{array}}{\begin{array}{l}
\mathcal{C} \vdash processDecl \implies \Pi \Rightarrow \texttt{[}\,\mathsf{gate}\langle\texttt{(}\$1 \texttt{=>} T_1\texttt{)}\rangle\texttt{,}\ldots\texttt{,}\mathsf{gate}\langle\texttt{(}\$1 \texttt{=>} T_m\texttt{)}\rangle\,\texttt{]} \\
\qquad\qquad\qquad\qquad\qquad \texttt{(}V_1\texttt{:}T_1', \ldots, V_p\texttt{:}T_p'\texttt{)} \\
\qquad\qquad\qquad\qquad\qquad \texttt{[}\,\mathsf{exn}\langle\texttt{(}\$1 \texttt{=>} T_1''\texttt{)}\rangle\texttt{,}\ldots\texttt{,}\mathsf{exn}\langle\texttt{(}\$1 \texttt{=>} T_n''\texttt{)}\rangle\,\texttt{]} \rightarrow \mathsf{guarded} \llparenthesis\, RT \,\rrparenthesis
\end{array}} \tag{9.5}$$

where

$$\begin{array}{l}
proc \equiv \Pi \Rightarrow \texttt{[}\,\mathsf{gate}\langle\texttt{(}\$1 \texttt{=>} T_1\texttt{)}\rangle\texttt{,}\ldots\texttt{,}\mathsf{gate}\langle\texttt{(}\$1 \texttt{=>} T_m\texttt{)}\rangle\,\texttt{]} \\
\qquad\qquad \texttt{(}V_1\texttt{:}T_1', \ldots, V_p\texttt{:}T_p'\texttt{)} \\
\qquad\qquad \texttt{[}\,\mathsf{exn}\langle\texttt{(}\$1 \texttt{=>} T_1''\texttt{)}\rangle\texttt{,}\ldots\texttt{,}\mathsf{exn}\langle\texttt{(}\$1 \texttt{=>} T_n''\texttt{)}\rangle\,\texttt{]} \rightarrow \mathsf{exit} \llparenthesis\, RT \,\rrparenthesis
\end{array}$$

Note that the type $\texttt{(}RT\texttt{)}$ is part of the process declaration, that is, the rule states that the body of the process has to produce (only) those bindings related to **out** parameters, those in $\texttt{(}RT\texttt{)}$.

And if the body B is an exit behaviour the inference rule is:

$$\frac{\begin{array}{l}
\forall i\,.\,1 \le i \le m\,.\,\mathcal{C} \vdash T_i \Rightarrow \mathsf{type} \\
\forall i\,.\,1 \le i \le p\,.\,\mathcal{C} \vdash T_i' \Rightarrow \mathsf{type} \\
\forall i\,.\,1 \le i \le n\,.\,\mathcal{C} \vdash T_i'' \Rightarrow \mathsf{type} \\
\mathcal{C} \vdash \texttt{(}RT\texttt{)} \Rightarrow \mathbf{type} \\
\mathcal{C}, G_1 \Rightarrow \mathsf{gate}\langle\texttt{(}\$1 \texttt{=>} T_1\texttt{)}\rangle, \ldots, G_m \Rightarrow \mathsf{gate}\langle\texttt{(}\$1 \texttt{=>} T_m\texttt{)}\rangle, \\
\quad V_1 \texttt{=>} T_1', \ldots, V_p \texttt{=>} T_p', \\
\quad X_1 \Rightarrow \mathsf{gate}\langle\texttt{(}\$1 \texttt{=>} T_1''\texttt{)}\rangle, \ldots, X_n \Rightarrow \mathsf{gate}\langle\texttt{(}\$1 \texttt{=>} T_n''\texttt{)}\rangle, \\
\quad proc \qquad\qquad\qquad\qquad\qquad\qquad \vdash B \implies \mathsf{exit} \llparenthesis\, RT \,\rrparenthesis
\end{array}}{\begin{array}{l}
\mathcal{C} \vdash processDecl \implies \Pi \Rightarrow \texttt{[}\,\mathsf{gate}\langle\texttt{(}\$1 \texttt{=>} T_1\texttt{)}\rangle\texttt{,}\ldots\texttt{,}\mathsf{gate}\langle\texttt{(}\$1 \texttt{=>} T_m\texttt{)}\rangle\,\texttt{]} \\
\qquad\qquad\qquad\qquad\qquad \texttt{(}V_1\texttt{:}T_1', \ldots, V_p\texttt{:}T_p'\texttt{)} \\
\qquad\qquad\qquad\qquad\qquad \texttt{[}\,\mathsf{exn}\langle\texttt{(}\$1 \texttt{=>} T_1''\texttt{)}\rangle\texttt{,}\ldots\texttt{,}\mathsf{exn}\langle\texttt{(}\$1 \texttt{=>} T_n''\texttt{)}\rangle\,\texttt{]} \rightarrow \mathsf{exit} \llparenthesis\, RT \,\rrparenthesis
\end{array}} \tag{9.6}$$

with the same *proc* included in the context.

Example 9.1 Let us see that the process declaration

$$\begin{array}{l}
\mathbf{process}\ \text{User2}\ \texttt{[}acc\texttt{:id,}abd\texttt{:id]}\,\texttt{(}myid\texttt{:id)}\ \mathbf{is} \\
\quad acc\texttt{(!}myid\texttt{);} \\
\quad abd\texttt{(!}myid\texttt{);} \\
\quad \text{User2}\ \texttt{[}acc\texttt{,}abd\texttt{]}\,\texttt{(}myid\texttt{)} \\
\mathbf{endproc}
\end{array}$$

is semantically correct in a context where the type id is defined, and what context it returns. The process declaration is translated to

$$\begin{array}{l}
processDecl \quad \equiv \\
\qquad \mathbf{process}\ \text{User2}\ \texttt{[}acc\texttt{:id,}abd\texttt{:id]}\,\texttt{(}myid\texttt{:id)}\,\texttt{:}\,\texttt{()}\ \mathbf{raises}\,\texttt{[ ]}\ \mathbf{is} \\
\qquad\quad acc\texttt{(}\$1 \texttt{=> !}\mathbf{exit}\texttt{(}\$1 \texttt{=>} myid\texttt{))}\ldots\texttt{;}abd\texttt{(}\$1 \texttt{=> !}\mathbf{exit}\texttt{(}\$1 \texttt{=>} myid\texttt{));} \\
\qquad\quad \text{User2}\ \texttt{[}acc\texttt{,}abd\texttt{]}\,\texttt{(}\mathbf{exit}\texttt{(}\$1 \texttt{=>} myid\texttt{))}\,\texttt{[ ]}
\end{array}$$

We use rule 9.5 as follows:

$$\frac{\begin{array}{ll}
\mathcal{C} \vdash \mathsf{id} \Rightarrow \mathit{type} & \\
\mathcal{C} \vdash \mathtt{()} \Rightarrow \mathbf{type} & \\
\mathcal{C}, acc \Rightarrow \mathit{gate}\langle (\$1 \mathrel{\mathtt{=>}} \mathsf{id}) \rangle, & \\
\quad abd \Rightarrow \mathit{gate}\langle (\$1 \mathrel{\mathtt{=>}} \mathsf{id}) \rangle, & \\
\quad myid \mathrel{\mathtt{=>}} \mathsf{id}, & \\
\quad \mathrm{User2} \Rightarrow [\, \mathit{gate}\langle (\$1 \mathrel{\mathtt{=>}} \mathsf{id}) \rangle , \mathit{gate}\langle (\$1 \mathrel{\mathtt{=>}} \mathsf{id}) \rangle \,] & \\
\qquad\qquad (myid{:}\mathsf{id})\, [\ ] \rightarrow \mathit{exit} \llparenthesis\ \rrparenthesis & \vdash acc(\$1 \mathrel{\mathtt{=>}} \mathtt{!}\mathbf{exit}(\$1 \mathrel{\mathtt{=>}} myid)) \ldots ; \\
 & \quad abd(\$1 \mathrel{\mathtt{=>}} \mathtt{!}\mathbf{exit}(\$1 \mathrel{\mathtt{=>}} myid)) \ldots ; \\
 & \quad \mathrm{User2}\ [acc\,,abd]\,(\mathbf{exit}(\$1 \mathrel{\mathtt{=>}} myid))\,[\ ] \Longrightarrow \mathit{guarded} \llparenthesis\ \rrparenthesis
\end{array}}{\mathcal{C} \vdash \mathit{processDecl} \Longrightarrow (\mathrm{User2} \Rightarrow [\, \mathit{gate}\langle (\$1 \mathrel{\mathtt{=>}} \mathsf{id}) \rangle , \mathit{gate}\langle (\$1 \mathrel{\mathtt{=>}} \mathsf{id}) \rangle \,]\, (myid{:}\mathsf{id})\, [\ ] \rightarrow \mathit{guarded} \llparenthesis\ \rrparenthesis)}$$

The last premise is checked using the inference rules of actions, sequential composition, and Example 7.26 in Section 7.26.

9.1.5 Sequential declarations

A sequential declaration $D_1\ D_2$ is semantically correct in the context $\mathcal{C}$ if D_1 is correct in this context, and D_2 is correct in a context where the declarations in D_1 has been included. The sequential declaration returns the information produce by both declarations which has to be disjoint.

$$\frac{\begin{array}{r}
\mathcal{C} \vdash D_1 \Longrightarrow \mathcal{C}' \\
\mathcal{C},\mathcal{C}' \vdash D_2 \Longrightarrow \mathcal{C}''
\end{array}}{\mathcal{C} \vdash D_1\ D_2 \Longrightarrow \mathcal{C}',\mathcal{C}''} \tag{9.7}$$

9.2 Dynamic Semantics

A specification starts to be executed by using the following rule

$$\frac{\begin{array}{ll}
\vdash \textit{m-body} \longrightarrow \mathcal{E} & \\
\mathcal{E}_0, \mathcal{E} \vdash \mathbf{rename} & \\
\qquad\quad \mathbf{gate}\ G_1(\$1 \mathrel{\mathtt{=>}} \mathtt{?}\$arg) : T_1\ \mathbf{is}\ G_1(\$1 \mathrel{\mathtt{=>}} \mathtt{!}\$arg) & \\
\qquad\qquad \ldots & \\
\qquad\quad \mathbf{gate}\ G_m(\$1 \mathrel{\mathtt{=>}} \mathtt{?}\$arg) : T_m\ \mathbf{is}\ G'_m(\$1 \mathrel{\mathtt{=>}} \mathtt{!}\$arg) & \\
\qquad \mathbf{in}\ B & \xrightarrow{\mu(RN)} B'
\end{array}}{\vdash \mathit{specification} \xrightarrow{\mu(RN)} B'} \tag{9.8}$$

where $\mathcal{E}_0$ is the initial dynamic context including the constants, types, and functions declared in the predefined library of E-LOTOS.

The first premise states that the “execution” of the declarations in *m-body* produces the context $\mathcal{E}$ that consists of the information needed to execute the behaviour B.

The second premise states that the behaviour B has as first possible transition the transition $\xrightarrow{\mu(RN)}$ becoming B'. The **rename** is used in order to provide the type of the values that can go through the gates declared by the specification. Writing this **rename** here allows to do not write it in the process instantiation as is done in [Que98]. Now all gates are declared in a **rename** or **hide**, and, therefore, the values that go through gates are always of the correct type.

In the following sections we define how the different declarations in *m-body* are "executed," and the contexts that these declarations returns. The judgements are of the form

$$\vdash D \longrightarrow \mathcal{E}$$

meaning that declaration D returns context $\mathcal{E}$ when it is executed. $\mathcal{E}$ includes the information needed when a behaviour using the declaration D is executed.

9.2.1 Synonymous type

The synonymous type declaration **type** S **renames** T returns $S \equiv T$, that is, T is the definition of type S.

$$\frac{}{\vdash \textbf{type } S \textbf{ renames } T \longrightarrow (S \equiv T)} \tag{9.9}$$

9.2.2 Named record type

The named record type declaration **type** S **is** (RT) returns that (RT) is the definition of type S.

$$\frac{}{\vdash \textbf{type } S \textbf{ is } (RT) \longrightarrow (S \equiv (RT))} \tag{9.10}$$

9.2.3 New data type

The user defined new data type

$$\begin{array}{l} \textbf{type } S \textbf{ is} \\ \quad C_1(RT_1) \mid \ldots \mid C_n(RT_n) \end{array}$$

returns the type of the constructors, needed to know the type of values of type S.

$$\frac{}{\vdash \textbf{type } S \textbf{ is } C_1(RT_1) \mid \ldots \mid C_n(RT_n) \longrightarrow (C_1 \Rightarrow (RT_1) \to S, \ldots, C_n \Rightarrow (RT_n) \to S)} \tag{9.11}$$

9.2.4 Process declaration

A process declaration

$$\begin{array}{l} \mathit{processDecl} \quad \equiv \\ \qquad \textbf{process } \Pi \; [\, G_1 : T_1, \ldots, G_n : T_n \,] \; (\, V_1 : T'_1, \ldots, V_p : T'_p \,) \; : (RT) \\ \qquad\qquad \textbf{raises } [\, X_1 : T''_1, \ldots, X_n : T''_n \,] \\ \qquad \textbf{is } B \end{array}$$

returns the information related to the type and name of formal gates, parameters, and exceptions; and the body of the process.

$$\frac{}{\vdash \mathit{processDecl} \longrightarrow \Pi \Rightarrow \lambda[\, G_1 : T_1, \ldots, G_m : T_m \,](V_1 : T'_1, \ldots, V_p : T'_p)[\, X_1 : T''_1, \ldots, X_n : T''_n \,] \to B} \tag{9.12}$$

9.2.5 Sequential declarations

A sequential declaration $D_1\ D_2$ returns the information returned by both D_1 and D_2.

$$\frac{\begin{array}{c}\vdash D_1 \longrightarrow \mathcal{E}\\ \vdash D_2 \longrightarrow \mathcal{E}'\end{array}}{\vdash D_1\ D_2 \longrightarrow \mathcal{E}, \mathcal{E}'} \tag{9.13}$$

Bibliography

[BB93] J. C. M. Baeten and J. A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3:142–188, 1993.

[Ber93] Gérard Berry. Preemption in concurrent systems. In *Proceedings of FSTTCS 93*, volume 761 of *Lecture Notes in Computer Science*, pages 72–93, Berlin, 1993. Springer Verlag.

[BL93] Rezki Boumezbeur and Luigi Logrippo. Specifying telephone systems in LOTOS. *IEEE Communications Magazine*, pages 38–45, August 1993.

[Bri88] Ed Brinksma. *On the Design of Extended LOTOS, a Specification Language for Open Distributed Systems.* PhD thesis, University of Twente, November 1988.

[dFLL$^+$95] D. de Frutos, G. Leduc, L. Léonard, L. Llana, C. Miguel, J. Quemada, and G. Rabay. Time extended LOTOS. In *Working draft on Enhancements to LOTOS.* ISO/IEC JTC1/SC21/WG1, 1995.

[Dij65] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University Eindhoven, 1965.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer-Verlag, 1985.

[Flo94] Thomas L. Floyd. *Digital Fundamentals.* Macmillan Publishing Company, 1994.

[Gam90] Mark Gamble. The CCSDS protocol validation programme inter-agency testing using LOTOS. In Juan Quemada, Jose A. Mañas, and Enrique Vázquez, editors, *Proc. Formal Description Techniques III.* North-Holland, Amsterdam, Netherlands, November 1990.

[GH93] Hubert Garavel and Rene-Pierre Hautbois. Experimenting with LOTOS in the aerospace industry. In Teodor Rus and Charles Rattray, editors, *Theories and Experiences for Real-Time System Development*, Computing: Vol 2. World Scientific, 1993.

[Gro93] J. F. Groote. Transition system specifications with negative premises. *Theoretical Computer Science*, 118:263–299, 1993.

[GS96] Hubert Garavel and Mihaela Sighireanu. On the introduction of exceptions in LOTOS. In Reinhard Gotzhein and Jan Bredereke, editors, *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'96 (Kaiserslautern, Germany)*, pages 469–484. Chapman & Hall, October 1996.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[ISO88] ISO/IEC. *LOTOS—A formal description technique based on the temporal ordering of observational behaviour.* International Standard 8807, International Organization for standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[JT97] He Ji and Kenneth J. Turner. Extended DILL: Digital logic in LOTOS. Technical Report CSM-142, Department of Computing Science and Mathematics, University of Stirling, UK, November 1997.

[LDV99] Luis Llana-Díaz and Alberto Verdejo. Time and urgency in E-LOTOS. Submitted for publication, March 1999.

[LFHH] L. Logrippo, M. Faci, and M. Haj-Hussein. An introduction to LOTOS: learning by examples. University of Ottawa.

[LL94] Guy Leduc and Luc Léonard. A formal definition of time in LOTOS. ISO/IEC, 1994. Source: JTC1/SC21/WG1/Q48.6.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Mun91] Harold B. Munster. LOTOS specification of the MAA standard, with an evaluation of LOTOS. NPL Report DITC 191/91, National Physical Laboratory, Teddington, Middlesex, UK, September 1991.

[NS91] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Computer Aided Design*, volume 575 of *Lecture Notes in Computer Science*, pages 376–398, 1991.

[NS94] X. Nicollin and J. Sifakis. The algebra of timed processes, ATP: Theory and application. *Information and Computation*, 114:131–178, 1994.

[Pec92] Charles Pecheur. Using LOTOS for specifying the CHORUS distributed operating system kernel. *Computer Communications*, 15(2):93–102, March 1992.

[Pec94] Charles Pecheur. A proposal for data types for E-LOTOS. Technical report, University of Liège, October 1994. Annex H of ISO/IEC JTC1/SC21/WG1 N1349 Working Draft on Enhancements to LOTOS.

[Que98] Juan Quemada, editor. Final committee draft on Enhancements to LOTOS. ISO/IEC JTC1/SC21/WG7 Project 1.21.20.2.3., May 1998.

[RR86] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. In *Proc. ICALP 86*, pages 314–323. Springer-Verlag, 1986. LNCS 226.

[Sch95] S. A. Schneider. An operational semantics for timed CSP. *Information and Computation*, 116(2):193–213, 1995.

[TS94] Kenneth J. Turner and Richard O. Sinnott. DILL: Specifying digital logic in LOTOS. In Richard L. Tenney, Paul D. Amer, and M. Ümit Uyar, editors, *Proc. Formal Description Techniques VI*, pages 71–86. North-Holland, Amsterdam, Netherlands, 1994.

[vEVD89] Peter H. J. van Eijk, Chris A. Vissers, and Michel Diaz. *The Formal Description Technique LOTOS: Results of the ESPRIT/SEDOS Project.* North-Holland, New York, 1989.

[Yi91] Wang Yi. CCS + time = an interleaving model for real time systems. In J. Leach Albert, B. Monien, and M. Rodríguez, editors, *Proc. ICALP 91*, pages 217–228. Springer-Verlag, 1991. LNCS 510.