Computer Science
University of Manchester

A Process Algebraic Semantics for Core ELLA

Howard Barringer, Graham Gough,

Brian Monahan, Alan Williams

Technical Report UMCS-93-2-1

FORMAL VERIFICATION SUPPORT FOR ELLA

IED 4/1/1357

A Process Algebraic Semantics for Core ELLA*

Howard Barringer   Graham Gough   Brian Monahan†   Alan Williams ‡

Department of Computer Science§
University of Manchester
Oxford Road, Manchester, UK.
email: howard@cs.man.ac.uk

November 1994

†Brian Monahan is now with Harlequin Ltd., Alderley Edge

‡Alan Williams acknowledges additional support from DRA (Malvern) and Harlequin Ltd (Cambridge)

§The University of Manchester acknowledges support from the SERC via research grant GR/F 38174.

Abstract

We give a presentation of a trace-based semantics of Core ELLA, a covering subset of the commercial hardware description language ELLA, using a synchronous process algebra called *ELLA PROCESS ALGEBRA* (*EPA*). This report is a semantics reference manual for ELLA, giving a detailed definition of the behaviour of ELLA expressions on a construct by construct basis.

EPA is an instantiation of a generic Process Algebra Framework (*PAF*). It supports parameterised process calls, and has highly structured actions representing pairs of input/output maps labelling its transitions. The *EPA* process terms reprsenting ELLA are constructed compositionally following the structure of ELLA expressions. Primitive terms, corresponding to atomic operations in ELLA, are assembled using a small set of process operators. In particular the parallel composition operator models synchronous communication between components and is effected by a special product operation on actions. The *EPA* terms resulting from the compilation of ELLA expressions can be translated into a special *normal form*, which is essentially a symbolic state machine representation. This forms the basis for semi-automated high-level symbolic manipulation and reasoning tools.

The work has been conducted as part of a collaborative project sponsored by the DTI and SERC, involving Manchester, DRA (Malvern) and Harlequin Ltd. (Cambridge), with GEC Plessey Semiconductors (Swindon) as evaluators. The aim of the project has been to provide formal verification support tools which are amenable to the industrial hardware designer. Using the formal semantics as a basis, software tools for ELLA compilation, normalisation, (ground-term/symbolic) simulation, (enumerated/symbolic) verification and ELLA-to-ELLA rewriting have been implemented by the project partners, and embedded into an integrated ELLA Design and Verification Environment.

Contents

1 Introduction

This report is a reference manual giving the formal semantic definition of the hardware description language (HDL) ELLA[MC93]. It forms part of the reference material describing the ELLA language, together with the standard documents, i.e. language reference manual, tutorial and user guide[MC93, Com91]. The report does *not* discuss formal semantics in general — see [Sch86] for such a text — and it is intended to complement rather than replace the standard ELLA documentation. It has not been our intention to change the meaning of ELLA, but to capture its essence, representing the 'understood' semantics exhibited by the standard commercial ELLA compiler and simulator. There *are* differences between this semantics and the 'understood' semantics, but these arise due essentially to efficiency mechanisms introduced into the simulator — see §1.4.2. Although this report concentrates on ELLA, the techniques and semantic model used have wider application to other HDLs and digital systems, which will be the subject of future work.

One major feature of the work presented here is that the semantics covers *full* ELLA, even though it is syntactically a large language. The report defines the semantics of subset of the full language, called Core ELLA. However, Core ELLA is a *covering* subset, in the sense that more complex language features can be translated into (informally) equivalent Core ELLA expressions, using a set of software transformations. Any design tools based on the semantics for Core ELLA will therefore also cover the full language, and so will not place restrictions on the supported constructs available to the hardware designer. Considering only the Core simplifies the task of defining the semantics, because the number of language constructs that need to be addressed is greatly reduced. Of course, the full-to-Core ELLA translation is not defined formally, in the way that the semantics of Core ELLA is defined in this report.

This work has been conducted as part of a collaborative project sponsored by the DTI and SERC, involving Manchester, DRA (Malvern) and Harlequin Ltd. (Cambridge), with GEC Plessey Semiconductors (Swindon) as evaluators. Using the formal semantics as a basis, software tools for ELLA compilation, normalisation, (ground-term/symbolic) simulation, (enumerated/symbolic) verification and ELLA-to-ELLA rewriting have been implemented by the project partners, and embedded into an integrated ELLA Design and Verification Environment. The aim of the project has been to provide formal verification support tools which are amenable to the industrial hardware designer.

The reader is assumed to have a general mathematical or computing background. The report aims to be self-contained, but does assume that the reader has some familiarity with: sets, predicate logic, and VDM-style notation. A collection of appendices introduces much of the notation used, in more familiar terms. A knowledge of the ELLA language and process algebras would be advantageous.

Note that there have been other recent approaches to providing a formal semantics for ELLA. At Edinburgh, operational semantics are provided for two simplified versions of ELLA, called *micro*ELLA[Goo90a] and *pico*ELLA[Goo90b]. At Cambridge, a system is being developed to *semantically embed* ELLA into HOL[BGHT90], so that the HOL verification system and its associated tools can be employed. Following the work described in this report, Formal Systems (Europe) Ltd. were commissioned to indicate how Receptive Process Theory could be developed, to encode a subset of the ELLA semantics.

1.1 Overview of Document

This section contains introductory material to ELLA, the synchronous process algebra called *ELLA Process Algebra* (*EPA*) used for defining the semantics, and a brief overview of the nformal semantics of computing languages. In the rest of the document, section 2 describes *EPA* in more detail. Some of the definitions are relegated to Appendix B. The section also includes an outline of the transition rule semantics for *EPA* process terms (§2.2). A method for performing symbolic verification of *EPA* systems is given in §3. Section 4 describes the approach we have adopted for presenting the semantic definitions, and defines the environments to be used. The main section 5 presents the semantic definition of Core ELLA, using the 'horizontal' tabulation described in§ 4; the syntax, static and dynamic semantics definitions are given on a construct-by-construct basis. It first defines a set of auxiliary functions which are used throughout the definitions. Finally, section 6 gives a summary of the work and indicates future directions.

A set of appendices provides further reference material. Appendix A gives the complete composite syntax for Core ELLA, in a BNF-style. Appendix B completes the definition of the syntax and semantics of *EPA*, started in §2, including a BNF-style presentation of the composite syntax for *EPA*, a definition of the type and value universes for ELLA, and a definition of an interpretation for the ELLA action algebra, which maps *EPA* type and value expressions into elements of the ELLA type and value universes. Appendix C provides information on the notation used in this report. Appendix D describes in detail how the static semantic definitions are applied to a small ELLA-text, to determine whether it is well-defined; appendix E repeats the exercise for the dynamic semantics, applying the semantic definitions given in §5 to a small ELLA example. Appendix F gives the semantics for a *recursive let* language construct, an alternative way for introducing mutually recursive signal declarations to the ELLA `MAKE` / `JOIN` approach given in §5.8.16.1. However, note that the recursive let is not part of the ELLA language. Finally, an index giving the location of the definitions of all static and dynamic semantic operations is given, so that these can be found quickly.

Although this report is large, most sections are self-contained and can be read independently. A suggested reading order is as follows:

1. Read the rest of §1 below, together with the following introductory sections 2 and 4.
2. Read Appendix C giving notational conventions, if these are unfamiliar.
3. Now concentrate on the tutorial examples in Appendix D and Appendix E, for the static and dynamic semantics respectively. Refer to semantic definitions, notational conventions, and ELLA syntax as necessary.
4. Consult the rest of the semantic definitions in §5 as required.

Note that although the static semantics is a pre-condition of the dynamic semantics, the static and dynamic semantics definitions for each ELLA construct can be read independently.

1.2 Acknowledgements

We would like to thank John Morison, Mike Hill and Peter Ryan (DRA, Malvern), and Matthew Arcus (Harlequin Ltd, Cambridge), for their valuable input to this paper.

1.3 ELLA

ELLA is a hardware design system created by DRA (Malvern, England) for high-level description of complex VLSI designs[MC93, MPT85] then subsequently developed as a commercial product by Praxis

Systems[Com91], and now available in the public domain[1]. The system includes the ELLA language and compiler, an application support environment, simulation and other tools. ELLA is thought of as a parallel language in which user defined processes/functions are used to model the behaviour of hardware structures — a number of characteristics ensure that any ELLA expression is realisable in hardware:

- explicit simple timing model
- finite types
- no recursive functions

Users define datatypes, to model signal values, and then build up complex systems by connecting together previously defined functions. ELLA uses a simple timing model, and is effectively deterministic, meaning that a deterministic state-based semantic model can often be used, with standard decidable equivalence procedures. In this report, we shall concentrate on the set of basic features included in Core ELLA.

1.4 Formal Semantics

The formal semantics of a language gives a meaning — typically a mathematical value — to every syntactically correct language expression. In the case of ELLA, the behaviour of each construct is given by a *synchronisation tree*[Win84] derived from the *EPA* term representing an ELLA expression , using a collection of transition rules. For deterministic systems, this can be reduced to an equivalent set of input/output traces, or input/output language[2]. A trace is a sequence of map pairs, containing one map from input channels to values, and another from output channels to values. In certain cases, for example if there is only a single input and output, then the channels may be dropped and the trace reduces to a sequence of value-pairs. A useful formal semantic definition will be able to express the meaning of each language construct in a natural, intuitive way — the users of the semantics will have a wide range of backgrounds, not necessarily in formal methods. The resulting semantic objects should ideally not be larger than that of the source language expressions. For example, it would be possible to define the ELLA semantics by simply stating the synchronisation tree or trace set for each component, and then to combine these in a suitable way when a composite ELLA expression is encountered. However, this representation is at a very low level, containing no structure, and will exhibit exponential growth with respect to the size of the corresponding ELLA expression. We adopt an approach where the semantic objects defined *represent* the synchronisation tree or trace sets in an efficient way.

1.4.1 Motivation

The main reason for developing of the formal semantics of ELLA has been to provide a foundation for formal reasoning tools, such as equivalence checkers and behaviour-preserving transformation tools. The *EPA* model is at an intermediate level, still retaining the design structure inherent in the original ELLA description. It provides a good basis for formal reasoning tools, which can exploit the structure in the *EPA* terms presented to them. Tractable methods for checking the equivalence of two *EPA* processes,

[1] The commercial ELLA system is currently marketed by R-Cube Systems, Bristol, England. The public domain version is available by anonymous ftp from Imperial College, London (`src.doc.ic.ac.uk:packages/ELLA`)

[2] For the (deterministic) examples used in this report, then the semantics will be presented in terms of traces in this report.

which use a *state bisimulation* rule operating on structured abstract terms, are outlined in §3, and are discussed further in [BGMW94]. This situation should be contrasted with previous formulations of the ELLA semantics, based on input/output automata — see §1.5 below.

However, the development of a formal semantics for ELLA also has benefits in other areas of design:

- *Language Definition*: the ELLA user can use the semantics in just the same way as the language reference manual and tutorial are used now, in order to learn the form and meaning of ELLA constructs. Of course the formal semantic definition has an advantage over an informal English language description because it is precise and unambiguous.
- *Tool Development*: up until now, the 'understood' behaviour of an ELLA expression has been given by the standard simulator[Com91]. The only way of determining whether a tool, such as a new simulator or compiler, is correct has been by comparing its behaviour with that of the standard simulator. With the introduction of the formal semantics, then there is the potential for *proving* that the tool has the correct behaviour, although of course this would currently be an extremely large undertaking. It has also been found that the semantics for ELLA has been beneficial in the rapid and accurate implementation of the compilation tools.
- *Language Transformation*: if a translation between ELLA and another style of description — perhaps another HDL — is required, then the the formal semantics (of both ELLA and the target format) would be used to check the correctness of the translation.

1.4.2 Outline of Approach

A semantic definition takes as a source each language construct and associates with it an element in the target *semantic domain*, either by using a valuation function which maps the construct to its denotation, or by supplying operational rules, which interpret each construct. For ELLA, the semantics is defined in distinct stages, using both of these techniques:

1. A (full) ELLA expression is translated into an(informally) equivalent Core ELLA expression by a suite of software transformation routines.
2. A Core ELLA parser/compiler constructs a syntax tree representing the Core ELLA expression, checking that it is well-formed. The annotated parse tree contains type information about each ELLA expression, for use in (3).
3. The Core ELLA *closure* is then compiled into a *context* of *EPA* process terms, proceeding in a natural and intuitive way, following the structure of each ELLA expression. Basic *EPA* terms represent atomic ELLA expressions, and the semantics of more complex expressions is defined by the composition of the *EPA* terms representing the ELLA components.
4. The *EPA* context, together with a collection of standard (generic) *EPA* definitions, are now interpreted, using a set of transition rules to evolve each term, and an action algebra to interpret value expressions. The result is a *synchronisation tree*, from which the input/output language for the original (full) ELLA expression can be extracted, when the system is deterministic.

The semantic definitions for (2), (3) and (4) all require the use of various *environments*, which store type information about language constructs, previously constructed definitions, and binding/assignment of language variables — see §4.1 below. A fuller discussion of these general issues can be found in [Sch86], for example.

The well-formedness check in (2) is used as a pre-condition for applying the (dynamic) semantic operations in (3). Additionally, the interpretation of the *EPA* context assumes that the *EPA* terms are well-defined — a separate well-formedness check is really required for *EPA*. However, it is assumed that an *EPA* term, produced by the dynamic semantic definitions from a well-formed ELLA expression, will itself be well-formed. Consequently, those pre-conditions for semantic operations which are implied by the well-formedness of the ELLA expression under consideration, are omitted from the definitions.

There is a very strong correspondence between the behaviour given by the formal semantics and that given by the (standard) ELLA simulator — the 'understood' semantics. However, the two behaviours are not precisely equivalent due to efficiency and other considerations in the operation of the simulator, most notably concerning the 'query' value in ELLA. This represents under-determined, over-determined and 'don't care' values in the simulator. Under certain circumstances the formal semantics will give a single (determined) value for an output, when the standard simulator returns 'query' — the simulator propagates the 'query' value in a pessimistic fashion. In other cases, for example when an attempt is made to assign inconsistent values to a node at the same time, the formal semantics will halt, whereas the standard simulator gives a continuation. The issue of simulator correctness with respect to the formal semantics is addressed in [Rya93].

The main part of this report is concerned with defining the static semantics for each ELLA language construct, which determines when the construct is well-formed, and then defining the *EPA* term which denotes its dynamic semantics. The transition-based semantics for *EPA* and the action algebra interpretation are covered in §2 and Appendix B. The composite syntax for both ELLA and *EPA* are presented using BNF in Appendix A and Appendix B.

1.5 Background

In previous work, we have developed and explored input/output automata (IOAs) as semantic models for a much simplified form of ELLA, called Boolean Kernel ELLA (*BKE*)[MDED91], which only allows rows of booleans as values. On the basis of this work, a compiler was developed that produced literal representations of these models, which were then used for verification.

The underlying semantic model was then extended to cover the additional datatypes and language constructs from Kernel ELLA and then the larger subset, Core ELLA. This semantics employed the same basic automaton constructions used in the *BKE* semantics, but allowed a much more complex structure of the input/output signatures. These were maps whose codomains represent the richer collection of datatypes available in both Core and Kernel ELLA. In order to set up the machinery for handling these datatypes we also needed to present the static semantics of the language explicitly, which was not done for *BKE*[3]. The IO automaton semantics for Core ELLA appears in [BGMW92].

Verification methods employing fully enumerated IOAs have always been known to have limitations due to 'state explosion' or even 'transition explosion'. In order to avoid this, the *EPA* approach described in this report was adopted. This uses an *EPA* process calculus representation of the semantic objects in Core ELLA, which can be considered to be at an intermediate level between the ELLA text and IO automata.

[3] Similar work on static semantics of Core and Kernel ELLA is contained in [HM92], which describes a transformation from Core to Kernel ELLA. This results in a presentation of the static semantics that is more complex than would have been necessary if it had been given separately.

With process terms as semantic objects, then algebraic methods such as term-rewriting can be used for verification. It is also possible to translate the terms into other representations, such as IO automata, if desired. Note that although the representation of the semantics has changed in the move from IOAs to *EPA* terms, the underlying semantics for Core ELLA remains unaltered.

1.6 Examples of Core ELLA

As an introduction to the ELLA language, three very simple examples of ELLA descriptions for hardware elements are given, starting with a two-valued enumerated type called `bool`, with values `f, t`[4]. The function declaration `NOR` shows one of the basic methods of describing functional behaviour in ELLA, by using a `CASE` statement. This has the usual meaning: if one of the chooser sets in the case-items (e.g `(f,f)` here), matches with the value of the chooser `(x,y)` then the value supplied by the unit on the right-hand side of that case-item (i.e., `t`) is chosen — otherwise the `ELSE` clause (i.e., `f`) is chosen. All of the chooser sets must be disjoint, and there is no delay implied — the output appears within the same transition as the corresponding inputs. Function `NOR` therefore represents the standard logic 'nor' of its inputs.

```
TYPE bool = NEW (f | t).

FN NOR = (bool: x, bool: y) -> bool:
  CASE (x,y) OF
     (f,f) : t
  ELSE f
  ESAC.

FN DEL = (bool) -> bool:
  IDELAY(f,1).
```

```
FN DELNOR = (bool: x, bool: y) -> bool:
BEGIN
  MAKE NOR: o1.
  MAKE DEL: o2.
  JOIN (x,y) -> o1.
  JOIN o1 -> o2.
  OUTPUT o2
END.
```

The delay function `DEL` shows how simple delays are defined in ELLA. Here the delay is unitary, so that at a time t the output value is the same as that input at time $t-1$. The initial value here is `f`. The function `DELNOR` shows how previously defined components are 'connected' in ELLA. First, instantiations of `NOR` and `DEL` are created by the two `MAKE` statements. Then these are composed by the `JOIN`, with the output of the `NOR` instantiation called `o1` supplying the input of the `DEL` instantiation called `o2`. Finally the `OUTPUT` statement determines the output value of `DELNOR` — in this case it is simply the output of the `DEL` instantiation. The output of `DELNOR` will initially be `f`, and then at each subsequent time step, it will be the 'nor' of the two values input at the previous time step. The `DELNOR` function can then be used to approximate the behaviour of a nor-gate component in hardware, which exhibits a notional delay of one (ELLA) time-unit.

1.7 Examples of *EPA*

Descriptions of the three examples introduced above — the NOR gate, unit-delay and delay NOR-gate — are now given using *EPA*, to indicate how the semantics of ELLA will be defined in this report. First, an outline of the meaning of an *EPA* term is presented.

[4]Note that the interpretation of the enumerated type is entirely user-defined. Of course in this case the intention is that `bool` represents the booleans, with the values `f` and `t` representing logical false and true; this will only be the case if the user defines operations over `bool` in an appropriate way.

An *EPA* process term evolves into a new term by performing *actions*; here an action describes the value expressions associated with input and output *channels*, which are name/value pairs. The channel names effectively provide labels for values, which must belong to a particular type. Processes are composed using a product operation, which combines component actions via an action product. This is defined essentially so that if an output channel in one component has the same name as an input channel in the other component, i.e., the two channels are to be 'connected', then a new valid action can only be formed when the channels are associated with equal values. The input channel is then removed, its value now 'supplied' by the output. This product models simultaneous communication between processes. Clearly, there must be restrictions on the names that channels can take in a valid action: outputs must all have distinct names, and the sets of inputs and outputs must be disjoint. An action can also be modified by hiding certain channels.

(a) Consider a process definition for a process called Nor_P representing the delay-less NOR-gate described above in ELLA:

$$\text{process } Nor_P(c_1, c_2\colon bool \;/\; c_3\colon bool) \underline{\triangle}$$
$$\quad \Box\, a,b\colon bool \;\cdot\; c_1(a), c_2(b) \;/\; c_3(nor(a,b)) :: Nor_P(c_1, c_2 \;/\; c_3)$$
$$\text{function } nor(a,b\colon bool) : bool \;\underline{\triangle}\; \textbf{case}\ (a,b) : \{(\mathtt{f},\mathtt{f})\} \rightarrow \mathtt{t}\ \textbf{else}\ \mathtt{f}\ \textbf{end}$$
$$\text{type } bool \,\underline{\triangle}\, \{\mathtt{f},\ \mathtt{t}\}$$

The left hand side of process declares two input channel variables c_1, c_2, and an output channel c_3, which range over channels of type *bool*. When a call to Nor_P is made, then c_1, c_2, c_3 will be substituted by actual channel parameters used in the call (see below). A call to the process will have the following behaviour: the part '$\Box\, a,b\colon bool \;\cdot$ ' means that values substituted for a, b are to be non-deterministically chosen from the set *bool* at every transition — in other words, a, b will be used here to range over the possible input values for the process. These are then used in the remaining term, which has a *prefix* action $c_1(a), c_2(b) \;/\; c_3(nor(a,b))$ and a *continuation* process term $Nor_P(c_1, c_2 \;/\; c_3)$. The action is performed before proceeding to the continuation. The values chosen for a, b are bound to the input channels substituted for c_1, c_2 — the process is *input deterministic*, because there is a unique output value and continuation for every input value. The value given by function $nor(a,b)$ is simultaneously placed on the output channel substituted for c_3 — the case statement in this function definition clearly evaluates to the logical 'nor' of its two arguments. The new process is just a call to itself. The set expression for *bool* represents the usual two-valued type $\{\mathtt{f},\mathtt{t}\}$, given by the type definition above. For example, the following sequence of (ground) transitions would be permitted by a call to $Nor_P(x, y \;/\; o1)$, where $x, y, o1$ represent concrete channels of type *bool*, with distinct channel names:

$$Nor_P(x,y \;/\; o1) \xrightarrow{x(\mathtt{t}),y(\mathtt{f})\;/\;o1(\mathtt{f})} Nor_P(x,y \;/\; o1) \xrightarrow{x(\mathtt{f}),y(\mathtt{t})\;/\;o1(\mathtt{f})}$$
$$Nor_P(x,y \;/\; o1) \xrightarrow{x(\mathtt{t}),y(\mathtt{t})\;/\;o1(\mathtt{f})} Nor_P(x,y \;/\; o1) \xrightarrow{x(\mathtt{f}),y(\mathtt{f})\;/\;o1(\mathtt{t})}$$
$$Nor_P(x,y \;/\; o1) \xrightarrow{x(\mathtt{f}),y(\mathtt{t})\;/\;o1(\mathtt{f})} Nor_P(x,y \;/\; o1) \xrightarrow{x(\mathtt{f}),y(\mathtt{f})\;/\;o1(\mathtt{t})}$$
$$Nor_P(x,y \;/\; o1) \xrightarrow{x(\mathtt{t}),y(\mathtt{t})\;/\;o1(\mathtt{f})} Nor_P(x,y \;/\; o1)$$

At the first transition the process call $Nor_P(x,y \,/\, o1)$ performs the action $x(\mathtt{t}),y(\mathtt{f}) \,/\, o1(\mathtt{f})$, and evolves into the process call $Nor_P(x,y \,/\, o1)$, which happens to be identical in this case to the original. The action binds the logic true and false values to input channels x and y respectively, and binds the logical 'nor' of these values, i.e., $\mathtt{f}$, to the output channel $o1$ — this is the result of interpreting the action algebra expression $nor(\mathtt{t},\mathtt{f})$. The sequence of transitions expanded so far shown represents the trace:

$$[(x(\mathtt{t}),y(\mathtt{f}) \,/\, o1(\mathtt{f})),\ (x(\mathtt{f}),y(\mathtt{t}) \,/\, o1(\mathtt{f})),\ (x(\mathtt{t}),y(\mathtt{t}) \,/\, o1(\mathtt{f})),\ (x(\mathtt{f}),y(\mathtt{f}) \,/\, o1(\mathtt{t})),$$
$$(x(\mathtt{f}),y(\mathtt{t}) \,/\, o1(\mathtt{f})),\ (x(\mathtt{f}),y(\mathtt{f}) \,/\, o1(\mathtt{t})),\ (x(\mathtt{t}),y(\mathtt{t}) \,/\, o1(\mathtt{f}))\ \ldots]$$

from the input/output language represented by Nor_P.

(b) A unit delay for data of any type *ty*, where *ty* is a variable ranging over possible concrete datatypes, can be represented by a *generic* process definition $udel_{[ty]}$ which has a 'state' variable w:

$$\text{process } udel_{[ty]}(w\colon ty \,;\, c_1\colon ty \,/\, c_2\colon ty) \mathrel{\underline{\triangle}} \square\, v\colon ty \,\cdot\, c_1(v) \,/\, c_2(w) :: udel_{[ty]}(v \,;\, c_1 \,/\, c_2)$$

At every transition, $udel_{[ty]}$ chooses a value for $v \in ty$ for input channel c_1, and binds the current state value w to output c_2. It then evolves into a recursive call, which takes the value assigned to v just received on its input; this will be output at the next transition, thus effecting a unit delay. For example, the process definition Del_P will represent the ELLA unit delay `DEL` for the booleans *bool*, with initial value `f`:

$$\text{process } Del_P(c_1\colon bool \,/\, c_2\colon bool) \mathrel{\underline{\triangle}} udel_{[bool]}(\mathtt{f} \,;\, c_1 \,/\, c_2)$$

i.e. the process $udel_{[bool]}$ is called with $w = \mathtt{f}$. A typical transition sequence represented by $Del_P(z \,/\, o2)$, where $z, o2$ represent concrete channels of type *bool* with distinct channel names, would be as follows:

$$Del_P(z \,/\, o2) \xrightarrow{z(\mathtt{f}) \,/\, o2(\mathtt{f})} udel_{[bool]}(\mathtt{f} \,;\, z \,/\, o2) \xrightarrow{z(\mathtt{t}) \,/\, o2(\mathtt{f})} udel_{[bool]}(\mathtt{t} \,;\, z \,/\, o2) \xrightarrow{z(\mathtt{f}) \,/\, o2(\mathtt{t})}$$
$$udel_{[bool]}(\mathtt{f} \,;\, z \,/\, o2) \xrightarrow{z(\mathtt{t}) \,/\, o2(\mathtt{f})} udel_{[bool]}(\mathtt{t} \,;\, z \,/\, o2) \xrightarrow{z(\mathtt{t}) \,/\, o2(\mathtt{t})} udel_{[bool]}(\mathtt{t} \,;\, z \,/\, o2)$$
$$\xrightarrow{z(\mathtt{f}) \,/\, o2(\mathtt{t})} udel_{[bool]}(\mathtt{f} \,;\, z \,/\, o2) \xrightarrow{z(\mathtt{f}) \,/\, o2(\mathtt{f})} udel_{[bool]}(\mathtt{f} \,;\, z \,/\, o2) \cdots$$

The compact representation of the semantics of ELLA becomes evident for *EPA* terms representing more complex structures; for example, consider the *EPA* term representing a unit delay when the cardinality of the data set *ty* is large.

(c) To represent the ELLA unit-delay-Nor gate `DELNOR`, then the two process terms Nor_P and Del_P must be combined to evolve in synchrony using a composition operator $|$, with a common channel c_4 used for the output of Nor_P and the input of Del_P, which is then hidden, indicated by the term '$(\_)\backslash\{c_4\colon bool\}$'.

$$\text{process } DelNor_P(c_1,c_2\colon bool \,/\, c_3\colon bool) \mathrel{\underline{\triangle}} \Big(Nor_P(c_1,c_2 \,/\, c_4) \mid Del_P(c_4 \,/\, c_3) \Big)\backslash\{c_4\colon bool\}$$

At each step, the actions from the component processes are combined with the action product operator '$\star$', which forms a valid composite action only when the values bound to like channel names, i.e. c_4, are in agreement. The continuation term is then the product of the component continuation terms. For example, the following is a (short) trace represented by $\mathit{DelNor}_P(x,y \;/\; o2)$, where $x,y,o2$ are distinct concrete channels of type *bool*:

$$
\begin{aligned}
&\mathit{DelNor}_P(x,y \;/\; o2) \xrightarrow{x(\mathtt{t}),y(\mathtt{f}) \;/\; o2(\mathtt{f})} \Big(\; \mathit{Nor}_P(x,y \;/\; o1) \;\mid\; \mathit{udel}_{[\mathit{bool}]}(\mathtt{f}\;;\; o1 \;/\; o2) \;\Big)\backslash\{o1\} \\
&\quad \xrightarrow{x(\mathtt{f}),y(\mathtt{f}) \;/\; o2(\mathtt{f})} \Big(\; \mathit{Nor}_P(x,y \;/\; o1) \;\mid\; \mathit{udel}_{[\mathit{bool}]}(\mathtt{t}\;;\; o1 \;/\; o2) \;\Big)\backslash\{o1\} \\
&\qquad \xrightarrow{x(\mathtt{f}),y(\mathtt{f}) \;/\; o2(\mathtt{t})} \Big(\; \mathit{Nor}_P(x,y \;/\; o1) \;\mid\; \mathit{udel}_{[\mathit{bool}]}(\mathtt{t}\;;\; o1 \;/\; o2) \;\Big)\backslash\{o1\} \;\cdots
\end{aligned}
$$

Each transition label is the result of interpreting the action product of the component actions, and then performing any channel hiding that is required. For example, the first transition label is a result of performing the following action product:

$$(x(\mathtt{t}),y(\mathtt{f}) \;/\; o1(\mathtt{f})) \star (o1(\mathtt{f}) \;/\; o2(\mathtt{f}))$$

where the first element comes from expanding the Nor_P process call, and the second element from expanding the Del_P call. The channel $o1$ is then hidden in a separate operation to produce the action as shown. Note, however, that an action product:

$$(x(\mathtt{t}),y(\mathtt{f}) \;/\; o1(\mathtt{f})) \star (o1(\mathtt{t}) \;/\; o2(\mathtt{f}))$$

would result in an *improper* action, because the component actions do not agree on the values assigned to the channel $o1$, and therefore cannot form part of a trace for DelNor_P.

2 *EPA*: ELLA Process Algebra

The semantics of (full) ELLA is defined by first performing software transformations, which take complex full ELLA constructs into corresponding Core ELLA expressions. A formal translation then maps Core ELLA-text into *EPA* process terms. These are interpreted, using the transition rule semantics for *EPA*, outlined below in §2.2, and the semantics for the ELLA action algebra, resulting in a *synchronisation tree*[Win84]. This report focusses on defining the formal translation from Core ELLA-text to *EPA* process terms; the interpretation for *EPA* is also given. For details of the full-to-Core ELLA transformations see [HM92].

The *EPA* semantics for ELLA is derived from the previous IO automaton-based version outlined above in §1.5 — the semantics for the basic ELLA components are now given as process definitions, and then the causal product and hiding operations are defined for *EPA*. Each ELLA function is mapped by the semantics into an *EPA* process definition. The result is a sequence of definitions for processes, types, and value expressions.

A collection of term languages is required to represent *EPA* — terms for processes, types, sets, value expressions and logical expressions are needed. Here, an overview of only the process terms is given — the remaining definitions are included in Appendix B.

2.1 *EPA* Term Language

In the following definitions, note that as usual, variables ranging over syntactic structures are introduced at the place of their definition, but may appear in other definitions. For example, *defn* ranges over *EPA* definitions, but is used previously in the *Context* syntax.

Context is a sequence of definitions of semantically relevant entities such as *process terms, types, and data functions*:

$$\mathit{Context} ::= \mathit{defn}_1 \ \cdots \ \mathit{defn}_k$$

General Definitions introduce named entities for use in forming term structures of various kinds. A definition takes the form:

$$\mathit{Defn} \;=\; \mathit{ProcDefn} \mid \mathit{TypeDefn} \mid \mathit{DataFunDefn} \mid \mathit{DataConstDefn} \mid \mathit{SetDefn} \mid \mathit{PredDefn}$$

These are *process term, type, data function, data constant, set* and *predicate* definitions.

***EPA* Process Definitions** take the form:

$$\begin{aligned} \mathit{ProcDefn} \quad &::= \quad \textsf{process}\, \mathit{pabs}(\mathit{vdecls}\ ;\ \mathit{icdecls}\ /\ \mathit{ocdecls}) \underline{\triangle}\, P \\ \mathit{pabs} \in \mathit{ProcAbstract} \quad &= \quad \mathit{ProcName} \mid \mathit{GenericProcAbstract} \\ \mathit{GenericProcAbstract} \quad &::= \quad \mathit{pnm}_{[v_1,\, \cdots\, ,v_k]} \end{aligned}$$

The left hand side of a process definition consists of a *process abstract* $pabs \in ProcAbstract$, followed by a list of formal parameters, divided into three groups: the first group *vdecls* are value-variable declarations which are assigned when the process is called. The second group *icdecls* are typed *input channel variable declarations*, and the third group *ocdecls* are typed *output channel variable declarations*. Note that in certain cases, the first and/or second groups may be empty, because the process definition does not require any value-variables or input channels.

The process abstract can either simply be a process name $pnm \in ProcName$, or it can be a more structured *generic process abstract*, containing a name and a list of arguments $v_1, \cdots, v_k \in Var$ which are supplied when an instantiation to the generic process is made. A generic process abstract could be viewed simply as a 'structured' process label or name. However, by introducing generic processes, reasoning about generic properties of these can be performed within *EPA*, instead of having to consider each instantiation separately. As a simple example, one property of the unit-delay generic process $udel_{[ty]}$, defined above, is that its output is always the same as the input value on the previous time-step; this property is independent of the type *ty* of data being delayed.

Typed channel declarations are simply declarations of typed channel variables:

$$
\begin{aligned}
icdecls, ocdecls \in ChannelDeclSeq \quad &::= \quad tcd_1, \ \cdots \ , tcd_n \\
tcd \in TyChanDecl \quad &::= \quad cv : te
\end{aligned}
$$

where $cv \in ChannelVarName$. Value variable declarations are declarations of data expression variables, which range over a particular type:

$$
\begin{aligned}
vdecls \in DataExprVarSeq \quad &::= \quad tvd_1, \ \cdots \ , tvd_n \\
tvd \in DataExprVarDecl \quad &::= \quad v : te
\end{aligned}
$$

where $v \in DataExprVar$.

A process term in *EPA* has the following structure:

$$
\begin{aligned}
P \in ProcExpr \quad ::= \quad & ae :: P && \textit{Prefixing} \\
| \quad & pabs(vs \ ; \ cs_1 \ / \ cs_2 \ ) && \textit{ProcCall} \\
| \quad & (P_1 \ | \ \cdots \ | \ P_{n+2}) && \textit{Product} \\
| \quad & (\ P \ )\backslash ch\text{-}set && \textit{Hiding} \\
| \quad & \Box xs \ \bullet \ P && \textit{ProcChoice}
\end{aligned}
$$

An *action expression* takes the form:

$$
\begin{aligned}
ae \in ActionExpr \quad &::= \quad io\text{-}ae_1 \ / \ io\text{-}ae_2 \\
io\text{-}ae \in IOActionExpr \quad &::= \quad be_1, \ \cdots \ , be_{n+1} \\
be \in BindingExpr \quad &::= \quad ce \ (\ ve \ )
\end{aligned}
$$

It consists of a pair of sequences of channel bindings, one for inputs and one for outputs. A binding associates a channel with a value expression. Note that the comma-separated list used in the *EPA* binding sequences is a convenient shorthand for *action summation*, introduced in the next section:

$$be_1, \cdots, be_{n+1} = be_1 \oplus \cdots \oplus be_{n+1}$$

A *channel expression* can either be a variable or a typed channel name:

$$\begin{aligned} ce \in \mathit{ChannelExpr} &= \mathit{ChannelVarName} \mid \mathit{ChannelLiteral} \\ \mathit{ChannelLiteral} &::= \mathit{chnm} : \mathit{te} \end{aligned}$$

Process calls take sequences of data values and channels as arguments:

$$\begin{aligned} vs \in \mathit{ValueArgs} &::= ve_1, \cdots, ve_n \\ cs \in \mathit{ChannelArgs} &::= ce_1, \cdots, ce_n \end{aligned}$$

The value expressions contain effectively the current state information for the process, and the channel arguments allow for input/output renaming. Note that the value argument and/or input channel lists may be empty.

The Hiding combinator is used to *hide* a set of channels within a process expression:

$$\mathit{ch\text{-}set} \in \mathit{ChannelSet} ::= \{tcd_1, \cdots, tcd_{k+1}\}$$

Note that the channel variables are *declared* in *ch-set* and are binding within the hiding expression.

The Process choice operator allows a choice of process expression to be made, one for each assignment of variables from a sequence of choice arguments:

$$\begin{aligned} xs \in \mathit{ChoiceArgs} &::= cv_1, \cdots, cv_k \\ cv \in \mathit{ChoiceVar} &::= v : \mathit{set\text{-}expr} \end{aligned}$$

where $v \in \mathit{DataExprVar}$.

2.2 Transition-Based Semantics for *EPA*

The *EPA* transition rules will now be given for *EPA* terms resulting from the compilation of ELLA expressions. Each rule will operate in an environment (σ, γ), where σ is an *EPA* context, containing type, function and process definitions, and γ is an *assignment* environment, a map from *EPA* variables to their currently assigned value:

$$\begin{aligned} \sigma &\subseteq \mathit{Context} \\ \gamma &= \mathit{EPAVar} \xrightarrow{\text{map}} \mathit{ALL\text{-}TYPE} \cup \mathit{ALL\text{-}VALUE} \\ \mathit{EPAVar} &= \mathit{DataExprVar} \cup \mathit{TypeExprVar} \end{aligned}$$

A function $M\text{-}[\![\, expr \,]\!](\sigma, \gamma)$ interprets each value expression *expr* with respect to the environment (σ, γ) to produce a value in $\mathit{ALL\text{-}TYPE} \cup \mathit{ALL\text{-}VALUE}$, where *ALL-TYPE* is the type universe for ELLA and *ALL-VALUE* is the value universe, defined in Appendix B. It is assumed that there is a suitable characterisation of the *equivalence* of such values, so that matching of, for example, value expressions required here can be performed. Action *expressions* $ae \in \mathit{ActionExpr}$ will therefore be interpreted to produce concrete actions $a \in \mathit{Act}$ as follows:

$$
\begin{array}{rcl}
a \in \mathit{Act} & ::= & \mathit{io\text{-}act}_1 \ / \ \mathit{io\text{-}act}_2 \\
\mathit{io\text{-}act} \in \mathit{IOAct} & ::= & bd_1, \ \cdots, bd_{n+1} \\
bd \in \mathit{Binding} & ::= & ce \ (\ v \ )
\end{array}
$$

where $ce \in \mathit{ChannelExpr}$, and $v \in \mathit{ALL\text{-}VALUE}$.

A transition relation $\longrightarrow_C$ is defined for *closed process terms* — terms in which there are no occurrences of free variables. Two additional special actions are defined: a *zero action* $\boxtimes$ which represents an illegal action for *EPA*, and an *empty action* ε, which is legal but does not contain any bindings. Actions within $\mathit{Act} \cup \{\varepsilon\}$ are called *proper actions* and $\boxtimes$ is an *improper action*. The semantics of ELLA is then taken to be the synchronisation tree, containing only *proper* actions, obtained from applying the following transition rules to the *EPA* process term representing the ELLA expression. For input deterministic processes, then the semantics is an equivalent set of input/output traces — see §5.10.1. The semantics of closed terms is given as the *least* transition relation between closed terms, indexed by actions from $\mathit{Act} \cup \{\boxtimes, \varepsilon\}$, satisfying the following rules:

Let $ae, ae_1, ae_2 \in \mathit{ActExpr} \cup \{\varepsilon\}$, $\alpha, \beta \in \mathit{Act} \cup \{\varepsilon\}$, $x \in \mathit{Var}$, $pnm \in \mathit{ProcName}$, $set \subseteq \mathit{TypeExpr}$, $P, P', P_1, P'_1, P_2, P'_2 \in$ *ProcExpr*:

Vacuum

$$\frac{}{(\sigma,\gamma) \vdash_{\mathit{EPA\text{-}TR}} P \xrightarrow{\boxtimes}_C P'}$$

Prefixing

$$\frac{\alpha = M\text{-}⦃\ ae\ ⦄(\sigma,\gamma)}{(\sigma,\gamma) \vdash_{\mathit{EPA\text{-}TR}} ae :: P \xrightarrow{\alpha}_C P}$$

ProcChoice

$$\frac{\begin{array}{ll}\text{(a)} & v \in M\text{-}⦃\ set\ ⦄(\sigma,\gamma) \\ \text{(b)} & (\sigma, \gamma \dagger \{x \mapsto v\}) \vdash_{\mathit{EPA\text{-}TR}} P \xrightarrow{\alpha}_C P'\end{array}}{(\sigma,\gamma) \vdash_{\mathit{EPA\text{-}TR}} \Box x\colon \ set \ \bullet \ P \xrightarrow{\alpha}_C P'}$$

Hiding

$$\frac{(\sigma,\gamma) \vdash_{\mathit{EPA\text{-}TR}} P \xrightarrow{\alpha}_C P'}{(\sigma,\gamma) \vdash_{\mathit{EPA\text{-}TR}} P \backslash \mathit{ch\text{-}set} \xrightarrow{\mathit{hide\text{-}act}(\alpha, \mathit{ch\text{-}set})(\sigma,\gamma)}_C P' \backslash \mathit{ch\text{-}set}}$$

Product

$$\frac{(\sigma,\gamma) \vdash_{\mathit{EPA\text{-}TR}} P_1 \xrightarrow{\alpha}_C P'_1, \quad (\sigma,\gamma) \vdash_{\mathit{EPA\text{-}TR}} P_2 \xrightarrow{\beta}_C P'_2}{(\sigma,\gamma) \vdash_{\mathit{EPA\text{-}TR}} P_1 \ | \ P_2 \xrightarrow{\alpha \star \beta}_C P'_1 \ | \ P'_2}$$

$$\boxed{\text{ProcDefn}}\ \frac{\begin{array}{ll}\text{(a)} & pdefn = pnm(fargs) \mathrel{\underline{\triangle}} P_2 \\ \text{(b)} & pdefn \in \sigma \\ \text{(c)} & (\sigma,\gamma) \vdash_{EPA\text{-}TR} P_2\,[\,args/fargs\,] \xrightarrow{\alpha}_C P'\end{array}}{(\sigma,\gamma) \vdash_{EPA\text{-}TR} pnm(args) \xrightarrow{\alpha}_C P'}$$

$$\boxed{\text{GenericProcDefn}}\ \frac{\begin{array}{ll}\text{(a)} & pdefn = pnm_{[fargs1]}(fargs_2) \mathrel{\underline{\triangle}} P_2 \\ \text{(b)} & pdefn \in \sigma \\ \text{(c)} & pdefn' = pdefn\,[\,args1/fargs1\,] \\ \text{(d)} & (\sigma \cup \{pdefn'\},\gamma) \vdash_{EPA\text{-}TR} pnm_{[args1]}(args_2) \xrightarrow{\alpha}_C P'\end{array}}{(\sigma,\gamma) \vdash_{EPA\text{-}TR} pnm_{[args1]}(args_2) \xrightarrow{\alpha}_C P'}$$

Note that in the above, $args, args1, args_2$ represent sequences of arguments to be used in (generic) process calls, including type expressions for generic process arguments, and value expressions, input channels and output channels for process calls. It is assumed that they will match the corresponding formal argument sequences $fargs, fargs1, fargs_2$.

Hiding The operation $hide\text{-}act(a, ch\text{-}set)$ simply removes any binding $ch(v)$ appearing in the action a, when ch: te is a channel declaration in the channel-set $ch\text{-}set \in ChannelSet$:

$$hide\text{-}act : Act \cup \{\boxtimes, \varepsilon\} \times ChannelSet \to Act \cup \{\boxtimes, \varepsilon\}$$
$$hide\text{-}act(\varepsilon, chset) = \varepsilon$$
$$hide\text{-}act(\boxtimes, chset) = \boxtimes$$
$$hide\text{-}act((a \mathbin{/} b), chset) = (hide\text{-}io\text{-}act(a, chset) \mathbin{/} hide\text{-}io\text{-}act(b, chset))$$

$$hide\text{-}io\text{-}act : IOAct \cup \{\boxtimes, \varepsilon\} \times ChannelSet \to IOAct \cup \{\boxtimes, \varepsilon\}$$
$$hide\text{-}io\text{-}act(\varepsilon, chset) = \varepsilon$$
$$hide\text{-}io\text{-}act(\boxtimes, chset) = \boxtimes$$
$$hide\text{-}io\text{-}act(bd, chset) = hide\text{-}prim\text{-}act(bd, chset)$$
$$hide\text{-}io\text{-}act(a \oplus b, chset) = hide\text{-}io\text{-}act(a, chset) \oplus hide\text{-}io\text{-}act(b, chset)$$

$$hide\text{-}prim\text{-}act : Binding \times ChannelSet \to Binding \cup \{\boxtimes, \varepsilon\}$$
$$hide\text{-}prim\text{-}act(bd, chset) = \text{if } chan\text{-}of(bd) \in chanvars \text{ then } \varepsilon \text{ else } bd$$
where
$$chanvars = chan\text{-}vars\text{-}of(chset)$$

$$chan\text{-}of : Binding \to ChannelExpr$$
$$chan\text{-}of(ch(v)) = ch$$

$$chan\text{-}vars\text{-}of : ChannelSet \to \mathcal{F}(ChannelVarName)$$
$$chan\text{-}vars\text{-}of(chset) = \{ch_1, \cdots, ch_k\}$$
where
$$⟦ \{chdecl_1, \cdots {}_{,:}\ k\} ⟧ = chset$$
$$\forall\, i \in \{1, \ldots, k\} \bullet ⟦\, ch_i \colon ty_i\, ⟧ = chdecl_i$$

Action Product ($\star$) has the same effect as the causal product for IO automata, where the input/output maps were combined using map union and map subtraction. This is achieved by using action summation and difference operators, $\oplus$ and $\ominus$ respectively:

$$\star : (Act \cup \{\boxtimes, \varepsilon\})^2 \to Act \cup \{\boxtimes, \varepsilon\}$$
$$\forall\, a_1, a_2, b_1, b_2 \in IOAct, \gamma \in Act \bullet$$
$$(a_1 \,/\, b_1) \star (a_2 \,/\, b_2) = ((a_1 \oplus a_2) \ominus (b_1 \oplus b_2)) \,/\, (b_1 \oplus b_2)$$
$$\boxtimes \star \gamma = \boxtimes = \gamma \star \boxtimes$$

Action Summation ($\oplus$) merges the bindings contained in two action expressions if they are consistent, and returns the illegal zero action $\boxtimes$ otherwise:

$$\oplus : Binding \cup \{\boxtimes, \varepsilon\} \times Binding \cup \{\boxtimes, \varepsilon\} \to IOAct \cup \{\boxtimes, \varepsilon\}$$
$$\begin{array}{rcl}
(ch_1 \neq ch_2) & \Rightarrow & ch_1(v_1) \oplus ch_2(v_2) = ch_1(v_1) \oplus ch_2(v_2) \\
 & & ch(v) \oplus ch(v) = ch(v) \\
(v_1 \neq v_2) & \Rightarrow & ch(v_1) \oplus ch(v_2) = \boxtimes \\
 & & bd \oplus \boxtimes = \boxtimes = \boxtimes \oplus bd \\
 & & bd \oplus \varepsilon = bd = \varepsilon \oplus bd
\end{array}$$

Note that action summation is associative and commutative.

Action Subtraction ($\ominus$) removes any bindings from its first argument which match exactly those in its second argument:

$$
\begin{array}{rcl}
\ominus : \mathit{Binding} \cup \{\boxtimes, \varepsilon\} \times \mathit{Binding} \cup \{\boxtimes, \varepsilon\} \rightarrow \mathit{IOAct} \cup \{\boxtimes, \varepsilon\} & & \\
& & ch(v) \ominus ch(v) = \varepsilon \\
(ch_1 \neq ch_2) & \Rightarrow & ch_1(v_1) \ominus ch_2(v_2) = ch_1(v_1) \\
& & bd \ominus \boxtimes = \boxtimes = \boxtimes \ominus bd \\
& & bd \ominus \varepsilon = bd \\
& & \varepsilon \ominus bd = \varepsilon
\end{array}
$$

Note that action difference distributes over summation so that $(a \oplus b) \ominus c = (a \ominus c) \oplus (b \ominus c)$.

The following examples illustrate how the *EPA* action product operates: Let w, x, y, z represent distinct concrete channels of type natural numbers:

$$
\begin{array}{ll}
(a) & (y(1) \mathbin{/} z(2)) \star (w(1) \mathbin{/} x(3)) = y(1), w(1) \mathbin{/} z(2), x(3) \\
(b) & (w(2) \mathbin{/} y(1)) \star (y(1) \mathbin{/} x(3)) = w(2) \mathbin{/} y(1), x(3) \\
(c) & (w(2) \mathbin{/} y(1)) \star (y(4) \mathbin{/} x(3)) = \boxtimes
\end{array}
$$

In (a) all the channels are different and so the product action is simply the combination of the primitive actions. In (b), input channel y is now 'connected' to the output y from the second action, and is removed from the resulting inputs. Finally, in (c), the value for y is now inconsistent and so $\boxtimes$ results. Also, note how comma-separated lists of bindings are used as shorthands for action summation.

Variable Binding is introduced by process definitions, hiding and process choice. For example, consider the following process definition, where *TY* is a literal type name, v, x are value variables, and $i, o, o2$ are channel variables:

$$
\begin{array}{l}
pnm(v : TY \,;\, i : TY \mathbin{/} o : TY) \triangleq \\
\qquad \Big(\Box x : TY \cdot i(x) \mathbin{/} o(v), o2(x) :: pnm(x \,;\, i \mathbin{/} o) \Big) \backslash \{o2 : TY\}
\end{array}
$$

Note first of all that the process is *closed*, because all of the variables are bound in the following way: the choice operator binds the value variable x, the hiding operator binds the channel variable $o2$, and the process definition binds the value variable v, and the channel variables i, o. The following process definition is therefore equivalent to the above, up to alpha-equivalence of the variable names:

$$
\begin{array}{l}
pnm(w : TY \,;\, r : TY \mathbin{/} s : TY) \triangleq \\
\qquad \Big(\Box y : TY \cdot r(y) \mathbin{/} s(w), t(y) :: pnm(y \,;\, r \mathbin{/} s) \Big) \backslash \{t : TY\}
\end{array}
$$

where w, y are value variables and r, s, t are channel variables. Notice that an expression '$i : TY$' used above is a declaration of a channel *variable* ranging over channels of type *TY*, while an expression '$nm : TY$' is a channel *literal* of type *TY*, where the channel is called $nm \in \mathit{ChannelName}$. The process call to the definition will only be well-defined if the input and output channels are disjoint, and all output channels are distinct.

In the *EPA* terms produced by the semantic definitions for ELLA, all of the process definitions are closed, and do not depend on particular (concrete) channel names — channels are referenced by (typed) channel variables declared either in the left-hand side of the process definition, or in a hiding term.

2.3 Normal Form for *EPA*

When *EPA* is used to represent the semantics of hardware systems described in ELLA, the resulting process definitions can be reduced to a pair of process definitions which constitute an *EPA normal form*. This arises because of the static structure of the ELLA hardware model, where the number and names of channels for each component remain constant. The normal form is the basis for (symbolic) simulation tools and verification using state bisimulation described below in §3.

The semantics of any ELLA expression can be represented by a process P which determines the initial 'state' conditions, and then calls a general process P' which gives the response and evolution behaviour:

$$
\begin{aligned}
&P(i \;/\; o) \mathrel{\underline{\triangle}} \Box\, y\colon se_1 \;\bullet\; P'(y \,;\, i \;/\; o) \\
&P'(v\colon te_1 \,;\, i\colon te_2 \;/\; o\colon te_3) \mathrel{\underline{\triangle}} \\
&\qquad \Box\, x\colon\; se_2, w\colon se_3 \;\bullet\; i(x) \;/\; o(r_P(x,v,w)) :: P'(e_P(x,v,w) \,;\, i \;/\; o)
\end{aligned}
$$

where r_P is a *response* function and e_P is an *evolution* function. The choice $y\colon se_1$, where y ranges over the initial state-space se_1, determines the initial state of the construct. The choice $x\colon se_2$ binds the variables occurring in the input actions, and $w\colon se_3$ represents variables used in the response and evolution functions which are not input-assigned, giving the scope for non-deterministic behaviour.

2.3.1 Rules for Normalisation of *EPA* Terms for ELLA

The *EPA* terms representing an ELLA expression are built up by composing calls to *atomic* process definitions, which represent basic ELLA constructs, using process call, product and hiding operations. The atomic process definitions are all in normal form, and a typical process definition for an ELLA function declaration will have the following structure:

$$
\begin{aligned}
&P(i\colon ity \;/\; o\colon oty \mathrel{\underline{\triangle}} \\
&\qquad \Big(\; P_1 \;\mid\; \ldots \;\mid\; P_k \;\Big) \backslash \{c_1\colon ty_1, \ldots c_m\colon ty_m\}
\end{aligned}
$$

Such a composite *EPA* process term can be normalised using a small set of rewrite rules — the three main rules are given below. Each rule is applied with respect to a context Γ of *EPA* definitions.

Process Call Rule: When the righthand side of a process has been normalised, then it will simply be a call to the new process definition constructed by normalisation; any call to the original process definition can now be replaced by a call to the new process: For example:

$$
\boxed{\text{Call}}\; \frac{\Gamma \vdash P(ic \;/\; oc)}{\Gamma \vdash Q(ic \;/\; oc)}
$$

where

$$
\begin{aligned}
&P(i_1\colon ty_1 \;/\; o_1\colon ty_2) \mathrel{\underline{\triangle}} Q(i_1 \;/\; o_1) \;\in\; \Gamma \\
&Q(i_2\colon ty_1 \;/\; o_2\colon ty_2) \mathrel{\underline{\triangle}} rhs \;\in\; \Gamma
\end{aligned}
$$

where i_1, i_2, o_1, o_2 are channel variables, and ic, oc are channels or channel variables. The rule states that if a set of *EPA* definitions, Γ, contains process definitions for P, Q, where P simply calls Q, then the call to P can be replaced by a call to Q,

Hiding Rule: The following rule constructs normal form process terms, from which any output channels contained in *chanset* have been removed. Note that i, o represent the sequences of channels declared in *idecl* and *odecl*, respectively:

$$\boxed{\text{Hide}}\ \frac{\Gamma \vdash P(i \ / \ o) \backslash chanset}{\Gamma \cup \{defn\} \vdash Q(i \ / \ o')}$$

where

$$P(idecl \ / \ odecl) \mathrel{\underline{\triangle}} \Box\, x{:}I \ \bullet \ prefix :: \ P(next \ ; \ i \ / \ o) \in \Gamma$$

$$defn \ = \ Q(idecl \ / \ odecl') \mathrel{\underline{\triangle}}$$

$$\Box\, x{:}I \ \bullet \ hide\text{-}act(prefix, chanset) :: \ Q(v \ ; \ i \ / \ o')$$

$$odecl' = odecl \mathbin{—}_{chdecl} chanset$$

$$o' = o \mathbin{—}_{chan} chanset$$

The operation $hide\text{-}act(prefix, chanset)$ removes any output bindings in the prefix action *prefix* which appear in *chanset*. The channel declaration sequence $odecl'$ is the result of removing declarations for channels in *chanset* from *odecl*, and the channel sequence o' is the result of removing removing channels in *chanset* from o.

Product Rule: replaces a product expression with a call to a single new process definition whose behaviour is equivalent to performing the product. The rule combines the two component actions, according to the product transition rule and the action product, so that in the resulting action there are no repeat input and output bindings, and any input binding which is also an output is removed from the final prefix term:

(a) $P(v_1\ ;\ idecl_1\ /\ odecl_1) \triangleq \square\, vdecl_1\ \bullet\ ibnd_1\ /\ obnd_1 :: P(e_1\ ;\ i_1\ /\ o_1)\ \in\ \Gamma$

(b) $Q(v_2\ ;\ idecl_2\ /\ odecl_2) \triangleq \square\, vdecl_2\ \bullet\ ibnd_2\ /\ obnd_2 :: Q(e_2\ ;\ i_2\ /\ o_2)\ \in\ \Gamma$

(c) $defn\ =\ R(v_1, v_2\ ;\ idecls\ /\ odecls) \triangleq$

$$\square\, vdecl\ \bullet\ prefix :: R(e'_1, e'_2\ ;\ ich\ /\ och)$$

$$\boxed{\text{Prod}}\ \frac{\Gamma \vdash P(i_1\ /\ o_1)\ \mid\ Q(i_2\ /\ o_2)}{\Gamma \cup \{defn\} \vdash R(ichans\ /\ ochans)}$$

where

$\sigma = unify(ibnd_1, obnd_1, ibnd_2, obnd_2)$

$allidecls = \sigma(idecl_1) \cup \sigma(idecl_2)$

$odecls = odecl_1, odecl_2$

$idecls = allidecls — odecls$

$vdecls = \sigma(vdecl_1) \cup \sigma(vdecl_2)$

$allibnds = \sigma(ibnd_1) \cup \sigma(ibnd_2)$

$obnds = \sigma((obnd_1, obnd_2))$

$ibnds = allibnds — obnds$

$prefix = ibnds\ /\ obnds$

$e'_1 = \sigma(e_1)$

$e'_2 = \sigma(e_2)$

$allichans = i_1 \cup i_2$

$ochans = (o_1, o_2)$

$ichans = allichans — ochans$

At the centre of this rule is the *unify* operation, which produces a substitution σ to unify its argument expressions. The substitution is applied to all components of the two processes. Corresponding element sequences from each process — input/output declarations, variable declarations, input/output bindings, input/output channels — are then merged using a sequence 'union' operation $\cup$; for each different type of object this concatenates the two sequences and systematically removes any duplicates from the result. Note that sequences involving the output channels can simply be concatenated, because the sets of output channels from the two processes must be disjoint. Finally any outputs occurring in the new sequences involving input channels are removed from those sequences.

The *EPA* normal form provides the basis for *symbolic* analysis, such as simulation and verification. A tractable approach to equivalence-checking, based on the *EPA* normal form for ELLA, and using the *state bisimulation* rule, is outlined below in §3.

2.4 Compiler Implementation

A compiler/parser for Core ELLA[5], which checks for the well-formedness of the ELLA expressions according to the static semantics given in §5, has been implemented by DRA (Malvern, England). This returns Common Lisp Object System (CLOS) data-structures for each ELLA expression, annotated with type information for the expression. It interfaces with an *EPA* compiler developed by Harlequin Ltd. (Cambridge, England), which is structured according to the semantic definitions given in §5. This converts the resulting process terms into a normal form, where the response and evolution function expressions are encoded as ordinary Lisp functions. The Lisp code can then be compiled and executed, forming the core of an efficient ground-value simulator, in which the executable code is derived from the original ELLA-text using the semantic definitions.

The simulator is embedded into a *Soft Logic Analyser* (SLA), a sophisticated graphical front-end/back-end interface which allows the user to set input values and then to view the results of simulation via a waveform-like display. The user can also set probes and forcers on circuit nodes, and can alter the format of the display. The SLA itself is then integrated into the ELLA Design Environment.

In the compiler implementation, the *EPA* term language hierarchy is reflected by the CLOS class definition hierarchy. The task of implementing the semantic definitions for the

Core ELLA-to-*EPA* translation is facilitated by the close match between the form of the semantic definitions and the corresponding Lisp function implementation. This approach led to rapid and accurate code-generation.

[5]All software has been written using the LispWorks Programmer's Environment for CLOS, from Harlequin Ltd., Cambridge.

3 Verification of *EPA*

Having compiled ELLA expressions into a set of *EPA* terms, and then transformed the set into *EPA* normal form, we can consider establishing equivalences between two *EPA* systems. In particular, the notion of (strong) bisimilarity[Mil89] will be used. In this section, we outline a *state bisimulation* rule, equivalent to strong bisimilarity, which operates on the abstract *EPA* terms produced from ELLA. An accompanying *state evolution* method generates a set of verification conditions needed to establish state bisimilarity. The approach offers the possibility of containing the usual growth in complexity of verification, from which state-based verification methods suffer. A high degree of automation is still possible, however. Although we will describe the method for deterministic systems, it is possible to extend it to non-deterministic systems. Note that this method is described in more detail in [BGMW94].

Each design is represented by a *deterministic EPA* process P in normal form:

$$P(i\ /\ o) \triangleq \square\, y\colon se_1\ \cdot\ P'(y\ ;\ i\ /\ o)$$

$$P'(\sigma_P\colon St_P\ ;\ i\colon I\ /\ o\colon O) \triangleq$$

$$\square\, x\colon St_P\ \cdot\ i(x)\ /\ o(r_P(x,\sigma_P)) :: P'(e_P(x,\sigma_P)\ ;\ i\ /\ o)$$

where the initial state se_1 contains a single value, i.e. $se_1 = \{i_P\}$, and all the variables in the response and evolution functions are either input-variables or state-variables. From this normal form is extracted a response function $r_P(x,\sigma_P)$, and an evolution function $e_P(x,\sigma_P)$, producing values belonging to a set of outputs O and a state-space St_P, respectively. At each step, these take an input value x from some set of inputs I and 'current-state' value σ_P from the state-space St_P. Note that the initial state value $i_P \in St_P$ for P is assigned to σ_P at the initial time-step 0.

There are a number of stages involved in establishing the bisimilarity of two processes when represented by response and evolution functions. First, we give the standard definition of bisimulation equivalence between two processes P and Q:

Definition 3.1 : (Strong) Bisimilarity

The relation S is a (strong) bisimulation if and only if:

- S is symmetric.
- $\langle P,Q\rangle \in S \ \Rightarrow$
  $\forall\, \alpha \in IOAct \cdot (P \xrightarrow{\alpha} P') \ \Rightarrow$
  $\exists\, Q' \cdot (Q \xrightarrow{\alpha} Q') \ \wedge\ \langle P',Q'\rangle \in SB$

We say that two processes $P,\ Q$ are *bisimilar*, denoted as $P \sim Q$ whenever there exists a strong bisimulation S such that $\langle P,Q\rangle \in S$.

Now we introduce the notion of State Bismulation which relates the state spaces of P and Q (as opposed to being a relation between process terms):

Definition 3.2 : State bisimulation

Let $SB \subseteq (St_P \times St_Q)$ be a binary relation between the *state spaces* of P and Q.

SB is a state bisimulation for $P,\ Q$ when:

1. $\langle i_P,\ i_Q \rangle \in SB$
2. if $\langle \sigma_P,\ \sigma_Q \rangle \in SB$ then for all x in I,
   (a) $r_P(x, \sigma_P) = r_P(x, \sigma_P)$, and
   (b) $\langle e_P(x, \sigma_P),\ e_Q(x, \sigma_Q) \rangle \in SB$

We now state the desired association between state and process:

Theorem 3.3 : Correctness of state bisimulation
Let $P,\ Q$ be deterministic systems in normal form:

$$(P \sim Q) \ \Leftrightarrow \ \exists\, SB \subseteq (St_P \times St_Q) \bullet SB \text{ is a state bisimulation for } P \text{ and } Q$$

3.1 Symbolic State Evolution method

The problem of establishing the strong bisimulation equivalence of two processes P and Q has now been transformed into the problem of establishing a state bisimulation between the response and evolution functions which represent P and Q. This can be performed in a truly symbolic manner by using the state evolution method. This generates a set of verification conditions whose validity would ensure that the systems were (state) bisimilar.

The idea is to take an arbitrary but general point, $\langle \sigma_P,\ \sigma_Q \rangle$, in the product state space $(St_P \times St_Q)$. As we know nothing about this point to start with, we can simply name it (or rather its components) using appropriate symbolic variables.

We then proceed to symbolically evolve both of the systems on the assumption that, for all inputs, the responses produced by each system should be in complete agreement at each stage. Now either:

1. we can show that, for some input, the responses clearly disagree at some stage (i.e. P and Q are different), *or*
2. we can demonstrate that the symbolic simulation is 'sufficiently complete', implying that the systems P and Q are behaviourally equivalent.

So the main issues become:

- what needs to be proven at each stage?
- what are the criteria for terminating?

Let us imagine that we are at stage N in applying the method. We have not already terminated, and so we consider the comparison of responses for the current step, based upon previous steps:

$$R_N(\sigma_P, \sigma_Q) \ \Leftrightarrow \ \forall\, x_N,\ \cdots\ x_1 \bullet r_P(x_N,\ sp_N) = r_Q(x_N,\ sq_N)$$

where sp_N and sq_N are expressions, dependent upon inputs $x_{N\text{-}1}, \cdots x_1$ and the initial product state $\langle \sigma_P,\ \sigma_Q \rangle$, describing in symbolic terms the state that is reached after N-1 input steps. We also need the following 'accumulating' version of predicate R:

$$AR_n(\sigma_P, \sigma_Q) \quad \Leftrightarrow \quad \bigwedge_{i=1}^{n} R_i(\sigma_P, \sigma_Q)$$

If valid, $AR_n(\sigma_P, \sigma_Q)$ says that $R_i(\sigma_P, \sigma_Q)$ is valid for all $1 \leq i \leq n$. In other words, the responses of systems P and Q agree for each step up to and including n, starting from states $\langle \sigma_P,\ \sigma_Q \rangle$. We should now check two things:

1. check that we continue to achieve agreement assuming that we had begun from the initial product state $\langle i_P,\ i_Q \rangle$. This amounts to instantiating the above formula with the initial state and showing that it is valid.

2. check to see if we can show that, on the basis of agreement on *earlier* steps, agreement of response is actually *forced* for the present step.

If the first check fails, then we have shown that the systems are manifestly different (after iterating from the initial state) and so we should then abort.

If both the first and second check are positive, then we stop with success since, intuitively, we have shown that the next step is implied by the immediate past history. Hence, the systems will continue to behave identically, once a product state is found where they will agree for N-1 steps. But the success of the first check shows that this has already been achieved from the initial state.

If the second check fails, then we proceed to the next stage of the iteration, since we have not been able to show one way or another whether the two systems are equivalent or not. Of course, we must also check that evolution from the initial states establishes agreement for as many steps.

Definition 3.4 : State Evolution rule

Using the notation introduced above we can now describe the State Evolution inference rule:

$$\boxed{\text{State Evolve}}\ \frac{\begin{array}{l}\text{For some fixed } N \in \mathbb{N}: \\ \quad \text{(a)} \quad \Gamma \vdash AR_N(i_P, i_Q) \\ \quad \text{(b)} \quad \Gamma \vdash \forall\, \sigma_P,\ \sigma_Q \ \cdot\ AR_N(\sigma_P, \sigma_Q) \ \Rightarrow\ R_{N+1}(\sigma_P, \sigma_Q)\end{array}}{\Gamma \vdash P \sim Q}$$

The above rule says that if we can establish for some *fixed N*:

(a) equivalent responses for N steps from the initial states, $\langle i_P,\ i_Q \rangle$, and:

(b) equivalent response for the $N+1$'th step from states $\langle \sigma_P,\ \sigma_Q \rangle$ follows from equivalent responses for all preceeding steps from the states $\langle \sigma_P,\ \sigma_Q \rangle$

then we can conclude that the two deterministic systems P and Q are strongly bisimilar.

3.2 Example: Verification Condition Generation

A verification condition generator (VCG) generates clauses which are logically equivalent to (a) and (b) in the state evolution rule, by performing case analysis on the response and evolution function terms for the two processes. For example, consider the following problem of determining whether two normal form *EPA* process terms, $P(i/o)$ and $Q(i/o)$ are equivalent:

$$
\begin{array}{ll}
P(v\colon \mathbb{Z}\ ;\ i\colon \mathbb{Z}\ /\ o\colon \mathbb{Z})\ \underline{\triangle} & Q(\langle w,s\rangle\colon \mathbb{Z}\times\mathbb{B}\ ;\ i\colon \mathbb{Z}\ /\ o\colon \mathbb{Z})\ \underline{\triangle} \\
\quad \Box x\colon \mathbb{Z}\ \cdot\ i(x)\ /\ o(v) & \quad \Box y\colon \mathbb{Z}\ \cdot\ i(y)\ /\ o(r_Q(y,w,s)) \\
\qquad ::\ P(x\ ;\ i\ /\ o) & \qquad ::\ Q(e_Q(y,w,s)\ ;\ i\ /\ o)
\end{array}
$$

where

$$
\begin{array}{ll}
r_Q(y,\langle w,s\rangle) = \textbf{case}\ s\ \textbf{in} & e_Q(y,\langle w,s\rangle) = \textbf{case}\ s\ \textbf{in} \\
\qquad t : w & \qquad t : \langle (y*10,f)\rangle \\
\quad \textbf{else}\ (w/10) & \quad \textbf{else}\ \langle y,t\rangle \\
\quad \textbf{end} & \quad \textbf{end}
\end{array}
$$

where the initial states are $i_P = 0$ and $i_Q = \langle 0,t\rangle$ (note that the response and evolution functions for P are simply $r_P(x,v) = v$ and $e_P(x,v) = x$ respectively). The process P represents a simple integer unit delay. Process Q performs the same operation, but on alternate time-steps it multiplies the input by 10 before storing it, and then divides this stored value by 10 again before outputting it on the next time-step. A toggle s controls the multiplication and division.

The state evolution rule premisses for P and Q are given by:

$$
\begin{array}{lrcl}
\text{(a)} & r_P(x_1,0) & = & r_Q(x_1,\langle 0,t\rangle) \\
\wedge & sp_2 & = & e_P(x_1,0) \\
\wedge & sq_2 & = & e_Q(x_1,\langle 0,t\rangle) \\
\Rightarrow & r_P(x_2,sp_2) & = & r_Q(x_2,sq_2) \\
\text{(b)} & r_P(x_1,sp_1) & = & r_Q(x_1,sq_1) \\
\wedge & sp_2 & = & e_P(x_1,sp_1) \\
\wedge & sq_2 & = & e_Q(x_1,sq_1) \\
\Rightarrow & r_P(x_2,sp_2) & = & r_Q(x_2,sq_2)
\end{array}
$$

Now consider premiss (b) — the VCG analyses the case structure of the expressions involved and will generate the following verification conditions (VCs), which are logically equivalent to (b):

$$
\begin{array}{llrcl}
\text{(b) (1)} & & r_P(x_1,v_1) & = & r_Q(x_1,\langle w_1,s_1\rangle) \\
\text{(2)} & \wedge & v_2 & = & e_P(x_1,v_1) \\
\text{(3)} & \wedge & \langle w_2,s_2\rangle & = & e_Q(x_1,\langle w_1,s_1\rangle) \\
\text{(4)} & \Rightarrow & r_P(x_2,sp_2) & = & r_Q(v_2,\langle w_2,s_2\rangle)
\end{array}
$$

with $sp_i = v_i$, $sq_i = \langle w_i, s_i \rangle$. Table 3.2 shows the eight VCs which are generated by expanding the three case statements contained in expression (b). Each VC is labelled by the value taken by the choosers s_1, s_1, s_2 in the expressions $r_Q(x_1, \langle w_1, s_1 \rangle)$, $e_Q(x_1, \langle w_1, s_1 \rangle)$, $r_Q(v_2, \langle w_2, s_2 \rangle)$ respectively. The elements in the first VC (ttt) have been labelled to show how they match with the different parts of expression (b) shown above. The VCG now extensively simplifies the set of clauses it generates, by checking for

	(ttt)	(ftt)	(tft)	$(*)(fft)$
(1)	$s_1 \in \{t\} \quad \wedge$	$s_1 \in \{f\} \quad \wedge$	$s_1 \in \{t\} \quad \wedge$	$s_1 \in \{f\} \quad \wedge$
	$v_1 = w_1 \quad \wedge$	$v_1 = w_1/10 \quad \wedge$	$v_1 = w_1 \quad \wedge$	$v_1 = w_1/10 \quad \wedge$
(2)	$v_2 = x_1 \quad \wedge$	$v_2 = x_1 \quad \wedge$	$v_2 = x_1 \quad \wedge$	$v_2 = x_1 \quad \wedge$
(3)	$s_1 \in \{t\} \quad \wedge$	$s_1 \in \{t\} \quad \wedge$	$s_1 \in \{f\} \quad \wedge$	$s_1 = f \quad \wedge$
	$w_2 = x_1 * 10 \quad \wedge$	$w_2 = x_1 * 10 \quad \wedge$	$w_2 = x_1 \quad \wedge$	$w_2 = x_1 \quad \wedge$
	$s_2 = f \quad \wedge$	$s_2 = f \quad \wedge$	$s_2 = t \quad \wedge$	$s_2 = t \quad \wedge$
(4)	$s_2 \in \{t\}$	$s_2 \in \{t\}$	$s_2 \in \{t\}$	$s_2 \in \{t\}$
	$\Rightarrow \; v_2 = w2$	$\Rightarrow \; v_2 = w2$	$\Rightarrow \; v_2 = w2$	$\Rightarrow \; v_2 = w2$

$(**)(ttf)$	(ftf)	(tff)	(fff)
$s_1 \in \{t\} \quad \wedge$	$s_1 \in \{f\} \quad \wedge$	$s_1 \in \{t\} \quad \wedge$	$s_1 \in \{f\} \quad \wedge$
$v_1 = w_1 \quad \wedge$	$v_1 = w_1/10 \quad \wedge$	$v_1 = w_1 \quad \wedge$	$v_1 = w_1/10 \quad \wedge$
$v_2 = x_1 \quad \wedge$	$v_2 = x_1 \quad \wedge$	$v_2 = x_1 \quad \wedge$	$v_2 = x_1 \quad \wedge$
$s_1 \in \{t\} \quad \wedge$	$s_1 \in \{t\} \quad \wedge$	$s_1 \in \{f\} \quad \wedge$	$s_1 \in \{f\} \quad \wedge$
$w_2 = x_1 * 10 \quad \wedge$	$w_2 = x_1 * 10 \quad \wedge$	$w_2 = x_1 \quad \wedge$	$w_2 = x_1 \quad \wedge$
$s_2 = f \quad \wedge$	$s_2 = f \quad \wedge$	$s_2 = t \quad \wedge$	$s_2 = t \quad \wedge$
$s_2 \in \{f\}$	$s_2 \in \{f\}$	$s_2 \in \{f\}$	$s_2 \in \{f\}$
$\Rightarrow \; v_2 = w2/10$	$\Rightarrow \; v_2 = w2/10$	$\Rightarrow \; v_2 = w2/10$	$\Rightarrow \; v_2 = w2/10$

Table 1: Verification Conditions for Latch Example

contradictions in the premisses and tautologies in the conclusion of each clause. In this example, all of the clauses except for (*) and (**) can be discarded, because there are contradictions in the premisses for the constraints and equalities involving s_1 or s_2. For example in clause (ttt), (3) requires $s_2 = f$ while (4) requires $s_2 \in \{t\}$. Then in (*) the conclusion is a tautology, when v_2 and w_2 have been substituted. This only leaves the single clause (**) to be proven; this requires the following fact about arithmetic:

$$\forall\, x, y \in \mathbb{Z} \bullet (y \neq 0) \;\Rightarrow\; ((y * x)/y = x)$$

which can be supplied to a standard theorem prover/proof assistant, such as PVS[6]. Thus the task of checking for equivalence between processes P and Q has been reduced by the VCG to that of proving a simple logical expression.

[6]PVS is available in the public domain from SRI International.

3.3 Formal Analysis Tool Implementation

A sophisticated *generic term* (GT) manipulation package[Mon94] for general data structures has been developed at Manchester. It provides facilities for the term manipulation expressions, such as substitution and rewriting, while automatically ensuring that common sub-terms are shared, for maximum time- and space-efficiency. These facilities are used extensively for implementing the following *EPA* term normalisation, symbolic simulation and verification tools.

A symbolic normaliser takes the raw compiler output and produces the corresponding *EPA* normal form terms. It incorporates various term simplification strategies, utilising the basic facilities provided by GT, for reducing the resulting term size, and for retaining the structure of the original ELLA text within the *EPA* term produced.

A verification condition generator (VCG) analyses the normal form terms and produces the set of VCs, which must be validated to establish state bisimulation using the state evolution method. The VCG simplifies the expressions generated to reduce the size and number of VCs left to be validated. It incorporates extensive set expression simplification, and can detect certain trivially valid or false expressions — any VC which is shown to be trivially valid can be discarded without having to be analysed further. The remaining VCs are then translated into a format accepted by a standard theorem-prover/proof assistant — at present the PVS Proof System [7] is used.

The *EPA* normal form also serves as a basis for a symbolic simulation (SymSim) tool. This is designed to operate in much the same way as the (ground) simulator included in the ELLA design environment, except that non-ground expressions are allowed as inputs, and thus non-ground expressions may be produced as output. The tool can therefore be used to examine the evolving design structure, and to analyse signal flow and control within the system. However, a designer can *selectively* monitor and explore the expressions generated, because the tool is able to exploit the structured nature of the process terms under analysis. This means that the size and number of *EPA* expressions displayed at any particular time can be be controlled.

The VCG and SymSim are both integrated into the ELLA design environment, and are operated via graphical interfaces.

[7] the PVS theorem-prover is available from SRI International, in the public-domain under special licence

4 Overview of semantic approach

The first stage in calculating the semantics for an ELLA-text is to flatten the name-space for functions and signals (see below). Then a *static semantics* relation determines type-checking and, as a side-effect, returns a (static) environment from which the output type of an ELLA expression can be calculated. In fact, for a well-typed ELLA-text there is a unique type for each expression. Note that it is only because the ELLA-text supplied to the static semantic relation has been flattened and has a global name-space, that it is possible to consider a single static environment — otherwise, the type of an ELLA function/variable may not be (globally) unique, because an already defined name may be redefined locally.

If the ELLA-text is well-typed, then its dynamic semantics can be calculated. The dynamic semantic operations build up an environment containing a semantic object, or *EPA* process definition, for each function declaration. They consult the static environment to obtain types of ELLA expressions. For each atomic ELLA structure, a call is made to a generic process definition which represents the structure; these are then composed using the causal product, hiding and choice operators, to form the process terms representing complex ELLA expressions. Instantiations of generic process definitions are *not* explicitly included in the dynamic environment — it is assumed that they will be available when the composite process terms are interpreted. In addition, the dynamic environment will contain *EPA* term definitions corresponding to ELLA type declarations, and auxiliary definitions for *EPA* value expressions as required. The semantic operations described here define a translation from Core ELLA-text into a sequence of *EPA* languge definitions. These will then be interpreted to produce a *synchronisation tree* from which a set of i/o traces can be derived for deterministic systems — this is taken to be the semantics of an ELLA expression.

4.1 Binding Environments

To present the semantics of Core ELLA, we shall need to use two main environment structures, which will accumulate static and dynamic information concerning the type and function object bindings. A local environment structure will also be needed to hold intermediate results when calculating the semantics of *Block*.

4.1.1 Static Environments

A *StaticEnvironment* object contains the naming information concerning type declarations and declarations of functions and their local signals.

> *Without loss of generality, we assume that the set Name is given sufficient structure so that* **distinct** *names can be used to separate uses of types, functions and any local variables.*

This constraint ensures that the same environment structure can be used for both kinds of semantic information.

$$StaticEnvironment = Name \xrightarrow{\text{map}} DeclInfo$$

The *DeclInfo* objects present static information declaring the kind of construct with which names are associated, and is specified in further detail in the next section.

We generally assume that $st\text{-}env, \rho \in StaticEnvironment$.

4.1.2 Dynamic Environments

A *DynamicEnvironment* object contains bindings of *EPA* language definitions that have been determined so far. When the dynamic semantics for all ELLA declarations has been calculated then the range of the resulting dynamic environment will be used to form an *EPA Context*.

$$\mathit{DynamicEnvironment} \;=\; \mathit{Name} \xrightarrow{\text{map}} \mathit{Defn}$$

We generally assume that $\mathit{dy\text{-}env}, \sigma \in \mathit{DynamicEnvironment}$.

4.1.3 Local Environment

A *LocalEnv* is required for *Block* to store intermediate information about the steps processed so far (see §5.8.16. It is simply a map from signal names to *EPA* language terms:

$$\mathit{LocalEnv} = \mathit{Signal} \xrightarrow{\text{map}} \mathit{ProcExpr} \cup \mathit{ProcName}$$

4.1.4 Environment

During the construction of the dynamic semantics for an ELLA-text, the dynamic environment is built up, holding *EPA* language terms representing previous type- and function-declarations. Static information concerning the output types of ELLA expressions is calculated from the static environment returned during the type-checking phase. Thus each semantic operation requires a pair of static and dynamic environments as an argument:

$$\mathit{Environment} = \mathit{StaticEnvironment} \times \mathit{DynamicEnvironment}$$

where we assume that $(\rho, \sigma) \in \mathit{Environment}$.

4.2 Horizontal and vertical organisation of semantic definitions

Semantic definitions are complex things and involve presenting the following information:

Abstract Syntax : The abstract structure of the language or data type.

Static Semantics : A collection of relations defined over the syntactic structure whose purpose is to ensure that the Dynamic Semantics is well-defined. In other words, the static semantics defines relationships between typing information and syntactic phrases. These relationships determine which phrases are deemed to be *legal* and for which the Dynamic Semantics is guaranteed to determine a well-defined *meaning* or value.

Dynamic Semantics : The relationship between phrases, as instances of language constructs, and their meanings which are often expressed mathematically as an element of some collection of denotable values. It is this that describes the fundamental characteristics of a construct.

Imagine that these semantic definitions were each laid out in some systematic fashion, perhaps determined by the structure of the abstract syntax definition. Then the material can be organised in two basic ways:

Vertical tabulation : Present each of the above in turn as an whole, by firstly defining the abstract syntax as a data type, then presenting the static semantics predicate by induction over this syntactic structure. Finally, the dynamic semantics function is then defined also by induction over the syntax. This has the advantage that each of the above are given conventional forms of definition that are familiar to mathematicians and computer scientists. Moreover, if one is interested in only one part of the semantics, then this allows the reader to concentrate solely upon that definition. As such, the vertical presentation is useful when making comparisons within a single semantic definition which relate different constructs.

Horizontal tabulation : Present each of the definitions on a construct-by-construct basis. Each construct is discussed in turn which involves giving first its Abstract Syntax, followed by any static semantic constraints associated with it and finally giving its Dynamic Semantics, often in some equational form.

This style naturally emphasises a compositional approach to the definition, so that objects are defined by reference to the meaning of their structural sub-components. Moreover, this style also indicates that constructs can be defined and understood a piece at a time rather than by analysing how each of semantic functions interact as a whole. As such, the horizontal presentation eases comparisons between static and dynamic semantics for particular constructs.

In the following work, we shall present the semantics using the horizontal format.

5 Semantic Definitions

This main section contains a Core ELLA semantics in the horizontal form discussed above. The additional notation used for presenting the semantics is given in Appendix C.

5.1 Auxiliary Operations

For the semantic definitions, most auxiliary operations are local to a particular ELLA constructions, and are defined in the same place. However, several auxiliary operations are used throughout, and their definitions are presented below.

The operation *type-of* accepts an ELLA-text expression and returns the *EPA* type expression corresponding to the output type $ty \in Type$ of the ELLA expression. It uses the static semantics relation ($\_ \vdash_{Unit}$ $\_$ has type $\_$) to find the type of the expression first:

$$
\begin{array}{l}
\textit{type-of} : \textit{Unit} \rightarrow \textit{Environment} \rightarrow \textit{TypeExpr} \\
\exists\, !ty \in \textit{TypeExpr} \bullet \\
\quad \textit{type-of}(\textit{unit})(\rho,\sigma) = \textit{M-Type}(ty)(\rho,\sigma) \\
\quad \text{where} \\
\qquad \rho \vdash_{Unit} \textit{unit} \text{ has type } ty
\end{array}
$$

The operation *proc-name* simply constructs a valid process name:

$$
\begin{array}{l}
\textit{proc-name} : \textit{Name} \rightarrow \textit{ProcName} \\
\textit{proc-name}(nm) = nm_P
\end{array}
$$

The operation $\textit{make-ExprVars}(n)$ generates a set of n distinct *EPA* value expression variables:

$$
\begin{array}{l}
\textit{make-ExprVars} : \mathbb{N} \rightarrow \mathcal{F}(\textit{DataExprVar}) \\
\textit{make-ExprVars}(n) = \{var_1, \cdots, var_n\} \\
\qquad \text{where} \\
\qquad\quad \forall\, i,j \in \{1,\ldots,n\} \bullet i \neq j \;\Rightarrow\; var_i \neq var_j \\
\qquad\quad \forall\, i \in \{1,\ldots,n\} \bullet var_i \in \textit{DataExprVar}
\end{array}
$$

$\textit{make-ChanVars}(n, nmset)$ generates a set of n new channel variable names, which are not within the set *nmset*:

$$
\begin{array}{l}
\textit{make-ChanVars} : \mathbb{N} \times \mathcal{F}(\textit{Name}) \rightarrow \mathcal{F}(\textit{ChannelVarName}) \\
\textit{make-ChanVars}(n, nmset) = \{nm_1, \cdots, nm_n\} \\
\quad \text{where} \\
\qquad \forall\, i,j \in \{1,\ldots,n\} \bullet nm_i \in (\textit{Name} - nmset) \;\wedge\; (i \neq j \;\Rightarrow\; nm_i \neq nm_j)
\end{array}
$$

5.1.1 The Operation *declash*

The operation *declash* flattens the name-space of an ELLA-text so that the names used for type-, function- and signal-declarations are globally unique. Instantiations of these are then renamed appropriately. The scope rules of ELLA are also obeyed. This is a reasonably straightforward but laborious process, since every different syntactic construction in ELLA must be addressed — therefore the full definition is not given here.

When an ELLA-text has been processed by *declash*, it is then possible to form a single global static environment for the resulting flattened text — this forms the basis for calculating the output type of any unit which occurs in the text. For example, in the following two fragments of (valid) ELLA, the text on the right is the result of applying *declash* to the text on the left. The original names used are int, F, w, x, y, the fresh names a, b, c have been introduced, and so the resulting names are int, F, w, x, y, a, b, c:

```
TYPE int = NEW i/(1..100).

FN F = (int : x) -> int:
BEGIN
     LET w = i/7.
     LET x = i/42.
     LET y = BEGIN
               LET x = w.
               LET y = x.
               OUTPUT y
             END.
     OUTPUT x
END.
```

```
TYPE int = NEW i/(1..100).

FN F = (int : x) -> int:
BEGIN
     LET w = i/7.
     LET a = i/42.
     LET y = BEGIN
               LET b = w.
               LET c = b.
               OUTPUT c
             END.
     OUTPUT a
END.
```

The full scope rules, and a definition of one method of producing a global name-space, are given in [HM92]. This also gives a definition of the static semantics of Core ELLA, as part of a translation from Core ELLA into a set of data structures called the *Kernel* of ELLA.

5.2 Declaration information

Structure:

$$
\boxed{
\begin{array}{rcl}
\textit{DeclInfo} & ::= & \textit{typedecl} \mid \textit{funtype} \mid \textit{sigtype} \\
\textit{FunType} & ::= & \texttt{funtype}\ \textit{type}_1 \rightarrow \textit{type}_2 \\
\textit{SigType} & ::= & \textit{lettype} \mid \textit{maketype} \\
\textit{LetType} & ::= & \texttt{let-type}\ \textit{type} \\
\textit{MakeType} & ::= & \texttt{make-type}\ \textit{type}_1\ \textit{type}_2
\end{array}
}
$$

The *declaration information* structures are not entities which are part of the syntax of Core ELLA, but instead represent syntactical structures capturing information concerning objects that can be declared using Core ELLA. As such, they refer to syntactical data extracted from declarations.

The following functions are used to manipulate these structures:

$$\textit{tags-of-env} : \textit{StaticEnvironment} \rightarrow \mathcal{F}(\textit{Tag})$$

$$\textit{tags-of-env}(\textit{st-env}) = \bigcup \{ \textit{tags-of}(\textit{td}) \mid \textit{td} \in (\textit{TypeDecl} \cap \mathsf{rng}\, \textit{st-env}) \}$$

$$\textit{tags-of} : \textit{TypeDecl} \rightarrow \mathcal{F}(\textit{Tag})$$

$$\textit{tags-of}[\![\textit{tynm} = \textit{ty}]\!] = \emptyset$$

$$\textit{tags-of}[\![\textit{tynm} = \texttt{NEW}\ \textit{nty}]\!] = \textit{tags-of-newtype}(\textit{nty})$$

$$\textit{tags-of-newtype} : \textit{NewType} \rightarrow \mathcal{F}(\textit{Tag})$$

$$\textit{tags-of-newtype}[\![(\textit{tagty}_1 \ \underline{|}\ \cdots\ \underline{|}\ \textit{tagty}_{k+1})]\!] = \{\textit{tag-of-tagtype}(\textit{tagty}_i) \mid i \in \{1, \ldots, k+1\}\}$$

$$\textit{tags-of-newtype}[\![\textit{tag}/(i\_j)]\!](\textit{env}) = \{\textit{tag}\}$$

$$\textit{tags-of-newtype}[\![\textit{tag}(''\textit{chr}_1 \ \underline{|}\ \cdots\ \underline{|}\ ''\textit{chr}_{k+1})]\!](\textit{env}) = \{\textit{tag}\}$$

$$\textit{tag-of-tagtype}[\![\textit{tag}]\!] = \textit{tag}$$

$$\textit{tag-of-tagtype}[\![\textit{tag}\ \&\ \textit{ty}]\!] = \textit{tag}$$

$$\textit{NewTypeParts} = \textit{TagType} \cup \textit{IntType} \cup \textit{CharType}$$

$$\textit{newtype-of} : \textit{TypeDecl} \rightarrow \mathcal{F}(\textit{NewTypeParts})$$

$$\textit{newtype-of}[\![\textit{tynm} = \textit{ty}]\!] = \emptyset$$

$$\textit{newtype-of}[\![\textit{tynm} = \texttt{NEW}\ \textit{nty}]\!] = \textit{newtype-parts-of}(\textit{nty})$$

$$newtype\text{-}parts\text{-}of : NewType \to \mathcal{F}(NewTypeParts)$$
$$newtype\text{-}parts\text{-}of[\![tagty_1 \ \underline{|} \ \cdots \ \underline{|} \ tagty_{k+1}]\!] = \{tagty_1, \ \cdots \ , tagty_{k+1}\}$$
$$newtype\text{-}parts\text{-}of[\![nty]\!] = \{nty\}$$

where

$$nty \in (IntType \cup CharType)$$

$$vartype\text{-}of : BlockSig \to Type$$
$$vartype\text{-}of[\![\ \texttt{let-type}\ obj\text{-}ty]\!] = obj\text{-}ty$$
$$vartype\text{-}of[\![\ \texttt{make-type}\ arg\text{-}tyobj\text{-}ty]\!] = obj\text{-}ty$$

The following defines a function for extracting the names that need to be joined to:

$$join\text{-}to\text{-}names : Step^* \to \mathcal{F}(Name)$$
$$join\text{-}to\text{-}names[\![st_1, \ \cdots \ , st_n]\!] = \bigcup\{join\text{-}to\text{-}name(st_i) \mid i \in \{1, \ldots, n\}\}$$

$$join\text{-}to\text{-}name : Step \to \mathcal{F}(Name)$$
$$join\text{-}to\text{-}name(step) = \emptyset$$

where

$$step \in (Step - Make)$$

$$join\text{-}to\text{-}name[\![\ \texttt{MAKE}\ fnname : signal.]\!] = \{\ signal\ \}$$

The following rules define various analysis predicates:

$$\_ \vdash_{is\text{-}char\text{-}type} \_ \subseteq (StaticEnvironment \times Type)$$
$$\forall\ k \in \mathbb{N}, chr_1, \ \cdots \ , chr_{k+1} \in Char, tag \in Tag, new\text{-}ty \in NewType,$$

$$\boxed{\text{is-char-type}}\ \frac{\begin{array}{ll} (a) & tynm \in \mathrm{dom}\, st\text{-}env \\ (b) & st\text{-}env(tynm) = [\![tynm = \ \texttt{NEW}\ new\text{-}ty]\!] \\ (c) & new\text{-}ty = [\![tag('' chr_1 \ \underline{|} \ \cdots \ \underline{|} \ '' chr_{k+1})]\!] \end{array}}{st\text{-}env \vdash_{is\text{-}char\text{-}type} tynm}$$

$$\_ \vdash_{is\text{-}num\text{-}type} \_ \subseteq (StaticEnvironment \times Type)$$
$$\forall\ i, j \in \mathbb{Z}, tag \in Tag, new\text{-}ty \in NewType,$$

$$\boxed{\text{is-num-type}}\ \frac{\begin{array}{ll} (a) & tynm \in \mathrm{dom}\, st\text{-}env \\ (b) & new\text{-}ty = [\![tag/(i \ \underline{..} \ j)]\!] \\ (c) & st\text{-}env(tynm) = [\![tynm = \ \texttt{NEW}\ new\text{-}ty]\!] \end{array}}{st\text{-}env \vdash_{is\text{-}num\text{-}type} tynm}$$

$(\_ \vdash_{\textit{is-tagelt-type}} \_) \subseteq (\textit{StaticEnvironment} \times \textit{Type})$

$\forall\ k \in \mathbb{N}, \textit{new-ty} \in \textit{NewType},$

$$\frac{\begin{array}{ll}(a) & \textit{tynm} \in \mathsf{dom}\,\textit{st-env} \\ (b) & \textit{new-ty} = [\![(\textit{tag}_1 \mid \ldots \mid \textit{tag}_k)]\!] \\ (c) & \forall\ i \in \{1,\ldots,k\} \cdot \textit{tag}_i \in \textit{Tag} \\ (d) & \textit{st-env}(\textit{tynm}) = [\![\textit{tynm} = \ \mathtt{NEW}\ \textit{new-ty}]\!]\end{array}}{\textit{st-env} \vdash_{\textit{is-tagelt-type}} \textit{tynm}}\ \boxed{\text{is-tagelt-type}}$$

$(\_ \vdash_{\textit{is-string-type}} \_) \subseteq (\textit{StaticEnvironment} \times \textit{Type})$

$\forall\ n \in \mathbb{N}, \textit{ty} \in \textit{Type},$

$$\frac{\begin{array}{ll}(a) & \textit{tynm} \in \mathsf{dom}\,\textit{st-env} \\ (b) & \textit{st-env}(\textit{tynm}) = [\![\textit{tynm} = \ \mathtt{STRING}\ [n]\textit{ty}]\!]\end{array}}{\textit{st-env} \vdash_{\textit{is-string-type}} \textit{tynm}}\ \boxed{\text{is-string-type}}$$

$(\_ \vdash_{\textit{is-prim-type}} \_) \subseteq (\textit{StaticEnvironment} \times \textit{Type})$

$$\frac{\textit{st-env} \vdash_{\textit{is-tagelt-type}} \textit{ty}}{\textit{st-env} \vdash_{\textit{is-prim-type}} \textit{ty}}\ \boxed{\text{is-prim-type: enum}}$$

$$\frac{\textit{st-env} \vdash_{\textit{is-num-type}} \textit{tyn}}{\textit{st-env} \vdash_{\textit{is-prim-type}} \textit{ty}}\ \boxed{\text{is-prim-type: num}}$$

$(\_ \vdash_{\textit{is-enum-type}} \_) \subseteq (\textit{StaticEnvironment} \times \textit{Type})$

$$\frac{\textit{st-env} \vdash_{\textit{is-tagelt-type}} \textit{ty}}{\textit{st-env} \vdash_{\textit{is-enum-type}} \textit{ty}}\ \boxed{\text{is-enum-type : tag}}$$

$$\frac{\textit{st-env} \vdash_{\textit{is-char-type}} \textit{ty}}{\textit{st-env} \vdash_{\textit{is-enum-type}} \textit{ty}}\ \boxed{\text{is-enum-type : char}}$$

The following function extracts set range information from explicit ranges within *NewType*:-

$\textit{range-set} : \textit{NewType} \to \mathcal{F}(\textit{GEN-VALUE})$

$\textit{range-set}[\![\textit{tagty}_1 \mid \cdots \mid \textit{tagty}_{k+1}]\!] = \emptyset$

$\textit{range-set}[\![\textit{tag}/(i \mathrel{..} j)]\!] = \{i,\ldots,j\}$

$\textit{range-set}[\![\textit{tag}(''\,\textit{chr}_1 \mid \cdots \mid ''\,\textit{chr}_k)]\!] = \{\textit{chr}_1, \cdots, \textit{chr}_k\}$

The following rules define context-sensitive predicates for constraining syntactical structure:

$$(\_ \vdash_{\textit{is-const-value}} \_) \subseteq (\textit{StaticEnvironment} \times \textit{Unit})$$

$$\boxed{\textsf{is-const-value : Enumerated}}\ \frac{\textit{st-env} \vdash_{\textit{Enumerated}} \textit{enum}}{\textit{st-env} \vdash_{\textit{is-const-value}} \textit{enum}}$$

$$\boxed{\textsf{is-const-value : Row}}\ \frac{\forall\, i \in \{1, \ldots, k+1\} \bullet \textit{st-env} \vdash_{\textit{is-const-value}} \textit{cnst}_i}{\textit{st-env} \vdash_{\textit{is-const-value}} [\![\textit{cnst}_1, \ \cdots \ , \textit{cnst}_{k+1}]\!]}$$

$$\boxed{\textsf{is-const-value : String}}\ \frac{\begin{array}{ll}\text{(a)} & n \geq 1 \\ \text{(b)} & \textit{st-env} \vdash_{\textit{is-char-value}} \textit{cnst}\end{array}}{\textit{st-env} \vdash_{\textit{is-const-value}} [\![\ \texttt{STRING}\ [n]\textit{cnst}]\!]}$$

$$\boxed{\textsf{is-const-value : Query}}\ \frac{\begin{array}{ll}\text{(a)} & \textit{ty} \in \textit{TypeName} \\ \text{(b)} & \textit{ty} \in \textsf{dom}\, \textit{st-env}\end{array}}{\textit{st-env} \vdash_{\textit{is-const-value}} [\![?\textit{ty}]\!]}$$

$$(\_ \vdash_{\textit{is-char-value}} \_) \subseteq (\textit{StaticEnvironment} \times \textit{Unit})$$

$$\boxed{\textsf{is-char-value}}\ \frac{\begin{array}{ll}\text{(a)} & \textit{chr} \in \textit{Unit} \\ \text{(b)} & \textit{st-env} \vdash_{\textit{Unit}} \textit{chr}\ \textsf{has type}\ \textit{chr-type} \\ \text{(c)} & \textit{st-env} \vdash_{\textit{is-char-type}} \textit{chr-type}\end{array}}{\textit{st-env} \vdash_{\textit{is-char-value}} \textit{chr}}$$

5.2.1 ELLA Type Equivalence

The following rules define a type equivalence relation, for use by the static semantics, over ELLA type constructs. A similar set of rules is needed for type equivalence over *EPA* type expressions in the dynamic semantics, and this is given in the following section.

$$(\_ \vdash_{E\text{-}TyEq} \_ \ \equiv \ \_) \subseteq (StaticEnvironment \times Type \times Type)$$

E-TyEq: Reflexivity

$$\dfrac{st\text{-}env \vdash_{Type} ty}{st\text{-}env \vdash_{E\text{-}TyEq} ty \ \equiv \ ty}$$

E-TyEq: Transitivity

$$\dfrac{\begin{array}{ll}(a) & st\text{-}env \vdash_{E\text{-}TyEq} tya \ \equiv \ tyb \\ (b) & st\text{-}env \vdash_{E\text{-}TyEq} tyb \ \equiv \ tyc\end{array}}{st\text{-}env \vdash_{E\text{-}TyEq} tya \ \equiv \ tyc}$$

E-TyEq: Symmetry

$$\dfrac{st\text{-}env \vdash_{E\text{-}TyEq} tyb \ \equiv \ tya}{st\text{-}env \vdash_{E\text{-}TyEq} tya \ \equiv \ tyb}$$

E-TyEq: TypeName

$$\dfrac{\begin{array}{ll}(a) & tynm \in \mathsf{dom}\, st\text{-}env \ \wedge \ st\text{-}env(tynm) = [\![tynm = rhs\text{-}ty]\!] \\ (b) & st\text{-}env \vdash_{E\text{-}TyEq} rhs\text{-}ty \ \equiv \ ty\end{array}}{st\text{-}env \vdash_{E\text{-}TyEq} tynm \ \equiv \ ty}$$

E-TyEq: TupleType

$$\dfrac{\begin{array}{ll}(a) & n \geq 2 \\ (b) & \forall\, i \in \{1, \ldots, n\} \bullet st\text{-}env \vdash_{E\text{-}TyEq} tya_i \ \equiv \ tyb_i\end{array}}{st\text{-}env \vdash_{E\text{-}TyEq} [\![(tya_1, \ \cdots \ , \ tya_n)]\!] \ \equiv \ [\![(tyb_1, \ \cdots \ , \ tyb_n)]\!]}$$

E-TyEq: PowerType (a)

$$\dfrac{\begin{array}{ll}(a) & st\text{-}env \vdash_{E\text{-}TyEq} [\![(ty_1, \ \cdots \ , \ ty_n)]\!] \equiv \ tyb \\ (b) & \forall\, i \in \{1, \ldots, n\} \bullet ty_i = tya \\ (c) & n \geq 2\end{array}}{st\text{-}env \vdash_{E\text{-}TyEq} [\![[n]tya]\!] \ \equiv \ tyb}$$

$$\boxed{\text{E-TyEq: PowerType (b)}}\ \dfrac{\textit{st-env} \vdash_{\textit{E-TyEq}} \textit{tya} \ \equiv \ \textit{tyb}}{\textit{st-env} \vdash_{\textit{E-TyEq}} [\![[n]\textit{tya}]\!] \ \equiv \ [\![[n]\textit{tyb}]\!]}$$

$$\boxed{\text{E-TyEq: StringType}}\ \dfrac{\textit{st-env} \vdash_{\textit{E-TyEq}} \textit{tya} \ \equiv \ \textit{tyb}}{\textit{st-env} \vdash_{\textit{E-TyEq}} [\![\ \texttt{STRING}\ [n]\textit{tya}]\!] \ \equiv \ [\![\ \texttt{STRING}\ [n]\textit{tyb}]\!]}$$

5.2.2 *EPA* Type Expression Equivalence

The following rules defin an equivalence relation for *EPA* type expressions. The only difference between these and the previous rules is that they are over *EPA* expressions and not ELLA expressions. Consequently, they operate with respect to an environment consisting of a set of *EPA* definitions:

$$(\_ \vdash_{\textit{EPA-TyEq}} \_ \ \equiv \ \_) \subseteq (\mathcal{F}(\textit{Defn}) \times \textit{TypeExpr} \times \textit{TypeExpr})$$

$$\boxed{\text{EPA-TyEq: Reflexivity}}\ \dfrac{\textit{ty} \in \textit{TypeExpr}}{\textit{epa-env} \vdash_{\textit{EPA-TyEq}} \textit{ty} \ \equiv \ \textit{ty}}$$

$$\boxed{\text{EPA-TyEq: Transitivity}}\ \dfrac{\begin{array}{ll}\text{(a)} & \textit{epa-env} \vdash_{\textit{EPA-TyEq}} \textit{tya} \ \equiv \ \textit{tyb} \\ \text{(b)} & \textit{epa-env} \vdash_{\textit{EPA-TyEq}} \textit{tyb} \ \equiv \ \textit{tyc}\end{array}}{\textit{epa-env} \vdash_{\textit{EPA-TyEq}} \textit{tya} \ \equiv \ \textit{tyc}}$$

$$\boxed{\text{EPA-TyEq: Symmetry}}\ \dfrac{\textit{epa-env} \vdash_{\textit{EPA-TyEq}} \textit{tyb} \ \equiv \ \textit{tya}}{\textit{epa-env} \vdash_{\textit{EPA-TyEq}} \textit{tya} \ \equiv \ \textit{tyb}}$$

$$\boxed{\text{EPA-TyEq: TypeName}}\ \dfrac{\begin{array}{ll}\text{(a)} & \textit{tydefn} \in \textit{epa-env} \\ \text{(b)} & \textit{tydefn} = (\![\ \text{type}\ \textit{tynm} \ \underline{\triangle} \ \textit{rhs-ty}\ ]\!) \\ \text{(c)} & \textit{epa-env} \vdash_{\textit{EPA-TyEq}} \textit{rhs-ty} \ \equiv \ \textit{ty}\end{array}}{\textit{epa-env} \vdash_{\textit{EPA-TyEq}} \textit{tynm} \ \equiv \ \textit{ty}}$$

(a) $\forall\, i \in \{1, \ldots, n\} \bullet \mathit{epa\text{-}env} \vdash_{\mathit{EPA\text{-}TyEq}} \mathit{tya}_i \;\equiv\; \mathit{tyb}_i$

(b) $n \geq 2$

EPA-TyEq: Tuple

$$\mathit{epa\text{-}env} \vdash_{\mathit{EPA\text{-}TyEq}} (\!\![\; \mathit{tya}_1 \times \cdots \times \mathit{tya}_n \;]\!\!) \;\equiv\; (\!\![\; \mathit{tyb}_1 \times \cdots \times \mathit{tyb}_n \;]\!\!)$$

(a) $\mathit{epa\text{-}env} \vdash_{\mathit{EPA\text{-}TyEq}} (\!\![\; \mathit{ty}_1 \times \cdots \times \mathit{ty}_n \;]\!\!) \equiv \mathit{tyb}$

(b) $\forall\, i \in \{1, \ldots, n\} \bullet \mathit{ty}_i = \mathit{tya}$

(c) $n \geq 2$

EPA-TyEq: Power

$$\mathit{epa\text{-}env} \vdash_{\mathit{EPA\text{-}TyEq}} (\!\![\; \mathit{tya} \uparrow n \;]\!\!) \;\equiv\; \mathit{tyb}$$

5.2.3 Free Signals Function

The *free-signals* function extracts those signals that statically occur freely within a *Unit* or a *Step* phrase. This is a static function in that it operates purely upon the static structure of the text and ignores any dynamic aspects. In particular, signals that occur statically but which cannot influence dynamic behaviour still count as occurring freely in a given circuit (cf: false paths).

The function is defined by induction over the structure of *Unit* and *Step* and is specified according to the entries in the following defining tables:

$$\begin{aligned}&\textit{free-signals} : (\textit{Unit} \cup \textit{Step} \cup \textit{Caseitem} \cup \textit{SignalDecl}\\ &\qquad \cup \textit{FnDecl} \cup \textit{Formula} \cup \textit{BoolFormula}) \to \mathcal{F}(\textit{Name})\end{aligned}$$

un : *Unit*	$\textit{free-signals}(un)$
( )	$\emptyset$
enum & *unit*	$\textit{free-signals}(\textit{unit})$
`STRING` $[\textit{nat}]\textit{unit}$	$\textit{free-signals}(\textit{unit})$
$[\ \textit{nat}\ ]\ \textit{unit}$	$\textit{free-signals}(\textit{unit})$
? *type*	$\emptyset$
$\textit{unit}\ [\ \textit{nat}\ ]$	$\textit{free-signals}(\textit{unit})$
$\textit{unit}_1\ [[\ \textit{unit}_2\ ]]$	$\textit{free-signals}(\textit{unit}_1) \cup \textit{free-signals}(\textit{unit}_2)$
`REPLACE` $(\textit{unit}_1, \textit{unit}_2, \textit{unit}_3)$	$\textit{free-signals}(\textit{unit}_1) \cup \textit{free-signals}(\textit{unit}_2)$ $\cup\, \textit{free-signals}(\textit{unit}_3)$
$\textit{unit}\ [\ \textit{nat}_1\ \underline{..}\ \textit{nat}_2\ ]$	$\textit{free-signals}(\textit{unit})$
$\textit{unit}_1$ `CONC` $\textit{unit}_2$	$\textit{free-signals}(\textit{unit}_1) \cup \textit{free-signals}(\textit{unit}_2)$
$(\textit{unit}_1\ ,\ \ldots\ ,\ \textit{unit}_{k+1})$	$\bigcup\{\textit{free-signals}(un_i) \mid i \in \{1,\ldots,k+1\}\}$
$\textit{unit}\ //\ \textit{enum}$	$\textit{free-signals}(\textit{unit})$
signal	$\{\textit{signal}\}$
fnname unit	$\textit{free-signals}(\textit{unit})$
`BEGIN` $\textit{step}_1 \ldots \textit{step}_{k+1}$ `OUTPUT` *unit* `END`	$(\ \textit{free-signals}(\textit{unit}) \cup$ $\bigcup\{\textit{free-signals}(\textit{step}_i) \mid i \in \{1,\ldots,k+1\}\}$ $)\ -\ \textit{local-signals}[[\textit{step}_1 \ldots \textit{step}_{k+1}]]$
`CASE` $\textit{unit}_1$ `OF` $\textit{caseitem}_1\ ,\ \ldots\ ,\ \textit{caseitem}_{k+1}$ `ELSE` $\textit{unit}_2$ `ESAC`	$\textit{free-signals}(\textit{unit}_1) \cup \textit{free-signals}(\textit{unit}_2) \cup$ $\bigcup\{\textit{free-signals}(\textit{caseitem}_i) \mid i \in \{1,\ldots,k+1\}\}$

$citm : Caseitem$	$free\text{-}signals(citm)$
$constset : unit$	$free\text{-}signals(unit)$

$step : Step$	$free\text{-}signals(step)$
$tydecl \in TypeDecl$	$\emptyset$
$fndecl \in FnDecl$	$free\text{-}signals(fndecl)$
`LET` $signal = unit$.	$free\text{-}signals(unit)$
`MAKE` $fnname : signal$.	$\emptyset$
`JOIN` $unit \rightarrow signal$.	$free\text{-}signals(unit)$

$formula : Formula$	$free\text{-}signals(formula)$
$monadic\ formula$	$free\text{-}signals(formula)$
$formula_1\ dyadic\ formula_2$	$free\text{-}signals(formula_1) \cup free\text{-}signals(formula_2)$
$signal$	$\{signal\}$
int	$\emptyset$
`IF` $boolformula$ `THEN` $formula_1$	$free\text{-}signals(boolformula) \cup free\text{-}signals(formula_1)$
`ELSE` $formula_2$ `FI` .	$\cup\ free\text{-}signals(formula_2)$

$boolformula : BoolFormula$	$free\text{-}signals(boolformula)$
`NOT` $boolformula$	$free\text{-}signals(boolformula)$
$formula_1\ binop\ formula_2$	$free\text{-}signals(formula_1) \cup free\text{-}signals(formula_2)$
$boolformula_1\ logicop\ boolformula_2$	$free\text{-}signals(boolformula_1) \cup free\text{-}signals(boolformula_2)$

$fndecl : FnDecl$	$free\text{-}signals(fndecl)$
$\texttt{FN}\, fnname = (sigdl_1,\ \cdots\ , sigdl_{k+1}) \rightarrow type : unit\ .$	$free\text{-}signals(unit) \cup \bigcup \{\, free\text{-}signals(sigdl_i) \mid i \in \{1, \ldots, k+1\} \}$
$\texttt{FN}\, fnname = (type_1) \rightarrow type : primitive\ .$	$\emptyset$
$\texttt{FN}\, fnname = (type_1, \ldots, type_4) \rightarrow type :\ \texttt{RAM}\ (unit)\ .$	$\emptyset$
$\texttt{FN}\, fnname = (type_1, \ldots, type_{k+1}) \rightarrow type :\ \texttt{REFORM .}$	$\emptyset$
$\texttt{FN}\, fnname = (type_1, \ldots, type_{k+1}) \rightarrow type :\ \texttt{BIOP}\ biopname.$	$\emptyset$
$\texttt{FN}\, fnname = (sigdl_1,\ \cdots\ , sigdl_{k+1}) \rightarrow type :\ \texttt{ARITH formula .}$	$free\text{-}signals(formula) \cup \bigcup \{\, free\text{-}signals(sigdl_i) \mid i \in \{1, \ldots, k+1\} \}$

$sigdl : SignalDecl$	$free\text{-}signals(sigdl)$
$type :\ signal$	$free\text{-}signals(signal)$

The auxiliary function *local-signals* calculates the defined signals in a sequence of block step elements:

$$local\text{-}signals : Step^* \rightarrow \mathcal{F}(Name)$$
$$local\text{-}signals[\![st_1,\ \cdots\ , st_k]\!] = \bigcup \{\ definition\text{-}signal(st_i) \mid i \in \{i, \ldots, k\}\ \}$$

The function *definition-signal* is used to extract the name of the signal that is defined by a *Step*:

$$definition\text{-}signal : Step \rightarrow \mathcal{F}(Name)$$
$$\forall\ decl \in Declaration \bullet$$
$$\quad definition\text{-}signal(decl) = \emptyset$$
$$definition\text{-}signal[\![\ \texttt{LET}\ sig = un\ .]\!] = \{sig\}$$
$$definition\text{-}signal[\![\ \texttt{MAKE}\ fnnm :\ sig\ .]\!] = \{sig\}$$
$$definition\text{-}signal[\![\ \texttt{JOIN}\ un \rightarrow sig\ .]\!] = \emptyset$$

5.3 New Types and Tag Types

Structure:

NewType	$::=$	*assoctype* $\mid$ *inttype* $\mid$ *chartype*
AssocType	$::=$	$(tagtype_1 \underline{\mid} \ldots \underline{\mid} tagtype_{k+1})$
IntType	$::=$	$tag/(int_1 \underline{..}\ int_2)$
CharType	$::=$	$tag(\ 'char_1 \underline{\mid} \ldots \underline{\mid}\ 'char_{k+1})$
TagType	$::=$	*tag* $\mid$ *tag* & *type*

The *NewType* constructs allows basic ELLA types to be formed. These are tagged integers or characters. The lower and upper bounds for integers are machine-dependent, and set here for when ELLA is run on a 32-bit machine. An association type forms a tagged union between a sequence of previously declared types. Note that within an association type declaration, it is possible to include single tags, which do not have an accompanying type. A common special case of this arises when there are no accompanying types, giving an enumeration type declaration containing simply a sequence of tags.

Static semantics rules:

$$\boxed{\text{wf-NewType : AssocType}}\ \frac{\begin{array}{ll}\text{(a)} & \textit{tag-set} = \textit{tags-of-newtype}[\![(tagty_1 \underline{\mid} \cdots \underline{\mid} tagty_{k+1})]\!] \\ \text{(b)} & \mathsf{card}\ \textit{tag-set} = k+1 \\ \text{(c)} & \forall\, i \in \{1,\ldots,k+1\} \bullet \textit{st-env} \vdash_{TagType} tagty_i\end{array}}{\textit{st-env} \vdash_{NewType} (tagty_1 \underline{\mid} \ldots \underline{\mid} tagty_{k+1})}$$

$$\boxed{\text{wf-NewType : IntType}}\ \frac{\begin{array}{ll}\text{(a)} & tag \notin \textit{tags-of-env}(\textit{st-env}) \\ \text{(b)} & -2^{31} \leq i \leq j \leq 2^{31}-1\end{array}}{\textit{st-env} \vdash_{NewType} tag/(i \underline{..}\ j)}$$

$$\boxed{\text{wf-NewType : CharType}}\ \frac{\begin{array}{ll}\text{(a)} & tag \notin \textit{tags-of-env}(\textit{st-env}) \\ \text{(b)} & \textit{char-set} = \textit{range-set}[\![tag(\ 'char_1 \underline{\mid} \ldots \underline{\mid}\ 'char_{k+1})]\!] \\ \text{(c)} & \mathsf{card}\ \textit{char-set} = k+1\end{array}}{\textit{st-env} \vdash_{NewType} tag(\ 'char_1 \underline{\mid} \ldots \underline{\mid}\ 'char_{k+1})}$$

$$\boxed{\text{wf-TagType: Tag}}\ \frac{tag \notin \textit{tags-of-env}(\textit{st-env})}{\textit{st-env} \vdash_{TagType} tag}$$

$$\frac{\text{(a)}\quad tag \notin \mathit{tags\text{-}of\text{-}env}(\mathit{st\text{-}env}) \qquad \text{(b)}\quad \mathit{st\text{-}env} \vdash_{\mathit{Type}} ty}{\mathit{st\text{-}env} \vdash_{\mathit{TagType}} tag\ \&\ ty} \quad \boxed{\text{wf-TagType: Assoc}}$$

A declaration in *NewType* is well-formed if:

- any tags introduced in a *NewType* have not already been used within the environment, and are not repeated in the declaration,
- the integer sub-range $(int_1 \mathbin{\underline{..}} int_2)$ in an *IntType* falls within the permitted range of ELLA-integers,
- the sequence of characters in a *CharType* contains no repetitions,
- any accompanying type *ty* in a *TagType* declaration *tag* & *ty* is well-formed.

The dynamic semantics for a *NewType* basically involves translating the ELLA type syntax into *EPA* syntax, which has a very similar structure. Little calculation of the corresponding sets of values, represented by each type expression, is performed at this stage — this occurs when the *EPA* terms are interpreted. In the following, the *EPA* expressions produced by the semantic definitions for *NewType* are naturally very similar to standard mathematical notation for set operations. However, at this stage they are simply terms of the *EPA* language, and are therefore distinguished by use of *EPA* brackets '⦇ _ ⦈':

Dynamic semantics equations:

$$\mathit{M\text{-}NewType} : \mathit{NewType} \to \mathit{Environment} \to \mathit{TypeExpr}$$

$$\mathit{M\text{-}NewType}[\![(tagty_1 \underline{|} \ldots \underline{|} tagty_{k+1})]\!](\rho,\sigma) = ⦇\ \{tag_1,\ \cdots\ ,\ tag_{k+1}\}\ ⦈$$

where

$$\forall\ i \in \{1,\ldots,k+1\} \bullet tagty_i = tag_i \wedge tag_i \in \mathit{Tag}$$

$$\mathit{M\text{-}NewType}[\![(tagty_1 \underline{|} \ldots \underline{|} tagty_{k+1})]\!](\rho,\sigma) = ⦇\ tyexpr_1 \cup\ \cdots\ \cup tyexpr_{k+1}\ ⦈$$

where

$$\forall\ i \in \{1,\ldots,k+1\} \bullet tyexpr_i = \mathit{M\text{-}TagType}(tagty_i)(\rho,\sigma)$$

$$\exists\ i \in \{1,\ldots,k+1\} \bullet tagty_i = [\![tag\ \&\ ty]\!]$$

$$\mathit{M\text{-}NewType}[\![tag/(i \mathbin{\underline{..}} j)]\!](\rho,\sigma) = ⦇\ \mathsf{tag}(tag, \mathsf{int}\ i\ j)\ ⦈$$

$$\mathit{M\text{-}NewType}[\![tag(\ 'char_1 \underline{|} \ldots \underline{|}\ 'char_{k+1})]\!](\rho,\sigma) = ⦇\ \mathsf{tag}(tag, \{char_1,\ \cdots\ , char_{k+1}\})\ ⦈$$

$$\mathit{M\text{-}TagType} : \mathit{TagType} \to \mathit{Environment} \to \mathit{TypeExpr}$$

$$\mathit{M\text{-}TagType}[\![tag\ \&\ ty]\!](\rho,\sigma) = ⦇\ \mathsf{tag}(tag, tagty)\ ⦈$$

where

$$tagty = \mathit{M\text{-}Type}(ty)(\rho,\sigma)$$

$$\mathit{M\text{-}TagType}[\![tag]\!](\rho,\sigma) = ⦇\ \{tag\}\ ⦈$$

5.4 Types

Structure:

$$
\boxed{\begin{array}{rcl}
\mathit{Type} & ::= & \mathit{typename} \mid \mathit{tupletype} \mid \mathit{powertype} \mid \mathit{stringtype} \mid \mathit{typevoid} \\
\mathit{TupleType} & ::= & (\mathit{type}_1, \ldots, \mathit{type}_{k+1}) \\
\mathit{PowerType} & ::= & [\mathit{nat}]\mathit{type} \\
\mathit{StringType} & ::= & \texttt{STRING}\ [\mathit{nat}]\mathit{type} \\
\mathit{TypeVoid} & ::= & (\ )
\end{array}}
$$

A *Type* construct allows more complex types to be built up from already-declared types, or it simply retrieves an existing type, permitting type aliasing. A tuple-type must be non-trivial, whereas a power-type can have unit-length. A string type takes a character-type *cty* and produces a set of character sequences; each sequence is tagged with the same tag used by *cty*.

The following relation and semantic function are defined on a construct-by-construct basis:-

$$(\_ \vdash_{\mathit{Type}} \_) \subseteq (\mathit{StaticEnvironment} \times \mathit{Type})$$

$$\mathit{M\text{-}Type} : \mathit{Type} \to \mathit{Environment} \to \mathit{TypeExpr}$$

Static semantics rules:

$$\boxed{\textsf{wf-Type : Typename}}\ \frac{tynm \in \textsf{dom}\, st\text{-}env}{st\text{-}env \vdash_{Type} tynm}$$

$$\boxed{\textsf{wf-Type : TupleType}}\ \frac{\forall\, i \in \{1, \ldots, k+1\} \bullet st\text{-}env \vdash_{Type} ty_i}{st\text{-}env \vdash_{Type} (ty_1, \ \cdots \ , ty_{k+1})}$$

$$\boxed{\textsf{wf-Type : PowerType}}\ \frac{\begin{array}{ll}\text{(a)} & n \in \mathbb{N} \wedge n \geq 1 \\ \text{(b)} & st\text{-}env \vdash_{Type} ty\end{array}}{st\text{-}env \vdash_{Type} [n]ty}$$

$$\boxed{\textsf{wf-Type : StringType}}\ \frac{\begin{array}{ll}\text{(a)} & st\text{-}env \vdash_{Type} ty \\ \text{(b)} & st\text{-}env \vdash_{is\text{-}char\text{-}type} ty \\ \text{(c)} & n \geq 1\end{array}}{st\text{-}env \vdash_{Type} \ \texttt{STRING}\ [n]ty}$$

$$\boxed{\textsf{wf-Type : TypeVoid}}\ \frac{\textsf{true}}{st\text{-}env \vdash_{Type} (\ )}$$

A *type* is well-formed precisely when it is:

- a typename that has already been declared, or
- a non-trivial tuple of types, each one of which is well-formed, or
- a power type of specified size and where the component type is also well-formed, or
- a string type where the component character type is well-formed, or
- a void type.

The semantics for *Type* simply converts the ELLA type expression syntax into *EPA* syntax, which is very similar. Note that when the tuple-type contains a single element, then the sequencing brackets are ignored and only the elemental type expression is returned.

Dynamic semantics equations:

$$\textit{M-Type}[\![tynm]\!](\rho,\sigma) = tynm$$

$$\textit{M-Type}[\![(ty_1)]\!](\rho,\sigma) = \textit{M-Type}[\![ty_1]\!](\rho,\sigma)$$

$$\textit{M-Type}[\![(ty_1, \cdots, ty_k)]\!](\rho,\sigma) = (\![\ type_1 \times \cdots \times type_k\ ]\!)$$

where

$\forall\, i \in \{1,\ldots,k\} \bullet type_i = \textit{M-Type}[\![ty_i]\!](\rho,\sigma)$

$k \geq 2$

$$\textit{M-Type}[\![[n]ty]\!](\rho,\sigma) = (\![\ type \uparrow n\ ]\!)$$

where

$type = \textit{M-Type}[\![ty]\!](\rho,\sigma)$

$$\textit{M-Type}[\![\texttt{STRING}\ [n]tynm]\!](\rho,\sigma) = (\![\ \mathsf{tag}(tag, type_1 \times \cdots \times type_{k+1})\ ]\!)$$

where

$\exists\, !char\text{-}type \in CharRange \bullet$

$\sigma(tynm) = \mathsf{tag}(tag, chr\text{-}type) \wedge$

$\forall\, i \in \{1,\ldots,k+1\} \bullet\ type_i = chr\text{-}type$

$$\textit{M-Type}[\![(\ )]\!](\rho,\sigma) = (\![\ \mathsf{unit}\ ]\!)$$

5.5 Type declarations

Structure:

TypeDecl	$::=$	*typenaming* | *typeformer*
TypeNaming	$::=$	`TYPE` *typename* $=$ *type*
TypeFormer	$::=$	`TYPE` *typename* $=$ `NEW` *newtype*

The *type declaration* introduces a name for a type structure. The first form above introduces a synonym for an type that is already formed, or for a complex type, whereas the second introduces a freshly named primitive structure. Any type aliasing is removed at this stage by dereferencing — in rule TypeNaming (b) for static checking, and in the first form of the dynamic semantic function *M-TypeDecl*.

Static semantics rules:

Let $ty \in Type$:

$$\textsf{wf-Typedecl : TypeNaming (a)}\quad \frac{\begin{array}{ll}\text{(a)} & tynm \notin \mathsf{dom}\, st\text{-}env \\ \text{(b)} & ty \notin TypeName \\ \text{(c)} & st\text{-}env \vdash_{Type} ty \\ \text{(d)} & td = [\![\, \texttt{TYPE}\ tynm = ty]\!] \\ \text{(e)} & new\text{-}st\text{-}env = st\text{-}env \cup \{tynm \mapsto td\}\end{array}}{st\text{-}env \vdash_{Typedecl} [\![\, \texttt{TYPE}\ tynm = ty]\!]\ \textsf{produces}\ new\text{-}st\text{-}env}$$

$$\textsf{wf-Typedecl : TypeNaming (b)}\quad \frac{\begin{array}{ll}\text{(a)} & tynm \notin \mathsf{dom}\, st\text{-}env \\ \text{(b)} & ty \in TypeName \\ \text{(c)} & st\text{-}env \vdash_{Type} ty \\ \text{(d)} & td = st\text{-}env(ty) \\ \text{(e)} & new\text{-}st\text{-}env = st\text{-}env \cup \{tynm \mapsto td\}\end{array}}{st\text{-}env \vdash_{Typedecl} [\![\, \texttt{TYPE}\ tynm = ty]\!]\ \textsf{produces}\ new\text{-}st\text{-}env}$$

$$\textsf{wf-Typedecl : TypeFormer}\quad \frac{\begin{array}{ll}\text{(a)} & tynm \notin \mathsf{dom}\, st\text{-}env \\ \text{(b)} & st\text{-}env \vdash_{Newtype} nty \\ \text{(c)} & td = [\![\, \texttt{TYPE}\ tynm = \ \texttt{NEW}\ nty]\!] \\ \text{(d)} & new\text{-}st\text{-}env = st\text{-}env \cup \{tynm \mapsto td\}\end{array}}{st\text{-}env \vdash_{Typedecl} [\![\, \texttt{TYPE}\ tynm = \ \texttt{NEW}\ nty]\!]\ \textsf{produces}\ new\text{-}st\text{-}env}$$

A type declaration is well-formed precisely when:

- the type name being introduced has not already been defined,
- synonym types must already be well-formed — any aliasing of names is resolved,

- a newtype is well-formed.

The semantics of each type declaration is to add a name binding for the declared type, after translating any newly-tagged type expressions. If the declaration produces a type alias, then the right-hand side type-name is replaced by the already constructed type.

Dynamic semantics equations:

$$\mathit{M\text{-}Typedecl} : \mathit{TypeDecl} \rightarrow \mathit{Environment} \rightarrow \mathit{DynamicEnvironment}$$

$$\mathit{M\text{-}Typedecl}[\![\ \texttt{TYPE}\ \mathit{tynm} = \mathit{ty}]\!](\rho,\sigma) = \mathit{tyd\text{-}env}$$

where

$$\mathit{tyd\text{-}env} = \sigma \cup \{\mathit{tynm} \mapsto \mathit{tyd}\}$$

$$\mathit{tyd} = (\!|\ \mathsf{type}\ \mathit{tynm}\ \underline{\triangle}\ \mathit{tyexpr}\ |\!)$$

$$\mathit{tyexpr} = \sigma(\mathit{ty})$$

$$\mathit{ty} \in \mathit{TypeName}$$

$$\mathit{M\text{-}Typedecl}[\![\ \texttt{TYPE}\ \mathit{tynm} = \mathit{ty}]\!](\rho,\sigma) = \mathit{tyd\text{-}env}$$

where

$$\mathit{tyd\text{-}env} = \sigma \cup \{\mathit{tynm} \mapsto \mathit{tyd}\}$$

$$\mathit{tyd} = (\!|\ \mathsf{type}\ \mathit{tynm}\ \underline{\triangle}\ \mathit{tyexpr}\ |\!)$$

$$\mathit{tyexpr} = \mathit{M\text{-}Type}[\![\mathit{ty}]\!](\rho,\sigma)$$

$$\mathit{ty} \in \mathit{Type} - \mathit{TypeName}$$

$$\mathit{M\text{-}Typedecl}[\![\ \texttt{TYPE}\ \mathit{tynm} =\ \texttt{NEW}\ \mathit{nty}]\!](\rho,\sigma) = \mathit{tyd\text{-}env}$$

where

$$\mathit{tyd\text{-}env} = \sigma \cup \{\mathit{tynm} \mapsto \mathit{tyd}\}$$

$$\mathit{tyd} = (\!|\ \mathsf{type}\ \mathit{tynm}\ \underline{\triangle}\ \mathit{tyexpr}\ |\!)$$

$$\mathit{tyexpr} = \mathit{M\text{-}Newtype}[\![\mathit{nty}]\!](\rho,\sigma)$$

5.6 Enumerated

Structure:

$$\begin{array}{|rcl|}\hline \mathit{Enumerated} & ::= & \mathit{tag} \mid \mathit{enumint} \mid \mathit{enumchar} \mid \mathit{stringenum} \\ \mathit{EnumInt} & ::= & \mathit{tag}/\mathit{int} \\ \mathit{EnumChar} & ::= & \mathit{tag}\,'\,\mathit{char} \\ \mathit{StringEnum} & ::= & \mathit{tag}\,''\,\mathit{char}_1\ \ldots\ \mathit{char}_{k+1}'' \\ \hline \end{array}$$

Enumerated simply returns a constant value v of the appropriate type. A basic process term is produced, which has no inputs, a single state, and returns the value v on its output. The enumerated value can be either a tag *taglit*, a tagged integer $\mathit{tag}/\mathit{taglit}$, a tagged character $\mathit{tag}'\,\mathit{chr}$, or a tagged string $\mathit{tag}\,''\,\mathit{ch}_1\ \ldots\ \mathit{ch}_{k+1}\,''$.

Static semantics rules:

(a) $\mathit{tyn} \in \mathsf{dom}\,\mathit{st\text{-}env} \wedge \mathit{newtype\text{-}of}(\mathit{st\text{-}env}(\mathit{tyn})) \subset \mathit{Tag}$

(b) $\mathit{tag} \in \mathit{newtype\text{-}of}(\mathit{st\text{-}env}(\mathit{tyn}))$

wf-Enumerated: Tag

$\mathit{st\text{-}env} \vdash_{\mathit{Enumerated}} [\![\mathit{tag}]\!]$ has type tyn

(a) $\mathit{tyn} \in \mathsf{dom}\,\mathit{st\text{-}env} \wedge [\![\mathit{tag}/(\mathit{lb}\underline{..}\mathit{ub})]\!] \in \mathit{newtype\text{-}of}(\mathit{st\text{-}env}(\mathit{tyn}))$

(b) $\mathit{int} \in \mathit{range\text{-}set}[\![\mathit{tag}/(\mathit{lb}\underline{..}\mathit{ub})]\!]$

wf-Enumerated: EnumInt

$\mathit{st\text{-}env} \vdash_{\mathit{Enumerated}} [\![\mathit{tag}/\mathit{int}]\!]$ has type tyn

(a) $\mathit{tyn} \in \mathsf{dom}\,\mathit{st\text{-}env} \wedge$
$[\![\mathit{tag}/(\mathit{ch}_1,\ldots,\mathit{ch}_{k+1})]\!] \in \mathit{newtype\text{-}of}(\mathit{st\text{-}env}(\mathit{tyn}))$

(b) $\mathit{char} \in \mathit{range\text{-}set}[\![(\mathit{ch}_1,\ldots,\mathit{ch}_{k+1})]\!]$

wf-Enumerated: EnumChar

$\mathit{st\text{-}env} \vdash_{\mathit{Enumerated}} [\![\mathit{tag}\,'\,\mathit{char}]\!]$ has type tyn

(a) $\mathit{chr\text{-}nty} \in \mathit{newtype\text{-}of}(\mathit{st\text{-}env}(\mathit{chty}))$

(b) $\mathit{tag} \in \mathit{tags\text{-}of\text{-}newtype}(\mathit{chr\text{-}nty})$

(c) $\forall\, i \in \{1,\ldots,k+1\} \bullet \mathit{ch}_i \in \mathit{range\text{-}set}[\![\mathit{chr\text{-}nty}]\!]$

(d) $k+1 = n$

wf-Enumerated: StringEnum

$\mathit{st\text{-}env} \vdash_{\mathit{Enumerated}} [\![\mathit{tag}''\,\mathit{ch}_1\ \ldots\ \mathit{ch}_{k+1}\,'']\!]$ has type $[\![\ \mathtt{STRING}\,[n]\mathit{chty}]\!]$

Enumerated values are well-formed when:

- the typename *tyn* returns a valid type,
- the enumerated value is a member of the set of values associated with *tyn*,
- for constant strings, the characters are members of the character set *ch-nty*, and the string is of the correct length n.

Enumerated returns a call $Constant_{[TY,enum]}$ to a process which has no inputs and has the value $enum \in TY$ on its output:

Dynamic semantics equations:

$$\textit{M-Enumerated} : \textit{Enumerated} \rightarrow \textit{ChannelVarName} \rightarrow \textit{Environment} \rightarrow \textit{ProcExpr}$$

$$\textit{M-Enumerated}[\![enum]\!](oc)(\rho,\sigma) = \langle\!\![\ Constant_{[TY,val]}(oc)\ ]\!\!\rangle$$

where

$$val = \textit{make-epa-constant}[\![enum]\!]$$

$$TY = \textit{type-of}[\![tag/int]\!](\rho,\sigma)$$

The operation *make-epa-constant*$[\![enum]\!]$ creates an *EPA* constant value for the ELLA constant *enum*:

$$\textit{make-epa-constant}: \textit{Enumerated} \rightarrow \textit{ALL-VALUE}$$

$$\begin{aligned} \textit{make-epa-constant}[\![tag]\!] &= \langle\!\![\ tag\ ]\!\!\rangle \\ \textit{make-epa-constant}[\![tag/int]\!] &= \langle\!\![\ \mathsf{tag}(tag,int)\ ]\!\!\rangle \\ \textit{make-epa-constant}[\![tag\ '\ chr]\!] &= \langle\!\![\ \mathsf{tag}(tag,chr)\ ]\!\!\rangle \\ \textit{make-epa-constant}[\![tag\ ''\ ch_1\ \ldots\ ch_{k+1}\ '']\!] &= \langle\!\![\ \mathsf{tag}(tag,\mathsf{tuple}(ch_1,\ldots,ch_{k+1}))\ ]\!\!\rangle \end{aligned}$$

The generic process definition $Constant_{[ty,val]}$ is given by:

$$\langle\!\![\ \mathsf{process}\ Constant_{[ty,val]}(o\colon ty)\ \underline{\triangle}\ o(val) :: Constant_{[ty,val]}(o)\ ]\!\!\rangle$$

where

$$\{o\} = \textit{make-ChanVars}(1,\emptyset)$$

Note that $Constant_{[ty,val]}$ only has an output channel declaration, and does not have any variable or input channel declarations.

5.7 Primitives

Structure:

$$\boxed{\mathit{Primitive} \quad ::= \quad \mathit{adelay} \mid \mathit{idelay} \mid \mathit{sample}}$$

An ELLA primitive function introduces various kinds of delay into a design. Its behaviour is represented by a normal form process definition, where the *initial* process definition makes a call to a generic process.

The following relation and semantic function are defined on a construct-by-construct basis:-

$$(\_ \vdash_{\mathit{Primitive}} \_ \text{ has type } \_) \subseteq (\mathit{StaticEnvironment} \times \mathit{Primitive} \times \mathit{Type})$$

$$\mathit{M\text{-}Primitive} : \mathit{Primitive} \rightarrow \mathit{Name} \rightarrow \mathit{Environment} \rightarrow \mathit{ProcDefn}$$

5.7.1 Ambiguity Delay

Structure:

$$\boxed{Adelay \quad ::= \qquad \texttt{DELAY}\,(unit_1, nat_1, unit_2, nat_2)}$$

The primitive function $\texttt{DELAY}\,(unm_1, del_1, unm_2, del_2)$ produces an initial output value given by unm_1. If the input changes, then after del_1 time-units the output value changes to the value given by unm_2; after a further $del_2 - del_1$ time-units the output value changes to the new input value. If, however, the input is stable for fewer than del_2 time-units, then the output changes to the value given by unm_2 after del_1 time-units — it remains this value until the input has been stable for del_2 time-units, when it changes to the new stable value.

Static semantics rules:

(a) $del_1 \geq 1 \wedge del_1 \leq del_2$

(b) $st\text{-}env \vdash_{Unit} unm_1$ has type ty

(c) $st\text{-}env \vdash_{Unit} unm_2$ has type ty

(d) $st\text{-}env \vdash_{is\text{-}const\text{-}value} unm_1$

(e) $st\text{-}env \vdash_{is\text{-}const\text{-}value} unm_2$

wf-Primitive: Adelay

$st\text{-}env \vdash_{Primitive} [\![\texttt{DELAY}\,(unm_1, del_1, unm_2, del_2)]\!]$ has type ty

The ambiguity delay is well-formed precisely when:

- $1 \leq del_1 \leq del_2$, and
- unm_1 and unm_2 are literals of type ty.

Dynamic semantics equations:

$$M\text{-}Primitive[\![\texttt{DELAY}\,(unm_1, del_1, unm_2, del_2)]\!](fnnm)(\rho, \sigma) =$$

$$(\!\![\text{ process } pnm(i\colon TY \;/\; o\colon TY) \;\underline{\triangle}$$

$$\square x\colon init_1 \;\uparrow\; k \;\cdot\; Adel\text{-}Aux_{[TY, n_2, k, init_2]}(x \,;\, i \;/\; o) \,]\!\!)$$

where

$pnm = proc\text{-}name(fnnm)$

$k = del_1 + del_2 - 1$

$init_1 = init\text{-}values[\![unm_1]\!](\rho, \sigma)$

$init_2 = init\text{-}values[\![unm_2]\!](\rho, \sigma)$

$\{i, o\} = make\text{-}ChanVars(2, \emptyset)$

$TY = type\text{-}of[\![unm_1]\!](\rho, \sigma)$

The process definition named $proc\text{-}name(fnnm)$ represents the semantics of the ambiguity delay function, by calling generic process $Adel\text{-}Aux_{[ty,n,k,set]}$ once for each value x containing elements from the set expression $init_1$, and taking an 'ambiguity' element from the set expression $init_2$. This process uses function $Adel_{[ty,n]}$:

$$\llbracket \text{process } \mathit{Adel\text{-}Aux}_{[ty,n,k,set]}(a\colon ty \uparrow k \,;\, i\colon ty \,/\, o\colon ty) \ \underline{\triangle}$$
$$\square x\colon ty,\ v\colon set \ \bullet$$
$$i(x) \,/\, o(\mathit{Adel}_{[ty,n]}(a,v)) :: \mathit{Adel\text{-}Aux}_{[ty,n,k,set]}([x] \frown \mathsf{trim}(a,1,k-1) \,;\, i \,/\, o) \rrbracket$$

where

$$\{i,o\} = \mathit{make\text{-}ChanVars}(2,\emptyset)$$
$$\{a\} = \mathit{make\text{-}ExprVars}(1)$$

$$\llbracket \text{function } \mathit{Adel}_{[ty,n]}(a\colon ty, v\colon ty) : ty \ \underline{\triangle} \ \textbf{case}\ \mathsf{subsequence}(a,n) :$$
$$\{\mathsf{true}\} \rightarrow \mathsf{index}(a,n)$$
$$\textbf{else}\ v$$
$$\textbf{end}\rrbracket$$

where

$$\{a,v\} = \mathit{make\text{-}ExprVars}(2)$$

Ambiguity delay is represented by the process *Adel-Aux*, which stores the last k inputs received in vector a. If there has been a sequence of at least n same-value inputs within this store, with $n < k$, then the n^{th} value of a is output (first case-item in function $\mathit{Adel}_{[ty,n]}$) otherwise the 'ambiguity' value v is output (second case-item in function $\mathit{Adel}_{[ty,n]}$), where $v \in set$. Clearly, v will be produced either when the input has changed between two stable values, or when it is unstable, i.e., not constant for more that n time-steps. The initial call to *Adel-Aux* is made with k copies of a value taken from $init_1$.

The *EPA* function expressions $[\_]$, $\_ \frown \_$ represent sequencing and sequence concatenation, respectively. The function expression $\mathsf{trim}(s,i,j)$ represents sub-sequencing, forming $s(i,j)$ from sequence s, and $\mathsf{index}(s,n)$ represents projection, returning the n^{th} element of s. The interpreted data function $\mathsf{subsequence}(l,n)$ will be true if there is a same-value subsequence with at least n elements in l, and false otherwise.

The operation $\mathit{init\text{-}values}(un)(\rho,\sigma)$ returns an *EPA* term representing a set expression for a constant, or the type, depending upon whether un is an enumeration value, 'query', row of constants or string constant:

$init\text{-}values : Unit \rightarrow Environment \rightarrow SetExpr$

$\forall\ un \in Enumerated \bullet$

$init\text{-}values[\![un]\!] = (\!\![\ \{ENUM\}\ ]\!\!)$

where

$ENUM = make\text{-}epa\text{-}constant[\![un]\!]$

$init\text{-}values[\![?\,type]\!](\rho,\sigma) = M\text{-}Type[\![type]\!](\rho,\sigma)$

$init\text{-}values[\![cnst_1,\ \cdots\ ,cnst_{k+1}]\!] = (\!\![\ cset_1 \times \ldots \times cset_{k+1}\ ]\!\!)$

where

$\forall\ i \in \{1,\ldots,k+1\} \bullet cset_i = init\text{-}values[\![const_i]\!]$

$init\text{-}values[\![\ \texttt{STRING}\ [n]\ tag''\ chr]\!] = (\!\![\ \{\ \mathsf{tag}(tag, \mathsf{tuple}(ch_1,\ldots,ch_n))\ \}\ ]\!\!)$

where

$\forall\ i \in \{1,\ldots,n\} \bullet ch_n = chr$

5.7.2 Inertial Delay

Structure:

$$\boxed{\textit{Idelay} \quad ::= \qquad \texttt{IDELAY}\ (\textit{unit}, \textit{nat})}$$

The primitive function $\texttt{IDELAY}\ (\textit{unm}, \textit{del})$ produces an initial output value given by *unm*, and then input values are reproduced on the output after a delay of *del* time-units. If, however, the input is stable for fewer than *del* time-units, then the input change is ignored. If *unm* returns 'query' then one from a set of initial output values is produced non-deterministically.

Static semantics rules:

$$\boxed{\text{wf-Primitive: Idelay}}\quad \frac{\begin{array}{ll}\text{(a)} & \textit{del} \geq 1 \\ \text{(b)} & \textit{st-env} \vdash_{\textit{Unit}} \textit{unm} \text{ has type } \textit{ty} \\ \text{(c)} & \textit{st-env} \vdash_{\textit{is-const-value}} \textit{unm}\end{array}}{\textit{st-env} \vdash_{\textit{Primitive}} [\![\ \texttt{IDELAY}\ (\textit{unm}, \textit{del})]\!] \text{ has type } \textit{ty}}$$

The inertial delay is well-formed precisely when:

- $\textit{del} \geq 1$, and
- *unm* is a literal of type *ty*.

Dynamic semantics equations:

$$\textit{M-Primitive}[\![\ \texttt{IDELAY}\ (\textit{unm}, \textit{del})]\!](\textit{fnnm})(\rho, \sigma) =$$

$$(\!\![\ \text{process}\ \textit{pnm}(i\!: \textit{TY}\ /\ o\!: \textit{TY})\ \triangleq \square\, v\!: \textit{initset} \cdot \textit{Idel-Aux}_{[\textit{TY},\textit{del}]}(\textit{del}, v, v\ ;\ i\ /\ o)\ ]\!\!)$$

where

$$\textit{pnm} = \textit{proc-name}(\textit{fnm})$$

$$\textit{initset} = \textit{init-values}[\![\textit{unm}]\!](\rho, \sigma)$$

$$\{i, o\} = \textit{make-ChanVars}(2, \emptyset)$$

$$\textit{TY} = \textit{type-of}[\![\textit{unm}]\!](\rho, \sigma)$$

The process definition *proc-name*(*fnnm*) represents the semantics of the inertial delay function for type *TY*, by calling generic process $\textit{Idel-Aux}_{[ty,n]}$ once for each value in the set *initset*. This uses two functions $\textit{Idel-1}_{[ty,n]}, \textit{Idel-2}_{[ty,n]}$:

$⟦$ process $\mathit{Idel\text{-}Aux}_{[ty,n]}(c\colon \textsf{int}\ 1\ n, a\colon ty, b\colon ty\ ;\ i\colon ty\ /\ o\colon ty) \,\underline{\triangle}$

$\quad \Box x\colon ty\ \bullet\ i(x)\ /\ o(\mathit{Idel\text{-}1}_{[ty,n]}(c,a,b))$

$\qquad :: \mathit{Idel\text{-}Aux}_{[ty,n]}(\mathit{Idel\text{-}2}_{[ty,n]}(c,a,b,x), x, \mathit{Idel\text{-}1}_{[ty,n]}(c,a,b)\ ;\ i\ /\ o)\ ⟧$

where

$\{i,o\} = \mathit{make\text{-}ChanVars}(2,\emptyset)$

$⟦$function $\mathit{Idel\text{-}1}_{[ty,n]}(c\colon \textsf{int}\ 1\ n, a\colon ty, b\colon ty) : ty\ \underline{\triangle}$

case $c = n$:

$\{\textsf{true}\} \to a$

else b

end$⟧$

where

$\{c,a,b\} = \mathit{make\text{-}ExprVars}(3)$

$⟦$function $\mathit{Idel\text{-}2}_{[ty,n]}(c\colon \textsf{int}\ 1\ n, a\colon ty, b\colon ty, x\colon ty) : \textsf{int}\ 1\ n\ \underline{\triangle}$

case $x = a$:

$\{\textsf{true}\} \to$ **case** $c = n$:

$\{\textsf{true}\} \to c$

else $c+1$

end

else 1

end$⟧$

where

$\{c,a,b,x\} = \mathit{make\text{-}ExprVars}(4)$

With inertial delay, *initset* is a set expression giving the initial values to be used from type *TY*, determined by *unm*. For example, if *TY* represents a boolean type *bool* with members `f`,`t`, then *initset* will be a singleton for *IDelay*$(\{\mathtt{f}\},3)$, or the complete set represented by *bool* in *IDelay*$(?\mathit{bool},5)$.

The 'state' variables are used as follows: counter c indicates the number of same-value inputs received, store a holds the last input value, and store b holds the last output value. Store a is always set to the current input value x. If $c = n$ then at least n same-value inputs a have been received so the current output and last-output store change to a. If the current input value is not the same as the last, held in a, then the counter is reset to 1. If $c < n$ then the last output store stays the same, and this value is output. If the input value is the same as the last, then the counter is incremented, since the length of the sequence of same-value inputs has grown, but otherwise it is reset to 1, since a new input sequence must begin.

5.7.3 Sample

Structure:

$$\boxed{\textit{Sample} \quad ::= \quad \texttt{SAMPLE}\ (nat_1, unit, int_2)}$$

The primitive function $\texttt{SAMPLE}\ (n_1, unm, i_2)$ selects the value of its input unit at regular intervals $n_1 \geq 1$ and then outputs this value for the duration of the interval. The start of the sampling process can also be skewed by an amount i_2, where an initial value *unm* is produced until the sampling process begins at time i_2. Fresh input samples are therefore taken at times $n_1 + i_2$, $2 \times n_1 + i_2$, and so on. Exactly the same conditions apply if $i_2 < 0$ — this would be the same as starting sampling at $n_1 - \mid i_2 \mid$.

Static semantics rules:

$$\frac{\begin{array}{ll}(a) & n_1 > \mid i_2 \mid \geq 0 \\ (b) & \textit{st-env} \vdash_{Unit} \textit{unm}\ \text{has type}\ ty \\ (c) & \textit{st-env} \vdash_{\textit{is-const-value}} \textit{unm}\end{array}}{\textit{st-env} \vdash_{Primitive} [\![\ \texttt{SAMPLE}\ (n_1, unm, i_2)]\!]\ \text{has type}\ ty}\ \boxed{\text{wf-Primitive: Sample}}$$

The sample is well-formed precisely when:

- $n_1 \ > \mid i_2 \mid$
- *unm* is a literal of type *ty*.

Dynamic semantics equations:

$$\textit{M-Primitive}[\![\ \texttt{SAMPLE}\ (n_1, unm, i_2)]\!](\textit{fnnm})(\rho, \sigma) =$$

$$(\!\![\ \text{process}\ pnm(i\colon TY\ /\ o\colon TY)\ \underline{\triangle}$$

$$\Box\, v\colon initset\ \bullet\ \textit{Sample-Aux}_{[TY,n_1]}(initcount, v\,;\ i\ /\ o)\ ]\!\!)$$

where

$$pnm = \textit{proc-name}(\textit{fnnm})$$

$$initset = \textit{init-values}[\![unm]\!](\rho, \sigma)$$

$$initcount = (n_1 - i_2)\ \mathrm{mod}\ n_1$$

$$TY = \textit{type-of}(unm)(\rho, \sigma)$$

$$\{i, o\} = \textit{make-ChanVars}(2, \emptyset)$$

This process definition represents the semantics of the sampling function, by calling generic process $\textit{Sample-Aux}_{[TY,n_1]}$ once for each element in *initset*, and exhibiting a sampling offset of i_2 time-steps. $\textit{Sample-Aux}_{[TY,n_1]}$ uses functions *Sample*-1 and *Sample*-2:

$\llbracket$ process $\mathit{Sample\text{-}Aux}_{[ty,del]}(c\colon \text{int}\ 0\ del-1, a\colon ty\ ;\ i\colon ty\ /\ o\colon ty) \underline{\triangle}$

$\quad \Box\, x : ty\ \bullet\ i(x)\ /\ o(\mathit{Sample}\text{-}1_{[ty,del]}(c,a,x))$

$\qquad ::\ \mathit{Sample\text{-}Aux}_{[ty,del]}(\mathit{Sample}\text{-}2_{[del]}(c), \mathit{Sample}\text{-}1_{[ty,del]}(c,a,x)\ ;\ i\ /\ o)\ \rrbracket$

where $\{i,o\} = \mathit{make\text{-}ChanVars}(2,\emptyset)$

$\llbracket$function $\mathit{Sample}\text{-}1_{[ty,del]}(c\colon \text{int}\ 0\ del-1, a\colon ty, x\colon ty) : ty\ \underline{\triangle}$

case $c = 0$:

$\{\text{true}\} \rightarrow x$

else a

end$\rrbracket$

where

$\{c,a,x\} = \mathit{make\text{-}ExprVars}(3)$

$\llbracket$ function $\mathit{Sample}\text{-}2_{[del]}(c\colon \text{int}\ 0\ del-1) : \text{int}\ 0\ del-1\ \ \underline{\triangle}$

$(c+1) \bmod del\ \rrbracket$

where

$\{c\} = \mathit{make\text{-}ExprVars}(1)$

With sample, *initset* is a set expression representing the initial value(s) from type *TY*, given by the unit *unm*. The main process definition — $\mathit{Sample\text{-}Aux}_{[TY,n_1]}$ — has two cases:

- $c = 0$: take a new sample x, store it and output it immediately,
- $c \neq 0$: add 1 to c and output sampled value a, ignoring new input x; if c is the maximum value n_1, then reset to 0.

5.8 Unit

Structure:

$$\boxed{\begin{array}{lll} \mathit{Unit} & ::= & \mathit{enumerated} \mid \mathit{signal} \mid \mathit{assoc} \mid \mathit{unitstr} \mid \mathit{replicate} \\ & \mid & \mathit{query} \mid \mathit{index} \mid \mathit{dyindex} \mid \mathit{replace} \mid \mathit{trim} \mid \mathit{conc} \\ & \mid & \mathit{sequence} \mid \mathit{extract} \mid \mathit{instance} \mid \mathit{case} \mid \mathit{unitvoid} \end{array}}$$

The following relations and semantic function are defined on a construct-by-construct basis:-

$$(\_ \vdash_{Unit} \_ \ \textsf{produces}\ \_) \subseteq (\mathit{StaticEnvironment} \times \mathit{Unit} \times \mathit{StaticEnvironment})$$
$$(\_ \vdash_{Unit} \_ \ \textsf{has type}\ \_) \subseteq (\mathit{StaticEnvironment} \times \mathit{Unit} \times \mathit{Type})$$

$$\mathit{M\text{-}Unit} : \mathit{Unit} \to \mathit{ChannelVarName} \to \mathit{Environment} \to (\mathit{ProcExpr} \times \mathit{DynamicEnvironment})$$

Note that *M-Unit* returns a pair, the first element being the process term representing the unit. The second term is a potentially extended dynamic environment — if any *local* type- or function-declarations have occurred in a block contained in the unit, then these must be added to the environment.

The relation $(\_ \vdash_{Unit} \_ \ \textsf{produces}\ \_)$ effectively controls the update of the static environment, when new type/function declarations are received. Of the unit constructs, only units $un \in \mathit{Block}$ can produce an altered static environment, by containing a local type/function declaration step. Therefore for the remainder, the relation $(\_ \vdash_{Unit} \_ \ \textsf{produces}\ \_)$ returns the environment produced by any sub-units, which may of course contain a *Block* structure. For these cases the definition of the relation is given below:

wf-Unit : unitvoid-prod

$$\overline{\mathit{st\text{-}env} \vdash_{Unit} [\![(\ )]\!] \ \textsf{produces}\ \mathit{st\text{-}env}}$$

wf-Unit : assoc-prod

$$\frac{\mathit{st\text{-}env} \vdash_{Unit} \mathit{un}\ \textsf{produces}\ \mathit{st\text{-}env}_1}{\mathit{st\text{-}env} \vdash_{Unit} [\![\mathit{enum}\ \&\ \mathit{un}]\!] \ \textsf{produces}\ \mathit{st\text{-}env}_1}$$

wf-Unit : string-prod

$$\frac{\mathit{st\text{-}env} \vdash_{Unit} \mathit{un}\ \textsf{produces}\ \mathit{st\text{-}env}_1}{\mathit{st\text{-}env} \vdash_{Unit} [\![\ \texttt{STRING}\ [i]\mathit{un}]\!] \ \textsf{produces}\ \mathit{st\text{-}env}_1}$$

wf-Unit : replicate-prod

$$\frac{\mathit{st\text{-}env} \vdash_{Unit} \mathit{un}\ \textsf{produces}\ \mathit{st\text{-}env}_1}{\mathit{st\text{-}env} \vdash_{Unit} [\![[\ \mathit{nat}\ ]\ \mathit{un}]\!] \ \textsf{produces}\ \mathit{st\text{-}env}_1}$$

$$\boxed{\textsf{wf-Unit : query-prod}}\;\; \frac{}{\textit{st-env} \vdash_{\textit{Unit}} [\![\ ?\ \textit{type}]\!] \text{ produces } \textit{st-env}}$$

$$\boxed{\textsf{wf-Unit : index-prod}}\;\; \frac{\textit{st-env} \vdash_{\textit{Unit}} \textit{un} \text{ produces } \textit{st-env}_1}{\textit{st-env} \vdash_{\textit{Unit}} [\![\textit{un}\ [\ n\ ]]\!] \text{ produces } \textit{st-env}_1}$$

$$\boxed{\textsf{wf-Unit : dyindex-prod}}\;\; \frac{\begin{array}{ll}\text{(a)} & \textit{st-env} \vdash_{\textit{Unit}} \textit{un}_1 \text{ produces } \textit{st-env}_1\\ \text{(b)} & \textit{st-env}_1 \vdash_{\textit{Unit}} \textit{un}_2 \text{ produces } \textit{st-env}_2\end{array}}{\textit{st-env} \vdash_{\textit{Unit}} [\![\textit{un}_1\ [\![\ \textit{un}_2\ ]\!]]\!] \text{ produces } \textit{st-env}_2}$$

$$\boxed{\textsf{wf-Unit : replace-prod}}\;\; \frac{\begin{array}{ll}\text{(a)} & \textit{st-env} \vdash_{\textit{Unit}} \textit{un}_1 \text{ produces } \textit{st-env}_1\\ \text{(b)} & \textit{st-env}_1 \vdash_{\textit{Unit}} \textit{un}_2 \text{ produces } \textit{st-env}_2\\ \text{(c)} & \textit{st-env}_2 \vdash_{\textit{Unit}} \textit{un}_3 \text{ produces } \textit{st-env}_3\end{array}}{\textit{st-env} \vdash_{\textit{Unit}} [\![\ \texttt{REPLACE}\ (\textit{un}_1, \textit{un}_2, \textit{un}_3)]\!] \text{ produces } \textit{st-env}_3}$$

$$\boxed{\textsf{wf-Unit : trim-prod}}\;\; \frac{\textit{st-env} \vdash_{\textit{Unit}} \textit{un} \text{ produces } \textit{st-env}_1}{\textit{st-env} \vdash_{\textit{Unit}} [\![\textit{un}\ [\ \textit{nat}_1\ \underline{..}\ \textit{nat}_2\ ]]\!] \text{ produces } \textit{st-env}_1}$$

$$\boxed{\textsf{wf-Unit : conc-prod}}\;\; \frac{\begin{array}{ll}\text{(a)} & \textit{st-env} \vdash_{\textit{Unit}} \textit{un}_1 \text{ produces } \textit{st-env}_1\\ \text{(b)} & \textit{st-env}_1 \vdash_{\textit{Unit}} \textit{un}_2 \text{ produces } \textit{st-env}_2\end{array}}{\textit{st-env} \vdash_{\textit{Unit}} [\![\textit{un}_1\ \texttt{CONC}\ \textit{un}_2]\!] \text{ produces } \textit{st-env}_2}$$

$$\boxed{\textsf{wf-Unit : sequence-prod}}\;\; \frac{\begin{array}{ll}\text{(a)} & \textit{st-env}_1 = \textit{st-env}\\ \text{(b)} & \forall\ i \in \{1, \ldots, k+1\} \bullet \textit{st-env}_i \vdash_{\textit{Unit}} \textit{un}_i \text{ produces } \textit{st-env}_{i+1}\end{array}}{\textit{st-env} \vdash_{\textit{Unit}} [\![(\textit{un}_1\ ,\ \ldots\ ,\ \textit{un}_{k+1})]\!] \text{ produces } \textit{st-env}_{k+2}}$$

$$\boxed{\textsf{wf-Unit : extract-prod}}\;\; \frac{\textit{st-env} \vdash_{\textit{Unit}} \textit{un} \text{ produces } \textit{st-env}_1}{\textit{st-env} \vdash_{\textit{Unit}} [\![\textit{un}\ //\ \textit{enum}]\!] \text{ produces } \textit{st-env}_1}$$

$$\boxed{\textsf{wf-Unit : signal-prod}}\;\; \frac{}{\textit{st-env} \vdash_{\textit{Unit}} [\![\textit{name}]\!] \text{ produces } \textit{st-env}}$$

$$\boxed{\textsf{wf-Unit : enumerated-prod}}\;\frac{}{\textit{st-env} \vdash_{\textit{Unit}} [\![\textit{enum}]\!] \text{ produces } \textit{st-env}}$$

$$\boxed{\textsf{wf-Unit : instance-prod}}\;\frac{\textit{st-env} \vdash_{\textit{Unit}} \textit{un} \text{ produces } \textit{st-env}_1}{\textit{st-env} \vdash_{\textit{Unit}} [\![\textit{fnnm un}]\!] \text{ produces } \textit{st-env}_1}$$

$$\boxed{\textsf{wf-Unit : case-prod}}\;\frac{\begin{array}{ll}\text{(a)} & \textit{st-env} \vdash_{\textit{Unit}} \textit{un}_1 \text{ produces } \textit{st-env}_1 \\ \text{(b)} & \forall\, i \in \{1, \ldots, k+1\} \bullet \textit{st-env}_i \vdash_{\textit{Caseitem}} \textit{ci}_i \text{ produces } \textit{st-env}_{i+1} \\ \text{(c)} & \textit{st-env}_{k+2} \vdash_{\textit{Unit}} \textit{un}_2 \text{ produces } \textit{st-env}_{k+3}\end{array}}{\textit{st-env} \vdash_{\textit{Unit}} [\![\ \texttt{CASE}\ \textit{un}_1\ \texttt{OF}\ \textit{ci}_1\ ,\ \ldots\ ,\ \textit{ci}_{k+1}\ \texttt{ELSE}\ \textit{un}_2\ \texttt{ESAC}\ ]\!] \text{ produces } \textit{st-env}_{k+3}}$$

$$(\_ \vdash_{\textit{Caseitem}} \_ \text{ produces } \_) \subseteq (\textit{StaticEnvironment} \times \textit{Caseitem} \times \textit{StaticEnvironment})$$

$$\boxed{\textsf{wf-CaseItem : caseitem-prod}}\;\frac{\textit{st-env} \vdash_{\textit{Unit}} \textit{un} \text{ produces } \textit{st-env}_1}{\textit{st-env} \vdash_{\textit{Caseitem}} [\![\textit{cns} : \textit{un}]\!] \text{ produces } \textit{st-env}_1}$$

5.8.1 UnitVoid

The semantics for unitvoid, the ELLA void type, is represented by a process which outputs the constant `unit` $\in$ *UnitType*:

Static semantics rules:

$$\boxed{\text{wf-Unit : UnitVoid}}\;\frac{\text{true}}{\mathit{st\text{-}env} \vdash_{\mathit{Unit}} [\![(\;)]\!] \text{ has type } [\![(\;)]\!]}$$

Dynamic semantics equations:

$$\mathit{M\text{-}Unit}[\![(\;)]\!](oc)(\rho,\sigma) = (\!\![\; \mathit{Constant}_{[\mathit{voidty},\, \texttt{unit}\,]}(oc) \;]\!\!)$$

where

$$\mathit{voidty} = \mathit{M\text{-}Type}[\![(\;)]\!](\rho,\sigma)$$

5.8.2 Enumerated Unit

An enumerated unit simply requires calls to the static and dynamic semantics for Enumerated:

Static semantics rules:

$$\boxed{\text{wf-Unit : Enumerated}}\;\frac{\mathit{st\text{-}env} \vdash_{\mathit{Enumerated}} [\![en]\!] \text{ has type } \mathit{ty}}{\mathit{st\text{-}env} \vdash_{\mathit{Unit}} [\![en]\!] \text{ has type } \mathit{ty}}$$

Dynamic semantics equations:

$$\mathit{M\text{-}Unit}[\![en]\!](oc)(\rho,\sigma) = (\mathit{term},\sigma)$$

where

$$\mathit{term} = \mathit{M\text{-}Enumerated}[\![en]\!](oc)(\rho,\sigma)$$

5.8.3 Assoc

Structure:

$$\boxed{Assoc \quad ::= \quad enum\ \&\ unit}$$

The association operation *en* & *unm* takes the output from unit *unm* and attaches a tag *en* to form a new associated value. The associated type of the output must have already been declared.

Static semantics rules:

$$\frac{\begin{array}{ll}\text{(a)} & tyn \in \mathsf{dom}\, st\text{-}env \wedge [\![en\ \&\ tagty]\!] \in newtype\text{-}of(st\text{-}env(tyn)) \\ \text{(b)} & st\text{-}env \vdash_{Unit}\ unm\ \mathsf{has\ type}\ tagty\end{array}}{st\text{-}env \vdash_{Unit} [\![en\ \&\ unm]\!]\ \mathsf{has\ type}\ tyn}\ \boxed{\text{wf-Unit: Assoc}}$$

Assoc calls the generic process $Assoc_{[ITY,OTY,tag]}$ which tags each input value $v \in ITY$ with a tag $tag \in Tag$:

Dynamic semantics equations:

$$M\text{-}Unit[\![en\ \&\ un]\!](oc)(\rho,\sigma) = (term, dy\text{-}env)$$

where

$$term = (\!\![\ \left(\ MU\ \mid\ Assoc_{[ITY,OTY,en]}(pipe\ /\ oc)\ \right) \setminus \{pipe\colon ITY\}\ ]\!\!)$$

$$(MU, dy\text{-}env) = M\text{-}Unit[\![un]\!](pipe)(\rho,\sigma)$$

$$ITY = type\text{-}of(un)(\rho,\sigma)$$

$$OTY = type\text{-}of[\![en\ \&\ un]\!](\rho,\sigma)$$

$$\{pipe\} = make\text{-}ChanVars(1, \mathsf{dom}\,\rho \cup \{oc\})$$

The generic process $Assoc_{[ity,oty,tag]}$ aassoicates a tag *tag* with its input value *v*, to form an association value $\mathsf{tag}(tag, v,)$:

$$(\!\![\ \mathsf{process}\ Assoc_{[ity,oty,tag]}(i\colon\ ity\ /\ o\colon\ oty) \triangleq$$

$$\square\, v\colon\ ity\ \bullet\ i(v)\ /\ o(\mathsf{tag}(tag, v)) :: Assoc_{[ity,oty,tag]}(i\ /\ o)\ ]\!\!)$$

where

$$\{i, o\} = make\text{-}ChanVars(2, \emptyset)$$

5.8.4 UnitStr

Structure:

$$\boxed{\mathit{Unitstr} \ ::= \quad \texttt{STRING}\ [\mathit{nat}]\mathit{unit}}$$

The unit string operation $\texttt{STRING}\ [n]\mathit{unm}$ creates a string containing n copies of the value produced by *unm*, which must be a character type:

Static semantics rules:

$$\frac{\begin{array}{ll}\text{(a)} & n \geq 1 \\ \text{(b)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \ \mathit{unm} \text{ has type } \mathit{cty} \\ \text{(c)} & \mathit{st\text{-}env} \vdash_{\mathit{is\text{-}char\text{-}type}} \mathit{cty}\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Unit}} [\![\ \texttt{STRING}\ [n]\mathit{unm}]\!] \text{ has type } [\![\ \texttt{STRING}\ \ [n]\ \mathit{cty}]\!]} \ \boxed{\text{wf-Unit: Unitstr}}$$

Unitstr is well-formed when:

- unit *unm* returns a character-type *cty*,
- a string type over *cty* has been declared.

Unit-str calls the generic process $\mathit{Unit\text{-}String}_{[\mathit{ITY},\mathit{OTY},n]}$; this produces an n-bit string of type *OTY*, where each element is its input character from type *ITY*:

Dynamic semantics equations:

$$\mathit{M\text{-}Unit}[\![\ \texttt{STRING}\ [n]\mathit{unm}]\!](\mathit{oc})(\rho,\sigma) = (\mathit{term}, \mathit{dy\text{-}env})$$

where

$$\mathit{term} = (\!\![\ \Big(\ \mathit{MU}\ \mid\ \mathit{Unit\text{-}String}_{[\mathit{ITY},\mathit{OTY},n]}(\mathit{pipe}\ /\ \mathit{oc})\ \Big) \setminus \{\mathit{pipe}\colon\ \mathit{ITY}\}\ ]\!\!)$$

$$(\mathit{MU}, \mathit{dy\text{-}env}) = \mathit{M\text{-}Unit}[\![\mathit{unm}]\!](\mathit{pipe})(\rho,\sigma)$$

$$\mathit{ITY} = \mathit{type\text{-}of}(\mathit{unm})(\rho,\sigma)$$

$$\mathit{OTY} = \mathit{type\text{-}of}[\![\ \texttt{STRING}\ \ [n]\ \mathit{unm}]\!](\rho,\sigma)$$

$$\{\mathit{pipe}\} = \mathit{make\text{-}ChanVars}(1, \mathrm{dom}\,\rho \cup \{\mathit{oc}\})$$

The generic process $\mathit{Unit\text{-}String}_{[\mathit{ity},\mathit{oty},n]}$ takes a tagged character v and produces a tagged string of length n; the value expressions $\mathrm{tag\text{-}of}(\mathit{tagexpr})$ and $\mathrm{value\text{-}of}(\mathit{tagexpr})$ represent the tag and value retrieved from a tagged expression *tagexpr*, and $\mathrm{copy}(\mathit{expr}, k)$ represents the tuple containing k copies of *expr*:

$$(\!\![\ \text{process } \mathit{Unit\text{-}String}_{[\mathit{ity},\mathit{oty},n]}(i\colon\ \mathit{ity}\ /\ o\colon\ \mathit{oty})\ \triangleq$$

$$\square\, v\colon\ \mathit{ity}\ \bullet\ i(v)\ /\ o(\mathrm{tag}(\mathrm{tag\text{-}of}(v), \mathrm{copy}(\mathrm{value\text{-}of}(v), n)))\ ::\ \mathit{Unit\text{-}String}_{[\mathit{ity},\mathit{oty},n]}(i\ /\ o)\ ]\!\!)$$

where

$$\{i, o\} = \mathit{make\text{-}ChanVars}(2, \emptyset)$$

5.8.5 Replicate

Structure:

$$\boxed{Replicate \quad ::= \qquad [\, nat \,]\ unit}$$

The replication operation $[\ k\ ]\ unm$ returns a tuple value where each component contains the value produced by *unm*. The type of the resulting unit is simply a power-type containing the type of *unm*. Note that a replicated value can contain a single element, when $k = 1$.

Static semantics rules:

$$\boxed{\text{wf-Unit: Replicate}}\ \frac{\begin{array}{ll}\text{(a)} & k \geq 1 \\ \text{(b)} & st\text{-}env \vdash_{Unit}\ unm \text{ has type } ty\end{array}}{st\text{-}env \vdash_{Unit} [\![[\ k\ ]\ unm]\!] \text{ has type } [\![[\ k\ ]\ ty]\!]}$$

The semantic operation simply calls the generic process $Replicate_{[ITY,OTY,k]}$ which replicates the values produced by *unm*:

Dynamic semantics equations:

$$M\text{-}Unit[\![[\ k\ ]\ unm]\!](oc)(\rho,\sigma) = (term, dy\text{-}env)$$

where

$$term = (\!\!|\ \Big(\ MU\ \mid\ Replicate_{[ITY,OTY,k]}(pipe\ /\ oc)\ \Big) \backslash \{pipe\colon ITY\}\ |\!\!)$$

$$ITY = type\text{-}of(unm)(\rho,\sigma)$$

$$(MU, dy\text{-}env) = M\text{-}Unit[\![unm]\!](pipe)(\rho,\sigma)$$

$$OTY = type\text{-}of[\![[\ k\ ]\ unm]\!](\rho,\sigma)$$

$$\{pipe\} = make\text{-}ChanVars(1, \mathsf{dom}\,\rho \cup \{oc\})$$

$$(\!\!|\ \textsf{process}\ Replicate_{[ity,oty,n]}(i\colon ity\ /\ o\colon oty)\ \underline{\triangle}$$

$$\square\, v\colon ity\ \bullet\ i(v)\ /\ o(\textsf{copy}(v,n)) :: Replicate_{[ity,oty,n]}(i\ /\ o)\ |\!\!)$$

where

$$\{i, o\} = make\text{-}ChanVars(2, \emptyset)$$

5.8.6 Query

Structure:

$$\boxed{Query \quad ::= \quad ?\ type}$$

The query construct $?\ ty$ captures the notion of an 'undefined' or 'undetermined' value. It non-deterministically returns the complete set of values in type ty.

Static semantics rules:

$$\boxed{\text{wf-Unit: Query}}\ \frac{st\text{-}env \vdash_{Type} ty}{st\text{-}env \vdash_{Unit} [\![\ ?\ ty]\!] \text{ has type } ty}$$

Query calls the generic process $Query_{[TY]}$ which has no inputs and non-deterministically outputs every member of type expression TY. There is no local change to the dynamic environment σ.

Dynamic semantics equations:

$$M\text{-}Unit[\![\ ?\ ty]\!](oc)(\rho,\sigma) = (term, \sigma)$$

where

$$term = (\![\ Query_{[TY]}(oc) \ ]\!)$$

$$TY = type\text{-}of[\![\ ?\ ty]\!](\rho,\sigma)$$

$$(\![\ \text{process } Query_{[ty]}(o\colon ty) \ \triangleq \square\, x\colon ty \ \bullet \quad /\ o(x) :: Query_{[ty]}(o) \ ]\!)$$

where

$$\{o\} = make\text{-}ChanVars(1, \emptyset)$$

5.8.7 Index

Structure:

$$\boxed{Index \quad ::= \quad unit\,[\,nat\,]}$$

The (static) indexing operation *unm* [*i*] accepts the output of *unm*, which must be either a sequence of type $(ty_1, \ldots, ty_k)$, or a string over characters of type *ty*. For the sequence, it returns a single value of type given by ty_i selected by *i*. For the string it returns a single (tagged) character of type *ty*, selected by *i*. The indexer must fall in the range of valid indices for the output of *unm*. There is a special case when *unm* is a power-type with a single element.

Static semantics rules:

$$\frac{\begin{array}{ll}\text{(a)} & \textit{st-env} \vdash_{Unit} \textit{unm} \text{ has type } ty \\ \text{(b)} & \textit{st-env} \vdash_{E\text{-}TyEq} ty \equiv [\![[1] ty_1]\!]\end{array}}{\textit{st-env} \vdash_{Unit} [\![\textit{unm}\,[\,1\,]]\!] \text{ has type } ty_1}\ \boxed{\text{wf-Unit: Index-row-elt}}$$

$$\frac{\begin{array}{ll}\text{(a)} & \textit{st-env} \vdash_{Unit} \textit{unm} \text{ has type } ty \\ \text{(b)} & \textit{st-env} \vdash_{E\text{-}TyEq} ty \equiv [\![(ty_1, \ldots, ty_k)]\!] \\ \text{(c)} & 1 \leq i \leq k\ \wedge k \geq 2\end{array}}{\textit{st-env} \vdash_{Unit} [\![\textit{unm}\,[\,i\,]]\!] \text{ has type } ty_i}\ \boxed{\text{wf-Unit: Index-row}}$$

$$\frac{\begin{array}{ll}\text{(a)} & \textit{st-env} \vdash_{Unit} \textit{unm} \text{ has type } ty \\ \text{(b)} & \textit{st-env} \vdash_{E\text{-}TyEq} ty \equiv [\![\ \mathtt{STRING}\,[k] cty]\!] \\ \text{(c)} & 1 \leq i \leq k\end{array}}{\textit{st-env} \vdash_{Unit} [\![\textit{unm}\,[\,i\,]]\!] \text{ has type } cty}\ \boxed{\text{wf-Unit: Index-str}}$$

The index operation is well-formed when:

- *unm* is either a sequence or a string,
- *i* falls in the range of valid indices.

The indexing operation is performed either by a process term $\textit{Index-row}_{[ITY,OTY]}$ or by $\textit{Index-str}_{[ITY,OTY]}$, depending upon whether *unm* is a tuple or string. Both processes extract the required element from the output of *unm*:

Dynamic semantics equations:

$$M\text{-}Unit[\![unm[\ i\ ]]\!](oc)(\rho,\sigma) = (term, dy\text{-}env)$$

where

$\textbf{case}\ is\text{-}tuple\text{-}type(ITY)(\sigma)\ \textbf{in}$

$\quad \{\mathtt{t}\} : term = (\!\!|\ \Big(\ MU\ \mid\ Index\text{-}row_{[ITY,OTY,i]}(pipe, oc)\ \Big)\backslash\{pipe\colon ITY\}\ |\!\!)$

$\textbf{else}\ term = (\!\!|\ \Big(\ MU\ \mid\ Index\text{-}str_{[ITY,OTY,i]}(pipe\ /\ oc)\ \Big)\backslash\{pipe\colon ITY\}\ |\!\!)$

$\textbf{end}$

$(MU, dy\text{-}env) = M\text{-}Unit[\![unm]\!](pipe)(\rho,\sigma)$

$ITY = type\text{-}of(unm)(\rho,\sigma)$

$OTY = type\text{-}of[\![unm[\ i\ ]]\!](\rho,\sigma)$

$\{pipe\} = make\text{-}ChanVars(1, \mathsf{dom}\,\rho \cup \{oc\})$

Two operations determine whether an *EPA* type expression represents a set containing tuples or one containing tagged sequence of characters. Note that the relation '$\_\ \vdash_{EPA\text{-}TyEq}\ \_\ \equiv\ \_$' determines equality of *EPA* type expressions, with respect to an environment containing a collection of *EPA* term definitions, i.e. the *range* of the dynamic environment σ.

$$is\text{-}tuple\text{-}type : TypeExpr \to DynamicEnvironment \to \mathbb{B}$$

$$is\text{-}tuple\text{-}type(ty)(\sigma) =$$

$$\quad \exists\, k \in \mathbb{N} \bullet$$

$$\qquad \exists\, ty_1,\ \cdots\ , ty_k \in TypeExpr \bullet$$

$$\qquad\quad \mathsf{rng}\,\sigma \vdash_{EPA\text{-}TyEq}\ ty\ \equiv\ (\!\!|\ ty_1 \times\ \cdots\ \times ty_k\ |\!\!)$$

$$is\text{-}string\text{-}type : TypeExpr \to DynamicEnvironment \to \mathbb{B}$$

$$is\text{-}string\text{-}type(ty)(\sigma) =$$

$$\quad \exists\, k \in \mathbb{N} \bullet$$

$$\qquad \exists\, tag \in TagName, ty_1,\ \cdots\ , ty_k \in CharRange \bullet$$

$$\qquad\quad \mathsf{rng}\,\sigma \vdash_{EPA\text{-}TyEq}\ ty\ \equiv\ (\!\!|\ \mathsf{tag}(tag, ty_1 \times\ \cdots\ \times ty_k)\ |\!\!)$$

A third operation determines whether a type expression corresponds to either of the above:

$$is\text{-}tuple\text{-}or\text{-}string\text{-}type : TypeExpr \to DynamicEnvironment \to \mathbb{B}$$

$$is\text{-}tuple\text{-}or\text{-}string\text{-}type(ty)(\sigma) =$$

$$\quad is\text{-}tuple\text{-}type(ty)(\sigma)\ \vee\ is\text{-}string\text{-}type(ty)(\sigma)$$

The generic process $Index\text{-}row_{[ity,oty,n]}$ takes an input sequence $\mathsf{tuple}(a_1,\ldots,a_k)$, and outputs the element a_n, and $Index\text{-}str_{[ity,oty,n]}$ takes a string $\mathsf{tag}(tag, \mathsf{tuple}(c_1,\ldots,c_k))$ and outputs a tagged character $\mathsf{tag}(tag, c_n)$:

$\llbracket$ process $Index\text{-}row_{[ity,oty,n]}(i\colon\ ity\ /\ o\colon\ oty)\ \underline{\triangle}$

$\square\, v\colon\ ity\ \cdot\ i(v)\ /\ o(\mathsf{index}(v,n)) :: Index\text{-}row_{[ity,oty,n]}(i\ /\ o)\ \rrbracket$

$\llbracket$ process $Index\text{-}str_{[ity,oty,n]}(i\colon\ ity\ /\ o\colon\ oty)\ \underline{\triangle}$

$\square\, s\colon\ ity\ \cdot\ i(s)\ /\ o(\mathsf{tag}(\mathsf{tag\text{-}of}(s), \mathsf{index}(\mathsf{value\text{-}of}(s),n))) :: Index\text{-}str_{[ity,oty,n]}(i\ /\ o)\ \rrbracket$

where

$\{i,o\} = make\text{-}ChanVars(2, \emptyset)$

The *EPA* value expression $\mathsf{value\text{-}of}(v)$ represents selection of the value part of the tagged value v. The *EPA* value expression $\mathsf{index}(v,i)$ represents selection of the i^{th} component of a tuple v.

5.8.8 Dyindex

Structure:

$$\boxed{\mathit{Dyindex} \quad ::= \quad \mathit{unit}_1 \; [[\; \mathit{unit}_2 \;]]}$$

The dynamic indexing operation unm_1 [[unm_2]] accepts the output of unm_1, which must be either a sequence of elements all having the same type *ty*, or a string over characters of type *ty*. It returns a single value of type given by *ty* selected by the output of unm_2, which must be an ELLA-integer falling in the range of valid indices for the output of unm_1.

Static semantics rules:

$$\frac{\begin{array}{ll} \text{(a)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{unm}_1 \text{ has type } \mathit{ty}_1 \\ \text{(b)} & \mathit{st\text{-}env} \vdash_{\mathit{E\text{-}TyEq}} \mathit{ty}_1 \equiv [\![[k]\mathit{ty}]\!] \\ \text{(c)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{unm}_2 \text{ has type } \mathit{nty} \\ \text{(d)} & \mathit{st\text{-}env} \vdash_{\mathit{E\text{-}TyEq}} \mathit{nty} \equiv [\![\mathit{tag}/(\mathit{lb}\underline{\;..\;}\mathit{ub})]\!] \\ \text{(e)} & 1 \le \mathit{lb} \wedge \mathit{lb} \le \mathit{ub} \wedge \mathit{ub} \le k \end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Unit}} [\![\mathit{unm}_1 \; [[\; \mathit{unm}_2 \;]]]\!] \text{ has type } \mathit{ty}} \; \boxed{\text{wf-Unit: Dyindex-row}}$$

$$\frac{\begin{array}{ll} \text{(a)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{unm}_1 \text{ has type } \mathit{ty}_1 \\ \text{(b)} & \mathit{st\text{-}env} \vdash_{\mathit{E\text{-}TyEq}} \mathit{ty}_1 \equiv [\![\; \mathtt{STRING} \; [k]\mathit{ty}]\!] \\ \text{(c)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{unm}_2 \text{ has type } \mathit{nty} \\ \text{(d)} & \mathit{st\text{-}env} \vdash_{\mathit{E\text{-}TyEq}} \mathit{nty} \equiv [\![\mathit{tag}/(\mathit{lb}\underline{\;..\;}\mathit{ub})]\!] \\ \text{(e)} & 1 \le \mathit{lb} \wedge \mathit{lb} \le \mathit{ub} \wedge \mathit{ub} \le k \end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Unit}} [\![\mathit{unm}_1 \; [[\; \mathit{unm}_2 \;]]]\!] \text{ has type } \mathit{ty}} \; \boxed{\text{wf-Unit: Dyindex-str}}$$

The dynamic index operation is well-formed when:

- unm_1 is either a sequence where each element has type given by *ty*, or unm_1 outputs a string,
- unm_2 returns an ELLA-integer which falls in the range of valid indices.

The dynamic index calls one of the processes $\mathit{Dyindex\text{-}row}_{[ITY,NTY,OTY]}$ or $\mathit{Dyindex\text{-}str}_{[ITY,NTY,OTY]}$, which extract the required element, depending upon whether unm_1 is a tuple or a string:

Dynamic semantics equations:

$$\textit{M-Unit}[\![\textit{unm}_1\ [[\ \textit{unm}_2\ ]]]\!](\textit{oc})(\rho,\sigma) = (\textit{term}, \textit{dy-env}_2)$$

where

case $\textit{is-tuple-type}(\textit{ITY})(\sigma)$ **in**

$\{\mathtt{t}\}$: $\textit{term} = \llparenthesis\ (\textit{MU}1\ \mid\ \textit{MU}2\ \mid$
$\qquad \textit{Dyindex-row}_{[\textit{ITY},\textit{NTY},\textit{OTY}]}(i_1, i_2\ /\ \textit{oc}))\ \backslash\ \{i_1\colon\ \textit{ITY}, i_2\colon\ \textit{NTY}\}\ \rrparenthesis$

else $\textit{term} = \llparenthesis\ (\textit{MU}1\ \mid\ \textit{MU}2\ \mid$
$\qquad \textit{Dyindex-str}_{[\textit{ITY},\textit{NTY},\textit{OTY}]}(i_1, i_2\ /\ \textit{oc}))\ \backslash\ \{i_1\colon\ \textit{ITY}, i_2\colon\ \textit{NTY}\}\ \rrparenthesis$

end

$(\textit{MU}1, \textit{dy-env}_1) = \textit{M-Unit}[\![\textit{unm}_1]\!](i_1)(\rho,\sigma)$

$(\textit{MU}2, \textit{dy-env}_2) = \textit{M-Unit}[\![\textit{unm}_2]\!](i_2)(\rho,\textit{dy-env}_1)$

$\textit{ITY} = \textit{type-of}(\textit{unm}_1)(\rho,\sigma)$

$\textit{OTY} = \textit{index-type}(\textit{ITY}, 1)(\sigma)$

$\textit{NTY} = \textit{type-of}(\textit{unm}_2)(\rho,\sigma)$

$\{i_1, i_2\} = \textit{make-ChanVars}(2, \mathsf{dom}\,\rho \cup \{\textit{oc}\})$

The generic processes $\textit{Dyindex-row}_{[\textit{ity},\textit{nty},\textit{oty}]}$ and $\textit{Dyindex-str}_{[\textit{ity},\textit{nty},\textit{oty}]}$ have input signatures $\textit{ity} \times \textit{nty}$ and output signatures $\textit{oty}$. The former takes as input a sequence $v = \mathsf{tuple}(a_1, \ldots, a_k) \in \textit{ity}$, and an indexer $x = \mathsf{tag}(\textit{tag}_1, i)$, and produces the element a_i. The latter takes a string $v = \mathsf{tag}(\textit{tag}_2, \mathsf{tuple}(c_1, \ldots, c_k))$ and the indexer x and produces the character $\mathsf{tag}(\textit{tag}_2, c_i)$:

$\llparenthesis$ process $\textit{Dyindex-row}_{[\textit{ity},\textit{nty},\textit{oty}]}(i_1\colon\ \textit{ity}, i_2\colon\ \textit{nty}\ /\ o\colon\ \textit{oty})\ \underline{\triangle}$
$\quad \square\, v\colon\ \textit{ity}, x\colon\ \textit{nty}\ \bullet\ i_1(v), i_2(x)\ /\ o(\mathsf{index}(v, \mathsf{value\text{-}of}(x)))$
$\qquad ::\ \textit{Dyindex-row}_{[\textit{ity},\textit{nty},\textit{oty}]}(i_1, i_2\ /\ o)\ \rrparenthesis$

$\llparenthesis$ process $\textit{Dyindex-str}_{[\textit{ity},\textit{nty},\textit{oty}]}(i_1\colon\ \textit{ity}, i_2\colon\ \textit{nty}\ /\ o\colon\ \textit{oty})\ \underline{\triangle}$
$\quad \square\, s\colon\ \textit{ity}, x\colon\ \textit{nty}\ \bullet\ i_1(s), i_2(x)\ /\ o(\mathsf{tag}(\mathsf{tag\text{-}of}(s), \mathsf{index}(\mathsf{value\text{-}of}(s), \mathsf{value\text{-}of}(x))))$
$\qquad ::\ \textit{Dyindex-str}_{[\textit{ity},\textit{nty},\textit{oty}]}(i_1, i_2\ /\ o)\ \rrparenthesis$

where

$\{i_1, i_2, o\} = \textit{make-ChanVars}(3, \emptyset)$

5.8.9 Replace

Structure:

$$\boxed{\mathit{Replace} \quad ::= \quad \texttt{REPLACE}\ (\mathit{unit}_1, \mathit{unit}_2, \mathit{unit}_3)}$$

The replacement operation $\texttt{REPLACE}\ (\mathit{unm}_1, \mathit{unm}_2, \mathit{unm}_3)$ accepts the output from unm_1 of type ty_1, which must be either a sequence of elements all having the same type ty_2, or a string over characters of type cty. It also takes a replacement unit unm_3, which has the same elemental output type ty_3, and an indexing value given by unm_2. It returns the modified output from unm_1 where the element indexed by unm_2 is replaced by unm_3. The output value of unm_2 must be an ELLA-integer which falls in the valid range of indices for elements in unm_1 output.

Static semantics rules:

$$\frac{\begin{array}{ll}
\text{(a)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{unm}_1 \text{ has type } \mathit{ty}_1 \\
\text{(b)} & \mathit{st\text{-}env} \vdash_{\mathit{E\text{-}TyEq}} \mathit{ty}_1 \equiv [\![[k]\mathit{ty}_2]\!] \\
\text{(c)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{unm}_2 \text{ has type } \mathit{nty} \\
\text{(d)} & \mathit{st\text{-}env} \vdash_{\mathit{E\text{-}TyEq}} \mathit{nty} \equiv [\![\mathit{tag}/(\mathit{lb}\underline{\ ..\ }\mathit{ub})]\!] \\
\text{(e)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{unm}_3 \text{ has type } \mathit{ty}_3 \\
\text{(f)} & \mathit{st\text{-}env} \vdash_{\mathit{E\text{-}TyEq}} \mathit{ty}_3 \equiv \mathit{ty}_2 \\
\text{(g)} & 1 \leq \mathit{lb} \wedge \mathit{lb} \leq \mathit{ub} \wedge \mathit{ub} \leq k
\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Unit}} [\![\texttt{REPLACE}\ (\mathit{unm}_1, \mathit{unm}_2, \mathit{unm}_3)]\!] \text{ has type } \mathit{ty}_1} \ \boxed{\text{wf-Unit: Replace-row}}$$

$$\frac{\begin{array}{ll}
\text{(a)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{unm}_1 \text{ has type } \mathit{ty}_1 \\
\text{(b)} & \mathit{st\text{-}env} \vdash_{\mathit{E\text{-}TyEq}} \mathit{ty}_1 \equiv [\![\texttt{STRING}\ [k]\mathit{ty}_3]\!] \\
\text{(b)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{unm}_2 \text{ has type } \mathit{nty} \\
\text{(c)} & \mathit{st\text{-}env} \vdash_{\mathit{E\text{-}TyEq}} \mathit{nty} \equiv [\![\mathit{tag}/(\mathit{lb}\underline{\ ..\ }\mathit{ub})]\!] \\
\text{(d)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{unm}_3 \text{ has type } \mathit{ty}_3 \\
\text{(e)} & 1 \leq \mathit{lb} \wedge \mathit{lb} \leq \mathit{ub} \wedge \mathit{ub} \leq k
\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Unit}} [\![\texttt{REPLACE}\ (\mathit{unm}_1, \mathit{unm}_2, \mathit{unm}_3)]\!] \text{ has type } \mathit{ty}_1} \ \boxed{\text{wf-Unit: Replace-str}}$$

The replacement operation is well-formed when:

- unm_1 is either a sequence or string, where each element is of type ty_3,
- unm_2 returns an ELLA-integer which falls in the range of valid indices,
- the output of unm_3 has the same type ty_3 as elements in unm_1.

Replace calls one of the generic processes $\mathit{Replace\text{-}row}_{[\mathit{STY},\mathit{NTY},\mathit{RTY}]}$, $\mathit{Replace\text{-}str}_{[\mathit{STY},\mathit{NTY},\mathit{RTY}]}$ which insert the value output by the new unit at the required index, in either a row or string:

Dynamic semantics equations:

$$M\text{-}Unit[\![\ \texttt{REPLACE}\ (unm_1, unm_2, unm_3)]\!](oc)(\rho,\sigma) = (term, dy\text{-}env_3)$$

where

case $is\text{-}tuple\text{-}type(STY)(\sigma)$ **in**

$\{\texttt{t}\} : term = (\!\!|\ (MU1 \mid MU2 \mid MU3 \mid Replace\text{-}row_{[STY,NTY,RTY]}(i_1, i_2, i_3, oc)) \setminus \{i_1\colon STY, i_2\colon NTY, i_3\colon RTY\}\ |\!\!)$

else $term = (\!\!|\ (MU1 \mid MU2 \mid MU3 \mid Replace\text{-}str_{[STY,NTY,RTY]}(i_1, i_2, i_3, oc)) \setminus \{i_1\colon STY, i_2\colon NTY, i_3\colon RTY\}\ |\!\!)$

end

$(MU1, dy\text{-}env_1) = M\text{-}Unit[\![unm_1]\!](i_1)(\rho,\sigma)$

$(MU2, dy\text{-}env_2) = M\text{-}Unit[\![unm_2]\!](i_2)(\rho, dy\text{-}env_1)$

$(MU3, dy\text{-}env_3) = M\text{-}Unit[\![unm_3]\!](i_3)(\rho, dy\text{-}env_2)$

$STY = type\text{-}of(unm_1)(\rho,\sigma)$

$NTY = type\text{-}of(unm_2)(\rho,\sigma)$

$RTY = type\text{-}of(unm_3)(\rho,\sigma)$

$\{i_1, i_2, i_3\} = make\text{-}ChanVars(3, \mathsf{dom}\,\rho \cup \{oc\})$

The generic process $Replace\text{-}row_{[sty,nty,rty]}$ takes as input a sequence $v = \mathsf{tuple}(a_1, \ldots, a_k)$, $v \in sty$, an indexer $x = \mathsf{tag}(tag_1, i)$, and a replacement value $w \in rty$. It returns a sequence $w = \mathsf{tuple}(a_1, \ldots, a_{i\text{-}1}, w, a_{i+1}, \ldots, a_k)$. $Replace\text{-}str_{[sty,nty,rty]}$ takes a string $v = \mathsf{tag}(tag_2, \mathsf{tuple}(c_1, \ldots, c_k))$ the indexer x and replacement value $\mathsf{tag}(tag_2, w)$ and returns the string $\mathsf{tag}(tag_2, \mathsf{tuple}(c_1, \ldots, c_{i\text{-}1}, w, c_{i+1}, \ldots, c_k))$:

$(\!\!|$ process $Replace\text{-}row_{[sty,nty,rty]}(i_1\colon sty, i_2\colon nty, i_3\colon rty\ /\ o\colon sty)\ \underline{\triangle}$

$\square\, v\colon sty, x\colon nty, a\colon rty\ \bullet\ i_1(v), i_2(x), i_3(a)\ /\ o(\mathsf{repl}(v, \mathsf{value\text{-}of}(x), a))$

$::\ Replace\text{-}row_{[sty,nty,rty]}(i_1, i_2, i_3\ /\ o)\ |\!\!)$

$(\!\!|$ process $Replace\text{-}str_{[sty,nty,rty]}(i_1\colon sty, i_2\colon nty, i_3\colon rty\ /\ o\colon sty)\ \underline{\triangle}$

$\square\, v\colon sty, x\colon nty, a\colon rty\ \bullet\ i_1(v), i_2(x), i_3(a)\ /$

$o(\mathsf{tag}(\mathsf{tag\text{-}of}(v), \mathsf{repl}(v, \mathsf{value\text{-}of}(x), \mathsf{value\text{-}of}(a))))$

$::\ Replace\text{-}str_{[sty,nty,rty]}(i_1, i_2, i_3\ /\ o)\ |\!\!)$

where

$\{i_1, i_2, i_3, o\} = make\text{-}ChanVars(4, \emptyset)$

The *EPA* value expression $\mathsf{repl}(v, n, a)$ represents the operation which replaces the n^{th} element of vector v with the value a.

5.8.10 Trim

Structure:

$$\boxed{\mathit{Trim} \quad ::= \quad \mathit{unit}\ [\ \mathit{nat}_1 \mathbin{\underline{..}} \mathit{nat}_2\ ]}$$

The trimming operation $unm[i \mathbin{\underline{..}} j\,]$ accepts the output of unm, which must be either a sequence $\mathsf{tuple}(v_1, \ldots, v_k)$ of type $(ty_1 \times \ldots \times ty_k)$, or a string $\mathsf{tag}(tag, \mathsf{tuple}(v_1, \ldots, v_k))$ over characters of type cty. For a sequence, it returns the subsequence of type $(ty_i \times \ldots \times ty_j)$. For the string, it returns the substring $\mathsf{tag}(tag, \mathsf{tuple}(v_i, \ldots, v_j))$. Clearly, $i \le j$, and the two indices must fall in the range of valid indices from unm. There are two special cases, treated in rules *Trim-row-elt* and *Trim-elt-elt*. The first occurs when $i = j$ and a sequence with one element results; this has type equivalent to $[1]typ$, and not (typ) where the brackets are ignored. In the second case, unm is a power-type with a single element, and $i = j = 1$.

Static semantics rules:

(a) $st\text{-}env \vdash_{Unit} unm$ has type ty

(b) $st\text{-}env \vdash_{E\text{-}TyEq} ty \equiv [\![(ty_1, \ldots, ty_k)]\!]$

(c) $i \in \{1, \ldots, k\}$

wf-Unit: Trim-row-elt

$st\text{-}env \vdash_{Unit} [\![unm[i \mathbin{\underline{..}} i\,]]\!]$ has type $[\![[1]ty_i]\!]$

(a) $st\text{-}env \vdash_{Unit} unm$ has type ty

(b) $st\text{-}env \vdash_{E\text{-}TyEq} ty \equiv [\![[1]ty_1]\!]$

wf-Unit: Trim-elt-elt

$st\text{-}env \vdash_{Unit} [\![unm[1 \mathbin{\underline{..}} 1\,]]\!]$ has type $[\![[1]ty_1]\!]$

(a) $st\text{-}env \vdash_{Unit} unm$ has type ty

(b) $st\text{-}env \vdash_{E\text{-}TyEq} ty \equiv [\![(ty_1, \ldots, ty_k)]\!]$

(c) $j \in \{1, \ldots, k\}$

(d) $i \in \{1, \ldots, j\}$

wf-Unit: Trim-row

$st\text{-}env \vdash_{Unit} [\![unm[i \mathbin{\underline{..}} j\,]]\!]$ has type $[\![(ty_i, \ldots, ty_j)]\!]$

(a) $st\text{-}env \vdash_{Unit} unm$ has type ty

(b) $m = j - i + 1$

(c) $st\text{-}env \vdash_{E\text{-}TyEq} ty \equiv [\![\ \mathtt{STRING}\ \ [k]cty]\!]$

(d) $j \in \{1, \ldots, k\}$

(e) $i \in \{1, \ldots, j\}$

wf-Unit: Trim-str

$st\text{-}env \vdash_{Unit} [\![unm[i \mathbin{\underline{..}} j\,]]\!]$ has type $[\![\ \mathtt{STRING}\ \ [m]cty]\!]$

The trimming operation is well-formed when:

- Unit *unm* is either a sequence or a string,

- the two indices can define a valid subsequence/substring from the output of the unit *unm*: they are in the index range of elements, and $i \leq j$.

Trim calls either the generic process $Select\text{-}row_{[ITY,i,j]}$ or the process $Select\text{-}str_{[ITY,i,j]}$ depending upon whether *unm* is a row or a string. The process then extracts the required elements or substring from its input:

Dynamic semantics equations:

$$M\text{-}Unit[\![unm[i\underline{\,..\,}j\,]]\!](oc)(\rho,\sigma) = (term, dy\text{-}env)$$

where

case $is\text{-}tuple\text{-}type(ITY)(\sigma)$ **in**

$\{\mathtt{t}\} : term = \llparenthesis \left(\; MU \mid Select\text{-}row_{[ITY,OTY,i,j]}(pipe \,/\, oc) \; \right) \backslash \{pipe\text{: } ITY\} \; \rrparenthesis$

else $term = \llparenthesis \left(\; MU \mid Select\text{-}str_{[ITY,OTY,i,j]}(pipe \,/\, oc) \; \right) \backslash \{pipe\text{: } ITY\} \; \rrparenthesis$

end

$(MU, dy\text{-}env) = M\text{-}Unit[\![unm]\!](pipe)(\rho,\sigma)$

$ITY = type\text{-}of(unm)(\rho,\sigma)$

$OTY = type\text{-}of[\![unm[i\underline{\,..\,}j\,]]\!](\rho,\sigma))$

$\{pipe\} = make\text{-}ChanVars(1, \mathsf{dom}\,\rho \cup \{oc\})$

The generic process $Select\text{-}row_{[ity,oty,m,n]}$ returns a subsequence of values $\mathsf{tuple}(v_m,\ldots,v_n) \in oty$ from an input $\mathsf{tuple}(v_1,\ldots,v_k) \in ity$, and $Select\text{-}str_{[ity,oty,m,n]}$ returns a substring $\mathsf{tag}(tag, \mathsf{tuple}(chr_m,\ldots,chr_n))$ from the input $\mathsf{tag}(tag, \mathsf{tuple}(chr_1,\ldots,chr_k)) \in ity$.

$\llparenthesis$ process $Select\text{-}row_{[ity,oty,m,n]}(i\text{: } ity \,/\, o\text{: } oty) \;\underline{\triangle}$

$\quad \square\, v\text{: } ity \;\cdot\; i(v) \,/\, o(\mathsf{trim}(v,m,n)) :: Select\text{-}row_{[ity,oty,m,n]}(i \,/\, o) \rrparenthesis$

$\llparenthesis$ process $Select\text{-}str_{[ity,oty,m,n]}(i\text{: } ity \,/\, o\text{: } oty) \;\underline{\triangle}$

$\quad \square\, v\text{: } ity \;\cdot\; i(v) \,/\, o(\mathsf{tag}(\mathsf{tag\text{-}of}(v), \mathsf{trim}(\mathsf{value\text{-}of}(v),m,n))) :: Select\text{-}str_{[ity,oty,m,n]}(i \,/\, o) \rrparenthesis$

where

$\{i,o\} = make\text{-}ChanVars(2,\emptyset)$

The *EPA* value expression $\mathsf{trim}(v,m,n)$ represents the operation which selects the subsequence $v(m, \cdots, n)$ from sequence v.

5.8.11 Conc

Structure:

$$\boxed{Conc \quad ::= \quad unit_1 \ \texttt{CONC} \ unit_2}$$

The concatenation operation unm_1 CONC unm_2 takes two units and outputs tuples or strings, which are a concatenation of the outputs from unm_1 and unm_2. Tuples and strings are handled separately — it is not possible to combine the two.

The concatenation operation is well-formed when, after any coercion, the two units unm_1 and unm_2 have elements from the same basic type, and at least one is a tuple or string:

Static semantics rules:

(a) $\mathit{st\text{-}env} \vdash_{Unit} \ unm_1 \text{ has type } ty_1 \ \wedge \ \mathit{st\text{-}env} \vdash_{E\text{-}TyEq} ty_1 \equiv [\![[i] typ]\!]$

(b) $\mathit{st\text{-}env} \vdash_{Unit} \ unm_2 \text{ has type } ty_2 \ \wedge \ \mathit{st\text{-}env} \vdash_{E\text{-}TyEq} ty_2 \equiv [\![[j] typ]\!]$

(c) $k = i + j$

wf-Unit: Conc-row1

$\mathit{st\text{-}env} \vdash_{Unit} [\![unm_1 \ \texttt{CONC} \ unm_2]\!] \text{ has type } [\![[k] typ]\!]$

(a) $\mathit{st\text{-}env} \vdash_{Unit} \ unm_1 \text{ has type } ty_1$

(b) $\mathit{st\text{-}env} \vdash_{Unit} \ unm_2 \text{ has type } ty_2 \ \wedge \ \mathit{st\text{-}env} \vdash_{E\text{-}TyEq} ty_2 \equiv [\![[i] ty_1]\!]$

(c) $k = i + 1$

wf-Unit: Conc-row2

$\mathit{st\text{-}env} \vdash_{Unit} [\![unm_1 \ \texttt{CONC} \ unm_2]\!] \text{ has type } [\![[k] ty_1]\!]$

(a) $\mathit{st\text{-}env} \vdash_{Unit} \ unm_1 \text{ has type } ty_1 \ \wedge \ \mathit{st\text{-}env} \vdash_{E\text{-}TyEq} ty_1 \equiv [\![[i] ty_2]\!]$

(b) $\mathit{st\text{-}env} \vdash_{Unit} \ unm_2 \text{ has type } ty_2$

(c) $k = i + 1$

wf-Unit: Conc-row3

$\mathit{st\text{-}env} \vdash_{Unit} [\![unm_1 \ \texttt{CONC} \ unm_2]\!] \text{ has type } [\![[k] ty_2]\!]$

(a) $\mathit{st\text{-}env} \vdash_{Unit} \ unm_1 \text{ has type } ty_1$

(b) $\mathit{st\text{-}env} \vdash_{E\text{-}TyEq} ty_1 \equiv [\![\ \texttt{STRING} \ [i] typ]\!]$

(c) $\mathit{st\text{-}env} \vdash_{Unit} \ unm_2 \text{ has type } ty_2$

(d) $\mathit{st\text{-}env} \vdash_{E\text{-}TyEq} ty_2 \equiv [\![\ \texttt{STRING} \ [j] typ]\!]$

(c) $k = i + j$

wf-Unit: Conc-str1

$\mathit{st\text{-}env} \vdash_{Unit} [\![unm_1 \ \texttt{CONC} \ unm_2]\!] \text{ has type } [\![\ \texttt{STRING} \ [k] typ]\!]$

(a) $\textit{st-env} \vdash_{Unit}\ unm_1$ has type ty_1

(b) $\textit{st-env} \vdash_{Unit}\ unm_2$ has type ty_2

(c) $\textit{st-env} \vdash_{E\text{-}TyEq} ty_2 \equiv [\![\ \texttt{STRING}\ [i] ty_1]\!]$

(d) $k = i + 1$

wf-Unit: Conc-str2

$\textit{st-env} \vdash_{Unit} [\![unm_1\ \texttt{CONC}\ unm_2]\!]$ has type $[\![\ \texttt{STRING}\ [k] ty_1]\!]$

(a) $\textit{st-env} \vdash_{Unit}\ unm_1$ has type ty_1

(b) $\textit{st-env} \vdash_{Unit}\ unm_2$ has type ty_2

(c) $\textit{st-env} \vdash_{E\text{-}TyEq} ty_1 \equiv [\![\ \texttt{STRING}\ [i] ty_2]\!]$

(d) $k = i + 1$

wf-Unit: Conc-str3

$\textit{st-env} \vdash_{Unit} [\![unm_1\ \texttt{CONC}\ unm_2]\!]$ has type $[\![\ \texttt{STRING}\ [k] ty_2]\!]$

Conc calls generic process $Conc_{[TY1,TY2,\sigma]}$ which performs the required concatenation:

Dynamic semantics equations:

$$\textit{M-Unit}[\![unm_1\ \texttt{CONC}\ unm_2]\!](oc)(\rho,\sigma) = (term, \textit{dy-env}_2)$$

where

$$term = (\!\!|\ \Big(\ MU1\ \mid\ MU2\ \mid\ Conc_{[TY1,TY2,OTY]}(i_1,i_2,oc)\ \Big) \backslash \{i_1,i_2\}\ |\!\!)$$

$$(MU1, \textit{dy-env}_1) = \textit{M-Unit}[\![unm_1]\!](i_1)(\rho,\sigma)$$

$$(MU2, \textit{dy-env}_2) = \textit{M-Unit}[\![unm_2]\!](i_2)(\rho,\textit{dy-env}_1)$$

$$TY1 = \textit{type-of}(unm_1)(\rho,\sigma)$$

$$TY2 = \textit{type-of}(unm_2)(\rho,\sigma)$$

$$OTY = \textit{type-of}[\![unm_1\ \texttt{CONC}\ unm_2]\!](\rho,\sigma)$$

$$\{i_1,i_2\} = \textit{make-ChanVars}(2, \mathsf{dom}\,\rho \cup \{oc\})$$

For rows, the generic process $Conc_{[TY1,TY2,OTY]}$ takes an input (v,w), with $v = \mathsf{tuple}(a_1,\ldots,a_m) \in TY1$, and $w = \mathsf{tuple}(b_1,\ldots,b_n) \in TY2$, and outputs the value $\mathsf{tuple}(a_1,\ldots,a_m,b_1,\ldots,b_n) \in OTY$. For strings, $Conc_{[TY1,TY2,OTY]}$ takes an input (v,w), with string $v = \mathsf{tag}(tag,\mathsf{tuple}(a_1,\ldots,a_m)) \in TY1$, and string $w = \mathsf{tag}(tag,\mathsf{tuple}(b_1,\ldots,b_n)) \in TY2$, and outputs the string $\mathsf{tag}(tag,\mathsf{tuple}(a_1,\ \cdots\ ,a_m,b_1,\ldots,b_n)) \in OTY$. Note that if either input type is an element/character-type of the row/string-type on the other input, then this is first coerced to a unit-row/string-type:

$$(\!\!|\ \text{process}\ Conc_{[ty1,ty2,oty]}(i_1\!: ty1, i_2\!: ty2\ /\ o\!: oty) \triangleq$$
$$\Box\, v\!: ty1, w\!: ty2\ \bullet\ i_1(v), i_2(w)\ /\ o(\mathsf{concat}_{[ty1,ty2]}(v,w)) :: Conc_{[ty1,ty2,oty]}(i_1,i_2\ /\ o)\ |\!\!)$$

where

$$\{i_1,i_2,o\} = \textit{make-ChanVars}(3,\emptyset)$$

The *EPA* value expression $\mathsf{concat}_{[ty1,ty2]}(a,b)$ concatenates a and b, which should both be sequences. If either a or b is an element of the other, then it is first coerced to form a unit sequence before concatenation.

5.8.12 Sequence

Structure:

$$\boxed{Sequence \quad ::= \quad (unit_1 \ , \ \ldots \ , \ unit_{m+1})}$$

The sequencing operation $(unm_1 \ , \ \ldots \ , \ unm_k)$ returns a tuple value containing component values produced by $unm_1, \ \ldots \ , \ unm_k$. The type of the resulting unit will be the product of the component unit types. Note that for a sequence containing a single element, the semantics of the *un-tupled* element is returned — the sequencing brackets are discarded.

Static semantics rules:

$$\boxed{\text{wf-Unit: Sequence(a)}} \quad \frac{st\text{-}env \vdash_{Unit} \ unm_1 \text{ has type } ty_1}{st\text{-}env \vdash_{Unit} [\![(unm_1)]\!] \text{ has type } ty_1}$$

$$\boxed{\text{wf-Unit: Sequence(b)}} \quad \frac{\begin{array}{ll}\text{(a)} & k \geq 2 \\ \text{(b)} & \forall i \in \{1,\ldots,k\} \cdot st\text{-}env \vdash_{Unit} \ unm_i \text{ has type } ty_i \\ \text{(c)} & st\text{-}env \vdash_{E\text{-}TyEq} ty \equiv [\![(ty_1,\ldots,ty_k)]\!]\end{array}}{st\text{-}env \vdash_{Unit} [\![(unm_1 \ , \ \ldots \ , \ unm_k)]\!] \text{ has type } ty}$$

Dynamic semantics equations:

$$M\text{-}Unit[\![(unm_1)]\!](oc)(\rho,\sigma) = M\text{-}Unit[\![unm_1]\!](oc)(\rho,\sigma)$$

$$M\text{-}Unit[\![(unm_1, \ \cdots \ ,unm_k)]\!](oc)(\rho,\sigma) = (term, dy\text{-}env_{k+2})$$

where

$$\begin{array}{l}
term = (\!\!| \ (MU_1 \ | \ \cdots \ | \ MU_k \ | \\
\qquad\qquad Sequence_{[oty_1, \ \cdots \ ,oty_k,OTY]}(ic_1, \ \cdots \ ,ic_k \ / \ oc)) \setminus \{ic_1\!: \ oty_1,\ldots,ic_k\!: \ oty_k\} \ |\!\!) \\
dy\text{-}env_1 = \sigma \\
\forall \ i \in \{1,\ldots,k\} \bullet (MU_i, dy\text{-}env_{i+1}) = M\text{-}Unit[\![unm_i]\!](ic_i)(\rho, dy\text{-}env_i) \\
\forall \ i \in \{1,\ldots,k\} \bullet oty_i = type\text{-}of(unm_i)(\rho,\sigma) \\
OTY = type\text{-}of[\![(unm_1, \ \cdots \ ,unm_k)]\!](\rho,\sigma) \\
k \geq 2 \\
\{ic_1,\ldots,ic_k\} = make\text{-}ChanVars(k, \mathrm{dom}\,\rho \cup \{oc\})
\end{array}$$

The generic process $Sequence_{[inty\text{-}seq,oty]}$ performs the required sequencing operation; it takes k inputs and produces a single output — a sequence containing the k inputs:

$$
\begin{array}{l}
[\![\ \textsf{process}\ Sequence_{[inty\text{-}seq,oty]}(i_1\colon ty_1,\ \cdots\ ,i_k\colon ty_k\ /\ o\colon oty) \ \underline{\triangle} \\
\quad \Box\, a_1\colon ty_1,\ \cdots\ ,a_k\colon ty_k\ \bullet \\
\qquad i_1(a_1),\ \cdots\ ,i_k(a_k)\ /\ o(\textsf{tuple}(a_1,\ \cdots\ ,a_k)) :: Sequence_{[inty\text{-}seq,oty]}(i\ /\ o)\ ]\!] \\
\ \textsf{where} \\
\quad \textsf{len}\, inty\text{-}seq = k\ \wedge\ inty\text{-}seq = [ty_1,\ \cdots\ ,ty_k] \\
\quad \{i_1,\ \cdots\ ,i_k,o\} = make\text{-}ChanVars(k+2,\emptyset)
\end{array}
$$

5.8.13 Extract

Structure:

$$\boxed{\mathit{Extract} \quad ::= \quad \mathit{unit} \; // \; \mathit{enum}}$$

The extraction operation $unm\ //\ tag_1$ takes a unit unm which has an associated type output value $v = label\text{-}value(tag_2, val)$; if the tag tag_2 in v is the same as tag_1 then the tagged value val is output, otherwise the output is non-deterministic.

Static semantics rules:

$$\frac{\begin{array}{ll}\text{(a)} & tyn \in \mathrm{dom}\, st\text{-}env \;\wedge\; [\![tag_1 \;\&\; tagty]\!] \in newtype\text{-}of(st\text{-}env(tyn)) \\ \text{(b)} & st\text{-}env \vdash_{Unit}\; unm \text{ has type } tyn\end{array}}{st\text{-}env \vdash_{Unit} [\![unm\ //\ tag_1]\!] \text{ has type } tagty}\ \boxed{\text{wf-Unit: Extract}}$$

Extract calls the generic process $Extract_{[ITY,OTY,tag_1]}$ which extracts, if possible, the tagged value from its input $v \in ITY$. When the tag in v does not match tag_1 then the output is non-deterministic:

Dynamic semantics equations:

$$M\text{-}Unit[\![unm\ //\ tag_1]\!](oc)(\rho,\sigma) = (term, dy\text{-}env)$$

where

$$term = \langle\!\!\!\langle \left(MU \mid Extract_{[ITY,OTY,tag_1]}(pipe\ /\ oc) \right) \backslash \{pipe\colon ITY\} \rangle\!\!\!\rangle$$
$$(MU, dy\text{-}env) = M\text{-}Unit[\![unm]\!](pipe)(\rho,\sigma)$$
$$ITY = type\text{-}of(unm)(\rho,\sigma)$$
$$OTY = type\text{-}of[\![unm\ //\ tag_1]\!](\rho,\sigma)$$
$$\{pipe\} = make\text{-}ChanVars(1, \mathrm{dom}\,\rho \cup \{oc\})$$

The generic process $Extract_{[ity,oty,tag_1]}$ extracts the value from an association value; it non-deterministically returns any value form oty, i.e., the ‘query’ value, if the association value is not associated with the tag tag_1:

$$\langle\!\!\!\langle \text{ process } Extract_{[ity,oty,tag_1]}(i\colon ity\ /\ o\colon oty) \ \underline{\triangle}$$
$$\square x\colon ity, y\colon oty \;\bullet$$
$$i(x)\ /\ o(\text{case tag-of}(x) : \{tag_1\} \;\rightarrow\; \text{value-of}(x) \text{ else } y \text{ end}) :: Extract_{[ity,oty,tag_1]}(i\ /\ o) \rangle\!\!\!\rangle$$

where

$$\{i, o\} = make\text{-}ChanVars(2, \emptyset)$$

5.8.14 Instance

Structure:

$$\boxed{\mathit{Instance} \quad ::= \quad \mathit{fnname}\ \mathit{unit}}$$

Instantiation simply creates a process call *proc-name*(*fnname*), with input and output channels *pipe*, *oc*, to the process representing *fnname*, already stored in the dynamic environment σ. Note that when the function was declared, then mutiple inputs were bundled together, to form a single input. The resulting term, whose output is named *pipe*, is then joined to the object representing the sequence of arguments to the function call; the connecting channel is then hidden.

Static semantics rules:

$$\begin{array}{ll}
 & \text{(a)} \quad \mathit{fnname} \in \mathsf{dom}\, \mathit{st\text{-}env} \\
 & \text{(b)} \quad \mathit{st\text{-}env}(\mathit{fnname}) = [\![\ \texttt{funtype}\ \mathit{ty}_1 \rightarrow \mathit{ty}_2]\!] \\
 & \text{(c)} \quad \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{unm}\ \mathsf{has\ type}\ \mathit{ty}_3 \\
 & \text{(d)} \quad \mathit{st\text{-}env} \vdash_{\mathit{E\text{-}TyEq}}\ \mathit{ty}_1\ \equiv\ \mathit{ty}_3 \\
\boxed{\mathsf{wf\text{-}Unit : Instance}} & \overline{\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
 & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} [\![\mathit{fnname}\ \mathit{unm}]\!]\ \mathsf{has\ type}\ \ \mathit{ty}_2
\end{array}$$

The instance is well-formed if the function *fnname* has already been declared, and if the output type of the argument unit *unm* is the same as the input type for *fnname*, stored in a *FunType* object in the static environment.

Dynamic semantics equations:

$$\mathit{M\text{-}Unit}[\![\mathit{fnname}\ \mathit{unm}]\!](\mathit{oc})(\rho,\sigma) = (\mathit{term}, \mathit{dy\text{-}env})$$

where

$$\begin{aligned}
&\mathit{term} = (\!|\ (\mathit{MU}\ \mid\ \mathit{FN}) \backslash \{\mathit{pipe}\colon\ \mathit{STY}\}\ |\!) \\
&\mathit{FN} = (\!|\ \mathit{pnm}(\mathit{pipe}\ /\ \mathit{oc})\ |\!) \\
&\mathit{pnm} = \mathit{proc\text{-}name}(\mathit{fnname}) \\
&(\mathit{MU}, \mathit{dy\text{-}env}) = \mathit{M\text{-}Unit}[\![\mathit{unm}]\!](\mathit{pipe})(\rho,\sigma) \\
&\mathit{STY} = \mathit{type\text{-}of}(\mathit{unm})(\rho,\sigma) \\
&\mathit{FTY} = \mathit{type\text{-}of}[\![\mathit{fnname}\ \mathit{unm}]\!](\rho,\sigma) \\
&\{\mathit{pipe}\} = \mathit{make\text{-}ChanVars}(1, \mathsf{dom}\,\rho \cup \{\mathit{oc}\})
\end{aligned}$$

5.8.15 Signals

Structure:

$$\boxed{Signal \quad ::= \quad name}$$

A signal *sig* simply enables different nodes in a circuit to be identified and connected together. It must have already been declared (e.g. in the formal arguments to a function declaration, or in a `MAKE` or `LET` statement), and its type *ty* is stored as a *VarType* object in the static environment.

Static semantics rules:

$$\boxed{\text{wf-Unit : Signal}}\ \frac{\begin{array}{ll}\text{(a)} & sig \in \mathrm{dom}\, st\text{-}env \\ \text{(a)} & ty = vartype\text{-}of(st\text{-}env(sig))\end{array}}{st\text{-}env \vdash_{Unit} sig \text{ has type } ty}$$

The dynamic semantics for *sig* simply calls the generic process definition $Chan\text{-}Proc_{[TY]}$, with the input channel *sig* and the output channel *oc*; any value appearing on channel *sig* will be copied to its output. In this way *sig* enables the 'connection' between the ports on different design modules. Clearly there is no change to the dynamic environment, which is copied through:

Dynamic semantics equations:

$$M\text{-}Unit(sig)(oc)(\rho,\sigma) = (term,\sigma)$$

where

$$term = ⟦\ Chan\text{-}Proc_{[TY]}(sig\ /\ oc)\ ⟧$$

$$TY = type\text{-}of(sig)(\rho,\sigma)$$

The generic process $Chan\text{-}Proc_{[ty]}$ simply copies the value of type *ty* on its input to its output:

$$⟦\ \text{process } Chan\text{-}Proc_{[ty]}(i\colon ty\ /\ o\colon ty) \triangleq \square x\colon ty \cdot i(x)\ /\ o(x) :: Chan\text{-}Proc_{[ty]}(i\ /\ o)\ ⟧$$

where

$$\{i,o\} = make\text{-}ChanVars(2,\emptyset)$$

5.8.16 Block structures

Structure:

$$\boxed{\mathit{Block} \quad ::= \qquad \texttt{BEGIN}\ \mathit{step}_1\ \ldots\ \mathit{step}_{k+1}\ \texttt{OUTPUT}\ \mathit{unit}\ \texttt{END}}$$

The block construct is the mechanism for (explicitly) instantiating previously defined functions and connecting them together in the required configuration. Within the block, local type- and function-declarations can be given, together with local signal bindings using `LET`. However, signals cannot be used before they have been bound. Note that the ELLA-text has already been filtered by the operation *declash* to remove any name clashes between local and external declarations — *declash* uses a set of scope rules associated with blocks to determine whether the local or external declaration of a variable, type or function should be used. However, it is applied to the ELLA-text *before* the static check has been performed; it therefore leaves any badly formed expressions, so that they can be rejected by the static semantics.

Static semantics rules:

$$\frac{\begin{array}{ll}\text{(a)} & \mathit{st\text{-}env} \vdash_{\mathit{Step}^*} [\![\mathit{st}_1\ \ldots\ \mathit{st}_{k+1}]\!]\ \textsf{produces}\ \mathit{st\text{-}env}_1 \\ \text{(b)} & \mathit{st\text{-}env}_1 \vdash_{\mathit{Unit}} \mathit{un}\ \ \textsf{produces}\ \ \mathit{st\text{-}env}_2\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Unit}} [\![\ \texttt{BEGIN}\ \mathit{st}_1\ \ldots\ \mathit{st}_{k+1}\ \texttt{OUTPUT}\ \mathit{un}\ \texttt{END}\ ]\!]\ \ \textsf{produces}\ \ \mathit{st\text{-}env}_2}\ \boxed{\textsf{wf-Unit : Block (a)}}$$

$$\frac{\begin{array}{ll}\text{(a)} & \vdash_{\mathit{redefinition}} [\![\mathit{st}_1\ \ldots\ \mathit{st}_{k+1}]\!] \\ \text{(b)} & \mathit{st\text{-}env} \vdash_{\mathit{Step}^*} [\![\mathit{st}_1\ \ldots\ \mathit{st}_{k+1}]\!]\ \textsf{produces}\ \mathit{st\text{-}env}_1 \\ \text{(c)} & \mathit{st\text{-}env}_1 \vdash_{\mathit{Unit}} \mathit{un}\ \ \textsf{has type}\ \ \mathit{ty} \\ \text{(d)} & \mathit{st\text{-}env}_2 \vdash_{\mathit{joined}} [\![\mathit{st}_1\ \ldots\ \mathit{st}_{k+1}]\!]\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Unit}} [\![\ \texttt{BEGIN}\ \mathit{st}_1\ \ldots\ \mathit{st}_{k+1}\ \texttt{OUTPUT}\ \mathit{un}\ \texttt{END}\ ]\!]\ \ \textsf{has type}\ \ \mathit{ty}}\ \boxed{\textsf{wf-Unit : Block (b)}}$$

$$\frac{\begin{array}{ll}\text{(a)} & \textsf{card}\, \mathit{local\text{-}signals}[\![\mathit{st}_1\ \ldots\ \mathit{st}_{k+1}]\!] = \mathit{count\text{-}signals}[\![\mathit{st}_1\ \ldots\ \mathit{st}_{k+1}]\!] \\ \text{(b)} & \mathit{st\text{-}env}_1 = \mathit{st\text{-}env} \\ \text{(c)} & \forall\ i \in \{1,\ldots,k+1\} \bullet \mathit{st\text{-}env}_i \vdash_{\mathit{Step}} \mathit{st}_i\ \textsf{produces}\ \mathit{st\text{-}env}_{i+1}\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Step}^*} [\![\mathit{st}_1\ \ldots\ \mathit{st}_{k+1}]\!]\ \textsf{produces}\ \mathit{st\text{-}env}_{k+2}}\ \boxed{\textsf{wf-Step sequence}}$$

The above static rules are fairly conventional, except for the 'redefinition' check for a block statement. This essentially forbids an existing signal to be redefined once it has been previously *mentioned* within the block. Note also that both relations ($\_ \vdash_{\mathit{Block}} \_$ has type $\_$) and ($\_ \vdash_{\mathit{Block}} \_$ produces $\_$) are defined for *Block*, since the output type of the block is required for type-checking in the calling expression, and the static environment must be updated, to include any new declarations made within the block.

Notice that clause (a) in *wf-Step sequence* requires that local bindings must use distinct signal names — any duplicates have been retained by *declash*, by renaming them to the same name. The operation *count-signals* finds the number of `LET` or `MAKE` steps occurring within a block. It exploits the operation *definition-signal*(st), which returns a singleton set if st is either a `LET` or `MAKE` step, and the empty set otherwise:

$$count\text{-}signals \;\; Step^* \to \mathbb{N}$$

$$count\text{-}signals[\![st_1, \ldots, st_k]\!] = \mathsf{card}\, definition\text{-}signal(st_1) + \cdots + \mathsf{card}\, definition\text{-}signal(st_k)$$

The auxiliary predicate for checking that all locally 'made' signals are 'joined' is defined as follows:

$$(\_ \vdash_{joined} \_) \subseteq (StaticEnvironment \times Step^*)$$

$$\boxed{\text{joined}}\; \frac{\forall\, nm \in (join\text{-}to\text{-}names(step\text{-}seq) \cap \mathsf{dom}\; st\text{-}env) \bullet st\text{-}env(nm) \in LetType}{st\text{-}env \vdash_{joined} step\text{-}seq}$$

The following auxiliary predicate enforces the 'redefinition' scoping property for block mentioned earlier:

$$(\vdash_{redefinition} \_) \subseteq Step^*$$

$$\boxed{\text{redefinition}}\; \frac{\begin{array}{l} \text{(a)} \quad free\text{-}sigs_1 = \emptyset \\ \text{(b)} \quad \forall\, i \in \{1, \ldots, k+1\} \bullet \\ \qquad free\text{-}sigs_{i+1} = free\text{-}sigs_i \cup free\text{-}signals(st_i) \wedge \\ \qquad definition\text{-}signal(st_i) \cap free\text{-}sigs_{i+1} = \emptyset \end{array}}{\vdash_{redefinition} [\![st_1 \; \ldots \; st_{k+1}]\!]}$$

The dynamic semantics for a block is evaluated by creating a temporary local environment $lcl\text{-}env \in LocalEnv$ which contains the bindings for the local signals introduced by the `LET` and `MAKE` statements. The dynamic environment is also updated to create *block-env*, which contains any local type or function declarations. The local declaration bindings are evaluated by traversing the sequence of steps $st_1, \ldots, st_{k+1}$. Each step st_i is evaluated within the environments resulting from the evaluation of step $st_{i\text{-}1}$. The unit *un* is evaluated in *block-env* to form *UN*. The causal product is then taken for all of the objects bound to the local signals, i.e., those contained in *lcl-env*, together with *UN*, using the operation *block-join-product*; this finally hides the local signals in the resulting object. Note that only the definitions for local type- and function-declarations are added to the dynamic environment *dy-env* — those intermediate objects bound to local signals in *lcl-env* are discarded once *term* has been formed by *block-join-product*.

Dynamic semantics equations:

$$M\text{-}Unit[\![\;\texttt{BEGIN}\; st_1 \;\ldots\; st_{k+1}\; \texttt{OUTPUT}\; un\; \texttt{END}\;]\!](oc)(\rho, \sigma) \;=\; (term, dy\text{-}env)$$

where

$$term = block\text{-}join\text{-}product(UN, chandecls)(lcl\text{-}env)$$

$$chandecls = local\text{-}channels(step\text{-}nms)(\rho, \sigma)$$

$$step\text{-}nms = \mathsf{dom}\, lcl\text{-}env$$

$$(UN, dy\text{-}env) = M\text{-}Unit[\![un]\!](oc)(\rho, block\text{-}env)$$

$$(block\text{-}env, lcl\text{-}env) = M\text{-}Step^*[\![st_1 \;\ldots\; st_{k+1}]\!](\rho, \sigma)$$

$$M\text{-}Step^* : Step^* \rightarrow Environment \rightarrow DynamicEnvironment \times LocalEnv$$

$$M\text{-}Step^*[\![st_1 \ \ldots \ st_{k+1}]\!](\rho, \sigma) \ = \ (block\text{-}env, lcl\text{-}env)$$

where

$$dy\text{-}ev_1 = \sigma$$

$$lcl\text{-}ev_1 = \{\}$$

$$(\forall\, i \in \{1, \ldots, k+1\} \cdot (dy\text{-}ev_{i+1}, lcl\text{-}ev_{i+1}) = M\text{-}Step(st_i)(\rho, dy\text{-}ev_i, lcl\text{-}ev_i))$$

$$block\text{-}env = dy\text{-}ev_{k+2}$$

$$lcl\text{-}env = lcl\text{-}ev_{k+2}$$

The operation *block-join-product* is used to form the causal product of the constituent components within a block — these are contained in the range of the local environment *lcl-env*, taken with the object *UN*. The local signals — the domain of *lcl-env* — are then hidden:

$$block\text{-}join\text{-}product : ProcExpr \times ChannelSet \rightarrow LocalEnv \rightarrow ProcExpr$$

$$block\text{-}join\text{-}product(UN, chandecls)(lcl\text{-}env) = (\!\![\left(\ P_1 \ | \ \cdots \ | \ P_k \ | \ UN \ \right) \backslash chandecls \,]\!\!)$$

where

$$objects = \mathsf{rng}\, lcl\text{-}env$$

$$k = \mathsf{card}\, objects$$

$$\mathsf{elems}\, [P_1, \ \cdots \ , P_k] = objects$$

The operation $local\text{-}channels(nm\text{-}set)(\rho, \sigma)$ produces a set expression containing channel declarations corresponding to the signals contained in *nm-set*:

$$local\text{-}channels : \mathcal{F}(Signal) \rightarrow Environment \rightarrow ChannelSet$$

$$local\text{-}channels(nm\text{-}set)(\rho, \sigma) = \{(\!\![\ nm : \ ty \,]\!\!) \mid nm \in nm\text{-}set \ \wedge \ ty = type\text{-}of(nm)(\rho, \sigma)\}$$

5.8.16.1 Step bindings within blocks

Structure:

$$\begin{array}{|rcl|}\hline \mathit{Step} & ::= & \mathit{declaration} \mid \mathit{let} \mid \mathit{make} \mid \mathit{join} \\ \mathit{Let} & ::= & \texttt{LET}\ \mathit{signal} = \mathit{unit}\ . \\ \mathit{Make} & ::= & \texttt{MAKE}\ \mathit{fnname} : \mathit{signal}\ . \\ \mathit{Join} & ::= & \texttt{JOIN}\ \mathit{unit} \rightarrow \mathit{signal}\ . \\ \hline \end{array}$$

A step can be a local type or function declaration, or a signal binding using either `LET` or `MAKE` – `JOIN` . The latter involves an explicit function instantiation under `MAKE` together with the binding under `JOIN` . A signal name *sig* has two roles — it must be considered to identify both a variable called *sig*, *and* a function instantiation whose output is named *sig*. This is implemented below by storing an intermediate object in a *local* environment, accessed by the signal name.

Note that an alternative method for introducing mutually recursive declarations for local signals uses a *recursive let* structure, which lifts the definition-before-use restriction imposed on the ELLA `LET` construct. The semantics for a `RECLET` construct is given in Appendix F.

Static semantics rules:

$\forall\ \mathit{decl} \in \mathit{Declaration}\ \bullet$

$$\boxed{\text{wf-Step : Declaration}}\ \frac{\mathit{st\text{-}env} \vdash_{\mathit{Declaration}} \mathit{decl}\ \text{produces}\ \mathit{new\text{-}st\text{-}env}}{\mathit{st\text{-}env} \vdash_{\mathit{Step}} \mathit{decl}\ \text{produces}\ \mathit{new\text{-}st\text{-}env}}$$

$$\boxed{\text{wf-Step : Let}}\ \frac{\begin{array}{ll} \text{(a)} & \mathit{sig} \notin \mathrm{dom}\, \mathit{st\text{-}env} \\ \text{(b)} & \mathit{st\text{-}env} \vdash_{\mathit{Unit}} \mathit{un}\ \text{ has type }\ \mathit{ty} \\ \text{(c)} & \mathit{sig\text{-}ty} = [\![\ \texttt{let-type}\ \mathit{ty}]\!] \\ \text{(d)} & \mathit{new\text{-}st\text{-}env} = \mathit{st\text{-}env}\ \cup\ \{\mathit{sig} \mapsto \mathit{sig\text{-}ty}\} \end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Step}} [\![\ \texttt{LET}\ \mathit{sig} = \mathit{un}.]\!]\ \text{produces}\ \mathit{new\text{-}st\text{-}env}}$$

$$\boxed{\text{wf-Step : Make}}\ \frac{\begin{array}{ll} \text{(a)} & \mathit{sig} \notin \mathrm{dom}\, \mathit{st\text{-}env} \\ \text{(b)} & \mathit{fn\text{-}nm} \in \mathrm{dom}\ \mathit{st\text{-}env} \\ \text{(c)} & \mathit{st\text{-}env}(\mathit{fn\text{-}nm}) = [\![\ \texttt{funtype}\ \mathit{ty}_1 \rightarrow \mathit{ty}_2]\!] \\ \text{(d)} & \mathit{sig\text{-}ty} = [\![\ \texttt{make-type}\ \mathit{ty}_1\ \mathit{ty}_2]\!] \\ \text{(e)} & \mathit{new\text{-}st\text{-}env} = \mathit{st\text{-}env} \cup \{\mathit{sig} \mapsto \mathit{sig\text{-}ty}\} \end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{Step}} [\![\ \texttt{MAKE}\ \mathit{fn\text{-}nm} :\ \mathit{sig}.]\!]\ \text{produces}\ \mathit{new\text{-}st\text{-}env}}$$

$$\begin{array}{ll}
\text{(a)} & sig \in \textsf{dom}\ st\text{-}env \\
\text{(b)} & st\text{-}env(sig) = [\![\ \texttt{make-type}\ ty_1\ ty_2]\!] \\
\text{(c)} & st\text{-}env \vdash_{Unit} un\ \ \textsf{has type}\ \ ty_3 \\
\text{(d)} & st\text{-}env \vdash_{E\text{-}TyEq} ty_1 \equiv ty_3 \\
\text{(e)} & sig\text{-}ty = [\![\ \texttt{let-type}\ ty_2]\!] \\
\text{(f)} & new\text{-}st\text{-}env = st\text{-}env\ \dagger\ \{sig \mapsto sig\text{-}ty\}
\end{array}$$

$$\boxed{\textsf{wf-Step : Join}}\ \frac{\qquad}{st\text{-}env \vdash_{Step} [\![\ \texttt{JOIN}\ un\ \rightarrow\ sig.]\!]\ \textsf{produces}\ new\text{-}st\text{-}env}$$

A step is well-formed under the following conditions:

- *Declaration*: the declaration *decl* is well-formed,
- *Make*: the function *fnname* has already been declared,
- *Join*: the signal name *sig* has been bound, and the (output) type of the unit *un* matches the (input) type of the function instantiation to which *sig* is bound.

Note that the new local signals introduced by `LET` or `MAKE` statements will be unique in the static environment ρ, because of the check in statement (a) of rule *wf-Step sequence* for block structures.

The dynamic semantics for each step will now be given.

Dynamic semantics equations:

$$M\text{-}Step : Step \rightarrow Environment \times LocalEnv \rightarrow DynamicEnvironment \times LocalEnv$$
$$M\text{-}Step(decl)(\rho, \sigma, lcl\text{-}env) = (M\text{-}Declaration(decl)(\rho, \sigma), lcl\text{-}env)$$

The semantics of a declaration step *decl* is therefore exactly the meaning of the declaration.

For $[\![\ \texttt{LET}\ sig = un]\!]$, the rhs *un* is evaluated, with its output named *sig*, to form a process term *LET-OBJ* — all the inputs have been 'connected'; *LET-OBJ* will be included later in the causal-product which forms the term representing the current block. Then *LET-OBJ* is stored in the local environment accessed by *sig*:

$$M\text{-}Step[\![\ \texttt{LET}\ sig = un.]\!](\rho, \sigma, lcl\text{-}env) = (unit\text{-}env, new\text{-}env)$$
where
$$(LET\text{-}OBJ, unit\text{-}env) = M\text{-}Unit(un)(sig)(\rho, \sigma)$$
$$new\text{-}env = lcl\text{-}env \cup \{sig \mapsto LET\text{-}OBJ\}$$

For $[\![\ \texttt{MAKE}\ fnname\text{: } sig]\!]$, the process name *proc-name(fnname)* is simply recorded at this stage. This enables a call to the process definition σ(*fnname*) with the same process name, representing the function *fnname*, to be made later when *sig* appears on the rhs of a `JOIN` statement. The process name is then stored in the local environment, accessed by *sig*:

$$M\text{-}Step[\![\ \texttt{MAKE}\ fnname:\ sig.]\!](\rho,\sigma,lcl\text{-}env) = (\sigma,new\text{-}env)$$

where

$$MAKE\text{-}OBJ = proc\text{-}name(fnname)$$
$$new\text{-}env = lcl\text{-}env \cup \{sig \mapsto MAKE\text{-}OBJ\}$$

For $[\![\ \texttt{JOIN}\ un \rightarrow sig]\!]$, the process associated with *sig*, whose name has been recorded as *OBJ* by a previous `MAKE` step, is now called with input channel *pipe* and output channel *sig*. This can now be connected to the evaluated unit *UN*, whose output is also labelled *pipe*, to create *COMP*. The connecting channel *pipe* is then hidden. Note that during the evaluation of *UN*, any `LET` or `MAKE` signal name appearing in *un* will result in the term representing the *variable* being used. The process term *COMP* is stored in the local environment accessed by *sig*, and replacing the previous object *OBJ* stored:

$$M\text{-}Step[\![\ \texttt{JOIN}\ un\ \rightarrow\ sig.]\!](\rho,\sigma,lcl\text{-}env) = (dy\text{-}env,new\text{-}env)$$

where

$$new\text{-}env = lcl\text{-}env \dagger \{sig \mapsto COMP\}$$
$$COMP = (\!|\ \Big(\ UN\ \mid\ OBJ(pipe\ /\ sig)\ \Big) \setminus \{pipe\colon\ UTY\}\ |\!)$$
$$(UN,dy\text{-}env) = M\text{-}Unit[\![un]\!](pipe)(\rho,\sigma)$$
$$lcl\text{-}env(sig) = OBJ$$
$$UTY = type\text{-}of(un)(\rho,\sigma)$$
$$FTY = type\text{-}of(sig)(\rho,\sigma)$$
$$\{pipe\} = make\text{-}ChanVars(1,\mathrm{dom}\,\rho)$$

Clearly a `LET` statement is equivalent to a `MAKE` / `JOIN` pair; for example, the following two expressions have the same meaning:

```
LET x = FUNC(a, b, c).              MAKE FUNC: x.
                                    JOIN (a, b, c) -> x.
```

Of course, any signal names appearing on the right-hand side of a `LET` statement must have previously been declared — hence the need for `JOIN` .

5.8.17 Case analysis

Structure:

Case	$::=$	`CASE` $unit_1$ `OF` $caseitem_1\ ,\ \ldots\ ,\ caseitem_{k+1}$ `ELSE` $unit_2$ `ESAC`
Caseitem	$::=$	$constset : unit$

The `CASE` statement works in the usual way: the output from the chooser unit *ch* is used to select one of the outputs from the units un_i in the case-items $cns_i : un_i$. If the value of *ch* does not match with any values determined by the constant set cns_i then the `ELSE` part of the case statement is selected and the output from *el* is produced. Note that the sets of values described by the cns_i are disjoint, ensuring that exactly one output from the branches can be selected for any particular value of *ch*.

Static semantics rules:

$$\frac{\begin{array}{ll} \text{(a)} & \textit{st-env} \vdash_{Unit} ch \text{ has type } \textit{ch-ty} \\ \text{(b)} & \textit{st-env} \vdash_{Unit} el \text{ has type } ty_1 \\ \text{(c)} & \forall\, i \in \{1, \cdots, (k+1)\} \bullet \textit{st-env} \vdash_{CaseItem} citm_i \text{ has type } (cty,\ ty_2) \\ \text{(d)} & \textit{st-env} \vdash_{E\text{-}TyEq} ty_1 \equiv ty_2 \\ \text{(e)} & \textit{st-env} \vdash_{E\text{-}TyEq} \textit{ch-ty} \equiv cty \\ \text{(f)} & \forall\, i, j \in \{1, \cdots, (k+1)\} \bullet (i \neq j) \Rightarrow \textit{st-env} \vdash_{disjoint} citm_i \not\sim citm_j \end{array}}{\textit{st-env} \vdash_{Unit} [\![\ \texttt{CASE}\ ch\ \texttt{OF}\ citm_1\ ,\ \ldots\ ,\ citm_{k+1}\ \texttt{ELSE}\ el\ \texttt{END}\ ]\!] \text{ has type } ty_2}\ \boxed{\text{wf-Unit : Case}}$$

$$\frac{\begin{array}{ll} \text{(a)} & \textit{st-env} \vdash_{ConstSet} cns \text{ has type } \textit{cns-ty} \\ \text{(b)} & \textit{st-env} \vdash_{Unit} un \text{ has type } \textit{un-ty} \end{array}}{\textit{st-env} \vdash_{CaseItem} [\![cns : un]\!] \text{ has type } (\textit{cns-ty},\ \textit{un-ty})}\ \boxed{\text{wf-CaseItem}}$$

A `CASE` statement is well-formed when:

- the values in each cns_i are of the same type *ch-ty* as the output of the chooser *ch*,
- the rhs units un_i of the case-items, and the else-unit *el* all have the same type ty_2,
- the constant sets formed from the cns_i in each case-item are disjoint.

The semantics for `CASE` constructs a multiplexor *MUX* with the the generic process $Chooser_{[CTY,OTY,setseq]}$. If the value output from the chooser term *CH* matches with the i^{th} chooser set cns_i, then $Chooser_{[CTY,OTY,setseq]}$ selects the output from P_i, which represents the i^{th} case item unit un_i. If there is no match then $Chooser_{[CTY,OTY,setseq]}$ selects the output from *EL*. Figure 1 shows how the various process are 'connected'.

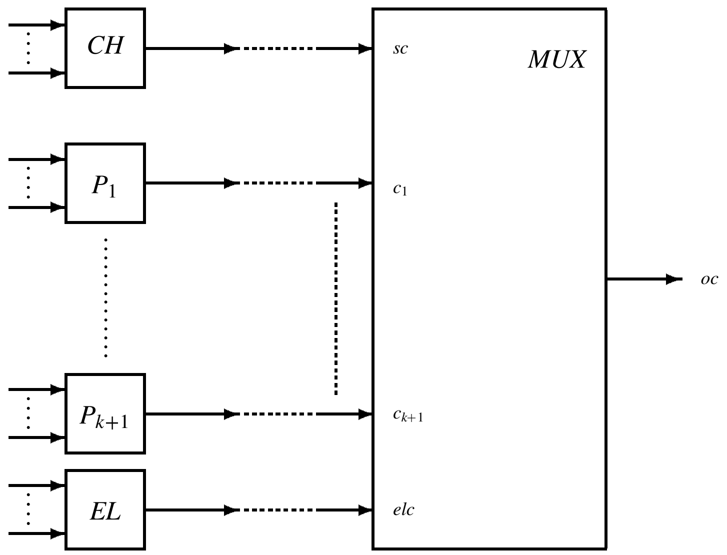

Figure 1: Components for `CASE`

Dynamic semantics equations:

$$M\text{-}Unit[\![\ \texttt{CASE}\ ch\ \texttt{OF}\ citm_1,\ \cdots\ ,citm_{k+1}\ \texttt{ELSE}\ el\ \texttt{ESAC}\ ]\!](oc)(\rho,\sigma) = (term, else\text{-}env)$$

where

$$term = (\!\![\ \Big(\ CH\ \mid\ P_1\ \mid\ \cdots\ \mid\ P_{k+1}\ \mid\ EL\ \mid\ MUX\ \Big)\backslash chandecls\ ]\!\!)$$

$$(CH, c\text{-}env_1) = M\text{-}Unit[\![ch]\!](sc)(\rho,\sigma)$$

$$(\forall\ i \in \{1,\ldots,k+1\} \bullet (P_i, c\text{-}env_{i+1}) = M\text{-}Unit(choice\text{-}of[\![citm_i]\!])(c_i)(\rho, c\text{-}env_i))$$

$$(EL, else\text{-}env) = M\text{-}Unit[\![el]\!](elc)(\rho, c\text{-}env_{k+2})$$

$$MUX = (\!\![\ Chooser_{[CTY,OTY,set_1,\ \cdots\ ,set_{k+1}]})(sc, c_1,\ \cdots\ ,c_{k+1}, elc\ /\ oc)\ ]\!\!)$$

$$\forall\ i \in \{1,\ldots,k+1\} \bullet set_i = M\text{-}ConstSet(constset\text{-}of[\![citm_i]\!])(\rho,\sigma)$$

$$CTY = type\text{-}of(ch)(\rho,\sigma)$$

$$OTY = type\text{-}of(el)(\rho,\sigma)$$

$$chandecls = (\!\![\ \{sc\colon\ CTY, c_1\colon\ OTY,\ \cdots\ ,c_{k+1}\colon\ OTY, elc\colon\ OTY\}\ ]\!\!)$$

$$\{sc, c_1,\ \cdots\ ,c_{k+1}, elc\} = make\text{-}ChanVars(k+3, \mathsf{dom}\,\rho \cup \{oc\})$$

The generic process $Chooser_{[ty,oty,se_1,\ \cdots\ ,se_{k+1}]}$ is simply a multiplexor, which attempts to match the value appearing on input channel sc against the choice set expressions $se_1,\ \cdots\ ,se_{k+1}$, and selects the value appearing on one of the input channels $c_1,\ \cdots\ ,c_{k+1}, el$, accordingly:

$$
\begin{array}{l}
[\![\text{ process } \mathit{Chooser}_{[ty,oty,se_1,\,\cdots\,,se_{k+1}]}(sc\colon ty, c_1\colon oty, \cdots, c_{k+1}\colon oty, elc\colon oty \;/\; oc\colon oty) \,\underline{\triangle} \\
\quad \Box\, s\colon ty, x_1\colon oty, \cdots, x_{k+1}\colon oty, y\colon oty \;\bullet \\
\qquad sc(s), c_1(x_1), \cdots, c_{k+1}(x_{k+1}), elc(y) \;/ \\
\qquad\quad oc(\text{case } s : se_1 \;\rightarrow\; x_1, \cdots, se_{k+1} \;\rightarrow\; x_{k+1} \text{ else } y \text{ end}) \\
\qquad\quad :: \mathit{Chooser}_{[ty,oty,se_1,\,\cdots\,,se_{k+1}]}(sc, c_1, \cdots, c_{k+1}, elc \;/\; oc) \,]\!]
\end{array}
$$

where

$$\{sc, c_1, \cdots, c_{k+1}, elc, o\} = \mathit{make\text{-}ChanVars}(k+3, \emptyset)$$

The operations *constset-of* and *choice-of* simply access the chooser set and unit, respectively, in a case item:

$$\mathit{constset\text{-}of} : \mathit{CaseItem} \rightarrow \mathit{ConstSet}$$

$$\mathit{constset\text{-}of}[\![cns : un]\!] = cns$$

$$\mathit{choice\text{-}of} : \mathit{CaseItem} \rightarrow \mathit{Unit}$$

$$\mathit{choice\text{-}of}[\![cns : un]\!] = un$$

The auxiliary case disjointness constraint predicate is defined in terms of a predicate stating what matching means:

$$(\_ \vdash_{disjoint} \_ \not\sim \_) \subseteq (\mathit{StaticEnvironment} \times \mathit{CaseItem} \times \mathit{CaseItem})$$

$$\boxed{\text{disjoint}}\;\frac{\neg(\mathit{st\text{-}env} \vdash_{matches} cns_1 \sim cns_2)}{\mathit{st\text{-}env} \vdash_{disjoint} [\![cns_1 : un_1]\!] \not\sim [\![cns_2 : un_2]\!]}$$

$$(\_ \vdash_{matches} \_ \sim \_) \subseteq (\mathit{StaticEnvironment} \times \mathit{ConstSet} \times \mathit{ConstSet})$$

$$\boxed{\text{matches : identity}}\;\frac{\text{true}}{\mathit{st\text{-}env} \vdash_{matches} cns \sim cns}$$

$$\boxed{\text{matches : symmetric}}\;\frac{\mathit{st\text{-}env} \vdash_{matches} cns_2 \sim cns_1}{\mathit{st\text{-}env} \vdash_{matches} cns_1 \sim cns_2}$$

$$\boxed{\text{matches : ConstSetAlt}}\;\frac{\exists\, i \in \{1, \ldots, j\} \bullet (\mathit{st\text{-}env} \vdash_{matches} cns_i \sim cns)}{\mathit{st\text{-}env} \vdash_{matches} [\![cns_1 \mid \cdots \mid cns_j]\!] \sim cns}$$

$$\fbox{matches : ConstSetString}\ \dfrac{\begin{array}{ll}\text{(a)} & \forall\, i \in \{1,\ldots,n\} \bullet (\textit{st-env} \vdash_{\textit{matches}} [\![\textit{tag}/\textit{ch}_i]\!] \sim \textit{cns}_1) \\ \text{(b)} & \textit{st-env} \vdash_{\textit{matches}} [\![\textit{tag}\,''\,\textit{ch}_1,\ \cdots\ ,\ \textit{ch}_n\,''\,)]\!] \sim \textit{cns}\end{array}}{\textit{st-env} \vdash_{\textit{matches}} [\![\ \texttt{STRING}\ [n]\textit{cns}_1]\!] \sim \textit{cns}}$$

$$\fbox{matches : ConstSetRow}\ \dfrac{\forall\, i \in \{1,\ldots,n\} \bullet \textit{st-env} \vdash_{\textit{matches}} \textit{acns}_i \sim \textit{bcns}_i}{\textit{st-env} \vdash_{\textit{matches}} [\![(\textit{acns}_1,\ \cdots\ ,\ \textit{acns}_n)]\!] \sim [\![(\textit{bcns}_1,\ \cdots\ ,\ \textit{bcns}_n)]\!]}$$

$$\fbox{matches : TypeName (a)}\ \dfrac{\begin{array}{ll}\text{(a)} & \textit{st-env} \vdash_{\textit{unit}} \textit{enum}\ \text{has type}\ \textit{ty} \\ \text{(b)} & \textit{st-env} \vdash_{\textit{E-TyEq}} \textit{ty} \equiv \textit{tynm}\end{array}}{\textit{st-env} \vdash_{\textit{matches}} \textit{tynm} \sim \textit{enum}}$$

$$\fbox{matches : TypeName (b)}\ \dfrac{\begin{array}{ll}\text{(a)} & \textit{tynm}_1 \in \text{dom}\ \textit{st-env} \\ \text{(b)} & [\![\textit{tag}\ \&\ \textit{tynm}_2]\!] \in \textit{newtype-parts-of}(\textit{st-env}(\textit{tynm}_1))\end{array}}{\textit{st-env} \vdash_{\textit{matches}} \textit{tynm}_1 \sim [\![\textit{tag}\ \&\ \textit{tynm}_2]\!]}$$

5.8.17.1 ConstSets – Case Pattern Matching

Structure:

$$
\begin{array}{|rcl|}
\hline
\mathit{ConstSet} & ::= & \mathit{enumerated} \mid \mathit{constsetalt} \mid \mathit{constsetstring} \\
 & \mid & \mathit{constsetrow} \mid \mathit{constsetassoc} \mid \mathit{constsettype} \\
\mathit{ConstSetAlt} & ::= & (\mathit{constset}_1 \,\underline{|}\, \ldots \,\underline{|}\, \mathit{constset}_{k+1}) \\
\mathit{ConstSetString} & ::= & \texttt{STRING}\ [\mathit{nat}]\mathit{constset} \\
\mathit{ConstSetRow} & ::= & (\mathit{constset}_1\ ,\ \ldots\ ,\ \mathit{constset}_{k+1}) \\
\mathit{ConstSetAssoc} & ::= & \mathit{enum}\ \&\ \mathit{constset} \\
\mathit{ConstSetType} & ::= & \mathit{typename} \\
\hline
\end{array}
$$

A Constset describes a set of alternative values $(cns_1 \,\underline{|}\, \cdots \,\underline{|}\, cns_k)$ for a case-item in a `CASE` statement that can match with the output of the case selector. If a typename *tynm* is included as an element in an alternative, then that element is matched with any value in type referred to — *tynm* is acting as a 'don't care' symbol.

Static semantics rules:

Let $en \in \mathit{Enumerated}$,

$$
\boxed{\text{wf-ConstSet : Enumerated}}\ \frac{\mathit{st\text{-}env} \vdash_{\mathit{Enumerated}} en \text{ has type } ty}{\mathit{st\text{-}env} \vdash_{\mathit{ConstSet}} en \text{ has type } ty}
$$

$$
\boxed{\text{wf-ConstSet : ConstSetAlt}}\ \frac{\forall\, i \in \{1,\ldots,k+1\} \bullet \mathit{st\text{-}env} \vdash_{\mathit{ConstSet}} cns_i \text{ has type } ty}{\mathit{st\text{-}env} \vdash_{\mathit{ConstSet}} [\![(cns_1 \,\underline{|}\, \cdots \,\underline{|}\, cns_{k+1})]\!] \text{ has type } ty}
$$

$$
\boxed{\text{wf-ConstSet : ConstSetString}}\ \frac{\begin{array}{ll}\text{(a)} & n \geq 1 \\ \text{(b)} & \mathit{st\text{-}env} \vdash_{\mathit{ConstSet}} cns \text{ has type } cty \\ \text{(c)} & \mathit{st\text{-}env} \vdash_{\mathit{is\text{-}char\text{-}type}} cty\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{ConstSet}} [\![\ \texttt{STRING}\ [n]cns]\!] \text{ has type } [\![\ \texttt{STRING}\ [n]\ cty]\!]}
$$

$$
\boxed{\text{wf-ConstSet : ConstSetRow}}\ \frac{\forall\, i \in \{1,\ldots,k+1\} \bullet \mathit{st\text{-}env} \vdash_{\mathit{ConstSet}} cns_i \text{ has type } ty_i}{\mathit{st\text{-}env} \vdash_{\mathit{ConstSet}} [\![(cns_1\ ,\ \cdots\ ,\ cns_{k+1})]\!] \text{ has type } [\![(ty_1,\ \cdots\ , ty_{k+1})]\!]}
$$

$$
\boxed{\text{wf-ConstSet : ConstSetAssoc}}\ \frac{\begin{array}{ll}\text{(a)} & tynm \in \text{dom}\, \mathit{st\text{-}env} \\ \text{(b)} & \mathit{st\text{-}env} \vdash_{\mathit{ConstSet}} cns \text{ has type } ty \\ \text{(c)} & [\![tag\ \&\ ty]\!] \in \mathit{newtype\text{-}of}(\mathit{st\text{-}env}(tynm))\end{array}}{\mathit{st\text{-}env} \vdash_{\mathit{ConstSet}} [\![tag\ \&\ cns]\!] \text{ has type } tynm}
$$

$$\frac{\begin{array}{ll}\text{(a)} & tynm \in Typename \\ \text{(b)} & st\text{-}env \vdash_{Type} tynm\end{array}}{st\text{-}env \vdash_{ConstSet} tynm \text{ has type } tynm}\ \boxed{\text{wf-ConstSet : ConstSetType}}$$

A ConstSet is well-formed simply if all of its alternative constants or types are well-formed.

The semantics of ConstSet creates an *EPA* set expression which yields values determined by the constant alternatives. For a row, note that each element is a *set* of values, and so the set of alternatives is formed from the product of these elemental sets. For a typename *tynm* then all of the values of the type to which *tynm* refers will be returned.

Dynamic semantics equations:

$$M\text{-}ConstSet : ConstSet \to Environment \to SetExpr$$

$$M\text{-}ConstSet[\![enum]\!](\rho,\sigma) = (\![\ \{enum\}\ ]\!)$$

$$M\text{-}ConstSet[\![(cns_1 \,\underline{|}\, \cdots \,\underline{|}\, cns_{k+1})]\!](\rho,\sigma) = (\![\ set_1 \cup \cdots \cup set_{k+1}\ ]\!)$$

where

$$\forall\, i \in \{1,\ldots,k+1\} \bullet set_i = M\text{-}ConstSet(cns_i)(\rho,\sigma)$$

$$\begin{aligned} M\text{-}ConstSet[\![\ \texttt{STRING}\ [n]cnst]\!](\rho,\sigma) = \\ (\![\{\text{tag}(tag, v_1, \cdots, v_n) \mid (\forall\, i \in \{1,\ldots,n\} \bullet v_i = ch) \wedge \\ tag = \text{tag-of}(val) \wedge ch = \text{value-of}(val) \wedge \\ val \in cset\}]\!) \end{aligned}$$

where

$$cset = M\text{-}ConstSet[\![cnst]\!](\rho,\sigma)$$

$$M\text{-}ConstSet[\![(cns_1\,, \cdots, cns_{k+1})]\!](\rho,\sigma) = (\![\ set_1 \times \cdots \times set_{k+1}\ ]\!)$$

where

$$\forall\, i \in \{1, \cdots, (k+1)\} \bullet set_i = M\text{-}ConstSet[\![cns_i]\!](\rho,\sigma)$$

$$M\text{-}ConstSet[\![tag \,\&\, cnst]\!](\rho,\sigma) = (\![\ \{\text{tag}(tag, v) \mid v \in cset\}\ ]\!)$$

where

$$cset = M\text{-}ConstSet[\![cnst]\!](\rho,\sigma)$$

$$M\text{-}ConstSet(tynm)(\rho,\sigma) = (\![\ tynm\ ]\!)$$

5.9 Function Declarations

Structure:

$$\boxed{\mathit{FnDecl} \quad ::= \quad \mathit{unitfnd} \mid \mathit{primfnd} \mid \mathit{ramfnd} \mid \mathit{reformfnd} \mid \mathit{biopfnd} \mid \mathit{arithfnd}}$$

Function declarations are divided into six groups. In the first group, the function body is a unit, and the function declaration contains information about the input signal names and types, and the output type. For the next four groups the function body is special and the number of inputs is determined by the type of primitive, so there is no need to explicitly list the input signal names. When the body is a primitive (a delay or sample), there is a single input, which must match the type of body. When the body is a `RAM` , there are four inputs for data, read- and write-address, and write-enable. The address types determine the size of the RAM, and the write-enable must have precisely two values. For reform and biop, the type information for the bodies is provided by the function declaration signature. In the case of biops, the signature must fit a generic 'template'. Finally, for *arithfnd*, the function body is a special form but the function can take any number of (explicitly named) input arguments.

The following relation and semantic function are defined on a construct-by-construct basis:

$$(\_ \vdash_{\mathit{FnDecl}} \_ \text{ produces } \_) \subseteq (\mathit{StaticEnvironment} \times \mathit{FnDecl} \times \mathit{StaticEnvironment})$$

$$\mathit{M\text{-}FnDecl} : \mathit{FnDecl} \rightarrow \mathit{Environment} \rightarrow \mathit{DynamicEnvironment}$$

5.9.1 UnitFnD

Structure:

$$\boxed{\textit{UnitFnD} \quad ::= \quad \texttt{FN}\, \textit{fnname} = (\textit{signaldecl}_1,\ \cdots\ ,\textit{signaldecl}_k) \to \textit{type} : \textit{unit}.}$$

The function declaration for units $[\![\ \texttt{FN}\, \textit{fnm} = (\textit{sd}_1,\ \cdots\ ,\textit{sd}_k) \to \textit{ty} : \textit{unm}.]\!]$ gives a sequence of signal declarations for its input, and has an output of type *ty*, which must be the same as the type of the function body *unm*.

Static semantics rules:

$$\frac{\begin{array}{ll}
\text{(a)} & \textit{fnnm} \notin \mathsf{dom}\, \textit{st-env} \\
\text{(b)} & \textit{inp-sigs} = \textit{extract-fndecl-signals}[\![\textit{sd}_1,\ldots,\textit{sd}_k]\!] \\
\text{(c)} & \mathsf{len}\, \textit{inp-sigs} = k \\
\text{(d)} & \textit{st-env}_1 = \textit{st-env} \cup \textit{extract-sigdecl-binding}[\![\textit{sd}_1,\ldots,\textit{sd}_k]\!] \\
\text{(e)} & \textit{st-env}_1 \vdash_{\textit{Unit}} \textit{unm}\ \mathsf{has\ type}\ \textit{ty}_1 \\
\text{(f)} & \textit{st-env}_1 \vdash_{\textit{E-TyEq}} \textit{ty}_1 \equiv \textit{ty} \\
\text{(g)} & \textit{st-env}_1 \vdash_{\textit{Unit}} \textit{unm}\ \mathsf{produces}\ \textit{st-env}_2 \\
\text{(h)} & \textit{ty-seq} = \textit{extract-sigdecl-types}[\![\textit{sd}_1,\ldots,\textit{sd}_k]\!] \\
\text{(j)} & \textit{st-env}_3 = \textit{st-env}_2 \cup \{\textit{fnm} \mapsto [\![\ \texttt{funtype}\ \textit{ty-seq} \to \textit{ty}]\!]\}
\end{array}}{\textit{st-env} \vdash_{\textit{FnDecl}} [\![\ \texttt{FN}\, \textit{fnnm} = (\textit{sd}_1,\ \cdots\ ,\textit{sd}_k) \to \textit{ty} : \textit{unm}.]\!]\ \mathsf{produces}\ \textit{st-env}_3}\ \boxed{\text{wf-FnDecl: UnitFnd}}$$

Unit function declaration is well-formed when:

- the function name *fnnm* being introduced has not already been defined.
- the input signal names are unique,
- the free variables in the function body *unm* are declared in the signal declaration list,
- the output type of *unm* is the same as the function declaration output.

The operation *extract-sigdecl-binding* finds the types of the input signal names, and returns a static environment storing the variable-types:

$$\textit{extract-sigdecl-binding} :\ \textit{SigDecl}^* \to \textit{StaticEnvironment}$$

$$\textit{extract-sigdecl-binding}[\![\textit{sigdec}_1,\ldots,\textit{sigdec}_k]\!] = \{\textit{nm}_1 \mapsto \textit{vt}_1,\ldots,\textit{nm}_k \mapsto \textit{vt}_k\}$$

where

$$\forall i \in \{1,\ldots,k\} \cdot [\![\textit{ty}_i :\ \textit{nm}_i]\!] = \textit{sigdec}_i \wedge \textit{vt}_i = [\![\ \texttt{let-type}\ \textit{ty}_i]\!]$$

Unit function declaration creates the semantic object for the body *unm*; if there is more than one input, then these are bundled together using *bundle-inputs*. With this resulting object as the rhs process body, a process definition called $\textit{proc-name}(\textit{fnnm})$ is formed. This is stored in the environment, accessed by the function name. When the function is instantiated, a process call to $\textit{pnm}(i\ /\ o)$ is made, where i,o are the input and output channels with the desired names, and of the appropriate types.

Dynamic semantics equations:

$$M\text{-}FnDecl[\![\ \texttt{FN}\, fnnm = (sigdecl\text{-}seq) \rightarrow ty : unm.]\!](\rho,\sigma) =$$

$$unit\text{-}env \cup \{fnnm \mapsto procdefn\}$$

where

case len *sigdecl-seq* **in**

$\{0\}$: $procdefn = (\![\ \text{process}\ pnm(oc\!: oty) \mathrel{\underline{\triangle}} term_1\ ]\!)$

$\{1\}$: $procdefn = (\![\ \text{process}\ pnm(inm\!: ity\ /\ oc\!: oty) \mathrel{\underline{\triangle}} term_1\ ]\!)$

$\wedge\ inm = inm\text{-}seq(1)$

else $procdefn = (\![\ \text{process}\ pnm(ic\!: ity\ /\ oc\!: oty) \mathrel{\underline{\triangle}} term_2\ ]\!)$

$\wedge\ term_2 = bundle\text{-}inputs(term_1, ic, inm\text{-}seq)(\rho,\sigma)$

$\wedge\ \{ic\} = make\text{-}ChanVars(1, \mathsf{dom}\,\rho \cup \{oc\})$

end

$pnm = proc\text{-}name(fnnm)$

$(term_1, unit\text{-}env) = M\text{-}Unit[\![unm]\!](oc)(\rho,\sigma)$

$inm\text{-}seq = extract\text{-}fndecl\text{-}signals(sigdecl\text{-}seq)$

$\{oc\} = make\text{-}ChanVars(1, \mathsf{dom}\,\rho)$

$ity = M\text{-}Type(ITY)(\rho,\sigma)$

$oty = M\text{-}Type(OTY)(\rho,\sigma)$

$[\![\ \texttt{funtype}\ \ ITY \rightarrow OTY]\!] = \rho(fnnm)$

Note that if the function has a single input, then the input name, given by $inm\text{-}seq(1)$ is used to name the formal argument of the process definition; otherwise a fresh name *ic* is used, to name the bundled input channel. The input and output type expressions for the ELLA function declaration are retrieved from the static environment ρ — these are then translated into *EPA* type expressions before use in the *EPA* process definition representing the function.

The operation *bundle-inputs* constructs a process that forms a single input for the object representing a function body, where the order of the components of the input is determined by the input signal declarations for the function:

$$bundle\text{-}inputs : ProcExpr \times ChannelVarName \times Name^{*} \rightarrow Environment \rightarrow ProcDefn$$

$$bundle\text{-}inputs(obj, ic, name\text{-}seq)(\rho,\sigma) =$$

$$(\![\ \Big(\ obj\ \mid\ Bundle_{[k,ty_1,\ \cdots\ ,ty_k]}(ic\ /\ oc_1,\ \cdots\ ,oc_k)\ \Big)\backslash\{oc_1\!:\ ty_1,\ \cdots\ ,oc_k\!:\ ty_k\}\ ]\!)$$

where

$k = \mathsf{len}\ name\text{-}seq$

$\forall\ i \in \{1,\ldots,k\} \bullet ty_i = type\text{-}of(name\text{-}seq(i))(\rho,\sigma)\ \wedge\ oc_i = name\text{-}seq(i)$

The generic process $Bundle_{[k,ty_1,\ \cdots\ ,ty_k]}$ accepts a single input vector *v* and produces *k* outputs of the elements of *v*, where $ty_1,\ \cdots\ ,ty_k$ is the sequence of output types.

$$\llparenthesis\ \textsf{process}\ Bundle_{[k,ty_1,\ \cdots\ ,ty_k]}(i\colon ity\ /\ o_1\colon ty_1,\ \cdots\ ,o_k\colon ty_k)\ \underline{\triangle}$$

$$\square\, v\colon ity\ \bullet\ i(v)\ /\ o_1(\textsf{index}(v,1)),\ \cdots\ ,o_k(\textsf{index}(v,k)) :: Bundle_{[k,ty_1,\ \cdots\ ,ty_k]}(i\ /\ o_1,\ \cdots\ ,o_k)\ \rrparenthesis$$

where

$$ity = \llparenthesis\ ty_1 \times\ \cdots\ \times ty_k\ \rrparenthesis$$

$$\{i, o_1,\ \cdots\ ,o_k\} = make\text{-}ChanVars(k+1, \emptyset)$$

The operation *extract-fndecl-signals* returns the sequence of signal names occurring in a sequence of signal declarations, *extract-fndecl-types* returns the sequence of types, and *extract-sigdecl-types* returns a type expression for the declarations:

$$extract\text{-}fndecl\text{-}signals : SigDecl^* \to Name^*$$

$$extract\text{-}fndecl\text{-}signals(sigdec\text{-}seq) = [nm_1,\ \cdots\ ,nm_k]$$

where

$$k = \textsf{len}\, sigdec\text{-}seq$$

$$\forall\ i \in \{1,\ldots,k\} \bullet [\![ty_i : nm_i]\!] = sigdec\text{-}seq(i)$$

$$extract\text{-}fndecl\text{-}types : SigDecl^* \to Type^*$$

$$extract\text{-}fndecl\text{-}types(sigdec\text{-}seq) = [ty_1,\ \cdots\ ,ty_k]$$

where

$$k = \textsf{len}\, sigdec\text{-}seq$$

$$\forall\ i \in \{1,\ldots,k\} \bullet [\![ty_i : nm_i]\!] = sigdec\text{-}seq(i)$$

$$extract\text{-}sigdecl\text{-}types : SigDecl^* \to Type$$

$$extract\text{-}sigdecl\text{-}types(sigdec\text{-}seq) = (ty_1,\ldots,ty_k)$$

where

$$k = \textsf{len}\, sigdec\text{-}seq$$

$$\forall\ i \in \{1,\ldots,k\} \bullet [\![ty_i : nm_i]\!] = sigdec\text{-}seq(i)$$

$$extract\text{-}sigdecl\text{-}types[\![ty : nm]\!] = ty$$

5.9.2 PrimFnD

Structure:

$$\boxed{PrimFnD \quad ::= \quad \texttt{FN}\, fnname = (type_1) \to ty : primitive.}$$

The function declaration for primitives $[\![\ \texttt{FN}\, fnm = (ty_1) \to ty : prim.]\!]$ has a single input of type ty_1 and an output of type ty, which must both be the same as the type of the primitive *prim*.

Static semantics rules:

$$\frac{\begin{array}{ll} (a) & fnm \notin \mathrm{dom}\, st\text{-}env \\ (b) & st\text{-}env \vdash_{E\text{-}TyEq} ty_1 \equiv ty \\ (c) & st\text{-}env \vdash_{Primitive} prim \text{ has type } ty_2 \\ (d) & st\text{-}env \vdash_{E\text{-}TyEq} ty_2 \equiv ty \\ (e) & st\text{-}env_1 = st\text{-}env \cup \{fnm \mapsto [\![\ \texttt{funtype}\ ty_1 \to ty]\!]\} \end{array}}{st\text{-}env \vdash_{FnDecl} [\![\ \texttt{FN}\, fnm = (ty_1) \to ty : prim.]\!] \text{ produces } st\text{-}env_1} \quad \boxed{\text{wf-FnDecl: PrimFnD}}$$

Primitive function declaration is well-formed when:

- the function name *fnm* being introduced has not already been defined.
- there is a single input with the same type *ty* as the output
- there is primitive as the function body with type *ty*.

Primitive function declaration simply returns the semantic structure for the primitive body, and then stores this in the environment, accessed by the function name:

Dynamic semantics equations:

$$M\text{-}FnDecl[\![\ \texttt{FN}\, fnm = (ty_1) \to ty : prim.]\!](\rho, \sigma) = dy\text{-}env$$

where

$$dy\text{-}env = \sigma \cup \{fnm \mapsto M\text{-}Primitive[\![prim]\!](fnm)(\rho, \sigma)\}$$

5.9.3 RamFnD

Structure:

$$\boxed{\textit{RamFnD} \quad ::= \qquad \texttt{FN}\, \textit{fnname} = (\textit{type}_1, \ldots, \textit{type}_4) \rightarrow \textit{type}:\ \texttt{RAM}\,(\textit{unit}).}$$

The function declaration for RAM $[\![\ \texttt{FN}\,\textit{fnm} = (ty_1, \ldots, ty_4) \rightarrow ty_5 :\ \texttt{RAM}\,(\textit{unm}).]\!]$ has exactly four inputs. The first input is for data to be stored and so its type ty_1 should match the output type ty_5, and the type of the initial value stored in the RAM, given by *unm*. The second and third inputs are the write- and read-addresses, and so the two types ty_2 and ty_3 must be the same integer type — the integer range determines the size of the RAM. The fourth input is the write-enable, whose type ty_4 must be an enumeration type with exactly two values.

Static semantics rules:

(a) $\textit{fnm} \notin \mathsf{dom}\, \textit{st-env}$

(b) $\textit{st-env} \vdash_{\textit{E-TyEq}} ty_1 \equiv ty$

(c) $\textit{st-env} \vdash_{\textit{is-num-type}} ty_2$

(d) $\textit{st-env} \vdash_{\textit{E-TyEq}} ty_2 \equiv ty_3$

(e) $ty_4 \in \mathsf{dom}\, \textit{st-env} \ \wedge\ \textit{ety} = \textit{newtype-of}(\textit{st-env}(ty_4))$

(f) $(\textit{st-env} \vdash_{\textit{is-enum-type}} ty_4) \ \wedge\ \textit{card-type}(\textit{ety}) = 2$

(g) $\textit{st-env} \vdash_{\textit{E-TyEq}} ty_5 \equiv ty$

(h) $\textit{st-env} \vdash_{\textit{Unit}} \textit{unm}\ \mathsf{has\ type}\ ty$

(j) $\textit{st-env} \vdash_{\textit{is-const-value}} \textit{unm}$

(k) $\textit{ty-seq} = [\![(ty_1, \ldots, ty_4)]\!]$

(l) $\textit{st-env}_1 = \textit{st-env} \cup \{\textit{fnm} \mapsto [\![\ \texttt{funtype}\ \textit{ty-seq} \rightarrow ty_5]\!]\}$

wf-FnDecl: RamFnD

$$\textit{st-env} \vdash_{\textit{FnDecl}} [\![\ \texttt{FN}\,\textit{fnm} = (ty_1, \ldots, ty_4) \rightarrow ty_5 :\ \texttt{RAM}\,(\textit{unm}).]\!]\ \mathsf{produces}\ \textit{st-env}_1$$

The operation $\textit{card-type}(ty)$ returns the number of tags or characters in type *ty*. If $ty \in \textit{IntType}$ then it returns zero:

$$\textit{card-type} : \textit{NewTypeParts} \rightarrow \mathbb{N}$$
$$\textit{card-type}(\{\textit{tag}_1, \cdots, \textit{tag}_k\}) = k$$
$$\textit{card-type}[\![\textit{tag}('\textit{ch}_1 \ \underline{|} \ \cdots \ \underline{|}\ '\textit{ch}_k)]\!] = k$$
$$\textit{card-type}(\textit{typ}) = 0$$

where

$$\textit{typ} \in \textit{IntType}$$

Function declaration for RAM is well-formed when:

- the function name *fnnm* being introduced has not already been defined.
- there are four inputs, and the inputs/output have the required types.

- Unit *unm* has the required type, and produces a literal.

A process definition called *proc-name*(*fnnm*) representing the RAM is constructed, and stored in the environment, accessed by the function name. The four inputs to the RAM are bundled together to form a single input channel *ic* to the process:

Dynamic semantics equations:

$$M\text{-}FnDecl[\![\ \texttt{FN}\ fnnm = (ty_1, \ldots, ty_4) \rightarrow ty_5 :\ \texttt{RAM}\ (unm).]\!](\rho, \sigma) =$$
$$\sigma \cup \{fnnm \mapsto procdefn\}$$

where

$$procdefn = (\!\![\ \text{process}\ pnm(ic\colon\ intype\ /\ oc\colon\ DTY) \triangleq$$
$$\square\ iv \in iset\ \uparrow\ size\ \bullet\ Ram\text{-}Aux_{[DTY,ITY,ETY,elt_1,lb,ub]}(iv\ ;\ ic\ /\ oc)\ ]\!\!)$$
$$pnm = proc\text{-}name(fnnm)$$
$$iset = init\text{-}values[\![unm]\!](\sigma)$$
$$lb = min\text{-}int\text{-}rng(ITY)(\sigma)$$
$$ub = max\text{-}int\text{-}rng(ITY)(\sigma)$$
$$size = ub - lb + 1$$
$$DTY = M\text{-}Type[\![ty_1]\!]\sigma$$
$$ITY = M\text{-}Type[\![ty_2]\!]\sigma$$
$$ETY = M\text{-}Type[\![ty_4]\!]\sigma$$
$$\langle elt_1, elt_2 \rangle = enable\text{-}elts(ETY)(\sigma)$$
$$intype = (\!\![\ DTY \times ITY \times ITY \times ETY\ ]\!\!)$$
$$\{ic, oc\} = make\text{-}ChanVars(2, \mathrm{dom}\,\rho)$$

The process definition *proc-name*(*fnnm*) represents an *initialised* RAM by calling the generic process $Ram\text{-}Aux_{[DTY,ITY,ETY,elt_1,lb,ub]}$ once for each element contained in *iset*.

The generic process $Ram\text{-}Aux_{[dty,ity,ety,elt_1,lb,ub]}$ represents a RAM initialised with value sequence *iv*, which stores data of type *dty*, and has read- and write-addresses of type *ity*. The write-enable is of type *ety*, containing exactly two enumeration values elt_1, elt_2.

$$\llbracket \text{ process } \mathit{Ram\text{-}Aux}_{[dty,ity,ety,elt_1,lb,ub]}(store\colon dty \uparrow k \,;\, i\colon inty \,/\, o\colon dty) \triangleq$$
$$\square x_1\colon dty, x_2\colon ity, x_3\colon ity, x_4\colon ety \,\bullet$$
$$i(\mathsf{tuple}(x_1, \cdots, x_4)) \,/\, o(\mathit{Ram\text{-}1}_{[dty,ity,ety,lb,k]}(store, x_1, x_2, x_3, x_4))$$
$$::\ \mathit{Ram\text{-}Aux}_{[dty,ity,ety,elt_1,lb,ub]}(\mathit{Ram\text{-}2}_{[dty,ity,ety,elt_1,lb,k]}(store, x_1, x_2, x_4) \,;\, i \,/\, o) \rrbracket$$

$$\llbracket \text{function } \mathit{Ram\text{-}1}_{[dty,ity,ety,elt_1,lb,k]}(store\colon dty \uparrow k, x_1\colon dty, x_2\colon ity, x_3\colon ity, x_4\colon ety) : dty \triangleq$$

case x_4 :

$\{elt_1\} \rightarrow$ **case** value-of(x_2) = value-of(x_3) :

$\{\mathtt{t}\} \rightarrow x_1$

else index$(store$, value-of$(x_3) - lb + 1)$

end

else index$(store$, value-of$(x_3) - lb + 1)$

end$\rrbracket$

$$\llbracket \text{function } \mathit{Ram\text{-}2}_{[dty,ity,ety,elt_1,lb,k]}(store\colon dty \uparrow k, x_1\colon ity, x_2\colon ity, x_4\colon ety) : dty \uparrow k \triangleq$$

case x_4 :

$\{elt_1\} \rightarrow$ repl$(store$, value-of$(x_2) - lb + 1, x_1)$

else *store*

end$\rrbracket$

where

$k = ub - lb + 1$

$\{i, o\} = \mathit{make\text{-}ChanVars}(2, \emptyset)$

$inty = \llbracket\, dty \times ity \times ity \times ety \,\rrbracket$

The type expression *ETY* yields a two-valued set equivalent to $\llbracket\, \{elt_1, elt_2\} \,\rrbracket$, where the value elt_1 indicates write-enable and elt_2 disables writing. The variable *store* holds the values in the RAM. When $en = elt_1$ then the memory is updated at the write-address value-of(x_2), and then the read-address value-of(x_3) accesses the *updated* store. When $en = elt_2$, then the memory cannot be updated, and so the store does not change — RAM returns the read-address value. The store is initialised with value(s) from *iset*; if *iset* returns a singleton, then there is only one initial store possible, but if *iset* returns $\llbracket\ ?\ DTY \rrbracket$ then the initial store can be filled with *any* sequence of data-values from *DTY*. Note that the read- and write-addresses are offset by an amount $-lb + 1$ before an element of the RAM is retrieved — the addresses range over $\{lb, \ldots, ub\}$ whereas the elements of the RAM are indexed over $\{1, \ldots, k\}$, where $k = ub - lb + 1$.

The operation *enable-elts* returns a tuple containing the two enable value-expressions, which will either be tags, or tagged characters:

$$enable\text{-}elts : TypeExpr \to DynamicEnvironment \to DataExpr^2$$

$$enable\text{-}elts(ety)(\sigma) = \langle tag_1, tag_2 \rangle$$

where

$$\mathsf{rng}\,\sigma \vdash_{EPA\text{-}TyEq} ety \ \equiv \ \{tag_1, tag_2\}$$

$$enable\text{-}elts(ety)(\sigma) = \langle (\!\![\ \mathsf{tag}(tag, chr_1) \ ]\!\!), (\!\![\ \mathsf{tag}(tag, chr_2) \ ]\!\!) \rangle$$

where

$$\mathsf{rng}\,\sigma \vdash_{EPA\text{-}TyEq} ety \ \equiv \ (\!\![\ \mathsf{tag}(tag, \{chr_1, chr_2\}) \ ]\!\!)$$

The operations *min-int-rng* and *max-int-rng* simply return the smallest and largest integers from tagged integer sets.

$$min\text{-}int\text{-}rng : TypeExpr \to DynamicEnvironment \to \mathbb{Z}$$

$$max\text{-}int\text{-}rng : TypeExpr \to \mathbb{Z}$$

$$max\text{-}int\text{-}rng(tag\text{-}int\text{-}set) = ub$$

$$\wedge$$

$$min\text{-}int\text{-}rng(tag\text{-}int\text{-}set) = lb$$

where

$$\mathsf{rng}\,\sigma \vdash_{EPA\text{-}TyEq} tag\text{-}int\text{-}set \ \equiv \ (\!\![\ \mathsf{tag}(tag, \mathsf{int}\ lb\ ub) \ ]\!\!)$$

5.9.4 ReformFnD

Structure:

$$\boxed{\textit{ReformFnD} \quad ::= \quad \texttt{FN}\, \textit{fnname} = (\textit{type}_1, \ldots, \textit{type}_{k+1}) \rightarrow \textit{type}\colon \texttt{REFORM}\,.}$$

The Reform function declaration $[\![\ \texttt{FN}\, \textit{fnm} = (\textit{ty}_1, \ldots, \textit{ty}_{k+1}) \rightarrow \textit{ty}\colon \texttt{REFORM}\,.]\!]$ takes a sequence of inputs with types $\textit{ty}_1, \ldots, \textit{ty}_{k+1}$ and produces an output of type *ty*, where the sequence of atomic input values, if each input was decomposed into values of atomic type, is the same as the sequence of atomic output values, if the output was similarly decomposed.

Static semantics rules:

(a) $\textit{fnm} \notin \mathsf{dom}\, \textit{st-env}$

(b) $\textit{flatten-types}[\![(\textit{ty}_1, \ldots, \textit{ty}_{k+1})]\!](\textit{st-env}) = \textit{type-seq}$

(c) $\textit{flatten-types}[\![\textit{ty}]\!](\textit{st-env}) = \textit{type-seq}$

(d) $\textit{ty-prod} = \textit{make-type-seq}([\![(\textit{ty}_1, \ldots, \textit{ty}_{k+1})]\!])$

(e) $\textit{new-st-env} = \textit{st-env}\ \{\textit{fnm} \mapsto [\![\ \texttt{funtype}\ \textit{ty-prod} \rightarrow \textit{ty}]\!]\}$

wf-FnDecl: ReformFnd

$$\textit{st-env} \vdash_{\textit{FnDecl}} [\![\ \texttt{FN}\, \textit{fnm} = (\textit{ty}_1, \ldots, \textit{ty}_{k+1}) \rightarrow \textit{ty}\colon \texttt{REFORM}\,.]\!]\ \mathsf{produces}\ \textit{new-st-env}$$

Reform is well-formed if the sequences of atomic types in the flattened input and output types are the same.

The operation *flatten-types* accepts a sequence of (possibly) composite types, and produces the corresponding sequence of atomic types.

$$\textit{flatten-types} : \textit{StaticEnvironment} \rightarrow \textit{Type} \rightarrow (\textit{NewType} \cup \textit{StringType} \cup \textit{TypeVoid})^*$$

$$\forall \textit{tyn} : \textit{TypeName} \cdot \textit{flatten-types}(\textit{tyn})(\textit{st-env}) = \textit{flatten-types}(\textit{ty})(\textit{st-env})$$

where

$\textit{ty} = \textit{st-env}(\textit{tyn})$

$\textit{ty} \in \textit{TypeNaming}$

$$\forall \textit{tyn} : \textit{TypeName} \cdot \textit{flatten-types}(\textit{tyn})(\textit{st-env}) = \langle \textit{newtype-of}(\textit{ty}) \rangle$$

where

$\textit{ty} = \textit{st-env}(\textit{tyn})$

$\textit{ty} \in \textit{TypeFormer}$

$\forall ty : TupleType \cdot \mathit{flatten\text{-}types}(ty)(st\text{-}env) =$

$\quad \mathit{flatten\text{-}types}(ty_1)(st\text{-}env) \frown \ldots \frown \mathit{flatten\text{-}types}(ty_k)(st\text{-}env)$

where

$[\![(ty_1, \ldots, ty_k)]\!] = ty$

$\forall ty : PowerType \cdot \mathit{flatten\text{-}types}(ty)(st\text{-}env) = fty_1 \frown \ldots \frown fty_k$

where

$[\![[k]typ]\!] = ty$

$\forall i \in \{1, \ldots, k\} \cdot fty_i = \mathit{flatten\text{-}types}(typ)(st\text{-}env)$

$\forall ty : StringType \cdot \mathit{flatten\text{-}types}(ty)(st\text{-}env) = \langle ty \rangle$

$\forall ty : TypeVoid \cdot \mathit{flatten\text{-}types}(ty)(st\text{-}env) = \langle ty \rangle$

$\mathit{make\text{-}type\text{-}seq} : TupleType \to Type$

$\mathit{make\text{-}type\text{-}seq}[\![(ty_1)]\!] = ty_1$

$\mathit{make\text{-}type\text{-}seq}[\![(ty_1, \ldots, ty_{k+1})]\!] = [\![(ty_1, \ldots, ty_{k+1})]\!]$

where

$k \geq 1$

Dynamic semantics equations:

$M\text{-}FnDecl[\![\ \mathtt{FN}\ fnnm = (ty_1, \ldots, ty_{k+1}) \to ty \colon \ \mathtt{REFORM}\ .]\!](fnnm)(\rho, \sigma) =$

$\quad \sigma \cup \{fnnm \mapsto procdefn\}$

where

$procdefn = (\!\!|\ \text{process}\ pnm(ic\colon\ ITY\ /\ oc\colon\ ty)\ \underline{\triangle}$

$\qquad \Box\, v\colon ty_1 \times\ \cdots\ \times ty_{k+1}\ \bullet$

$\qquad\quad ic(v), oc(\text{reform-values}(v, ty)) :: pnm(ic\ /\ oc)\ |\!\!)$

$pnm = \mathit{proc\text{-}name}(fnnm)$

$ITY = (\!\!|\ ty_1 \times \ldots \times ty_{k+1}\ |\!\!)$

$\{ic, oc\} = \mathit{make\text{-}ChanVars}(2, \mathrm{dom}\, \rho)$

The value expression reform-values(v, ty) represents the operation which restructures the elements value v to produce a new value which is a member of type ty. The inputs have been bundled together.

5.9.5 BiopFnD

Structure:

$$\boxed{BiopFnD \quad ::= \qquad \texttt{FN}\, fnname = (type_1,\ldots,type_{k+1}) \to type\colon \texttt{BIOP}\; biopname.}$$

The *BIOP* function declaration has a sequence of inputs of type ty_i, $i \in \{1,\ldots,k+1\}$ and an output of type *ty*. This signature must be compatible with a type 'template' associated with each built-in operation *biopnm*.

Static semantics rules:

$$\frac{\begin{array}{ll}(a) & fnm \notin \mathsf{dom}\, st\text{-}env \\ (b) & st\text{-}env \vdash_{BIOP\text{-}type}\ biopnm\ \mathsf{matches\ type}\ \langle [\![(ty_1,\ \cdots\ ,ty_{k+1})]\!], ty\rangle \\ (c) & ty\text{-}seq = make\text{-}type\text{-}seq[\![ty_1,\ \cdots\ ,ty_{k+1}]\!] \\ (d) & st\text{-}env_1 = st\text{-}env \cup \{fnm \mapsto [\![\ \texttt{funtype}\ ty\text{-}seq \to ty]\!]\}\end{array}}{st\text{-}env \vdash_{FnDecl} [\![\ \texttt{FN}\, fnm = (ty_1,\ldots,ty_{k+1}) \to ty\colon\ \texttt{BIOP}\ biopnm.]\!]\ \mathsf{produces}\ st\text{-}env_1}\ \boxed{\text{wf-BiopFnD}}$$

A biop function is well-formed if:

- the function name *fnm* has not been used before,
- the signature of the function is compatible with the type template required by the particular (generic) biop *biopnm*

The relation $st\text{-}env \vdash_{BIOP\text{-}type}\ biopnm$ matches type $\langle intype, outtype\rangle$ checks that the input and output types *intype* and *outtpye* supplied to the biop are compatible with the type template expected by *biopnm*, with respect to the static environment *st-env*. A complete definition is not given here; refer to [HM92]. Table 2 shows type templates for a small selection of BIOPs; *ttype* is a two-valued enumeration type, *type* is a basic enumeration type, and *string* represents a (bit) string type. Note that *AND* can apply either to a two-valued enumeration type or to a string. The operation *make-type-seq* was defined in §5.9.4.

BIOP name	: input template	→ output
AND	: (ttype, ttype)	→ ttype
AND	: (string, string)	→ string
EQ	: (type, type)	→ ttype
EQ_US	: (string,string)	→ ttype
PLUS_US	: (string,string)	→ string
DIVIDE_S	: (string,string)	→ (ttype,string,string)

Table 2: Type Templates for BIOPS

The dynamic semantics of a biop produces a process $Biop_{[ity,oty,biopnm]}$ which simply instantiates the generic function *biopnm* for the input and output types *ity*, *oty* and applies it to the function arguments:

$$M\text{-}FnDecl[\![\ \texttt{FN}\, fnm = (ty_1, \ldots, ty_{k+1}) \to ty\colon\ \texttt{BIOP}\ biopnm.]\!](\rho, \sigma) =$$
$$\sigma \cup \{fnnm \mapsto procdefn\}$$

where

$$procdefn = (\!\![\ Biop_{[ity,oty,biopnm]}(ic\colon\ \ /\ oc\colon\ oty) \underline{\triangle}$$
$$\square\, v\colon\ ity, w\colon\ oty\ \bullet\ ic(v)\ /\ oc(funapp) :: Biop_{[ity,oty,biopnm]}(ic\ /\ oc)\ ]\!\!)$$
$$funapp = (\!\![\ \mathsf{biop}\ biopnm\,[ity, oty]\ (\ v, w\ )\ ]\!\!)$$
$$\{ic, oc\} = make\text{-}ChanVars(2, \mathsf{dom}\,\rho)$$

When the *EPA* expression is interpreted the generic operation *biopnm* will be instantiated for types *ity*, *oty*, and then applied to the input argument *v* from *ity*. If an error occurs — such as an out-of-range result for integer operations — then the *biopnm* outputs the non-deterministic value *w*, from *oty*. See Appendix B for interpretations of the built-in operators.

5.9.6 ArithFnD

Structure:

$\mathit{ArithFnD}$	$::=$	$\texttt{FN}\,\mathit{fnname} = (\mathit{signaldecl}_1, \ldots, \mathit{signaldecl}_k) \rightarrow \mathit{type}\colon\ \texttt{ARITH}\,\mathit{formula}.$
$\mathit{Formula}$	$=$	$\mathit{unexpr} \mid \mathit{binexpr} \mid \mathit{ifexpr} \mid \mathit{signal} \mid \mathit{int}$
UnExpr	$::=$	$\mathit{unsym}\,\mathit{formula}$
$\mathit{BinExpr}$	$::=$	$\mathit{formula}_1\ \mathit{binsym}\ \mathit{formula}_2$
IfExpr	$::=$	$\texttt{IF}\ \mathit{boolexpr}\ \texttt{THEN}\ \mathit{formula}_1\ \texttt{ELSE}\ \mathit{formula}_2\ \texttt{FI}$
UnSym	$::=$	$+ \mid - \mid \texttt{INOT} \mid \texttt{ABS} \mid \texttt{SQRT}$
BinSym	$::=$	$+ \mid - \mid \texttt{*} \mid \texttt{\%} \mid \texttt{IDIV} \mid \texttt{SL} \mid \texttt{SR} \mid \texttt{IAND} \mid \texttt{IOR} \mid \texttt{MOD} \mid \texttt{ILOG} \mid \texttt{IEXP}$
$\mathit{BoolExpr}$	$=$	$\mathit{intlogpred} \mid \mathit{unlogpred} \mid \mathit{binlogpred}$
$\mathit{IntLogPred}$	$::=$	$\mathit{formula}_1\ \mathit{boolop}\ \mathit{formula}_2$
$\mathit{UnLogPred}$	$::=$	$\texttt{NOT}\ \mathit{boolexpr}$
$\mathit{BinLogPred}$	$::=$	$\mathit{boolexpr}_1\ \mathit{logop}\ \mathit{boolexpr}_2$
BoolOp	$::=$	$\texttt{=} \mid \texttt{/=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=}$
LogOp	$::=$	$\texttt{AND} \mid \texttt{OR}$

The `ARITH` function declaration $[\![\ \texttt{FN}\,\mathit{fnnm} = (\mathit{sd}_1, \ldots, \mathit{sd}_k) \rightarrow \mathit{ty}\colon\ \texttt{ARITH}\,\mathit{formula}.]\!]$ takes a sequence of ELLA integers as inputs, and evaluates a standard arithmetic expression *formula*, which may include conditionals, using just the pure integer part of the inputs. The integer result is then coerced, if possible, to be a value of the output type *ty*, which must be a *primitive* type, i.e. an ELLA integer- or enumeration-type. For the former, then the appropriate tagged integer is output; for the latter, then the integer result is mapped onto a value from the enumeration type, according to the order of values appearing in the declaration. However, if the result from *formula* is out-of-range, then the output from the function is non-deterministically chosen. Note that there may be no inputs.

Static semantics rules:

(a) $\mathit{fnm} \notin \mathsf{dom}\,\mathit{st\text{-}env}$

(b) $\mathit{inp\text{-}sigs} = \mathit{extract\text{-}fndecl\text{-}signals}[\![\mathit{sd}_1, \ldots, \mathit{sd}_k]\!]$

(c) $\mathsf{len}\,\mathit{inp\text{-}sigs} = k$

(d) $\mathit{ty\text{-}seq} = \mathit{extract\text{-}fndecl\text{-}types}[\![\mathit{sd}_1, \ldots, \mathit{sd}_k]\!]$

(e) $\forall\, i \in \{1, \ldots, k\} \bullet \mathit{st\text{-}env} \vdash_{\mathit{is\text{-}num\text{-}type}} \mathit{ty\text{-}seq}(i)$

(f) $\mathit{st\text{-}env}_1 \vdash_{\mathit{is\text{-}prim\text{-}type}} \mathit{ty}$

(g) $\mathit{free\text{-}signals}(\mathit{formula}) \subseteq \mathsf{elems}\,\mathit{inp\text{-}sigs}$

(h) $\mathit{st\text{-}env}_1 = \mathit{st\text{-}env} \cup \mathit{extract\text{-}sigdecl\text{-}binding}[\![\mathit{sd}_1, \ldots, \mathit{sd}_k]\!]$

(j) $\mathit{inp\text{-}ty} = \mathit{extract\text{-}sigdecl\text{-}types}[\![\mathit{sd}_1, \ldots, \mathit{sd}_k]\!]$

(k) $\mathit{st\text{-}env}_2 = \mathit{st\text{-}env}_1 \cup \{\mathit{fnm} \mapsto [\![\ \texttt{funtype}\ \mathit{inp\text{-}ty} \rightarrow \mathit{ty}]\!]\}$

wf-FnDecl: ArithFnd

$\mathit{st\text{-}env} \vdash_{\mathit{FnDecl}} [\![\ \texttt{FN}\,\mathit{fnm} = (\mathit{sd}_1, \ldots, \mathit{sd}_k) \rightarrow \mathit{ty}\colon\ \texttt{ARITH}\,\mathit{formula}.]\!]$ produces $\mathit{st\text{-}env}_2$

ELLA `ARITH` function declaration is well-formed when:

- the function name *fnm* being introduced has not already been defined,
- the input signal names are unique, and the input types are ELLA integers,
- the output type *ty* of the function declaration is a primitive type, i.e., an ELLA integer or an enumerated type.
- the free variable declarations in the function body *formula* are contained within the the signal declaration list,

An `ARITH` function declaration is represented by a process definition called *proc-name*(*fnm*), which accepts an input of type *ity* and produces an output of type *oty*, given by the coerced value for the arithmetic expression *expr*. Notice that the (ELLA) input signal names *inm-seq* are used directly as value expression variables — this interpretation should be contrasted to that of ELLA *Unit* signal names, which are represented by process calls $\mathit{Chan\text{-}Proc}_{[ty]}(sig)$. The use of the signal names in this way simplifies the semantics for *Formula* objects, and still ensures that the desired connectivity of the inputs to the function is obtained when the process is called. The process definition is stored in the environment, accessed by the function name.

Dynamic semantics equations:

$$M\text{-}FnDecl[\![\ \texttt{FN}\, fnm = (sigdecl\text{-}seq) \rightarrow ty\colon\ \texttt{ARITH}\, formula.]\!](\rho,\sigma) =$$

$$unit\text{-}env \cup \{fnm \mapsto procdefn\}$$

where

case k **in**

$\{0\}$: $procdefn = (\!\![\ \mathsf{process}\ pnm(oc\colon\ oty) \underline{\triangle} \prod undefvar\colon\ oty \cdot\ oc(expr) :: pnm(oc)\ ]\!\!)$

else $procdefn = (\!\![\ \mathsf{process}\ pnm(ic\colon\ ity \ /\ oc\colon\ oty) \underline{\triangle}$

$\prod nm_1\colon\ ty_1,\ldots,nm_k\colon\ ty_k, undefvar\colon\ oty \cdot\ ic(\mathsf{tuple}(nm_1,\ldots,nm_k)) \ /\ oc(expr)$

$:: pnm(ic \ /\ oc)\ ]\!\!)$

end

$pnm = proc\text{-}name(fnm)$

$k = \mathsf{len}\, sigdecl\text{-}seq$

$inm\text{-}seq = extract\text{-}fndecl\text{-}signals(sigdecl\text{-}seq)$

$intype\text{-}seq = extract\text{-}sigdecl\text{-}types(sigdecl\text{-}seq)$

$[nm_1,\ldots,nm_k] = inm\text{-}seq$

$[ty_1,\ldots,ty_k] = intype\text{-}seq$

$expr = \mathsf{convert\text{-}types}_{[oty]}(undefvar, arithexpr)$

$arithexpr = M\text{-}Formula[\![formula]\!])$

$\{undefvar\} = make\text{-}ExprVars(1)$

$\{ic, oc\} = make\text{-}ChanVars(2, \emptyset)$

$[\![\ \texttt{funtype}\ \ ITY \rightarrow OTY]\!] = \rho(fnm)$

$ity = M\text{-}Type(ITY)(\rho,\sigma)$

$oty = M\text{-}Type(OTY)(\rho,\sigma)$

The operation $\mathsf{convert\text{-}types}_{[ty]}(v, val)$ coerces, if possible, the output value given by *arithexpr* into a value from type *ty*. If this is not possible, then the (non-deterministic) value which will be bound to v is returned. The operation $M\text{-}Formula[\![formula]\!]$ builds an *EPA* value expression representing *formula*, to be embedded into the output of the process; this is naturally very similar to the original `ARITH` expression:

$M\text{-}Formula : Formula \rightarrow GenDateExpr$

$M\text{-}Formula[\![monsym\, formula]\!] = (\![\; monsym(M\text{-}Formula[\![formula]\!])\;]\!)$

where

$monsym \in \{+, -, \texttt{INOT}, \texttt{ABS}, \texttt{SQRT}\}$

$M\text{-}Formula[\![formula_1\; dysym\; formula_2]\!] =$
$(\![\; (M\text{-}Formula[\![formula_1]\!]\; dysym\; M\text{-}Formula[\![formula_2]\!])\;]\!)$

where

$dysym \in \{+, -, \texttt{*}, \texttt{\%}, \texttt{IDIV}, \texttt{SL}, \texttt{SR}, \texttt{IAND}, \texttt{IOR}, \texttt{MOD}, \texttt{ILOG}, \texttt{IEXP}\}$

$M\text{-}Formula[\![\; \texttt{IF}\; boolformula\; \texttt{THEN}\; formula_1\; \texttt{ELSE}\; formula_2\; \texttt{FI}\;]\!] =$

$(\![$**case** $M\text{-}Formula[\![boolformula]\!]$:

true $\rightarrow M\text{-}Formula[\![formula_1]\!]$

else $M\text{-}Formula[\![formula_2]\!]$

end$]\!)$

$M\text{-}Formula[\![signal]\!] = signal$

$M\text{-}Formula[\![int]\!] = int$

The boolean expressions again are simply translated directly into *EPA*, to be interpreted later in the usual way:

$M\text{-}Formula[\![formula_1\; binsym\; formula_2]\!] =$
$(\![\; (M\text{-}Formula[\![boolformula_1]\!]\; binsym\; M\text{-}Formula[\![boolformula_2]\!])\;]\!)$

where

$binsym \in \{ =, /=, <, >, <=, >= \}$

$M\text{-}Formula[\![boolformula_1 \;\wedge\; boolformula_2]\!] =$
$(\![\; (M\text{-}Formula[\![boolformula_1]\!] \;\wedge\; M\text{-}Formula[\![boolformula_2]\!])\;]\!)$

$M\text{-}Formula[\![boolformula_1\; \texttt{OR}\; boolformula_2]\!] =$
$(\![\; (M\text{-}Formula[\![boolformula_1]\!] \;\vee\; M\text{-}Formula[\![boolformula_2]\!])\;]\!)$

$M\text{-}Formula[\![\; \texttt{NOT}\; boolformula]\!] = (\![\; \neg\; (M\text{-}Formula[\![boolformula]\!])\;]\!)$

5.10 Closures and Declarations

Structure:

$$\begin{array}{|rcl|} \hline \textit{Closure} & ::= & \textit{declaration}_1 \ldots \textit{declaration}_{k+1} \\ \textit{Declaration} & ::= & \textit{typedecl} \mid \textit{fndecl} \\ \hline \end{array}$$

The *closure* consists of a non-empty sequence of well-formed declarations.

Static semantics rules:

$$\boxed{\text{wf-Closure}} \quad \frac{\emptyset \vdash_{Declaration^*} [\![d_1 \ \cdots \ d_{k+1}]\!] \text{ produces } \textit{st-env}}{\vdash_{Closure} [\![d_1 \ \cdots \ d_{k+1}]\!]}$$

$$\boxed{\text{wf-Declaration sequence}} \quad \frac{\begin{array}{ll} \text{(a)} & \textit{st-ev}_1 = \textit{st-env} \\ \text{(b)} & \forall j \in \{1, \ldots, k\text{-}1\} \bullet \textit{st-ev}_j \vdash_{Declaration} d_j \text{ produces } \textit{st-ev}_{j+1} \\ \text{(c)} & \textit{new-st-env} = \textit{st-ev}_{k+1} \end{array}}{\textit{st-env} \vdash_{Declaration^*} [\![d_1 \ \cdots \ d_k]\!] \text{ produces } \textit{new-st-env}}$$

$$\boxed{\text{wf-Declaration : FnDecl}} \quad \frac{\begin{array}{ll} \text{(a)} & \textit{fndecl} \in \textit{FnDecl} \\ \text{(b)} & \textit{st-env} \vdash_{FnDecl} \textit{fndecl} \text{ produces } \textit{new-st-env} \end{array}}{\textit{st-env} \vdash_{Declaration} \textit{fndecl} \text{ produces } \textit{new-st-env}}$$

$$\boxed{\text{wf-Declaration : TypeDecl}} \quad \frac{\begin{array}{ll} \text{(a)} & \textit{tydecl} \in \textit{TypeDecl} \\ \text{(b)} & \textit{st-env} \vdash_{TypeDecl} \textit{tydecl} \text{ produces } \textit{new-st-env} \end{array}}{\textit{st-env} \vdash_{Declaration} \textit{tydecl} \text{ produces } \textit{new-st-env}}$$

A closure is well-formed precisely when:

- Each of the declarations is well-formed, within the environment produced by processing earlier declarations.

The dynamic semantics for a closure returns a dynamic environment containing a sequence of *EPA* language definitions representing the ELLA type and function declarations in the closure. It first flattens the name-space of the ELLA-text using *declash* and then checks for well-formedness using the relation $\_ \vdash_{Declaration^*} \_$ produces $\_$ — the non-emptiness of this is a pre-condition for the dynamic semantics. The process terms are then found by applying *M-Declaration** to the flattened ELLA-text, using the static environment produced by the static semantics relation. It is assumed that the necessary instantiations of generic process definitions, representing atomic ELLA constructs, will be supplied when the process terms for the closure are interpreted.

Dynamic semantics equations:

$$M\text{-}Closure : Declaration^* \rightarrow Context$$

$$M\text{-}Closure[\![d_1 \ \cdots \ d_{k+1}]\!] = pd_1, \ \cdots \ , pd_i$$

where

$\text{elems}\,[pd_1, \ \cdots \ , pd_i] = \text{rng}\, dy\text{-}env$

$dy\text{-}env = M\text{-}Declaration^*[\![d'_1 \ \cdots \ d'_i]\!](init\text{-}env)$

$init\text{-}env = \langle st\text{-}env, \emptyset \rangle$

$\exists \ st\text{-}env \in StaticEnvironment \bullet$

$\quad \emptyset \vdash_{Declaration^*} [\![d'_1 \ \cdots \ d'_i]\!] \text{ produces } st\text{-}env$

$[\![d'_1 \ \cdots \ d'_i]\!] = declash[\![d_1 \ \cdots \ d_i]\!](\emptyset)$

$(i \in \{1, \ldots, k+1\}) \wedge (d_i \in FnDecl)$

$(\forall \ j \in \{i+1, \ldots, k+1\} \bullet d_j \notin FnDecl)$

$$M\text{-}Declaration^*[\![dcl_1, \ \cdots \ , dcl_n]\!](\rho, \sigma) = new\text{-}env$$

where

$dy\text{-}env_1 = \sigma$

$(\forall \ i \in \{1, \ldots, n\} \bullet dy\text{-}env_{i+1} = M\text{-}Declaration[\![dcl_i]\!](dy\text{-}env_i))$

$new\text{-}env = dy\text{-}env_{n+1}$

$$M\text{-}Declaration : Declaration \rightarrow DynamicEnvironment$$

$\forall \ fndecl \in Fndecl \bullet$

$\quad M\text{-}Declaration[\![fndecl]\!](\rho, \sigma) = M\text{-}Fndecl[\![fndecl]\!](\rho, \sigma)$

$\forall \ tydecl \in Typedecl \bullet$

$\quad M\text{-}Declaration[\![tydecl]\!](\rho, \sigma) = M\text{-}Typedecl[\![tydecl]\!](\rho, \sigma)$

Notice that the semantics ignores any type declarations appearing after the final function declaration.

5.10.1 The Semantics of ELLA: Trace Set Generation

We now show how the semantics of a deterministic ELLA system — the set of input/output traces — is obtained, using the ELLA-to-*EPA* translation defined by the preceeding semantic definitions, and the transition rule semantics for the resulting *EPA* terms, defined in §2.2.

First, define the relation $(\_ \vdash_{EPA\text{-}TR^*} \_ \xrightarrow{\_}^{+} \_) \subseteq Context \times ProcExpr \times Act^{+} \times ProcExpr$ as follows:

$$\sigma \vdash_{EPA\text{-}TR^*} P \xrightarrow{\alpha\text{-}seq}^{+} P' \Leftrightarrow$$

$$\forall\, i \in \{1, \ldots, k\} \bullet (\sigma, \{\,\}) \vdash_{EPA\text{-}TR} P_i \xrightarrow{\alpha_i}_C P_{i+1}$$

where

$$P_1 = P$$

$$P_{k+1} = P'$$

$$\alpha\text{-}seq = \alpha_1, \cdots, \alpha_k$$

$$\mathsf{len}\, \alpha\text{-}seq = k$$

It determines those transition sequences of length k given by the *EPA* transition rules, which start with process $P \in ProcExpr$, and finish with process $P' \in ProcExpr$, having as labels the sequence of proper actions $\alpha\text{-}seq = \alpha_1, \cdots, \alpha_k$.

The set of traces $traces(P)(\sigma)$ for a process term $P \in ProcExpr$ with respect to a context of *EPA* definitions σ is then given by the set of action sequences labelling a transition sequence for P:

$$traces(P)(\sigma) = \{\alpha\text{-}seq \mid (\sigma \vdash_{EPA\text{-}TR^*} P \xrightarrow{\alpha\text{-}seq}^{+} P') \wedge P' \in ProcExpr \wedge$$
$$k \in \mathbb{N} \wedge \alpha\text{-}seq = \alpha_1, \cdots, \alpha_k \wedge \forall\, i \in \{1, \ldots, k\} \bullet \alpha_i \in Act\}$$

If a sequence of ELLA definitions $ed_1, \cdots, ed_m$ compiles to the context $pd_1, \cdots, pd_n$, then the behaviour of the ELLA-text is given by:

$$ELLA\text{-}Semantics\colon Decl^{+} \rightarrow \mathbb{P}((ALL\text{-}VALUE \times ALL\text{-}VALUE)^*)$$

$$ELLA\text{-}Semantics[\![ed_1, \cdots, ed_m]\!] = val\text{-}tset$$

where

$$val\text{-}tset = remove\text{-}traceset\text{-}chan(lbl\text{-}tset)$$

$$lbl\text{-}tset = traces(pcall)(context)$$

$$pcall = make\text{-}pcall\text{-}of\text{-}pdefn(pd_n)(ich, och)$$

$$context = \mathsf{elems}\, pd_1, \cdots, pd_n \cup generic\text{-}defns$$

$$[pd_1, \cdots, pd_n] = M\text{-}Closure[\![ed_1, \cdots, ed_m]\!]$$

$$ich, och \in ChannelName \wedge ich \neq och$$

The set *generic-defns* is a predefined collection of all of the generic *EPA* definitions appearing in the dynamic semantics. Notice that the trace set *lbl-tset* depends on the names of (concrete) channels used, which label the input/output values in each action. These are removed from each trace by *remove-trace-set-chan* to return a set of traces containing input/output value pairs.

$remove\text{-}ioact\text{-}chan\colon IOAct \rightarrow ALL\text{-}VALUE$

$remove\text{-}ioact\text{-}chan(ch(v)) = v$

$remove\text{-}ioact\text{-}chan(ch_1(v_1),\ldots,ch_k(v_k)) = \langle v_1,\ldots,v_k\rangle$

$remove\text{-}act\text{-}chan\colon Act \rightarrow (ALL\text{-}VALUE \times ALL\text{-}VALUE)$

$remove\text{-}act\text{-}chan(ioact_1 \ / \ ioact_2) = \langle remove\text{-}ioact\text{-}chan(ioact_1), remove\text{-}ioact\text{-}chan(ioact_2)\rangle$

$remove\text{-}trace\text{-}chan\colon Act^* \rightarrow (ALL\text{-}VALUE \times ALL\text{-}VALUE)^*$

$remove\text{-}trace\text{-}chan(\alpha_1,\ldots,\alpha_k) = [io_1,\ldots,io_k]$

where

$\forall\ i \in \{1,\ldots,k\} \bullet io_i = remove\text{-}act\text{-}chan(\alpha_i)$

$remove\text{-}traceset\text{-}chan\colon\ \mathbb{P}(Act^*) \rightarrow\ \mathbb{P}((ALL\text{-}VALUE \times ALL\text{-}VALUE)^*)$

$remove\text{-}traceset\text{-}chan(\{tr_1,\ldots,tr_k\}) = \{iotr_1,\ldots,iotr_k\}$

where

$\forall\ i \in \{1,\ldots,k\} \bullet iotr_i = remove\text{-}trace\text{-}chan(tr_i)$

The operation $make\text{-}pcall\text{-}of\text{-}pdefn(pd)(chnm\text{-}seq)$ simply constructs a process call from the process definition pd, using the channel name sequence $chnm\text{-}seq$:

$make\text{-}pcall\text{-}of\text{-}pdefn(pd)(chnm\text{-}seq) = pnm(ich_1,\ \cdots\ ,ich_m \ / \ och_1,\ \cdots\ ,och_n)$

where

$pd = (\!|\ pnm(icdecl_1,\ \cdots\ ,icdecl_m \ / \ ocdecl_1,\ \cdots\ ,ocdecl_n) \ \underline{\triangle}\ rhs\ |\!)$

$chnm\text{-}seq = ich_1,\ \cdots\ ,ich_m, och_1,\ \cdots\ ,och_n$

$\forall\ j \in \{1,\ldots,m\} \bullet icdecl_j = (\!|\ ivar_j\colon ity_j\ |\!) \wedge ivar_j \in ChannelVarName$

$\forall\ j \in \{1,\ldots,n\} \bullet ocdecl_j = (\!|\ ovar_j\colon oty_j\ |\!) \wedge ovar_j \in ChannelVarName$

$\forall\ j \in \{1,\ldots,m\} \bullet ich_j = (\!|\ i_j\colon ity_j\ |\!) \wedge ich_j \in ChannelLiteral$

$\forall\ j \in \{1,\ldots,n\} \bullet och_j = (\!|\ o_j\colon oty_j\ |\!) \wedge och_j \in ChannelLiteral$

pre

$pd \in ProcDefn$

Note that process definitions representing ELLA functions will always have single inputs and outputs. Of course, the ELLA-text may not be well-formed, in which case the semantics is undefined.

6 Summary

This document has presented the formal semantics for the hardware description language ELLA. The synchronisation tree defining the behaviour of an ELLA expression is determined in a number of stages; for input deterministic systems, this is equivalent to determining the set of input/output traces, or input/output language. The *EPA* process algebra gives an intermediate-level semantic model for ELLA. *EPA* is similar to SCCS or (synchronous) Π-calculus. It offers simultaneous input/output via structured actions, which describe the value expressions associated with channel labels. Processes are composed using a specialised product operation, giving synchronous communication. Process definitions can be parameterised to effect value-passing and channel renaming.

The static semantics for ELLA is defined in terms of relationships between various syntactic data and a static environment representing the user defined ELLA functions and types that are currently available for use. It assumes that the name-space used in the ELLA-text is global, so that there are no variable redefinitions. The purpose of these static relations is to *restrict* the syntactic data to the well-formed structures, which in turn can be assigned a meaning via the dynamic semantics. These relations are defined via a collection of inference rules, whose overall organisation is determined by the composite syntax of Core ELLA. As a side-effect a static environment is produced which is used by the dynamic semantics to calculate the output types of expressions. The well-formedness of the ELLA-text is taken as a pre-condition to all ensuing semantic operations.

The dynamic semantics is defined in terms of inductively defined partial functions which associate well-formed phrases with *EPA* process terms as their denotations. The semantics of ELLA is taken as the synchronisation tree resulting from interpreting the process terms using the process transition rules, and applying the interpretation function for the ELLA action algebra. An equivalent set of input/output traces can be extracted from the synchronisation tree for input deterministic systems. A *normal form* is defined for those *EPA* processes which represent the behaviour of ELLA expressions.

Behavioural equivalence is naturally taken as *(strong) bisimulation equivalence*, which corresponds to equality of the IO languages for deterministic process terms. A tractable method is presented for determining the bisimulation equivalence of normal form *EPA* terms, using *state bisimulation* and *state evolution*.

An ELLA-to-*EPA* parser and compiler have been developed in Common Lisp by DRA (Malvern, England) and Harlequin Ltd. (Cambridge, England), respectively; they implement the static and dynamic semantics presented here, including normalisation and some simplification of the resulting process definitions. A ground simulator has also been implemented, where the *EPA* terms are translated in Lisp functions, which are then compiled using the standard (Lisp) compiler. The simulator forms the core of a *soft logic analysis* (SLA) tool.

Symbolic analysis and verification tools have been developed at Manchester. A symbolic normaliser produces *EPA* normal forms from the raw compiler output. A verification condition generator (VCG) generates a set of logical expressions required to establish design equivalence using state bisimulation and state evolution. The designer can also analyse signal flow and design structure using a symbolic simulation tool (SymSim). These tools are built on top of a sophisticated data structure manipulation package called GT, which supplies basic functionality for efficient term manipulation, employing term-sharing and *lazy substitution*.

The ELLA compilation and analysis tools are fully integrated and embeddded within an ELLA Design Environment, which offers the designer a uniform graphical interface to the tools, and which operates at the designer-level, i.e. much of the various underlying compilation operation is performed automatically as necessary, and any segment of the resulting *EPA* semantic representation which needs to be displayed to the designer (e.g. verification conditions) is presented as an ELLA expression whenever possible.

The techniques described in this report for providing formal semantics, and accompanying analysis tools, will be applied in future work to other computer languages. They will also be extended to model asynchronous systems.

7 References

[BGHT90] R Boulton, M Gordon, J Herbert, and J Van Tassel. The HOL Verification of ELLA Designs. Technical report, University of Cambridge Computer Laboratory, 1990.

[BGL+91] H. Barringer, G. Gough, T. Longshaw, B. Monahan, M. Peim, and A. Williams. Semantics and verification for Boolean Kernel ELLA using IO automata. In *Advanced Research Workshop on Correct Hardware Design Methodologies (CHARME'91)*, Turin, Italy, May 1991.

[BGL+92] H. Barringer, G.D. Gough, T. Longshaw, B.Q. Monahan, M. Peim, and A. Williams. A Semantics and Verification Framework for ELLA. Technical Report UMCS-92-4-6, University of Manchester, March 1992.

[BGMW92] H. Barringer, G. Gough, B. Monahan, and A. Williams. A semantics for core ella. Technical Report D2.3b, Formal Verification Support for Ella, IED project 4/1/1357, November 1992.

[BGMW94] H. Barringer, G. Gough, B. Monahan, and A. Williams. Verifying Deterministic Systems using a State Evolution Method. Technical report, University of Manchester, submitted to CAV '94, January 1994.

[Com91] Computer General Electronic Design, Chippenham, Wiltshire, United Kingdom. *The ELLA Language Reference Manual*, 5.1 edition, February 1991.

[Goo90a] K.G.W. Goossens. An Operational Semantics for a Subset of the CHDDL ELLA. Technical report, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, April 1990.

[Goo90b] K.G.W. Goossens. Semantics for picoELLA. Technical report, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, June 1990.

[HM92] M.G. Hill and J.D. Morison. Semantics and basic laws of ella. Technical Report D1.2, Formal Verification Support for Ella, IED project 4/1/1357, May 1992.

[MC93] J D Morison and A S Clarke. *ELLA2000: A language for Electronic System Design*. McGraw-Hill, 1993.

[MDED91] University of Manchester and RSRE Malvern DRA Electronics Division. Semantics and Verification Foundations for ELLA. Technical Report IED 4/1/1357 Deliverable D1.1+D2.1, May 1991.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mon94] B Monahan. Generic Term Structures — a Meta-Logical Programmer Interface. Technical report, Department of Computer Science, University of Manchester, April 1994.

[MPT85] J.D. Morison, N.E. Peeling, and T.L. Thorp. The design rationale of Ella, a hardware design and description language. In *CHDL*, Tokyo, 1985.

[Pau87] L.C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.

[Qui40] W.B. Quine. *Mathematical Logic*. Harper and Row, 1940.

[Rya93] P Y A Ryan. Simulators vs. semantics. Technical Report Memo, DRA, 1993.

[Sch86] D.A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[Sto77] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[Win84] Glynn Winskel. Synchronisation trees. *TCS*, 34:33 – 82, 1984.

¡

A Core ELLA : Composite Syntax

$$
\begin{array}{rcl}
\mathit{Closure} & ::= & \mathit{declaration}_1 \ldots \mathit{declaration}_{k+1} \\
\mathit{Declaration} & ::= & \mathit{typedecl} \mid \mathit{fndecl}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{TypeDecl} & ::= & \mathit{typenaming} \mid \mathit{typeformer} \\
\mathit{TypeNaming} & ::= & \texttt{TYPE}\ \mathit{typename} = \mathit{type} \\
\mathit{TypeFormer} & ::= & \texttt{TYPE}\ \mathit{typename} =\ \texttt{NEW}\ \mathit{newtype}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Type} & ::= & \mathit{typename} \mid \mathit{tupletype} \mid \mathit{powertype} \mid \mathit{stringtype} \mid \mathit{typevoid} \\
\mathit{TupleType} & ::= & (\mathit{type}_1, \ldots, \mathit{type}_{k+1}) \\
\mathit{PowerType} & ::= & [\mathit{nat}]\mathit{type} \\
\mathit{StringType} & ::= & \texttt{STRING}\ [\mathit{nat}]\mathit{typename} \\
\mathit{TypeVoid} & ::= & (\ ) \\
 & & \\
\mathit{NewType} & ::= & \mathit{enumtype} \mid \mathit{assoctype} \mid \mathit{inttype} \mid \mathit{chartype} \\
\mathit{AssocType} & ::= & (\mathit{tagtype}_1 \underline{|} \ldots \underline{|} \mathit{tagtype}_{k+1}) \\
\mathit{IntType} & ::= & \mathit{tag}/(\mathit{int}_1\ \underline{..}\ \mathit{int}_2) \\
\mathit{CharType} & ::= & \mathit{tag}(\ \texttt{'}\mathit{char}_1 \underline{|} \ldots \underline{|}\ \texttt{'}\mathit{char}_{k+1}) \\
\mathit{TagType} & ::= & \mathit{tag} \mid \mathit{tag}\ \&\ \mathit{type}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{FnDecl} & ::= & \mathit{unitfnd} \mid \mathit{primfnd} \mid \mathit{ramfnd} \mid \mathit{reformfnd} \mid \mathit{biopfnd} \\
\mathit{UnitFnD} & ::= & \texttt{FN}\ \mathit{fnname} = (\mathit{signaldecl}_1,\ \cdots\ , \mathit{signaldecl}_{k+1}) \rightarrow \mathit{type} : \mathit{unit}. \\
\mathit{PrimFnD} & ::= & \texttt{FN}\ \mathit{fnname} = (\mathit{type}_1) \rightarrow \mathit{type} : \mathit{primitive}. \\
\mathit{RamFnD} & ::= & \texttt{FN}\ \mathit{fnname} = (\mathit{type}_1, \ldots, \mathit{type}_4) \rightarrow \mathit{type} :\ \texttt{RAM}\ (\mathit{unit}). \\
\mathit{ReformFnD} & ::= & \texttt{FN}\ \mathit{fnname} = (\mathit{type}_1, \ldots, \mathit{type}_{k+1}) \rightarrow \mathit{type} :\ \texttt{REFORM}\ . \\
\mathit{BiopFnD} & ::= & \texttt{FN}\ \mathit{fnname} = (\mathit{type}_1, \ldots, \mathit{type}_{k+1}) \rightarrow \mathit{type} :\ \texttt{BIOP}\ \mathit{biopname}. \\
\mathit{ArithFnD} & ::= & \texttt{FN}\ \mathit{fnname} = (\mathit{signaldecl}_1, \ldots, \mathit{signaldecl}_k) \rightarrow \mathit{type} :\ \texttt{ARITH}\ \mathit{formula}. \\
\mathit{SignalDecl} & ::= & \mathit{type} : \mathit{signal} \\
 & & \\
\mathit{Unit} & ::= & \mathit{enumerated} \mid \mathit{signal} \mid \mathit{assoc} \mid \mathit{unitstr} \mid \mathit{replicate} \\
 & \mid & \mathit{query} \mid \mathit{index} \mid \mathit{dyindex} \mid \mathit{replace} \mid \mathit{trim} \\
 & \mid & \mathit{conc} \mid \mathit{sequence} \mid \mathit{extract} \mid \mathit{instance} \mid \mathit{case} \mid \mathit{block} \\
 & \mid & \mathit{unitvoid}
\end{array}
$$

Unitvoid	$::=$	( )
Assoc	$::=$	*enum* & *unit*
Unitstr	$::=$	STRING [*nat*]*unit*
Replicate	$::=$	[*nat*] *unit*
Query	$::=$	? *type*
Index	$::=$	*unit* [*nat*]
Dyindex	$::=$	$unit_1$ [[$unit_2$]]
Replace	$::=$	REPLACE $(unit_1, unit_2, unit_3)$
Trim	$::=$	*unit* [$nat_1 \ \underline{..}\ nat_2$]
Conc	$::=$	$unit_1$ CONC $unit_2$
Sequence	$::=$	$(unit_1\ ,\ \ldots\ ,\ unit_{k+1})$
Extract	$::=$	*unit* // *enum*
Signal	$::=$	*name*
Instance	$::=$	*fnname sequence*
Block	$::=$	BEGIN $step_1 \ldots step_{k+1}$ OUTPUT *unit* END

Case	$::=$	CASE $unit_1$ OF $caseitem_1\ ,\ \ldots\ ,\ caseitem_{k+1}$ ELSE $unit_2$ ESAC
Caseitem	$::=$	*constset* : *unit*
Step	$::=$	*declaration* | *let* | *make* | *join*
Let	$::=$	LET *signal* = *unit*.
Make	$::=$	MAKE *fnname* : *signal*.
Join	$::=$	JOIN *unit* $\rightarrow$ *signal*.

ConstSet	$::=$	*enumerated* | *constsetalt* | *constsetstring*
	|	*constsetrow* | *constsetassoc* | *constsettype*
ConstSetAlt	$::=$	$(constset_1\ \underline{\vert}\ \ldots\ \underline{\vert}\ constset_{k+1})$
ConstSetString	$::=$	STRING [*nat*]*constset*
ConstSetRow	$::=$	$(constset_1\ ,\ \ldots\ ,\ constset_{k+1})$
ConstSetAssoc	$::=$	*enum* & *constset*
ConstSetType	$::=$	*typename*

Enumerated	$::=$	*tag* | *enumint* | *enumchar* | *stringenum*
EnumInt	$::=$	*tag*/*int*
EnumChar	$::=$	*tag* ' *char*
StringEnum	$::=$	*tag* " $char_1\ \ldots\ char_{k+1}$"

$$
\begin{array}{rcl}
\mathit{Primitive} & ::= & \mathit{adelay} \mid \mathit{idelay} \mid \mathit{sample} \\
\mathit{Adelay} & ::= & \texttt{DELAY}\ (\mathit{unit}_1, \mathit{nat}_1, \mathit{unit}_2, \mathit{nat}_2) \\
\mathit{Idelay} & ::= & \texttt{IDELAY}\ (\mathit{unit}, \mathit{nat}) \\
\mathit{Sample} & ::= & \texttt{SAMPLE}\ (\mathit{nat}_1, \mathit{unit}, \mathit{nat}_2)
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Formula} & = & \mathit{unop} \mid \mathit{binop} \mid \mathit{ifop} \mid \mathit{tupop} \mid \mathit{signal} \mid \mathit{int} \\
\mathit{UnOp} & ::= & \mathit{unsym}\ \mathit{formula} \\
\mathit{BinOp} & ::= & \mathit{formula}_1\ \mathit{binsym}\ \mathit{formula}_2 \\
\mathit{IfOp} & ::= & \texttt{IF}\ \mathit{boolexpr}\ \texttt{THEN}\ \mathit{formula}_1\ \texttt{ELSE}\ \mathit{formula}_2\ \texttt{FI} \\
\mathit{UnSym} & ::= & + \mid - \mid \texttt{INOT} \mid \texttt{ABS} \mid \texttt{SQRT} \\
\mathit{BinSym} & ::= & + \mid - \mid \texttt{*} \mid \texttt{\%} \mid \texttt{IDIV} \mid \texttt{SL} \mid \texttt{SR} \mid \texttt{IAND} \mid \texttt{IOR} \mid \texttt{MOD} \mid \texttt{ILOG} \mid \texttt{IEXP}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{BoolExpr} & = & \mathit{intlogpred} \mid \mathit{unlogpred} \mid \mathit{binlogpred} \\
\mathit{IntLogPred} & ::= & \mathit{formula}_1\ \mathit{boolop}\ \mathit{formula}_2 \\
\mathit{UnLogPred} & ::= & \texttt{NOT}\ \mathit{boolexpr} \\
\mathit{BinLogPred} & ::= & \mathit{boolexpr}_1\ \mathit{logop}\ \mathit{boolexpr}_2 \\
\mathit{BoolOp} & ::= & \texttt{=} \mid \texttt{/=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \\
\mathit{LogOp} & ::= & \texttt{AND} \mid \texttt{OR}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{DeclInfo} & ::= & \mathit{typedecl} \mid \mathit{funtype} \mid \mathit{vartype} \\
\mathit{FunType} & ::= & \texttt{funtype}\ \mathit{type}_1 \rightarrow\ \mathit{type}_2 \\
\mathit{SigType} & ::= & \mathit{lettype} \mid \mathit{maketype} \\
\mathit{LetType} & ::= & \texttt{let-type}\ \mathit{type} \\
\mathit{MakeType} & ::= & \texttt{make-type}\ \mathit{type}_1\ \mathit{type}_2
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Nat} & = & \mathbb{N} \\
\mathit{Int} & = & \mathbb{Z} \\
\mathit{Tag} & = & \mathit{Name} \\
\mathit{Typename} & = & \mathit{Name} \\
\mathit{Fnname} & = & \mathit{Name} \\
\mathit{Biopname} & = & \mathit{Name}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Name} & = & \textbf{LEXICAL ITEM} \\
\mathit{Char} & = & \textbf{LEXICAL ITEM}
\end{array}
$$

B *EPA* Term Language

In this section, we complete the documentation of the term languages which are involved in formulating the *result* of compiling Core ELLA.

B.1 General Type Language

A *general type* resembles the Core ELLA types in many respects:

$$\begin{array}{rcl} te \in TypeExpr & = & TypeName \mid Tuple \mid Power \mid TagTypeExpr \mid UnionType \mid AtomicType \\ Tuple & ::= & te_1 \times \ldots \times te_k \\ Power & ::= & te \uparrow nat \\ TagTypeExpr & ::= & \mathsf{tag}(tag\text{-}nm, te) \\ UnionType & ::= & te_1 \cup \ldots \cup te_{k+1} \end{array}$$

$$\begin{array}{rcl} AtomicType & = & All \mid UnitType \mid EnumType \mid IntRange \mid CharRange \\ All & ::= & \mathsf{all} \\ UnitType & ::= & \mathsf{unit} \\ EnumType & ::= & \{tag\text{-}nm_1, \cdots, tagnm_{k+1}\} \\ IntRange & ::= & \mathsf{int}\ nat_1\ nat_2 \mid \mathsf{integers} \\ CharRange & ::= & \{chr_1, \cdots chr_{k+1}\} \mid \mathsf{characters} \end{array}$$

The main deviation from the ELLA type system is the inclusion of the type all. This a type expression denoting the collection of all denotable values in Core ELLA, *ALL-VALUE*. The type unit may be identified with the empty tuple type given by tuple - this denotes a set containing a single point (i.e. the `VOID` type in ELLA).

B.2 General Set Language

A *general set expression* stands for *typed* subsets of those sets given by types. These set expressions take the following form:

$$\begin{array}{rcl} se \in SetExpr & ::= & snm \mid te \\ & \mid & \{ve_1, \cdots, ve_k\} \\ & \mid & \{ve \mid vdfn_1, \cdots, vdfn_k \bullet le\} \\ & \mid & se_1 \times \cdots \times se_{k+1} \\ & \mid & se_1 \cup \cdots \cup se_{k+1} \\ & \mid & se_1 \cap \cdots \cap se_{k+2} \\ & \mid & se_1 - se_2 \end{array}$$

B.2.1 Set Definitions

A *general set definition* takes the form:

$$SetDefn ::= \textsf{set}\ snm \subseteq te \triangleq se$$

B.3 General Logic Language

The logic language consists of the following *quantifier-free* forms:

$$\begin{array}{rcl}
le \in LogicExpr & = & AtomicLogicExpr \mid NegLogicExpr \mid BinaryLogicExpr \\
AtomicLogicExpr & ::= & \textsf{true} \mid \textsf{false} \mid pred\text{-}nm(ve_1, \cdots, ve_{n+1}) \\
& \mid & (ve_1 = ve_2) \mid (ve_1 < ve_2) \mid (ve_1 > ve_2) \mid (ve_1 \leq ve_2) \\
& \mid & (ve_1 \geq ve_2) \mid (ve \in se) \\
NegLogicExpr & ::= & \neg\ le \\
BinaryLogicExpr & ::= & le_1 \wedge le_2 \\
& \mid & le_1 \vee le_2 \\
& \mid & le_1 \Rightarrow le_2 \\
& \mid & le_1 \Leftrightarrow le_2
\end{array}$$

B.3.1 General Predicate Definitions

A *general predicate definition* takes the form:

$$\begin{array}{rcl}
PredDefn & ::= & \textsf{predicate}\ pred\text{-}nm(vdfn_1, \cdots vdfn_{n+1}) \Leftrightarrow le \\
vdfn \in VarDefn & ::= & vnm : tynm
\end{array}$$

B.4 General Data Expressions

General data expressions also have the following form:

$$\begin{array}{rcl}
ve \in DataExpr & = & Literals \mid NamedDataObj \mid Application \\
& \mid & TagExpr \mid CaseExpr \mid Tuple \mid LetExpr \\
\\
Literals & = & CharLit \mid IntLit \mid EnumLit \\
CharLit & = & char \\
IntLit & ::= & int \\
EnumLit & ::= & tag\text{-}nm \\
\\
NamedDataObj & = & ConstName \mid Var \\
\\
Application & ::= & fun\text{-}name(\ value\text{-}args\ ) \mid biop\text{-}appl
\end{array}$$

$$
\begin{array}{rcl}
\mathit{TagExpr} & = & \mathit{MakeTagExpr} \mid \mathit{TagOfExpr} \mid \mathit{ValueOfExpr} \\
\mathit{MakeTagExpr} & ::= & \mathsf{tag}(\mathit{tag\text{-}nm}, \mathit{ve}) \\
\mathit{TagOfExpr} & ::= & \mathsf{tag\text{-}of}(\mathit{ve}) \\
\mathit{ValueOfExpr} & ::= & \mathsf{value\text{-}of}(\mathit{ve})
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{CaseExpr} & ::= & \mathsf{case}\ \mathit{ve}_1 : \mathit{alternatives}\ \mathsf{else}\ \mathit{ve}_2 \\
\mathit{Alternatives} & ::= & \mathit{alt}_1, \cdots, \mathit{alt}_{k+1} \\
\mathit{Alt} & ::= & \mathit{match} \rightarrow \mathit{ve} \\
\mathit{Match} & ::= & \mathit{se} \\
\mathit{Tuple} & ::= & \mathsf{tuple}(\mathit{ve}_1, \cdots, \mathit{ve}_{k+1}) \\
\mathit{LetExpr} & ::= & \mathsf{let}\ \mathit{var}\colon\ \mathit{te}\ = \mathit{ve}_1\ \mathsf{in}\ \mathit{ve}_2
\end{array}
$$

The BIOP function calls have additional generic parameters, such as types and numeric constants. In other words, these functions have already been schematically defined using such parameters:

$$
\mathit{BIOPAppl}\ ::=\ \mathsf{biop}\ \mathit{biop\text{-}name}\ \mathit{generic\text{-}biop\text{-}arguments}\ (\ \mathit{value\text{-}args}\ )
$$

$$
\begin{array}{rcl}
\mathit{GenericBIOPArguments} & ::= & [\mathit{biop\text{-}arg}_1, \cdots, \mathit{biop\text{-}arg}_n] \\
\mathit{BIOPArg} & ::= & \mathit{int} \mid \mathit{tynm}
\end{array}
$$

B.4.1 Data Function and Constant Definitions

A *data function definition* takes the form:

$$
\begin{array}{rcl}
\mathit{DataFunDefn} & ::= & \mathsf{function}\ \mathit{fnabs}(\mathit{vdfn}_1, \cdots\ \mathit{vdfn}_{n+1}) : \mathit{tynm} \underline{\triangle}\ \mathit{ve} \\
\mathit{fnabs} \in \mathit{FunAbstract} & = & \mathit{FunName} \mid \mathit{GenericFunAbstract} \\
\mathit{GenericFunAbstract} & ::= & \mathit{fnm}_{[v_1, \cdots, v_k]}
\end{array}
$$

As with process definitions, a function definition can be *generic*; the function abstraction *fnabs* can either simply be a function name $\mathit{fnm} \in \mathit{FunName}$, or it can be a structured *generic function abstract*, containing a function name and a list of argument $v_1, \cdots, v_k$, which are supplied when a call to the generic function is made.

A *data constant definition* takes the form:

$$
\mathit{DataConstDefn}\ ::=\ \mathsf{constant}\ \mathit{const\text{-}nm} : \mathit{tynm} \underline{\triangle}\ \mathit{ve}
$$

B.5 Names and identifiers

A *name* is essentially a textual representation of an identifier for some given kind of object. An *identifier* then provides a (sufficiently) unique way of referring to a defined entity, whereas names could possibly be dereferenced to various identifiers, depending upon the context and the kind of object that is being referred to. For example, it is possible that the same name may refer to a type and also to a function. Confusion of intention does not arise since these entities can only be syntactically referred to in certain ways within the syntax. Accordingly, any suitably declared name can always be dereferenced to the unique identifier for the entity in question.

The class of name objects is as follows:

tag-name : used to label components. Can appear in both *type* and *data* expressions.

tynm $\in$ *type-name* **:** used to name defined types.

snm $\in$ *set-name* **:** used to name defined sets.

prednm $\in$ *pred-name* **:** used to name defined predicates.

vnm $\in$ *var-name* **:** used as a (typed) variable, ranging over data values.

fnm $\in$ *fun-name* **:** used to name defined functions.

biop-name **:** used to name a specific BIOP operator.

const-nm $\in$ *const-name* **:** used to name a defined constant value.

pnm $\in$ *proc-name* **:** used to name a defined process.

cv $\in$ *channel-var-name* **:** used as a (typed) variable, ranging over channels.

chnm $\in$ *channel-name* **:** used as a literal name for a channel (i.e. `/dev/ttyp2` in UNIX)

B.6 Semantic Structures for ELLA Action Algebra

In this section, we define some supporting semantic structures needed to present the semantic definition of the ELLA action algebra.

B.6.1 Assumed structures

We shall assume the following sets as being given:

$\mathbb{B}$	The set of booleans.
$\mathbb{Z}$	The set of integers.
$\mathbb{N}$	The set of natural numbers, including zero.
Char	Some finite set of character codes, such as the ASCII set.
Name	A countable collection of identifier symbols.

Standard arithmetical operators acting upon $\mathbb{B}, \mathbb{Z}, \mathbb{N}$ can be assumed.

B.6.2 Core ELLA type semantics

Type structures in ELLA take the following general forms:

Associated types : The associated types provide a discriminated *union* of types, enabling type structures to be 'collected' together (cf logical disjunction). Values of such types are *associated* with a unique name (or *tag*) so that distinctions can be drawn based upon which component the value came from.

Enumeration types : The enumeration types are a primitive form of type and which are essentially a special case of an associated type — where there is no other value associated with the tag, other than the tag itself. Such types are implicitly ordered by the sequence that tags are given within the definition.

Fixed size rows, strings and homogeneous vectors : Row and vector types provide a *product* of types, enabling type structures to be 'bundled' together (cf. logical conjunction). Each row or vector value consists of a sequence of values, the type of which is determined by ordering. For the purpose of providing a type semantics, strings are essentially rows of characters[8].

Integer subrange and character subset types : ELLA types may also be (labelled) integer or character ranges, each specifying a finite subset of either the set of 32-bit integers or the set of available characters.

In the following section, a mathematical model of a suitably large universe of types is given.

B.6.3 Mathematical definition of ELLA type universe

We give an ab-initio description of a semantic structure used to give mathematical meaning to Core ELLA type structures. This structure is defined to be the least family of sets satisfying the equations given below. These equations define a number of *structured* sets, each of which contribute various kinds of type structure to the type universe. Note that this type structure is identical to that for the IO automaton semantics[BGMW92].

As it stands, the definition for *GEN-TYPE* given below is slightly more general than is needed to describe the meaning of Core ELLA types. But certainly, every set that can be specified by a well-formed definition within Core ELLA can naturally be regarded as an element of *GEN-TYPE*. Accordingly, this universe provides an adequate semantic basis for Core ELLA types.

1. $\mathit{GEN\text{-}TYPE} =$

   $(\mathit{ROW} \cup \mathit{TAG\text{-}TYPE} \cup \mathit{GEN\text{-}UNION} \cup \mathit{ENUM} \cup \mathit{VOIDTYPE}) - \{\emptyset\}$

   $\mathit{GEN\text{-}VALUE} = \bigcup \mathit{GEN\text{-}TYPE}$

   $\mathit{ALL\text{-}TYPE} = \mathit{GEN\text{-}TYPE} \cup \mathit{ATOMIC}$

   $\mathit{ALL\text{-}VALUE} = \bigcup \mathit{ALL\text{-}TYPE}$

   The universe of (non-empty) ELLA types *GEN-TYPE* is built from the union of four main subsets consisting of a set of finite row types, given by *ROW*, a set of tagged types, given by *TAG-TYPE*, a

[8] The behaviour of ELLA strings with respect to '?' is determined by the dynamic semantics.

collection of tagged union types, given by *GEN-UNION*, a collection of enumerated types, given by *ENUM* and finally the void type, *VOIDTYPE*. The set *GEN-VALUE* consists of all the values that populate the types belonging to *GEN-TYPE*, and so we have that $\mathit{GEN\text{-}TYPE} \subseteq \mathbb{P}(\mathit{GEN\text{-}VALUE})$. The set *GEN-VALUE* is one sort of the specialisation algebra for *EPA*.

The set of all non-empty *extended* types is defined to be *ALL-TYPE* and consists of the *GEN-TYPE* family, together with the (untagged) atomic types, *ATOMIC* which contains finite, compact integer intervals, of character subsets and sets of all bounded length strings. The set *ALL-VALUE* simply consists of all values that populate the types belonging to *ALL-TYPE*. Since $\mathit{GEN\text{-}TYPE} \subseteq \mathit{ALL\text{-}TYPE}$, we must have that $\mathit{GEN\text{-}VALUE} \subseteq \mathit{ALL\text{-}VALUE}$.

2. $\mathit{ROW} = \{\mathit{tuple\text{-}product}(\mathit{ty\text{-}seq}) \in \mathit{GEN\text{-}TYPE} \mid \mathit{ty\text{-}seq} \in \mathit{GEN\text{-}TYPE}^+\}$

   Every finite sequence of general types can be put alongside each other (i.e. by Cartesian product) to form a collection of rows, which is also a general type. The function *tuple-product* is defined by:-

   $$\mathit{tuple\text{-}product}\colon \mathit{GEN\text{-}TYPE}^+ \to \mathit{GEN\text{-}TYPE}$$
   $$\mathit{tuple\text{-}product}(\langle ty_1, \cdots, ty_{k+1}\rangle) =$$
   $$\{\langle v_1, \cdots, v_{k+1}\rangle \mid \forall\, i \in \{1, \ldots, k+1\} \bullet\ v_i \in ty_i\}$$

3. $\mathit{TAG\text{-}TYPE} = \mathit{label\text{-}type}(\lvert \mathit{Tag} \times \mathit{ALL\text{-}TYPE} \rvert)$

   Every general extended type can be 'labelled' by an arbitrary tag name (i.e. every value in type *gty* can be paired with tag name *tag*). The function *label-type* and associated functions are specified by:-

   $$\mathit{Tag} = \mathit{Name}$$

   $$\mathit{label\text{-}type} : \mathit{Tag} \times \mathit{ALL\text{-}TYPE} \to \mathit{GEN\text{-}TYPE}$$
   $$\mathit{label\text{-}type}(tag, aty) = \mathit{label\text{-}value}(\lvert \{tag\} \times aty \rvert)$$

   $$\mathit{TAG\text{-}VALUE} ::= \ \mathtt{tagval}\ \ tag\ \mathit{all\text{-}value}$$

   $$\mathit{label\text{-}value} : \mathit{Tag} \times \mathit{ALL\text{-}VALUE} \to \mathit{TAG\text{-}VALUE}$$
   $$\mathit{label\text{-}value}(tag, v) = [\![\ \mathtt{tagval}\ \ tag\ v]\!]$$

$$extract\text{-}tag : TAG\text{-}VALUE \rightarrow Tag$$
$$extract\text{-}tag[\![\, \texttt{tagval}\ \ tag\ v]\!] = tag$$

$$extract\text{-}value : TAG\text{-}VALUE \rightarrow ALL\text{-}VALUE$$
$$extract\text{-}value[\![\, \texttt{tagval}\ \ tag\ v]\!] = v$$

$$extract\text{-}label : TAG\text{-}TYPE \rightarrow Tag$$
$$extract\text{-}label(tag\text{-}ty) = \iota\ tag \in Tag \bullet tag \in tag\text{-}set$$
where
$$tag\text{-}set = extract\text{-}tag(\,|tag\text{-}ty|\,)$$

$$extract\text{-}type : TAG\text{-}TYPE \rightarrow \mathbb{P}(ALL\text{-}VALUE)$$
$$extract\text{-}type(tag\text{-}ty) = extract\text{-}value(\,|tag\text{-}ty|\,)$$

The above definition of *extract-label* relies upon the fact that every tagged type from *TAG-TYPE* is necessarily non-empty and that the function *extract-tag* is constant when applied to any tagged value from a tagged type.

4. $GEN\text{-}UNION =$

$$\begin{aligned}&\{\bigcup tag\text{-}type\text{-}set \in GEN\text{-}TYPE\\ &\mid\ tag\text{-}type\text{-}set \in \mathcal{F}(TAG\text{-}TYPE) \bullet\\ &\quad \mathsf{card}\, tag\text{-}type\text{-}set = \mathsf{card}\, extract\text{-}label(\,|tag\text{-}type\text{-}set|\,) \wedge\\ &\quad tag\text{-}type\text{-}set\ \neq\ \emptyset\\ &\}\end{aligned}$$

The union of a finite collection of disjointly tagged types is again a general type. Notice that only (non-empty) tagged types can participate in a union of types, reflecting the constraint on type unions in ELLA.

5. $ENUM = \mathcal{F}(Tag) - \{\emptyset\}$

Every non-empty, finite subset of *Name* is an atomic general type, each representing an ELLA enumerated types as a set of values. The ordering aspect of finite enumerated types is more conveniently considered elsewhere.

6. $VOIDTYPE = \{\{\ \texttt{unitvoid}\ \}\}$

The singleton set, $\{\ \texttt{unitvoid}\ \}$ in *GEN-TYPE* provides the denotation for the Core ELLA type, *typevoid*.

7. $ATOMIC = INT\text{-}RANGE \cup CHAR\text{-}SET \cup STRING \cup LOGIC$

The atomic types added within *ALL-TYPE* consist of compact integer intervals, character subsets and sets of bounded character strings, and logical values. These are individually specified as follows:

(a) $INT\text{-}RANGE = \{\ \{n,\ldots,m\} \in GEN\text{-}TYPE \mid n,m \in \mathbb{Z} \bullet n \leq m\}$
Every non-empty closed *subinterval* of the integers is an atomic general type.

(b) $CHAR\text{-}SET = \mathbb{P}(Char) - \{\emptyset\}$
Any non-empty subset of the character codes is an atomic general type.

(c) $STRING = \{\{str \in chars^{+} \mid \mathsf{len}\, str = n\} \in GEN\text{-}TYPE \mid$
$n \in \mathbb{N}, chars \in CHAR\text{-}SET \wedge n \geq 1$
$\}$
Every set of character strings of some specified length $n \geq 1$, is an atomic general type.

(d) $\text{LOGIC} = \{\mathtt{t},\mathtt{f}\}$
where $\mathtt{t},\mathtt{f}$ are symbols for the standard logical values.

The family of sets *GEN-TYPE* is now defined to be the least, sufficiently general family of sets satisfying the above equations. An iterative set-theoretical construction can be given that produces this least solution, without making any unnecessary identifications (e.g. the constant `unitvoid` is not identified with any other generated value).

B.7 Interpretation of ELLA Action Algebra

When the *EPA* transition rules are applied to an *EPA* process term, the action-, type-, and set-expressions are interpreted to give ground values, i.e., values from *ALL-VALUE* and *ALL-TYPE*. This section describes the interpretation function $M\text{-}⟦\ e\ ⟧(\sigma,\gamma)$ for an *EPA* expression *e*, relative to a *semantic environment* (σ,γ), consisting of the context σ of *EPA* definitions, and an assignment map γ, where values for *EPA* term variables are placed, but which is initially empty.

$$M\text{-}⟦\ \_\ ⟧(\sigma,\gamma) : GenTypeExpr \cup GenDataExpr \cup GenSetExpr \cup GenLogicExpr \rightarrow ALL\text{-}TYPE \cup ALL\text{-}VALUE$$

There is often a naturally strong similarity between the syntax of an *EPA* expression, and the syntax of its interpretation, using standard mathematical notation for set operations, and for values from *ALL-TYPE* and *ALL-VALUE* given above. The *EPA* syntax will therefore be surrounded with semantic brackets '⟦ _ ⟧'.

B.7.1 Type Expressions

The structure of *EPA* type (and set) expressions is very close to the usual mathematical notation, and the interpreted expressions will have their usual meaning. When a type name is encountered, then the corresponding type definition is found in the context σ, and the right-hand side of the definition is then interpreted to give the set of values required.

$$M\text{-}[\![\ tynm\ ]\!](\sigma,\gamma) = M\text{-}[\![\ te\ ]\!](\sigma,\gamma)$$

where

$$[\![\ \mathsf{type}\ tynm\ \underline{\triangle}\ te\ ]\!] \in \sigma$$

$$M\text{-}[\![\ te_1 \times \cdots \times te_k\ ]\!](\sigma,\gamma) = \mathit{tuple\text{-}product}(\langle M\text{-}[\![\ te_1\ ]\!](\sigma,\gamma), \cdots, M\text{-}[\![\ te_k\ ]\!](\sigma,\gamma)\rangle)$$

$$M\text{-}[\![\ te \uparrow n\ ]\!](\sigma,\gamma) = \mathit{tuple\text{-}product}(\langle ty_1, \cdots, ty_n\rangle)$$

where

$$M\text{-}[\![\ te\ ]\!](\sigma,\gamma) = ty$$

$$\forall\ i \in \{1,\ldots,n\} \bullet ty_i = ty$$

$$M\text{-}[\![\ \mathsf{tag}(tag\text{-}nm, te)\ ]\!](\sigma,\gamma) = \mathit{label\text{-}type}(tag\text{-}nm, M\text{-}[\![\ te\ ]\!](\sigma,\gamma))$$

$$M\text{-}[\![\ te_1 \cup \cdots \cup te_{k+1}\ ]\!](\sigma,\gamma) = M\text{-}[\![\ te_1\ ]\!](\sigma,\gamma) \cup \cdots \cup M\text{-}[\![\ te_{k+1}\ ]\!](\sigma,\gamma)$$

$$M\text{-}[\![\ \mathsf{all}\ ]\!](\sigma,\gamma) = \mathit{ALL\text{-}VALUE}$$

$$M\text{-}[\![\ \mathsf{unit}\ ]\!](\sigma,\gamma) = \{\ \mathtt{unitvoid}\ \}$$

$$M\text{-}[\![\ \{tag\text{-}nm_1, \cdots, tag\text{-}nm_{k+1}\}\ ]\!](\sigma,\gamma) = \{tag\text{-}nm_1, \cdots, tag\text{-}nm_{k+1}\}$$

$$M\text{-}[\![\ \mathsf{int}\ n_1\ n_2\ ]\!](\sigma,\gamma) = \{n \mid n \in \mathbb{N} \wedge n \geq n_1 \wedge n \leq n_2\}$$

$$M\text{-}[\![\ \{chr_1, \cdots chr_{k+1}\}\ ]\!](\sigma,\gamma) = \{chr_1, \cdots chr_{k+1}\}$$

Note that variables over type expressions, used for example with generic process and function definitions, will have been substituted by type expressions before evaluation.

B.7.2 Set Expressions

$M\text{-}[\![\ snm\ ]\!](\sigma,\gamma) = M\text{-}[\![\ se\ ]\!](\sigma,\gamma)$

where

$[\![\ \text{set}\ snm\ \underline{\triangle}\ se\ ]\!] \in \sigma$

$M\text{-}[\![\ \{ve_1,\ \cdots\ ,\ ve_k\}\ ]\!](\sigma,\gamma) = \{v_1,\ \cdots\ , v_k\}$

where

$\forall\ i \in \{1,\ldots,k\} \bullet v_i = M\text{-}[\![\ ve_i\ ]\!](\sigma,\gamma)$

$M\text{-}[\![\ \{ve \mid vdecl_1,\ \cdots\ ,\ vdecl_k \bullet le\}\ ]\!](\sigma,\gamma) =$

$\{M\text{-}[\![\ ve\ ]\!](\sigma,\gamma') \mid \forall\ i \in \{1,\ldots,k\} \bullet v'_i \in set_i \wedge M\text{-}[\![\ le\ ]\!](\sigma,\gamma'\}$

where

$\forall\ i \in \{1,\ldots,k\} \bullet [\![\ v_i : se_i\ ]\!] = vdecl_i \wedge set_i = M\text{-}[\![\ se_i\ ]\!](\sigma,\gamma)$

$\gamma' = \gamma \cup \{v_1 \mapsto v'_1,\ \cdots\ , v_k \mapsto v'_k\}$

$M\text{-}[\![\ se_1 \times\ \cdots\ \times se_{k+1}\ ]\!](\sigma,\gamma) = tuple\text{-}product(\langle set_1,\ \cdots\ , set_{k+1}\rangle)$

where

$\forall\ i \in \{1,\ldots,k+1\} \bullet set_i = M\text{-}[\![\ se_i\ ]\!](\sigma,\gamma)$

$M\text{-}[\![\ se_1 \cup\ \cdots\ \cup se_{k+1}\ ]\!](\sigma,\gamma) = set_1 \cup\ \cdots\ \cup set_{k+1}$

where

$\forall\ i \in \{1,\ldots,k+1\} \bullet set_i = M\text{-}[\![\ se_i\ ]\!](\sigma,\gamma)$

$M\text{-}[\![\ se_1 \cap\ \cdots\ \cap se_{k+2}\ ]\!](\sigma,\gamma) = set_1 \cap\ \cdots\ \cap set_{k+1}$

where

$set_1 = M\text{-}[\![\ se_1\ ]\!](\sigma,\gamma) \wedge set_2 = M\text{-}[\![\ se_2\ ]\!](\sigma,\gamma)$

$M\text{-}[\![\ se_1 — se_2\ ]\!](\sigma,\gamma) = set_1 - set_2$

where

$set_1 = M\text{-}[\![\ se_1\ ]\!](\sigma,\gamma) \wedge set_2 = M\text{-}[\![\ se_2\ ]\!](\sigma,\gamma)$

B.7.3 General Logic Language

Again, the structure of *EPA* logic expressions is very close to usual mathematical notation, and the meaning is intended to be the same.

$$
\begin{array}{rcl}
M\text{-}[\![\ \mathsf{true}\ ]\!](\sigma,\gamma) & = & \mathtt{t} \\
M\text{-}[\![\ \mathsf{false}\ ]\!](\sigma,\gamma) & = & \mathtt{f} \\
M\text{-}[\![\ (ve_1 = ve_2)\ ]\!](\sigma,\gamma) & = & (M\text{-}[\![\ ve_1\ ]\!](\sigma,\gamma) = M\text{-}[\![\ ve_2\ ]\!](\sigma,\gamma)) \\
M\text{-}[\![\ (ve_1 < ve_2)\ ]\!](\sigma,\gamma) & = & (M\text{-}[\![\ ve_1\ ]\!](\sigma,\gamma) < M\text{-}[\![\ ve_2\ ]\!](\sigma,\gamma)) \\
M\text{-}[\![\ (ve_1 > ve_2)\ ]\!](\sigma,\gamma) & = & (M\text{-}[\![\ ve_1\ ]\!](\sigma,\gamma) > M\text{-}[\![\ ve_2\ ]\!](\sigma,\gamma)) \\
M\text{-}[\![\ (ve_1 \leq ve_2)\ ]\!](\sigma,\gamma) & = & (M\text{-}[\![\ ve_1\ ]\!](\sigma,\gamma) \leq M\text{-}[\![\ ve_2\ ]\!](\sigma,\gamma)) \\
M\text{-}[\![\ (ve_1 \geq ve_2)\ ]\!](\sigma,\gamma) & = & (M\text{-}[\![\ ve_1\ ]\!](\sigma,\gamma) \geq M\text{-}[\![\ ve_2\ ]\!](\sigma,\gamma)) \\
M\text{-}[\![\ (ve \in se)\ ]\!](\sigma,\gamma) & = & M\text{-}[\![\ ve\ ]\!](\sigma,\gamma) \in M\text{-}[\![\ se\ ]\!](\sigma,\gamma) \\
M\text{-}[\![\ \neg\ le\ ]\!](\sigma,\gamma) & = & \neg\ M\text{-}[\![\ le\ ]\!](\sigma,\gamma) \\
M\text{-}[\![\ le_1\ \wedge\ le_2\ ]\!](\sigma,\gamma) & = & M\text{-}[\![\ le_1\ ]\!](\sigma,\gamma)\ \wedge\ M\text{-}[\![\ le_2\ ]\!](\sigma,\gamma) \\
M\text{-}[\![\ le_1\ \vee\ le_2\ ]\!](\sigma,\gamma) & = & M\text{-}[\![\ le_1\ ]\!](\sigma,\gamma)\ \vee\ M\text{-}[\![\ le_2\ ]\!](\sigma,\gamma) \\
M\text{-}[\![\ le_1\ \Rightarrow\ le_2\ ]\!](\sigma,\gamma) & = & M\text{-}[\![\ le_1\ ]\!](\sigma,\gamma)\ \Rightarrow\ M\text{-}[\![\ le_2\ ]\!](\sigma,\gamma) \\
M\text{-}[\![\ le_1\ \Leftrightarrow\ le_2\ ]\!](\sigma,\gamma) & = & M\text{-}[\![\ le_1\ ]\!](\sigma,\gamma)\ \Leftrightarrow\ M\text{-}[\![\ le_2\ ]\!](\sigma,\gamma)
\end{array}
$$

B.7.4 Value Expressions

The interpretation of *EPA* value expressions proceeds in a natural fashion. The result will be a value from the ELLA value universe, either directly, or by evaluation of a function application. If a variable is encountered, then its assigned value is looked up in the assignment map γ — it is assumed that every variable will have a unique assignment. The let value expression assigns a value to a typed variable, which is then used in the second value expression. On evaluation, the assignment map γ is used to hold the value assigned to the let variable. A function application will either involve an auxiliary function, or a built-in operation — the semantics of these are defined in the following sections – or will require an *EPA* function definition to be retrieved from the context σ. Note that for a generic function application, the generic function definition is first instantiated for the actual (generic) parameters, and then the (instantiated) call is interpreted as a simple function, with respect to a context augmented by the instantiated function definition.

$M\text{-}\llparenthesis\ literal\ \rrparenthesis(\sigma,\gamma) = literal$

$M\text{-}\llparenthesis\ fnm(ve_1,\ \cdots\ ,ve_k)\ \rrparenthesis(\sigma,\gamma) = M\text{-}\llparenthesis\ rhs\ \rrparenthesis(\sigma,\gamma')$

where

$\llparenthesis$ function $fnm\ (vdecl_1,\ \cdots\ ,vdecl_k) : te \mathrel{\underline{\triangle}} rhs\ \rrparenthesis \in \sigma$

$\forall\ i \in \{1,\ldots,k\} \bullet \llparenthesis\ v_i : te_i\ \rrparenthesis = vdecl_i \wedge val_i = M\text{-}\llparenthesis\ ve_i\ \rrparenthesis(\sigma,\gamma)$

$\gamma' = \gamma \dagger \{v_1 \mapsto val_1,\ \cdots\ ,v_k \mapsto val_k\}$

$M\text{-}\llparenthesis\ fnm_{[a_1,\ \cdots\ ,a_m]}(ve_1,\ \cdots\ ,ve_k)\ \rrparenthesis(\sigma,\gamma) = M\text{-}\llparenthesis\ fnm_{[a_1.\ \cdots\ ,a_m]}(ve_1,\ \cdots\ ,ve_k)\ \rrparenthesis(\sigma',\gamma)$

where

$fdefn = \llparenthesis$ function $fnm_{[fa_1,\ \cdots\ ,fa_m]}\ (vdecl_1,\ \cdots\ ,vdecl_k) : te \mathrel{\underline{\triangle}} rhs\ \rrparenthesis$

$fdefn \in \sigma$

$fdefn' = fdefn\,[\ a_1/fa_1,\ \cdots\ ,a_m/fa_m\ ]$

$\sigma' = \sigma \dagger \{fdefn'\}$

$M\text{-}\llparenthesis$ tag$(tag\text{-}nm, ve)\ \rrparenthesis(\sigma,\gamma) = label\text{-}value(tag\text{-}nm, M\text{-}\llparenthesis\ ve\ \rrparenthesis(\sigma,\gamma))$

$M\text{-}\llparenthesis$ tag-of$(ve)\ \rrparenthesis(\sigma,\gamma) = extract\text{-}tag(M\text{-}\llparenthesis\ ve\ \rrparenthesis(\sigma,\gamma))$

$M\text{-}\llparenthesis$ value-of$(ve)\ \rrparenthesis(\sigma,\gamma) = extract\text{-}value(M\text{-}\llparenthesis\ ve\ \rrparenthesis(\sigma,\gamma))$

$M\text{-}\llparenthesis\ \textbf{case}\ ve_{k+2} : alt_1,\ \cdots\ ,alt_{k+1}\ \textbf{else}\ ve_{k+3}\ \textbf{end}\ \rrparenthesis(\sigma,\gamma) =$

case $M\text{-}\llparenthesis\ ve_{k+2}\ \rrparenthesis(\sigma,\gamma)$ **in**

$M\text{-}\llparenthesis\ se_1\ \rrparenthesis(\sigma,\gamma) : M\text{-}\llparenthesis\ ve_1\ \rrparenthesis(\sigma,\gamma)$

. : .

. : .

$M\text{-}\llparenthesis\ se_{k+1}\ \rrparenthesis(\sigma,\gamma) : M\text{-}\llparenthesis\ ve_{k+1}\ \rrparenthesis(\sigma,\gamma)$

else $M\text{-}\llparenthesis\ ve_{k+3}\ \rrparenthesis(\sigma,\gamma)$

end

where

$\forall\ i \in \{1,\ldots,k+1\} \bullet [\![se_i \rightarrow ve_i]\!] = [\![alt_i]\!]$

$M\text{-}\llparenthesis$ tuple$(ve_1,\ \cdots\ ,ve_{k+1})\ \rrparenthesis(\sigma,\gamma) = \langle M\text{-}\llparenthesis\ ve_1\ \rrparenthesis(\sigma,\gamma),\ \cdots\ ,M\text{-}\llparenthesis\ ve_{k+1}\ \rrparenthesis(\sigma,\gamma)\rangle$

$M\text{-}\llparenthesis\ var\ \rrparenthesis(\sigma,\gamma) = \gamma(var)$

where

$var \in DataExprVar$

$M\text{-}\llparenthesis$ let var: $te\ = ve_1$ in $ve_2\ \rrparenthesis(\sigma,\gamma) = M\text{-}\llparenthesis\ ve_2\ \rrparenthesis(\sigma,\gamma')$

where

$\gamma' = \gamma \dagger \{var \mapsto M\text{-}\llparenthesis\ ve_1\ \rrparenthesis(\sigma,\gamma)\}$

B.8 Auxiliary Functions

The semantics for the auxiliary value function expressions used by the semantic operations is now defined. For the concatentation operation $\mathsf{concat}_{[ty1,ty2]}$, three different cases are given, both for sequence and string concatentation, so that coersion of elements to unit-length sequences/strings can be handled:

$$M\text{-}⟦\ \mathsf{subsequence}(lst,k)\ ⟧(\sigma,\gamma) =$$
$$\exists s_1, s_2 \in \mathit{GEN\text{-}VALUE}^*, v \in \mathit{GEN\text{-}VALUE}\ \cdot\ seq = s_1 \frown [v]^k \frown s_2$$
where
$$seq = M\text{-}⟦\ lst\ ⟧(\sigma,\gamma)$$

$$M\text{-}⟦\ \mathsf{copy}(expr,k)\ ⟧(\sigma,\gamma) = \langle v_1, \cdots, v_k \rangle$$
where
$$v = M\text{-}⟦\ expr\ ⟧(\sigma,\gamma)$$
$$\forall\ i \in \{1,\ldots,k\} \bullet v_i = v$$

$$M\text{-}⟦\ \mathsf{index}(expr,i)\ ⟧(\sigma,\gamma) = v(i)$$
where
$$v = M\text{-}⟦\ expr\ ⟧(\sigma,\gamma)$$

$$M\text{-}⟦\ \mathsf{repl}(expr,n,elt)\ ⟧(\sigma,\gamma) = \langle v(1), \cdots, v(n-1), x, v(n+1), \cdots, v(k) \rangle$$
where
$$v = M\text{-}⟦\ expr\ ⟧(\sigma,\gamma)$$
$$x = M\text{-}⟦\ elt\ ⟧(\sigma,\gamma)$$
$$\mathsf{len}\, v = k$$
$$1 \le n \le k$$

$$M\text{-}⟦\ \mathsf{trim}(expr,m,n)\ ⟧(\sigma,\gamma) = \langle v(m), \cdots, v(n) \rangle$$
where
$$v = M\text{-}⟦\ expr\ ⟧(\sigma,\gamma)$$
$$\mathsf{len}\, v = k$$
$$1 \le m \le n \le k$$

$$M\text{-}⟦\ \mathsf{reform\text{-}values}(expr,tyexpr)\ ⟧(\sigma,\gamma) = x$$
where
$$type = M\text{-}⟦\ tyexpr\ ⟧(\sigma,\gamma)$$
$$v = M\text{-}⟦\ expr\ ⟧(\sigma,\gamma)$$
$$\exists\ !x \in type \bullet \mathit{flatten\text{-}values}(x) = \mathit{flatten\text{-}values}(v)$$

$\forall\ v \in \mathit{ALL\text{-}VALUE} - \bigcup \mathit{ROW} \bullet$

$\quad \mathit{flatten\text{-}values}(v) = \langle v \rangle$

$\forall\ v \in \bigcup \mathit{ROW} \bullet$

$\quad \mathit{flatten\text{-}values}(v) = \mathit{flatten\text{-}values}(v_1) ^\frown \cdots ^\frown \mathit{flatten\text{-}values}(v_k)$

$\quad\quad$ where

$\quad\quad\quad\quad v = \langle v_1, \cdots, v_k \rangle$

$\forall\ ty \in \mathit{GEN\text{-}TYPE}, m, n \in \mathbb{N} \bullet$

$\quad TY1 = ty^m \ \wedge\ TY2 = ty^n \ \Rightarrow$

$\quad\quad M\text{-}[\![\ \mathsf{concat}_{[ty1,ty2]}(expr_1, expr_2)\ ]\!](\sigma, \gamma) = \langle a(1), \cdots, a(m), b(1), \cdots, b(n) \rangle$

$\quad\quad\quad$ where

$\quad\quad\quad\quad \mathsf{len}\, a = m \wedge \mathsf{len}\, b = n$

$\quad TY1 = ty^m \ \wedge\ TY2 = ty \ \Rightarrow$

$\quad\quad M\text{-}[\![\ \mathsf{concat}_{[ty1,ty2]}(expr_1, expr_2)\ ]\!](\sigma, \gamma) = \langle a(1), \cdots, a(m), b \rangle$

$\quad\quad\quad$ where

$\quad\quad\quad\quad \mathsf{len}\, a = m$

$\quad TY1 = ty \ \wedge\ TY2 = ty^n \ \Rightarrow$

$\quad\quad M\text{-}[\![\ \mathsf{concat}_{[ty1,ty2]}(expr_1, expr_2)\ ]\!](\sigma, \gamma) = \langle a, b(1), \cdots, b(n) \rangle$

$\quad\quad\quad$ where

$\quad\quad\quad\quad \mathsf{len}\, b = n$

$\quad$ where

$\quad\quad a = M\text{-}[\![\ expr_1\ ]\!](\sigma, \gamma)$

$\quad\quad b = M\text{-}[\![\ expr_2\ ]\!](\sigma, \gamma)$

$\quad\quad TY_1 = M\text{-}[\![\ ty_1\ ]\!](\sigma, \gamma)$

$\quad\quad TY_2 = M\text{-}[\![\ ty_2\ ]\!](\sigma, \gamma)$

$$
\begin{array}{l}
\forall\ cty \in \mathit{CHAR\text{-}SET}, tag \in \mathit{ENUM}, m, n \in \mathbb{N} \bullet \\
\quad TY1 = \mathit{label\text{-}type}(tag, cty^m)\ \wedge\ TY2 = \mathit{label\text{-}type}(tag, cty^n)\ \Rightarrow \\
\qquad M\text{-}[\![\ \mathsf{concat}_{[ty1,ty2]}(expr_1, expr_2)\ ]\!](\sigma,\gamma) = \mathit{label\text{-}value}(tag, \langle x(1), \cdots, x(m), y(1), \cdots, y(n)\rangle) \\
\qquad\quad \mathsf{where} \\
\qquad\qquad \mathsf{len}\, x = m \\
\qquad\qquad \mathsf{len}\, y = n \\
\quad TY1 = \mathit{label\text{-}type}(tag, cty^m)\ \wedge\ TY2 = \mathit{label\text{-}type}(tag, cty)\ \Rightarrow \\
\qquad M\text{-}[\![\ \mathsf{concat}_{[ty1,ty2]}(expr_1, expr_2)\ ]\!](\sigma,\gamma) = \mathit{label\text{-}value}(tag, \langle x(1), \cdots, x(m), y\rangle) \\
\qquad\quad \mathsf{where} \\
\qquad\qquad \mathsf{len}\, x = m \\
\quad TY1 = \mathit{label\text{-}type}(tag, cty)\ \wedge\ TY2 = \mathit{label\text{-}type}(tag, cty^n)\ \Rightarrow \\
\qquad M\text{-}[\![\ \mathsf{concat}_{[ty1,ty2]}(expr_1, expr_2)\ ]\!](\sigma,\gamma) = \mathit{label\text{-}value}(tag, \langle x, y(1), \cdots, y(n)\rangle) \\
\qquad\quad \mathsf{where} \\
\qquad\qquad \mathsf{len}\, y = n \\
\quad \mathsf{where} \\
\qquad \mathit{label\text{-}value}(tag, x) = M\text{-}[\![\ expr_1\ ]\!](\sigma,\gamma) \\
\qquad \mathit{label\text{-}value}(tag, y) = M\text{-}[\![\ expr_2\ ]\!](\sigma,\gamma) \\
\qquad TY_1 = M\text{-}[\![\ ty_1\ ]\!](\sigma,\gamma) \\
\qquad TY_2 = M\text{-}[\![\ ty_2\ ]\!](\sigma,\gamma) \\
\forall\ a, b \in ROW \bullet \\
\quad M\text{-}[\![\ a \frown b\ ]\!](\sigma,\gamma) = \langle a(1), \cdots, a(m), b(1), \cdots, b(n)\rangle \\
\qquad \mathsf{where} \\
\qquad\qquad \mathsf{len}\, a = m\ \wedge\ \mathsf{len}\, b = n
\end{array}
$$

B.9 ARITHs

`ARITH` operations in ELLA are performed over (untagged) integers, and include boolean operations which are translated in *EPA* logical expressions described above. The result of the `ARITH` expression is then coerced into the required output type *oty*, which is either a tagged integer or tag type, using $\mathsf{convert\text{-}types}_{[oty]}$:

$$M\text{-}[\![\ \textsf{convert-types}_{[oty]}(var, ve)\ ]\!](\sigma,\gamma) = \textbf{case } val \geq lb \wedge val \leq ub \textbf{ in}$$

$$\{\texttt{t}\} : label\text{-}value(tag, val)$$

$$\textbf{else } varval$$

$$\textbf{end}$$

where

$$oty = \textsf{tag}(tag, \textsf{int}\ lb\ ub)$$

$$val = M\text{-}[\![\ ve\ ]\!](\sigma,\gamma)$$

$$varval = M\text{-}[\![\ var\ ]\!](\sigma,\gamma)$$

$$M\text{-}[\![\ \textsf{convert-types}_{[oty]}(var, ve)\ ]\!](\sigma,\gamma) = \textbf{case } val \geq 1 \wedge val \leq k \textbf{ in}$$

$$\{\texttt{t}\} : \langle tag_1, \ldots, tag_k \rangle (val)$$

$$\textbf{else } varval$$

$$\textbf{end}$$

where

$$oty = [\![\ \{tag_1, \cdots, tag_k\}\ ]\!]$$

$$val = M\text{-}[\![\ ve\ ]\!](\sigma,\gamma)$$

$$varval = M\text{-}[\![\ var\ ]\!](\sigma,\gamma)$$

Most of the integer operations are standard and have their usual integer arithmetic meaning, always returning an integer value. The remainder perform operations on the bit-vector representation of their arguments, and then convert the resulting bit-vector back into an integer representation. Table 3 briefly describes each operation; more details can be found in [HM92, MC93].

Operator	Description
$+, -$, `ABS` , `SQRT`	standard (monadic) plus, minus, absolute and square root operators
$+, -$, `*` , `%` , `IDIV`	standard addition, subtraction, multiplication and division operators
`SL` , `SR` , `IAND` , `IOR` , `INOT`	operators on bit-vector representation: shift-left, shift-right, logical conjunction, disjunction and negation of individual bits
`MOD` , `ILOG` , `IEXP`	standard modulus, logarithm and exponentiation operators

Table 3: *EPA* Integer Operations for ELLA

Note that `%` and `IDIV` are synonyms; `SL` and `SR` provide multiplication and division by powers of two.

B.10 BIOPs

It is assumed that a built-in operator application is well-formed, i.e., that the input and output types conform to the type template associated with the operator. This is ensured by the well-formedness check for ELLA BIOP expressions in the static semantics.

Biops operate on integers, bit-vectors, and strings. They therefore require interface functions to translate input values into a standard form, and then to translate the standard operator output into the form required by the target output type. If an error occurs, then a special error-value is selected, which is supplied as an input to the operator — this will be a non-deterministically chosen value.

$$enum2bool_{[ty]} : Context \to ENUM \to LOGIC$$

$$enum2bool_{[ty]}(v)(\sigma) = \text{case } v \text{ in } elt_1 : \mathtt{t}\text{else } \mathtt{f} \text{ end}$$

where

$$\text{elems}\,\sigma \vdash_{EPA\text{-}TyEq} \llparenthesis\ \{elt_1, elt_2\}\ \rrparenthesis \equiv oty$$

$$bool2enum_{[elt_1,elt_2]} : Context \to LOGIC \to ENUM$$

$$bool2enum_{[ty]}(v)(\sigma) = :\ oty \mathbin{\underline{\triangle}} \text{case } v \text{ in } \mathtt{t} : elt_1\text{else } elt_2 \text{ end}$$

where

$$\text{elems}\,\sigma \vdash_{EPA\text{-}TyEq} \llparenthesis\ \{elt_1, elt_2\}\ \rrparenthesis \equiv ty$$

The functions *enum2bool* and *bool2enum* convert between the enumeration values elt_1, elt_2 which are matched with logical true and false values $\mathtt{t}, \mathtt{f}$ respectively. Other pairs of conversion functions will be required for bit- and integer-strings — refer to [HM92].

The semantics of each built-in operation simply assembles the appropriate conversion functions with a standard operator corresponding to the particular BIOP. For example, the *AND* built-in operator will be defined as follows, for enumerated types:

$$M\text{-}\llbracket\ \text{biop}\ AND\ [ity, oty](v, w)\ \rrbracket(\sigma, \gamma) = val$$

where

$$val = bool2enum_{[oty]}(res)(\sigma)$$

$$res = a \wedge b$$

$$a = enum2bool_{[ity]}(v(1))(\sigma)$$

$$b = enum2bool_{[ity]}(v(2))(\sigma)$$

Notice that all of the inputs are bundled together into the argument v. Complications arise if an error occurs during evaluation. For example, integer arithmetic operations may take the converted result outside of the specified result type. In these cases, then the operator would return its second input argument w, which will be non-deterministically assigned a value from the result type, representing the ELLA ‘query’ value.

For details of the semantics of other BIOPs, refer to [HM92]. Note that the convention adopted there is slightly different from that outlined above — the input and output types in the BIOP call are *dynamically* checked, and the ‘query’ value is returned if the call is not well-formed.

B.11 Action Algebra Rewrite Rules

It is often convenient to consider expressions from the ELLA action algebra at the *term* level and to apply semantically-preserving rewrite rules to the terms. This has a strong correspondance at the *EPA* level, to

ELLA transformations presented in [HM92]. The idea here is to reduce the size of terms as far as possible, without grounding and interpreting them, so that subsequent reasoning, such as determining a symbolic bisimulation, can be performed efficiently. Of course, in certain cases the simplifying rewrite rule has the same effect as evaluation.

The following shows some of the rules that can be usefully applied when attempting to reduce the expression size — they mostly concern terms resulting from 'plumbing' operations at the ELLA level. In these, it is assumed that the expressions are well-formed, for example that indices are within range. Some of the rules have been slightly simplified — for example rule Tup-Ind, does not address the case when value expression e is a variable.

$$\boxed{\text{Ind-Tup}}\ \frac{\mathsf{index}(i, \mathsf{tuple}(e_1, \cdots, e_k))}{e_i}$$

$$\boxed{\text{Tup-Ind}}\ \frac{\begin{array}{ll}\text{(a)} & \mathsf{index}(i, \mathsf{tuple}(\mathsf{index}(1,e), \cdots, \mathsf{index}(k,e))) \\ \text{(b)} & e = \mathsf{tuple}(e_1, \ldots, e_k)\end{array}}{e}$$

$$\boxed{\text{Ind-Copy}}\ \frac{\mathsf{index}(i, \mathsf{copy}(e,n))}{e}$$

$$\boxed{\text{Ind-Trim}}\ \frac{\mathsf{index}(i, \mathsf{trim}(e,m,n))}{\mathsf{index}(i+m-1, e)}$$

$$\boxed{\text{Trim-Tup}}\ \frac{\mathsf{trim}(\mathsf{tuple}(e_1, \cdots, e_k), m, n))}{\mathsf{tuple}(e_m, \cdots, e_n)}$$

$$\boxed{\text{Trim-Copy}}\ \frac{\mathsf{trim}(\mathsf{copy}(e,k), m, n)}{\mathsf{copy}(e, n-m+1)}$$

$$\boxed{\text{Ind-Case}}\ \frac{\mathsf{index}(i, \mathsf{case}\ expr_1 : m_1 \to e_1, \cdots, m_k \to e_k\ \mathsf{else}\ expr_2\ \mathsf{end})}{\mathsf{case}\ expr_1 : m_1 \to \mathsf{index}(i, e_1), \cdots, m_k \to \mathsf{index}(i, e_k)\ \mathsf{else}\ \mathsf{index}(i, expr_2)\ \mathsf{end})}$$

$$\boxed{\text{Conc1}}\ \frac{\mathsf{concat}(\mathsf{tuple}(a_1, \cdots, a_m), \mathsf{tuple}(b_1, \cdots, b_n))}{\mathsf{tuple}(a_1, \cdots, a_m, b_1, \cdots, b_n)}$$

(a) $\mathsf{index}(i, \mathsf{concat}(\mathsf{tuple}(a_1, \cdots, a_m), \mathsf{tuple}(b_1, \cdots, b_n))$

(b) $i \leq m$

Ind-Conc1

---

a_i

(a) $\mathsf{index}(i, \mathsf{concat}(\mathsf{tuple}(a_1, \cdots, a_m), \mathsf{tuple}(b_1, \cdots, b_n))$

(b) $i > m$

Ind-Conc2

---

b_{i-m}

C How to read the semantics definition

Each entry in the horizontal semantic definition is given a certain amount of form and structure, as indicated by the small example of binary trees below.

Construct

Structure:

$$
\begin{array}{rcl}
\mathit{BinTree} & ::= & \mathit{nil} \mid \mathit{tip} \mid \mathit{node} \\
\mathit{Nil} & ::= & \texttt{emptytree} \\
\mathit{Tip} & ::= & \texttt{tip}\ \mathit{data} \\
\mathit{Node} & ::= & \texttt{node}\ \mathit{data}\ \texttt{of}\ \mathit{bintree}_1\ \texttt{and}\ \mathit{bintree}_2 \\
 & & \\
\mathit{Data} & ::= & \mathit{nat} \mid \mathit{name} \\
\mathit{Nat} & = & \mathbb{N} \\
\mathit{Name} & = & \textbf{LEXICAL ITEM}
\end{array}
$$

This paragraph will usually describe aspects of the structural forms defined above. For example, binary trees have three possible shapes: an empty tree, a tree tip or a node of a tree. The data that can be carried in the tree's is also specified above.

Static semantics rules:

$$\text{inference rule}_1$$
$$\ldots$$
$$\text{inference rule}_k$$

The static semantic rules above will be discussed here.

Dynamic semantics equations:

equation_1

$\ldots$

equation_n

Dynamic semantics rules:

$$\text{inference rule}_1$$
$$\ldots$$
$$\text{inference rule}_m$$

The dynamic semantic equation and/or rules presented above would then be discussed here.

C.1 Inference rule structure

The static semantics is written using inference rules, a natural style for inductively defining complex relations over a syntax. The idea is that applications of these rules constitute a step in a logical 'proof' of some particular statement.

Conversely, given a statement to be proven (using some particular static environment), any inference rule whose conclusion "pattern matches" the goal statement can be used to reduce the goal to a number of (hopefully simpler) sub-goal statements. This is a useful way of building such proof trees (or proof diagrams) in a top-down, goal directed manner. Even so, this general technique usually requires further guidance for proof attempts to succeed, assuming that this is possible. Fortunately, this problem is known to be computable and so such guidance can always be given.

The notation used for inference rules is as follows:

$$\boxed{\text{rulename 1}}\ \frac{\textit{premiss}}{\textit{conclusion}}$$

This is essentially a standard *if — then* form of rule: if the *premiss* holds, then so does the *conclusion*, assuming that all free variables for the entire rule are globally universally quantified. When there are several premisses to be satisfied then they may be written in a vertical format as follows:

$$\boxed{\text{rulename 2}}\ \frac{\begin{array}{ll}\text{(a)} & \textit{premiss}_1 \\ \ldots & \ldots \\ \text{(n)} & \textit{premiss}_n\end{array}}{\textit{conclusion}}$$

Finally, sometimes a rule acts exclusively, giving an *if and only if* behaviour. This is indicated graphiclly by using a double inference line as follows:

$$\boxed{\text{rulename 3}}\ \begin{array}{c}\textit{premiss} \\ \hline\hline \textit{conclusion}\end{array}$$

C.2 Semantic Brackets — 'Anti-Quotation'

The structural syntax notation used above introduces collections of structured data values. These may be created and also accessed by pattern-matching using Strachey's *semantic bracket* notation. In terms of logic, this is also known as *anti-quotation* (see [Pau87], [Qui40]). This notation is widely used, within for example Denotational Semantics (see [Sto77], [Sch86]) to indicate structured phrase information.

By way of illustration, instances of the binary tree structure could be written as follows:

1. $[\![\ \texttt{emptytree}\ ]\!]$
2. $[\![\ \texttt{node}\ \boxed{\text{abc}}\ \texttt{of emptytree and}\ (\ \texttt{tip}\ 3)]\!]$

Notice that we have allowed brackets to be inserted to group structural forms as needed. Name literals have been distinguished by placing a box around them e.g $\boxed{\text{abc}}$.

The main use of this semantic notation is to provide an elegant and simple way of structurally analysing data forms. For example, we may have a function defined like this:

$$
\begin{aligned}
&\mathit{sumtree} : \mathit{BinTree} \rightarrow \mathbb{N}\\
&\quad \mathit{sumtree}[\![\ \texttt{emptytree}\ ]\!] = 0\\
&\quad \mathit{sumtree}[\![\ \texttt{tip}\ n]\!] = \mathit{num\text{-}of}(n)\\
&\quad \mathit{sumtree}[\![\ \texttt{node}\ d\ \texttt{of}\ bt_1\ \texttt{and}\ bt_2]\!] = n_1 + n_2 + 2.n_3\\
&\qquad \text{where}\\
&\qquad\quad n_1 = \mathit{sumtree}(bt_1)\\
&\qquad\quad n_2 = \mathit{num\text{-}of}(d)\\
&\qquad\quad n_3 = \mathit{sumtree}(bt_2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{num\text{-}of} : \mathit{Data} \rightarrow \mathbb{N}\\
&\quad \forall\ nm \in \mathit{Name}, n \in \mathit{Nat} \bullet\\
&\qquad \mathit{num\text{-}of}(n) = n\\
&\qquad \mathit{num\text{-}of}(nm) = 1
\end{aligned}
$$

The semantic bracket notation is used to *pattern-match* the inputs to *sumtree*, by using *variables* in the composite syntax template to bind structured values. For example, the function *sumtree* behaves as follows:

$$\mathit{sumtree}[\![\ \texttt{node}\ \boxed{\text{abc}}\ \texttt{of emptytree and}\ (\ \texttt{tip}\ 3)]\!] = 7$$

C.3 *EPA* Brackets

In a similar vein to the semantic brackets described above, a second quotation mechanism has been introduced to delimit those meta-level expressions which will result in object-level *EPA* process definitions or terms. For example, if there are concrete process names *procname*1, *procname*2, signal names *inm*, *hidename*, *onm* and type expression *TY*, then a description of a process expression *procexpr* may be given in the meta-level language by:

$$
\begin{aligned}
&\mathit{procexpr} = (\![\ (MU_1\ \mid\ MU_2)\backslash \mathit{nmset}\ ]\!)\\
&\quad \text{where}\\
&\qquad MU_1 = (\![\ \mathit{procname}1(\mathit{inm} : \mathit{TY}\ /\ \mathit{hidename} : \mathit{TY}\ ]\!)\\
&\qquad MU_1 = (\![\ \mathit{procname}2(\mathit{hidename} : \mathit{TY}\ /\ \mathit{onm} : \mathit{TY})\ ]\!)\\
&\qquad \mathit{nmset} = \{\mathit{hidename}\}
\end{aligned}
$$

Of course, the expressions assigned to MU_1, MU_2 may well have been generated by seperate semantic operations. This would result in the following object-level process term expression:

$$\Big(\ \mathit{procname}1(\mathit{inm} : \mathit{TY}\ /\ \mathit{hidename} : \mathit{TY})\ \mid\ \mathit{procname}2(\mathit{hidename} : \mathit{TY}\ /\ \mathit{onm} : \mathit{TY})\ \Big)\backslash\{\mathit{hidename}\}$$

C.4 The where notation

The where clause is 'syntactic sugar' for logical implication and global universal quantification of free variables:

logical-formula
  where
    $logical\text{-}formula_1$
    $\cdots$
    $logical\text{-}formula_n$

will be logically equivalent to:

$$\forall\, fv_1 \in ty_1, \cdots, fv_k \in ty_k \bullet$$
$$logical\text{-}formula_1 \wedge \cdots \wedge logical\text{-}formula_n \;\Rightarrow\; logical\text{-}formula$$

where $fv_i \in ty_i$ are appropriately typed free variables. Clearly, equations are suitable as logical formulae!

C.5 The case notation

The **case** notation is 'syntactic sugar' for combining similar but slightly different semantic definitions:

$log\text{-}formulae_1$
case *expr* **in**
    $chset_1 : logical\text{-}formula_1$
    $\cdots : \cdots$
    $chset_{k+1} : logical\text{-}formula_{k+1}$
else $logical\text{-}formula_{k+2}$
end
$log\text{-}formulae_2$

where *expr* produces a value in a set TY, with $chset_1, \cdots, chset_k \subseteq TY$, where the $chset_i$ are non-overlapping. This is equivalent to:

$$
\begin{aligned}
&expr \in chset_1 \;\wedge\; log\text{-}formulae_1\\
&\qquad\qquad\qquad \wedge\, logical\text{-}formula_2\\
&\qquad\qquad\qquad \wedge\, log\text{-}formulae_2\\
&\qquad\quad \vee \;\cdots\; \vee\\
&expr \in chset_{k+1} \;\wedge\; log\text{-}formulae_1\\
&\qquad\qquad\qquad \wedge\, logical\text{-}formula_{k+1}\\
&\qquad\qquad\qquad \wedge\, log\text{-}formulae_2\\
&\qquad\quad \vee\\
&expr \in TY - \bigcup\{chset_1,\;\cdots\;,chset_k\} \;\wedge\; log\text{-}formulae_1\\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge\, logical\text{-}formula_{k+2}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge\, log\text{-}formulae_2
\end{aligned}
$$

C.6 Other Notation

Several VDM-style operators are used for map, set and list manipulation. They are summarised below, with examples illustrating their effect. Let

$$
\begin{aligned}
m &= \{a \mapsto 1, b \mapsto 3, c \mapsto 3, d \mapsto 5\}\\
s &= \{u, v, w, x, y, z\}\\
l_1 &= [A, B, C, B, D]\\
l_2 &= [A, E, F, A]
\end{aligned}
$$

then

Operator	Name	Example
$\mathsf{dom}\ \_$	domain	$\mathsf{dom}\, m = \{a, b, c, d\}$
$\mathsf{rng}\ \_$	range	$\mathsf{rng}\, m = \{1, 3, 5\}$
$\_ \mathbin{⩤} \_$	domain subtraction	$\{b\} ⩤ m = \{a \mapsto 1, c \mapsto 3, d \mapsto 5\}$
$\_ \lhd \_$	domain restriction	$\{b\} \lhd m = \{b \mapsto 3\}$
$\_ \dagger \_$	map overwrite	$m \dagger \{c \mapsto 7\} = \{a \mapsto 1, b \mapsto 3, c \mapsto 7, d \mapsto 5\}$
$\mathsf{card}\ \_$	size of set	$\mathsf{card}\, s = 6$
$\mathsf{elems}\ \_$	set of elements in sequence	$\mathsf{elems}\, l_1 = \{A, B, C, D\}$
$\mathsf{len}\ \_$	sequence length	$\mathsf{len}\, l_1 = 5$
$\_ \curvearrowright \_$	sequence concatenation	$l_1 \curvearrowright l_2 = [A, B, C, B, D, A, E, F, A]$
$\_(\_)$	sequence indexing	$l_2(3) = F$

C.7 Semantic functions and pre-conditions

We have generally marked pre-conditions when they provide useful documentation of the function they qualify. Frequently, however, the functions comprising the dynamic semantics have pre-conditions essentially delimited by the constraint imposed by the static semantics. In such cases, this pre-condition information is *not* restated — the reader should interpret the static semantics to determine pre-conditions for the dynamic semantic functions.

When the pre-conditions are explicitly stated, we use the following notations. We allow pre-condition information to be stated both globally and locally — the *full* pre-condition is then determined by taking each of the local conditions and adding the global constraint as well.

A global constraint is associated with the type signature of the function:

$$
\begin{array}{l}
\textit{my-fun}(a,b,c):\ \mathbb{N}\times\mathbb{N}\times\mathbb{N}\rightarrow\mathbb{R}\\
\quad \textsf{pre}\\
\qquad a+b\text{-}c\neq 0\\
\qquad \textit{my-fun}(x,y,z)=\frac{x.y}{x+y\text{-}z}
\end{array}
$$

Combined global and local constraints can be specified as follows:

$$
\begin{array}{l}
\textit{div-tree}(t){:}\,\textit{BinTree}\rightarrow\mathbb{R}\\
\quad \textsf{pre}\\
\qquad t\neq\ \texttt{emptytree}\\
\textit{div-tree}[\![\ \texttt{tip}\ d]\!]=\frac{2}{d}\\
\quad \textsf{pre}\\
\qquad d\in\mathbb{N}\wedge d>0\\
\textit{div-tree}[\![\ \texttt{node}\ d\ \texttt{of}\ bt_1\ \texttt{and}\ bt_2]\!]=v\\
\quad \textsf{where}\\
\qquad v=2.d+\textit{div-tree}(bt_1)\ \textit{div-tree}(bt_2)\\
\quad \textsf{pre}\\
\qquad d\in\mathbb{N}\wedge d>0
\end{array}
$$

Any local pre-condition's are simply conjoined (with the global condition) to the other predicates in the context of a where clause.

C.8 Type parameterisation

We have introduced 'generic' functions by allowing type parameters to be used, such as:

$$
\begin{array}{l}
\textit{reverse}_{[A]}\ :\ A^*\rightarrow A^*\\
\textit{reverse}_{[B]}([\,])=[\,]\\
\textit{reverse}_{[C]}(a::l)=\textit{reverse}_{[C]}(l)\frown[a]
\end{array}
$$

In some sense, this denotes a parametric family of ordinary functions, each of which satisfy the same definition. Clearly, the type parameters range over some suitable universe of set types.

Such a function can then be used as follows:

$$reverse([1,2,3,4,5]) = [5,4,3,2,1]$$

$$snoc_{[B]} : B^* \times B \to B^*$$
$$snoc(l,x) = reverse(a :: reverse(l))$$

$$snoc_{[\mathbb{N}]}([1,2,3,4],5) = [1,2,3,4,5]$$

Note that we have omitted type parameter information in the above when this may be *inferred* from the context.

D Example: Static Semantics

The static semantics check is preceded by applying the *declash* operation(§5.1.1) to the ELLA-text, to make sure that all of the type/function/signal names used are (globally) unique. In the example used below, this is the case, and therefore the ELLA-text is unaltered by *declash*. The static semantics check is a pre-condition of applying the dynamic semantics operations. Additionally, the static semantics identifies a static environment *st-env* which contains the information necessary for the dynamic semantics operations to calculate the type of each ELLA expression. In fact *st-env* is unique for a well-formed ELLA-text.

A very simple ELLA-text example is used to demonstrate how the static semantic rules are applied:

```
TYPE chr3 = NEW (a | b | c).
FN VAR = (chr3 : v) -> chr3:  v.
```

This contains a single type declaration, which gives a 3-value enumeration type called `chr3`, containing the tags `a, b, c`. A single function declaration is given for a function called *VAR*, which takes a single input `v` of type `chr3` and simply returns its value.

The example proceeds simply by presenting a numbered list of the static semantic rules that are applied. A 'tree' of numbered rules is formed, as shown in Figure 2 — the root of the tree is rule 1. Each rule is presented in the same form as it was in the static semantic definitions. A rule has the following meaning:

> *'for the conclusion below the inference line to hold, then the premiss(es) above the line must (all) hold'.*

Each rule takes the label supplied in the static semantics description, so that it is straightforward to compare an occurrence of a rule in this example with the original definition. For example, rule 2 is an instance of the static semantic rule *wf-Declaration sequence*(§5.10), and if it holds then Rule 1 also holds, and the complete ELLA-text is well-formed. Further, for (2) to hold then (3)–(6) must also hold, and so on.

In the following, segments of the ELLA-text such as `chr3` or `NEW(a | b | c)` have been underlined, to distinguish them from variables and constructs local to the rules. The ELLA-text for the type and function declarations `chr3` and `VAR` given above is represented by **td** and **fd** respectively. The static semantic relation only holds if the ELLA-text is well-typed. If the relation holds then it gives a static environment *st-env* which contains all the information necessary to calculate the type of any expression in the ELLA-text supplied.

$$\boxed{\text{1: wf-Closure}}\ \frac{2{:}\ \emptyset \vdash_{Declaration^*} \mathbf{td},\mathbf{fd}\ \text{ produces } \textit{st-env}}{\vdash_{Closure} \mathbf{td},\mathbf{fd}}$$

$$\boxed{\text{2: wf-Declaration sequence}}\ \frac{\begin{array}{ll} 3{:} & \textit{st-ev}_1 = \emptyset \\ 4{:} & \textit{st-ev}_1 \vdash_{Declaration} \mathbf{td}\ \text{ produces } \textit{st-ev}_2 \\ 5{:} & \textit{st-ev}_2 \vdash_{Declaration} \mathbf{fd}\ \text{ produces } \textit{st-ev}_3 \\ 6{:} & \textit{st-env} = \textit{st-ev}_3 \end{array}}{\emptyset \vdash_{Declaration^*} \mathbf{td},\mathbf{fd}\ \text{ produces } \textit{st-env}}$$

Rule 2 passes along the list of declarations, the type declaration **td** in the empty environment $st\text{-}ev_1$, and then checking the function declaration **fd** in the environment $st\text{-}ev_2$ produced by **td**. Note that rules such as (3), which are effectively assignments, are trivially true. Now the type declaration can be checked:

$$\boxed{\text{4: wf-Declaration : TypeDecl}}\ \frac{\begin{array}{ll} 7: & \mathbf{td} \in \mathit{TypeDecl} \\ 8: & \emptyset \vdash_{\mathit{TypeDecl}} \mathbf{td} \ \text{ produces } st\text{-}ev_2 \end{array}}{\emptyset \vdash_{\mathit{Declaration}} \mathbf{td} \ \text{ produces } st\text{-}ev_2}$$

$$\boxed{\text{8: wf-TypeDecl : TypeFormer}}\ \frac{\begin{array}{ll} 11: & \underline{\texttt{chr3}} \notin \mathsf{dom}\,\emptyset \\ 12: & \emptyset \vdash_{\mathit{NewType}} \underline{\texttt{(a | b | c)}} \\ 13: & \mathit{tyd} = \underline{\texttt{TYPE chr3 = NEW (a | b | c)}} \\ 14: & st\text{-}ev_2 = \emptyset \cup \{\underline{\texttt{chr3}} \mapsto \mathit{tyd}\} \end{array}}{\emptyset \vdash_{\mathit{TypeDecl}} \mathbf{td} \ \text{ produces } st\text{-}ev_2}$$

Line (11) checks that the type name $\underline{\texttt{chr3}}$ has not been used previously, and (13,14) add the necessary information about **td** to the static environment to form $st\text{-}ev_2$.

Rule 12 checks that the new enumeration type is well-formed — each new tag must be unique in $\underline{\texttt{chr3}}$, and must not have been used in a previous type declaration.

$$\boxed{\text{12: wf-NewType : AssocType}}\ \frac{\begin{array}{ll} 15: & \mathit{tag\text{-}set} = \mathit{tags\text{-}of\text{-}newtype}(\underline{\texttt{(a | b | c)}}) \\ 16: & \mathsf{card}\,\mathit{tag\text{-}set} = 3 \\ 17: & \emptyset \vdash_{\mathit{TagType}} \underline{\texttt{a}} \\ 18: & \emptyset \vdash_{\mathit{TagType}} \underline{\texttt{b}} \\ 19: & \emptyset \vdash_{\mathit{TagType}} \underline{\texttt{c}} \end{array}}{\emptyset \vdash_{\mathit{NewType}} \underline{\texttt{(a | b | c)}}}$$

It passes along the sequence of tags $\underline{\texttt{a | b | c}}$ and checks each element separately in (17)–(19). For example, Rule 17 expands to:

$$\boxed{\text{17: wf-TagType: Tag}}\ \frac{\underline{\texttt{a}} \notin \mathit{tags\text{-}of\text{-}env}(\emptyset)}{\emptyset \vdash_{\mathit{TagType}} \underline{\texttt{a}}}$$

and similarly for (18) and (19). The variable *tag-set* gives the set of tags in the type:

$$\begin{aligned} \mathit{tag\text{-}set} &= \mathit{tags\text{-}of\text{-}newtype}(\underline{\texttt{(a | b | c)}}) \\ &= \{\underline{\texttt{a}}, \underline{\texttt{b}}, \underline{\texttt{c}}\} \end{aligned}$$

The resulting environment $st\text{-}ev_2$ is given by:

$$\mathit{st\text{-}ev}_2 = \{\ \underline{\texttt{chr3}}\ \mapsto\ \underline{\texttt{TYPE chr3 = NEW (a | b | c)}}\ \}$$

Now the function declaration **fd** is checked, in the environment $\mathit{st\text{-}ev}_2$ produced by (4) above:

$$\boxed{\text{5: wf-Declaration : FnDecl}}\ \frac{\begin{array}{ll} 9: & \mathbf{fd} \in \mathit{FnDecl} \\ 10: & \mathit{st\text{-}ev}_2 \vdash_{\mathit{FnDecl}} \mathbf{fd}\ \text{ produces } \mathit{st\text{-}ev}_3 \end{array}}{\mathit{st\text{-}ev}_2 \vdash_{\mathit{Declaration}} \mathbf{fd}\ \text{ produces } \mathit{st\text{-}ev}_3}$$

$$\boxed{\text{10: wf-FnDecl: UnitFnd}}\ \frac{\begin{array}{ll} 20: & \underline{\texttt{VAR}} \notin \text{dom}\, \mathit{st\text{-}ev}_2 \\ 21: & [\ \underline{\texttt{v}}\ ] = \mathit{extract\text{-}fndecl\text{-}signals}(\ \underline{\texttt{chr3 : v}}\ ) \wedge \text{len}[\ \underline{\texttt{v}}\ ] = 1 \\ 22: & \mathit{fnd\text{-}st\text{-}env} = \mathit{st\text{-}ev}_2 \cup \mathit{extract\text{-}sigdecl\text{-}binding}(\ \underline{\texttt{chr3 : v}}\ ) \\ 23: & \mathit{fnd\text{-}st\text{-}env} \vdash_{\mathit{Unit}} \underline{\texttt{v}}\ \text{ has type } \underline{\texttt{chr3}} \\ 24: & \mathit{fnd\text{-}st\text{-}env} \vdash_{\mathit{E\text{-}TyEq}} \underline{\texttt{chr3}} \equiv \underline{\texttt{chr3}} \\ 25: & \mathit{fnd\text{-}st\text{-}env} \vdash_{\mathit{Unit}} \underline{\texttt{v}}\ \text{ produces } \mathit{fnd\text{-}st\text{-}env} \\ 26: & \underline{\texttt{chr3}} = \mathit{extract\text{-}sigdecl\text{-}types}(\ \underline{\texttt{chr3 : v}}\ ) \\ 27: & \mathit{st\text{-}ev}_3 = \mathit{fnd\text{-}st\text{-}env} \cup \{\ \underline{\texttt{VAR}} \mapsto \texttt{funtype}\ \underline{\texttt{chr3}} \rightarrow \underline{\texttt{chr3}}\ \} \end{array}}{\mathit{st\text{-}ev}_2 \vdash_{\mathit{FnDecl}} \mathbf{fd}\ \text{ produces } \mathit{st\text{-}ev}_3}$$

Line (20) checks that the function has not been previously declared. Line (21) checks that all of the signal declarations are unique — this is trivially true here, since only one signal is declared. Line (22) adds the function signal declaration binding to $\mathit{st\text{-}ev}_2$ to form the environment $\mathit{fnd\text{-}st\text{-}env}$ — the maplet $\{v \mapsto \texttt{let-type}\ \underline{\texttt{chr3}}\ \}$ is added. Rule (23) checks that the function body is well-formed:

$$\boxed{\text{23: wf-Unit : Signal}}\ \frac{\begin{array}{ll} 28: & \underline{\texttt{v}} \in \text{dom}\, \mathit{fnd\text{-}st\text{-}env} \\ 29: & \underline{\texttt{chr3}} = \mathit{vartype\text{-}of}(\mathit{fnd\text{-}st\text{-}env}(\ \underline{\texttt{v}}\ )) \end{array}}{\mathit{fnd\text{-}st\text{-}env} \vdash_{\mathit{Unit}} \underline{\texttt{v}}\ \text{ has type } \underline{\texttt{chr3}}}$$

Here, it checks that the signal $\underline{\texttt{v}}$ has already been declared. The type of $\underline{\texttt{v}}$ must be the same as the type of the function output — this is checked by Rule (24):

$$\boxed{\text{24: E-TyEq: Reflexivity}}\ \frac{30:\ \mathit{fnd\text{-}st\text{-}env} \vdash_{\mathit{Type}} \underline{\texttt{chr3}}}{\mathit{fnd\text{-}st\text{-}env} \vdash_{\mathit{E\text{-}TyEq}} \underline{\texttt{chr3}} \equiv \underline{\texttt{chr3}}}$$

$$\boxed{\text{30: wf-Type : Typename}}\ \frac{31:\ \underline{\texttt{chr3}} \in \text{dom}\, \mathit{fnd\text{-}st\text{-}env}}{\mathit{fnd\text{-}st\text{-}env} \vdash_{\mathit{Type}} \underline{\texttt{chr3}}}$$

Here, the type expressions for the function body and output happen to be identical.

Rule (25) updates the static environment with any local declarations contained in the function body — here, the environment produced is unaltered, since the body does not contain any local declarations. Finally, lines (26,27) add the necessary information about the function declaration to the environment, to form *st-ev*$_3$:

$$
\begin{aligned}
\textit{st-ev}_3 = \{\ & \underline{\texttt{chr3}} \mapsto \underline{\texttt{TYPE chr3 = NEW (a | b | c)}}\,, \\
& \underline{\texttt{v}} \mapsto \texttt{let-type}\ \underline{\texttt{chr3}}\,, \\
& \underline{\texttt{VAR}} \mapsto \texttt{funtype}\ \underline{\texttt{chr3}} \to \underline{\texttt{chr3}}\ \}
\end{aligned}
$$

This contains information about all of the type/function/signal declarations in the ELLA-text so that the dynamic semantic operations can calculate the type of any expression within the ELLA-text.

The following operations are used above to extract name and type information about the input signal declaration, so that the static environment can be updated. Here, only one signal declaration for $\underline{\texttt{v}}$ is given:

$$
\begin{aligned}
& \textit{extract-fndecl-signals}(\underline{\texttt{chr3 : v}}) = \{\underline{\texttt{v}}\} \\
& \textit{extract-sigdecl-binding}(\underline{\texttt{chr3 : v}}) = \{v \mapsto \texttt{let-type}\ \underline{\texttt{chr3}}\} \\
& \textit{extract-sigdecl-types}(\underline{\texttt{chr3 : v}}) = (\underline{\texttt{chr3}}) \\
& \textit{vartype-of}(\textit{new-st-env}(\underline{\texttt{v}})) = \underline{\texttt{chr3}}
\end{aligned}
$$

All the sub-rules applied to the ELLA-text hold, and so the text is well-formed. Figure 2 shows the structure of the tree constructed during this type-checking, where the node labels refer to the rules and clauses numbered above.

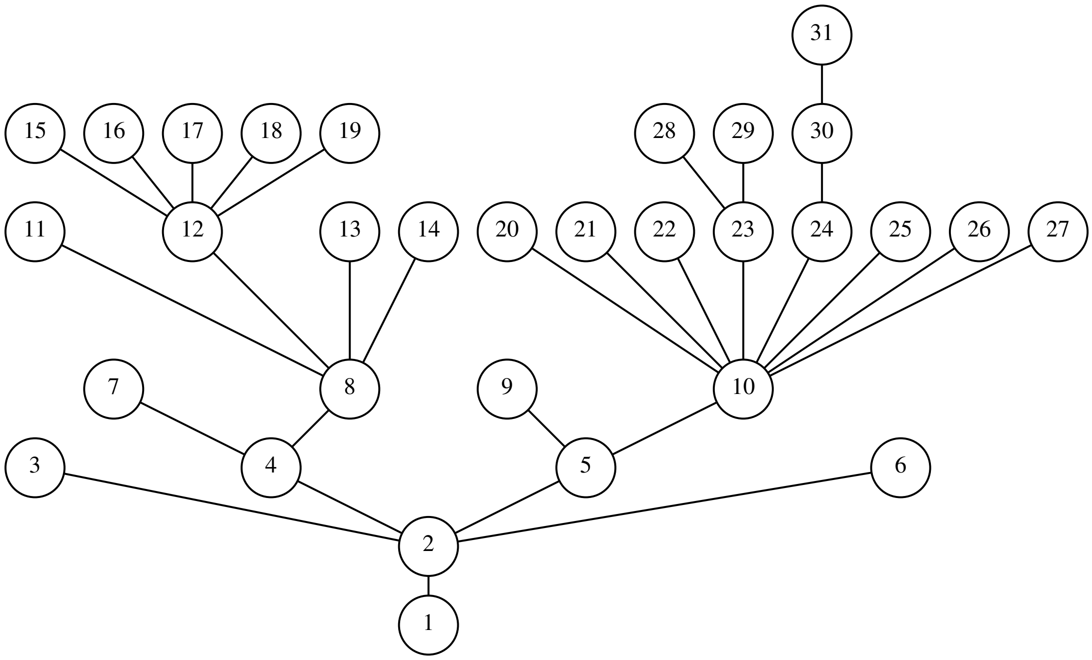

Figure 2: Proof-Tree for Well-Formedness Check on ELLA-Text Example

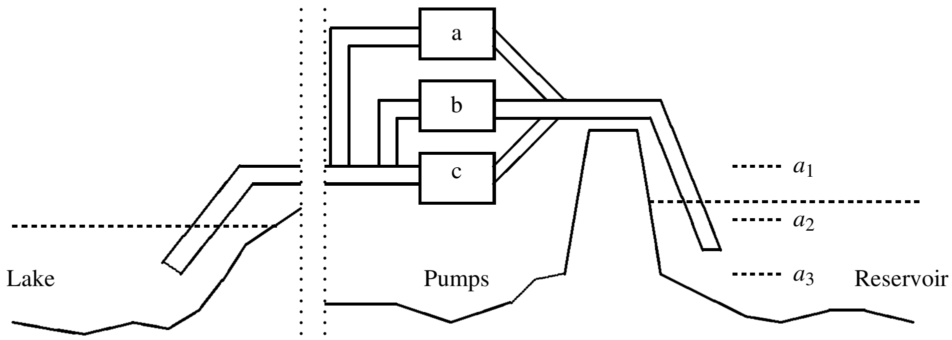

Figure 3: Reservoir, Lake and Pumps

E Example: Dynamic Semantics for Pump Controller

This section describes in some detail the construction of the semantic object corresponding to a small ELLA-text, using the definition of the formal semantics for Core ELLA given in §5. A Core ELLA description of a three-pump controller is given. The original definition of the controller from [MDED91] is reproduced here:

> *A reservoir is connected to a lake by a pipe line. Water is taken from the lake to the reservoir by a system of three pumps.*
>
> *Three level sensors are installed on the reservoir. Their outputs are denoted by signals a_1, a_2, a_3. Signal a_i is 0 when the water is above level i, for i = 1,2,3 and has a value 1 when the water is below level i. The number of pumps that are on at any one time depends on the water level in the reservoir. In particular, if the water level is between level 1 and 2, then one pump should be in operation; if the water level is between level 2 and 3, then two pumps should be in operation; if the water level is below level 3, then three pumps should be in operation. Of course, if the water level is above level 1 then no pumps should be in operation. In order to equalise wear on the pumps, they should come into operation in a cyclic manner.*

Figure 3 gives a sketch of the system, showing the water level in the reservoir lying between sensors a_1 and a_2; this means that one pump should be in operation.

The following description is a slightly simplified version of that appearing in [MDED91]; additionally, minor changes have been made to translate the (full) ELLA into Core ELLA. Two enumeration types are used: the first type called 'pump' denotes which pumps are actually operating, the pumps being known as 'a', 'b' and 'c'. The second type 'level' denotes the level of water in the reservoir, with zero representing a full reservoir.

The function `CONTROL` is the pump controller and its `CASE` clause determines which pumps are switched on, depending on the previous state of the pumps held in `store` and the current water level `in`:

```
TYPE  pump  = NEW (none | a | b | ab | c | ca | bc | abc ).
TYPE  level = NEW lv/(0..3).

FN DEL = (pump) -> (pump): IDELAY(none, 1).
```

```
FN CONTROL = ( level: in ) -> pump:
BEGIN
   MAKE DEL: store.
   LET newstore = CASE in OF
                    lv/0: none,
                    lv/1: CASE store OF
                              a | ca: b,
                              b | ab: c
                          ELSE a
                          ESAC,
                    lv/2: CASE store OF
                              a | ab: bc,
                              b | bc: ca
                          ELSE ab
                          ESAC
                  ELSE abc
                  ESAC.
   JOIN newstore -> store.
   OUTPUT newstore
END.
```

E.1 Stage 1: *declash* and Static Checking

The first task is to pass the ELLA-text through *declash* which removes any name-clashes from the text; however, in this example there is no problem with redeclaration of type-, function- or signal-names, and so *declash* leaves the text unaltered.

The ELLA-text is well-formed, and so satisfies the static semantics relation — this operation is illustrated by the previous example, and is not dealt with further here. It is assumed that the static check yields a static environment ρ, containing all the necessary typing information; when the operation $\textit{type-of}(\textit{txt})(\rho,\sigma)$ is called upon a piece of ELLA-text *txt* then the correct output type for *txt* is assumed to be returned. In an implementation, the parser would perform type-checking, and then the output type of each ELLA expression would be stored, perhaps using an attributed parse tree; the operation *type-of* would then simply interrogate the appropriate part of the tree to retrieve the type information.

E.2 Stage 2: Dynamic Semantics

This main section contains details of the construction of the process term(s) representing the semantics of the ELLA-text. ELLA-text expressions are surrounded by semantic brackets $[\![\ \_\ ]\!]$. Those parts of the text which constitute *EPA* expressions have been surrounded by '*EPA* brackets' $(\![\ \_\ ]\!)$; this helps to tackle the problem of distinguishing meta-level text from object-level text.

Preliminaries A dynamic environment $\textit{dy-env} \in \textit{DynamicEnvironment}$ holds the final mapping of declaration names to *EPA* terms, as described in §4. Additionally, it is assumed that the *static* environment ρ is available, as stated above, which contains any typing information for ELLA constructs, required by the dynamic semantics to create channels of the correct type, for example. In this example, most semantic operations do not affect the dynamic environment — although a new environment should be returned by the semantic operations, this will be in the most part remain unaltered. As far as possible, the notation used in the relevant sections of the dynamic semantics has been followed, so that it is easier to compare instances of the semantic operations with their definitions. Names for meta-variables have been chosen so that corresponding expressions in different semantic operations have been given the same name, and distinct expressions have been given distinct names. This is only to assist the reader tracing the progress of expressions through the semantic operations; however, any information required by a semantic operation must of course be supplied to it through its arguments. For example, the dynamic environment produced by *M-Declaration** below is assigned to *dy-env* — this name is also used later when the detailed operation of *M-Declaration** is given.

Closure Construction of the dynamic semantics for the pump controller begins with the closure:

$$\textit{M-Closure}[\![\ \texttt{TYPE}\ \textit{pump}\ \cdots\ \texttt{OUTPUT}\ \textit{store}\ \texttt{END}\ .]\!] = [\textit{Pty}, \textit{Lty}, \textit{Pdefn}_1, \textit{Pdefn}_2]$$

where

$$\begin{aligned}
\textit{dy-env} &= \textit{M-Declaration}^*[\![\ \texttt{TYPE}\ \textit{pump}\ \cdots\ \texttt{OUTPUT}\ \textit{newstore}\ \texttt{END}\ .]\!](\rho, \{\,\}) \\
&= \textit{M-Declaration}[\![\ \texttt{FN}\ \textit{CONTROL}\ \cdots\ \texttt{OUTPUT}\ \textit{newstore}\ \texttt{END}\ .]\!](\rho, \textit{env}_4) \\
&= \{\textit{pump} \mapsto \textit{Pty}, \textit{level} \mapsto \textit{Lty}, \textit{DEL} \mapsto \textit{Pdefn}_1, \textit{CONTROL} \mapsto \textit{Pdefn}_2\} \\
\textit{env}_4 &= \textit{M-Declaration}[\![\ \texttt{FN}\ \textit{DEL}\ \cdots\ \textit{none}, 1).]\!](\rho, \textit{env}_3) \\
\textit{env}_3 &= \textit{M-Declaration}[\![\ \texttt{TYPE}\ \textit{level} = \texttt{NEW}\ (\ \cdots\ ).]\!](\rho, \textit{env}_2) \\
\textit{env}_2 &= \textit{M-Declaration}[\![\ \texttt{TYPE}\ \textit{pump} = \texttt{NEW}\ (\ \cdots\ ).]\!](\rho, \{\,\})
\end{aligned}$$

The semantics for each type- and function-declaration is calculated in turn, using the environment generated by the previous declaration, to build up the final dynamic environment *dy-env*. From this, an *EPA context* is formed, consisting of a sequence of *EPA* language definitions.

Type Declarations for ***pump*** **and** ***level*** are first evaluated and added to an initially empty dynamic environment $\{\,\}$:

$$\textit{M-Typedecl}[\![\ \texttt{TYPE}\ \textit{pump} = \texttt{NEW}\ (\ \cdots\ ).]\!](\rho, \{\,\}) = \textit{env}_2$$

where

$$\begin{aligned}
\textit{env}_2 &= \{\,\} \cup \{\textit{pump} \mapsto \textit{Pty}\} \\
\textit{Pty} &= (\!\![\ \text{type}\ \textit{pump} \underline{\triangle}\ \textit{nty}_1\ ]\!\!) \\
\textit{nty}_1 &= \textit{M-Newtype}[\![(\textit{none} \mid a \mid b \mid ab \mid c \mid ca \mid bc \mid abc)]\!](\rho, \emptyset) \\
&= (\!\![\ \{\textit{none}, a, b, ab, c, ca, bc, abc\}\ ]\!\!)
\end{aligned}$$

$$\textit{M-Typedecl}[\![\ \texttt{TYPE}\ \textit{level} = \texttt{NEW}\ (\ \cdots\ ).]\!](\rho, \textit{env}_2) = \textit{env}_3$$

where

$$\begin{aligned}
\textit{env}_3 &= \textit{env}_2 \cup \{\textit{level} \mapsto \textit{Lty}\} \\
\textit{Lty} &= (\!\![\ \text{type}\ \textit{level} \underline{\triangle}\ \textit{nty}_2\ ]\!\!) \\
\textit{nty}_2 &= \textit{M-Newtype}[\![\textit{lv}/(0..3)]\!](\rho, \emptyset) \\
&= (\!\![\ \text{tag}\ \textit{lv}\ \text{int}\ 0\ 3\ ]\!\!)
\end{aligned}$$

The resulting environment env_3 is a map containing two definition terms:

$$env_3 = \{pump \mapsto Pty, level \mapsto Lty\}$$

where

$$Pty = (\!|\ \text{type } pump \ \underline{\triangle}\ \{none, a, b, ab, c, ca, bc, abc\}\ |\!)$$

$$Lty = (\!|\ \text{type } level \ \underline{\triangle}\ \text{tag } lv \text{ int } 0\ 3\ |\!)$$

Note that the ordering for elements in the enumeration type *pump* can be obtained from *Pty*, which is simply an *expression* representing the set of enumerated values, to be interpreted at a later stage. This ordering information may be required because in ELLA it is possible to compute, for example, the successor of *b*, obtaining *ab* — it is not required in this example, however. Throughout the rest of this section, the types will be referenced by their names *pump* and *level*.

Function Declaration for *DEL* can now be evaluated to form the process definition DEL_P, which is added to env_3 to form env_4:

$$M\text{-}FnDecl[\![\ \texttt{FN}\ \ DEL\ \cdots\ none, 1).]\!](\rho, env_3) = env_4$$

where

$$env_4 = env_3 \cup \{DEL \mapsto Pdefn_1\}$$

$$Pdefn_1 = M\text{-}Primitive[\![\ \texttt{IDELAY}\ (none, 1)]\!](DEL)(\rho, env_3)$$

The body of the declaration is a unit inertial delay primitive, with initial value *none*. The process named, by convention, DEL_P simply calls the generic process definition, named $Idel\text{-}Aux_{[pump,1]}$, with initial value $v = none$ and $count = 1$:

$$M\text{-}Primitive[\![\ \texttt{IDELAY}\ (none, 1)]\!](DEL)(\rho, env_3) = Pdefn_1$$

where

$$Pdefn_1 = (\!|\ \text{process } DEL_P(ic_1\colon pump\ /\ oc_1\colon pump)\ \underline{\triangle}$$
$$\square\, v : \{none\}\ \bullet\ Idel\text{-}Aux_{[pump,1]}(1, v, v\ ;\ ic_1\ /\ oc_1)\ |\!)$$

$$\{none\} = init\text{-}values[\![none]\!]$$

$$\{ic_1, oc_1\} = make\text{-}ChanVars(2, \emptyset)$$

The set of initial values is a singleton in this example, containing the value *none*; the set expression is returned by the operation *init-values*.

The operation *make-ChanVars* generates two fresh channel variables ic_1, oc_1, which will be used in the formal arguments of the process definition $Pdefn_1$.

When the generic process definition $Idel\text{-}Aux_{[pump,1]}$ is instantiated later, then the sequence $[IDEL, FD_1, FD_1]$ of definitions will be added to the *EPA* context:

$$
\begin{aligned}
IDEL = \llparenthesis\ & \textsf{process} \\
& \mathit{Idel\text{-}Aux}_{[pump,1]}(count\colon \textsf{int } 1\ 1, a\colon pump, b\colon pump\ ;\ i\colon pump\ /\ o\colon pump)\ \underline{\triangle} \\
& \square x\colon pump\ \cdot\ i(x)\ /\ o(\mathit{Idel\text{-}1}_{[pump,1]}(count, a, b, 1)) \\
& \quad :: \mathit{Idel\text{-}Aux}_{[pump,1]}(\mathit{Idel\text{-}2}_{[pump,1]}(count, a, b, x), x, \mathit{Idel\text{-}1}_{[pump,1]}(count, a, b, 1)\ ;\ i\ /\ o)\ \rrparenthesis
\end{aligned}
$$

$$
\begin{aligned}
FD_1 = \llparenthesis & \textsf{function } \mathit{Idel\text{-}1}_{[pump,1]}(count\colon \textsf{int } 1\ 1, a\colon pump, b\colon pump)\colon pump\ \underline{\triangle} \\
& \textbf{case } count = 1 : \\
& \quad \{\textsf{true}\} \rightarrow a \\
& \textbf{else } b \\
& \textbf{end} \rrparenthesis
\end{aligned}
$$

$$
\begin{aligned}
FD_2 = \llparenthesis & \textsf{function } \mathit{Idel\text{-}2}_{[pump,1]}(count\colon \textsf{int } 1\ 1, a\colon pump, b\colon pump, x\colon pump)\colon \mathbb{N}\ \underline{\triangle} \\
& \textbf{case } x = a : \\
& \quad \{\textsf{true}\} \rightarrow \textbf{case } count = 1 : \\
& \qquad\qquad\qquad \{\textsf{true}\} \rightarrow count \\
& \qquad\qquad\quad \textbf{else } count + 1 \\
& \qquad\qquad\quad \textbf{end} \\
& \textbf{else } 1 \\
& \textbf{end} \rrparenthesis
\end{aligned}
$$

Note that for *unit* delay, then the process and function definitions can be simplified further, since the case statements degenerate into non-branching expressions; only the following process definition $IDEL'$ needs to be added to the definition sequence:

$$
\begin{aligned}
IDEL' = \llparenthesis\ & \textsf{process } \mathit{Unit\text{-}Del}_{[pump]}(v\colon pump\ ;\ i\colon pump\ /\ o\colon pump)\ \underline{\triangle} \\
& \square w\colon pump\ \cdot\ i(w)\ /\ o(v) :: \mathit{Unit\text{-}Del}_{[pump]}(w\ ;\ i\ /\ o)\ \rrparenthesis
\end{aligned}
$$

Function Declaration for *CONTROL* can now be evaluated:

$$
\mathit{M\text{-}FnDecl}[\![\ \texttt{FN}\ \ CONTROL = (level\colon in)\ \cdots\ \texttt{OUTPUT}\ newstore.]\!](\rho, env_4) = \mathit{dy\text{-}env}
$$

where

$$
\begin{aligned}
& \mathit{dy\text{-}env} = ev_1 \cup \{CONTROL \mapsto Pdefn_2\} \\
& Pdefn_2 = \llparenthesis\ \textsf{process } CONTROL_P(in\colon level\ /\ oc_2\colon pump)\ \underline{\triangle}\ body\ \rrparenthesis \\
& (body, ev_1) = \mathit{M\text{-}Unit}[\![\ \texttt{BEGIN}\ \cdots\ \texttt{END}\ ]\!](oc_2)(\rho, env_4) \\
& \{oc_2\} = \mathit{make\text{-}ChanVars}(1, \emptyset)
\end{aligned}
$$

A process definition, called by convention $CONTROL_P$ after the name of the function declaration, is added to the dynamic environment. The right-hand side of this definition contains the process term *body* representing the function body, i.e. the block $[\![$ `BEGIN` $\cdots$ `END` .$]\!]$:

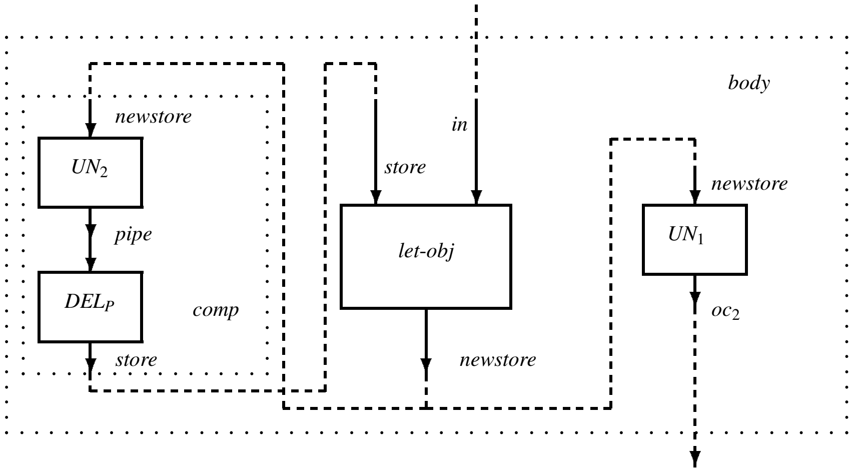

Figure 4: Objects produced by Step Sequence

$$M\text{-}Unit[\![\ \texttt{BEGIN}\ \cdots\ \texttt{END}\ ]\!](oc_2)(\rho, env_4) = (body, ev_1)$$

where

$$\begin{aligned}
body &= block\text{-}join\text{-}product(UN_1, chandecls_1)(lcl\text{-}env)\\
&= (\!\![\ \Big(\ comp\ \mid\ let\text{-}obj\ \mid\ UN_1\ \Big)\backslash chandecls_1\ ]\!\!)\\
chandecls_1 &= local\text{-}channels(store, newstore)(lcl\text{-}env)\\
&= (\!\![\ \{store\colon pump, newstore\colon pump\}\ ]\!\!)\\
(UN_1, ev_1) &= M\text{-}Unit[\![newstore]\!](oc_2)(\rho, block\text{-}env)\\
(block\text{-}env, lcl\text{-}env) &= M\text{-}Step^*[\![\ \texttt{MAKE}\ DEL\ \cdots\ \texttt{JOIN}\ newstore \rightarrow store.]\!](\rho, env_4)
\end{aligned}$$

The object *comp* represents the dynamic semantics of the `LET` step, and the object *let-obj* represents the dynamic semantics of the `MAKE` / `JOIN` steps; the details of their construction will given below. The objects are stored in the local environment *lcl-env*, accessed by *store* and *newstore* respectively. The output from the whole block is determined by UN_1, and is named oc_2 — this is a formal argument to the process definition *CONTROL$_P$* representing the function declaration *CONTROL*. The environments env_4, ev_1, and *block-env* are all identical in this case.

The process terms are given the necessary input and output names so that the causal product performed by *block-join-product* ‘connects’ the required channels. The `LET` creates a process to represent the expression on its right-hand side, and names the output *newstore*. The `MAKE` statement simply records the name of the process — *DEL$_P$* — whose output will be named *store*. The `JOIN` creates the actual process call to *DEL$_P$* naming the input *newstore* and the output *store*. Finally, in this example, the `OUTPUT` unit simply determines which channel will be visible externally to the block unit. Figure 4 show how the elements in the block are ‘connected’.

Before this causal product is taken, the objects *comp* and *let-obj* must be constructed using *M-Step**, and placed in the local environment *lcl-env*:

$$M\text{-}Step^*[\![\texttt{MAKE}\ \ DEL \cdots\ \texttt{JOIN}\ \ newstore \rightarrow store.]\!](\rho, env_4) = (block\text{-}env, lcl\text{-}env)$$

where

$$(blk\text{-}ev_1, lcl\text{-}ev_1) = (env_4, \{\,\})$$
$$(blk\text{-}ev_2, lcl\text{-}ev_2) = M\text{-}Step[\![\texttt{MAKE}\ DEL\text{: } store]\!](\rho, blk\text{-}ev_1, lcl\text{-}ev_1)$$
$$(blk\text{-}ev_3, lcl\text{-}ev_3) = M\text{-}Step[\![\texttt{LET}\ \ newstore = \ \texttt{CASE}\ \cdots\ \texttt{ESAC}\ .]\!](\rho, blk\text{-}ev_2, lcl\text{-}ev_2)$$
$$(blk\text{-}ev_4, lcl\text{-}ev_4) = M\text{-}Step[\![\texttt{JOIN}\ \ newstore \rightarrow store.]\!](\rho, blk\text{-}ev_3, lcl\text{-}ev_3)$$
$$(block\text{-}env, lcl\text{-}env) = (blk\text{-}ev_4, lcl\text{-}ev_4)$$

First the MAKE step is evaluated:

$$M\text{-}Step[\![\texttt{MAKE}\ DEL\text{: } store]\!](\rho, blk\text{-}ev_1, lcl\text{-}ev_1) = (blk\text{-}ev_1, \{store \mapsto DEL_P\})$$

where

$$DEL_P = proc\text{-}name(DEL)$$

This simply records the name of the process DEL_P — generated by the operation *proc-name* — which will be used later in the JOIN statement to construct the process *call* to the term representing the delay function *DEL*.

Now the LET statement is evaluated:

$$M\text{-}Step[\![\texttt{LET}\ \ newstore = \ \texttt{CASE}\ \cdots\ \texttt{ESAC}\ .]\!](\rho, blk\text{-}ev_2, lcl\text{-}ev_2) =$$
$$(ev_2, lcl\text{-}ev_2 \cup \{newstore \mapsto let\text{-}obj\})$$

where

$$(let\text{-}obj, ev_2) = M\text{-}Unit[\![\texttt{CASE}\ \ in\ \texttt{OF}\ \cdots\ \texttt{ESAC}\ .]\!](newstore)(\rho, blk\text{-}ev_2)$$

This constructs the process term *let-obj* representing the CASE statement, with the output channel named *newstore*; details of this appear below. Again, in this case, the environment ev_2 is unchanged by calculation of the right-hand side of the LET statement, and so it is identical to $blk\text{-}ev_2$.

Now the JOIN statement is evaluated:

$$M\text{-}Step[\![\texttt{JOIN}\ \ newstore \rightarrow store.]\!](\rho, blk\text{-}ev_3, lcl\text{-}ev_3) =$$
$$(ev_3, lcl\text{-}ev_3 \dagger \{newstore \mapsto comp\})$$

where

$$comp = (\!\![\left(UN_2 \mid DEL_P(pipe\ /\ store) \right) \backslash \{pipe\text{: } pump\}]\!\!)$$
$$(UN_2, ev_3) = M\text{-}Unit[\![newstore]\!](pipe)(\rho, blk\text{-}ev_3)$$
$$blk\text{-}ev_3(store) = (\!\![\ DEL_P\ ]\!\!)$$
$$\{pipe\} = make\text{-}ChanVars(1, \emptyset)$$

This constructs a process term *comp* where the output of UN_2 has been piped into DEL_P, whose output is named *store*. The pipe channel, *pipe*, is then hidden. The effect of this is simply to rename the input of DEL_P to *newstore*. The process name DEL_P is determined by looking up the object associated with signal *store* in the local environment $lcl\text{-}ev_3$, resulting from the previous `MAKE` step. The process term UN_2, representing the left-hand side of the `JOIN` statement, is simply a call to $Chan\text{-}Proc_{[pump]}$ with the input channel named *newstore*:

$$M\text{-}Unit[\![newstore]\!](pipe)(\rho, blk\text{-}ev_3) = (UN_2, blk\text{-}ev_3)$$

where

$$UN_2 = \langle\!\langle\ Chan\text{-}Proc_{[pump]}(newstore\ /\ pipe)\ \rangle\!\rangle$$

The environment ev_3 is unchanged by this application of *M-Unit* and therefore it is the same as $blk\text{-}ev_3$.

Instantiation of the generic process definition $Chan\text{-}Proc_{[pump]}$ will result in a process which simply copies the value x of type *pump* on its input i to its output o, as required:

$$\langle\!\langle\ \text{process } Chan\text{-}Proc_{[pump]}(i\colon pump\ /\ o\colon pump)\ \underline{\triangle}$$
$$\square x\colon pump\ \cdot\ i(x)\ /\ o(x) :: Chan\text{-}Proc_{[pump]}(i\ /\ o)\ \rangle\!\rangle$$

where

$$\{i, o\} = make\text{-}ChanVars(2, \emptyset)$$

Having described how the step elements are assembled to form the semantic object representing the block body of function declaration *CONTROL*, the remaining task is to examine how the process term *let-obj* is created, for the right-hand side of the `LET`. This involves evaluating $M\text{-}Case[\![$ `CASE` $in\ \cdots$ `ESAC` $.]\!]$. The procedure will be described in detail below for the first inner case statement only, and outlined for other `CASE` expressions, starting with the outer `CASE` clause. The output of the resulting term is to be named *newstore*, the left-hand side signal-name of the `LET` :

$$M\text{-}Unit[\![\ \texttt{CASE}\ \ in\ \ \texttt{OF}\ \ \cdots\ \ \texttt{ESAC}\ .]\!](newstore)(\rho, blk\text{-}ev_2) = (let\text{-}obj, ev_2)$$

where

$$\begin{array}{rcl}
let\text{-}obj & = & (\!\![\left(\ CH_1 \ \mid \ P_1 \ \mid \ P_2 \ \mid \ P_3 \ \mid \ EL_1 \ \mid \ MUX_1 \ \right) \backslash chandecls_2 \ ]\!\!) \\
(CH_1, c\text{-}ev_1) & = & M\text{-}Unit[\![in]\!](sc_1)(\rho, blk\text{-}ev_2) \\
(P_1, c\text{-}ev_2) & = & M\text{-}Unit(choice\text{-}of[\![lv/0\text{: } none]\!])(c_1)(\rho, c\text{-}ev_1) \\
(P_2, c\text{-}ev_3) & = & M\text{-}Unit(choice\text{-}of[\![lv/1\text{: } \texttt{CASE}\ store\ \cdots\ a\ \texttt{ESAC}\ ]\!])(c_2)(\rho, c\text{-}ev_2) \\
(P_3, c\text{-}ev_4) & = & M\text{-}Unit(choice\text{-}of[\![lv/2\text{: } \texttt{CASE}\ store\ \cdots\ ab\ \texttt{ESAC}\ ]\!])(c_3)(\rho, c\text{-}ev_3) \\
set_1 & = & M\text{-}Constset(constset\text{-}of[\![lv/0\text{: } none]\!])(\rho, \sigma) \\
 & = & \mathsf{tag}(lv, 0) \\
set_2 & = & M\text{-}Constset(constset\text{-}of[\![lv/1\text{: } \texttt{CASE}\ store\ \cdots\ a\ \texttt{ESAC}\ ]\!])(\rho, \sigma) \\
 & = & \mathsf{tag}(lv, 1) \\
set_3 & = & M\text{-}Constset(constset\text{-}of[\![lv/2\text{: } \texttt{CASE}\ store\ \cdots\ ab\ \texttt{ESAC}\ ]\!])(\rho, \sigma) \\
 & = & \mathsf{tag}(lv, 2) \\
(EL_1, ev_3) & = & M\text{-}Unit[\![abc]\!](elc_1)(\rho, c\text{-}ev_4) \\
MUX_1 & = & (\!\![\ Chooser_{[level, pump, [set_1, set_2, set_3]]}(sc_1, c_1, c_2, c_3, elc_1\ /\ newstore)\ ]\!\!) \\
chandecls_2 & = & (\!\![\ \{sc_1\text{: } level, c_1\text{: } pump, c_2\text{: } pump, c_3\text{: } pump, elc_1\text{: } pump\}\ ]\!\!) \\
\{sc_1, c_1, c_2, c_3, elc_1\} & = & make\text{-}ChanVars(5, \mathsf{dom}\,\rho \cup \{newstore\})
\end{array}$$

Note that the environments $blk\text{-}ev_2, c\text{-}ev_1, \cdots, c\text{-}ev_4, ev_3$ are all the same in this case. The term *let-obj* representing the right-hand side of the `LET` statement, is now added to the local environment $lcl\text{-}ev_2$, and is accessed by *newstore*.

Now the construction of the process term for the right-hand side of the second case-item from above will be described in detail:

$$\begin{array}{l}
M\text{-}Unit(choice\text{-}of[\![lv/1\text{: } \texttt{CASE}\ store\ \cdots\ a\ \texttt{ESAC}\ ]\!])(c_2)(\rho, c\text{-}ev_2) \\
\quad = M\text{-}Unit[\![\ \texttt{CASE}\ store\ \cdots\ a\ \texttt{ESAC}\ ]\!](c_2)(\rho, c\text{-}ev_2) \\
\quad = (P_2, c\text{-}ev_3)
\end{array}$$

The process P_2 represents the first inner case statement:

```
CASE store OF
   a | ca: b,
   b | ab: c
ELSE a
ESAC,
```

This is evaluated as follows:

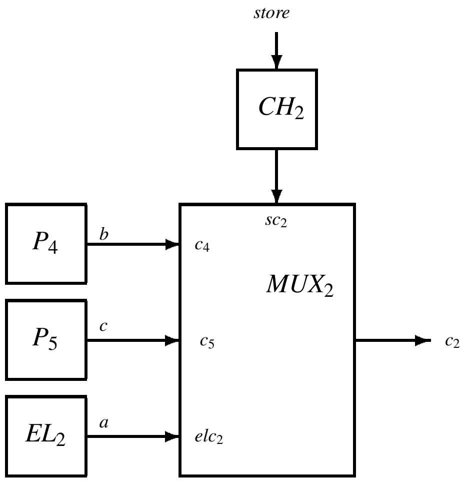

Figure 5: Connections for Inner Case Statement

$$M\text{-}Unit[\![\ \texttt{CASE}\ store\ \cdots\ a\ \texttt{ESAC}\ ]\!](c_2)(\rho, c\text{-}ev_2)$$
$$= M\text{-}Case[\![\ \texttt{CASE}\ \ store\ \texttt{OF}\ \ \cdots\ \ \texttt{ELSE}\ a\ \texttt{ESAC}\ ]\!](c_2)(\rho, c\text{-}ev_2)$$
$$= (P_2, c\text{-}ev_3)$$

where

$$P_2 = (\!\!|\ \Big(\ CH_2\ \mid\ P_4\ \mid\ P_5\ \mid\ EL_2\ \mid\ MUX_2\ \Big)\backslash chandecls_3\ |\!\!)$$
$$(CH_2, c\text{-}ev_5) = M\text{-}Unit[\![store]\!](sc_2)(\rho, c\text{-}ev_2)$$
$$(P_4, c\text{-}ev_6) = M\text{-}Unit[\![b]\!](c_4)(\rho, c\text{-}ev_5)$$
$$(P_5, c\text{-}ev_7) = M\text{-}Unit[\![c]\!](c_5)(\rho, c\text{-}ev_6)$$
$$(EL_2, c\text{-}ev_3) = M\text{-}Unit[\![a]\!](elc_2)(\rho, c\text{-}ev_7)$$
$$MUX_2 = (\!\!|\ Chooser_{[pump, pump, [set_4, set_5]]}(sc_2, c_4, c_5, elc_2\ /\ oc_2)\ |\!\!)$$
$$set_4 = M\text{-}ConstSet[\![a\ \mid\ ca]\!](\rho, \sigma)$$
$$set_5 = M\text{-}ConstSet[\![b\ \mid\ ab]\!](\rho, \sigma)$$
$$chandecls_3 = (\!\!|\ \{sc_2\text{: } pump, c_4\text{: } pump, c_5\text{: } pump, elc_2\text{: } pump\}\ |\!\!)$$
$$\{sc_2, c_4, c_5, elc_2\} = make\text{-}ChanVars(4, \mathsf{dom}\,\rho \cup \{c_2\})$$

with $c\text{-}ev_2 = c\text{-}ev_3 = c\text{-}ev_5 = c\text{-}ev_6 = cev_7$, in this case. Note that the chooser sets set_4, set_5 are given by set *expressions*. Figure 5 shows how the different terms 'connect' together.

The chooser unit CH_2 is simply a signal, and so a call to *Chan-Proc* is made, with the input *store*, and the output sc_2; the chooser can then act as the controller input to the multiplexor process $Chooser_{[pump, pump, [set_4, set_5]]}$:

$$M\text{-}Unit[\![store]\!](sc_2)(\rho, c\text{-}ev_2) = (CH_2, c\text{-}ev_5)$$

where

$$CH_2 = (\!\!|\ Chan\text{-}Proc_{[pump]}(store\ /\ sc_2)\ |\!\!)$$

The two case-items and else-item can now be evaluated. These are all simply constants and so they each call the primitive process $Constant_{[pump,tag]}$, where *tag* is either b, c or a. For example, the first arm of the case statement yields the following:

$$M\text{-}Unit[\![b]\!](c_4)(\rho, c\text{-}ev_5) = (P_4, c\text{-}ev_6)$$

where

$$P4 = (\!|\ Constant_{[pump,b]}(c_4)\ |\!)$$

The other two items are similar with b replaced by c and a respectively, and the appropriate channel name used.

Now the multiplexor process MUX_2 can be evaluated. First, the operation *M-ConstSet* is used to extract the chooser set expressions for each arm in the case statement. Here, it yields the set expressions:

$$\begin{aligned}
set_4 &= M\text{-}ConstSet(constset\text{-}of([\![a \mid ca\colon b]\!])(\rho,\sigma) \\
&= (\!|\ \{a\} \cup \{ca\}\ |\!) \\
set_5 &= M\text{-}ConstSet(constset\text{-}of([\![b \mid ab\colon c]\!])(\rho,\sigma) \\
&= (\!|\ \{b\} \cup \{ab\}\ |\!)
\end{aligned}$$

The instantiation of the generic process $Chooser_{[pump,pump,[set_4,set_5]]}$ results in the following process definition:

$$\begin{aligned}
CD_1 = (\!|\ &\text{process } Chooser_{[pump,pump,[set_4,set_5]]}(sc_2, c_4, c_5, elc_2\colon pump \ /\ oc_2\colon pump) \triangleq \\
&\square\, v,\ x_1,\ x_2,\ y\colon pump \ \cdot\ sc_2(v), c_4(x_1), c_5(x_2), elc_2(y), oc_2(caseexpr) \\
&\quad ::\ Chooser_{[pump,pump,[set_4,set_5]]}(sc_2, c_4, c_5, elc_2 \ /\ oc_2)\ |\!)
\end{aligned}$$

where

$$\begin{aligned}
caseexpr = (\!|\ &\textbf{case } v: \\
&\quad set_4 \rightarrow x_1 \\
&\quad set_5 \rightarrow x_2 \\
&\textbf{else } y \\
&\textbf{end}\ |\!) \\
set_4 = (\!|\ &\{a\} \cup \{ca\}\ |\!) \\
set_5 = (\!|\ &\{b\} \cup \{ab\}\ |\!)
\end{aligned}$$

The process term P_2 corresponds to the intuitive meaning of the first inner case statement. The output channel sc_2 of CH_2 connects to the input sc_2 of the multiplexor MUX_2. Each different output from CH_2, on input channel *store*, enables a different term in one of P_4, P_5 or EL_2 to be produced on the output. For example, when $store = b$ then $oc_2 = c$.

The second inner case statement is handled in exactly the same way and results in a term P_3; the details of its construction are omitted. Now P_2 and P_3 can be incorporated into the term *let-obj*, representing the outer case statement.

F Recursive Let - Local Binding of Signals

Structure:

$$\begin{array}{|rcl|} \hline \textit{RecLet} & ::= & \texttt{RECLET}\ \textit{signalbind}_1\ ,\ \ldots\ ,\ \textit{signalbind}_{k+1}\ \texttt{IN}\ \textit{unit} \\ \textit{SignalBind} & ::= & \textit{signal} : \textit{type}\ =\ \textit{unit} \\ \hline \end{array}$$

The recursive let construct is an alternative method for introducing mutually recursive signal binding. It does not form part of Core ELLA, which would use `LET` / `MAKE` / `JOIN` constructs to produce the same effect. It was first used in *BKE*[BGL+91, BGL+92] as the method for mutually recursive signal binding. For example, the following two expressions have the same meaning, the left-hand one using `LET` / `MAKE` / `JOIN` , and the right-hand one using recursive let:

```
BEGIN
     MAKE F: x.
     MAKE G: y.
     JOIN y -> x.
     JOIN x -> y.
     OUTPUT H(x,y)
END.
```

```
RECLET x = F(y),
       y = G(x)
IN H(x,y).
```

The recursive let identifies a local signal name sig_i with the output of the right-hand side unint un_i for each signal binding $sigbnd_i$ in a binding sequence $sigbnd_1, \cdots, sigbnd_{k+1}$. These local signals may then be used in the final unit *un* which determines the output of the recursive let statement.

Static semantics rules:

(a) $\mathsf{card}\ \textit{extract-local-signals}[\![\textit{sigbnd}_1\ ,\ \ldots\ ,\ \textit{sigbnd}_{k+1}]\!] = k+1$

(b) $\textit{lcl-st-env} = \textit{extract-local-binding}[\![\textit{sigbnd}_1\ ,\ \ldots\ ,\ \textit{sigbnd}_{k+1}]\!]$

(c) $\textit{new-st-env} = \textit{st-env} \dagger \textit{lcl-st-env}$

(d) $\forall\ i \in \{1,\ldots,k+1\} \cdot \textit{new-st-env} \vdash_{\textit{SignalBind}} \textit{sigbnd}_i$

(e) $\textit{new-st-env} \vdash_{\textit{Unit}} \textit{un}$ has type $\textit{ty}$

wf-Unit : RecLet

$\textit{st-env} \vdash_{\textit{Unit}} [\![\ \texttt{let}\ \textit{sigbnd}_1\ ,\ \ldots\ ,\ \textit{sigbnd}_{k+1}\ \texttt{in}\ \textit{un}]\!]$ has type $\textit{ty}$

(a) $\textit{st-env} \vdash_{\textit{Unit}} \textit{un}$ has type $\textit{ty}_1$

(b) $\textit{st-env} \vdash_{\textit{E-TyEq}} \textit{ty}\ \equiv\ \textit{ty}_2$

wf-SignalBind

$\textit{st-env} \vdash_{\textit{SignalBind}} [\![\textit{sig} : \textit{ty}\ =\ \textit{un}]\!]$

The above static rules state that a recursive let is well-formed if:

- the local signal names are unique within the recursive let statement
- each right-hand side unit unm_i and the output unit *unm* must be well-formed, with respect to a local static environment containing the local signal definitions, and each output type must match that of the local signal to which it is bound.

The dynamic semantics for recursive let is very similar to that for the ELLA block construct. A local environment *local-var-env* is created containing *EPA* terms for the right-hand side units unm_i, $i \in$

$\{1,\ldots,k+1\}$, with their outputs named to the local signals sig_i. The dynamic environment is also updated to create *dy-env*, which contains any local type or function declarations in the right-hand side units. The output unit *un* is evaluated in *dy-env* to form *UN*. The operation *block-join-product*, defined previously for the ELLA block semantics, is again used to form the causal product of *UN* and the objects representing the local bindings, i.e., those contained in *local-var-env*. It finally hides the local signals in the resulting object. Note that only the definitions for local type- and function-declarations are added to the dynamic environment *dy-env*.

Dynamic semantics equations:

$$M\text{-}Unit[\![\ \texttt{RECLET}\ sigbnd_1\ ,\ \ldots\ ,\ sigbnd_{k+1}\ \texttt{IN}\ un]\!](onm)(\rho,\sigma)\ =\ (term, new\text{-}dy\text{-}env)$$

where

$$term = block\text{-}join\text{-}product(UN, ch\text{-}set)(lcl\text{-}env)$$
$$(UN, new\text{-}dy\text{-}env) = M\text{-}Unit[\![un]\!](onm)(\rho, dy\text{-}env),$$
$$dy\text{-}ev_1 = \sigma$$
$$lcl\text{-}ev_1 = \{\}$$
$$(\forall\ i \in \{1,\ \cdots\ ,(k+1)\} \bullet$$
$$(dy\text{-}ev_{i+1}, lcl\text{-}ev_{i+1}) = M\text{-}SignalBind(sigbnd_i)(\rho, dy\text{-}ev_i, lcl\text{-}ev_i)$$
$$)$$
$$dy\text{-}env = dy\text{-}ev_{k+2}$$
$$lcl\text{-}env = lcl\text{-}ev_{k+2}$$
$$ch\text{-}set = local\text{-}channels(\mathsf{dom}\, lcl\text{-}ev_{k+2})(\rho,\sigma)$$

$$M\text{-}SignalBind[\![sig:\ ty\ =\ un]\!](\rho, dy\text{-}env, lcl\text{-}env) = (new\text{-}dy\text{-}env, new\text{-}lcl\text{-}env)$$

where

$$(UN, new\text{-}dy\text{-}ev) = M\text{-}Unit(un)(sig)(\rho, dy\text{-}env)$$
$$new\text{-}lcl\text{-}env = lcl\text{-}env \cup \{sig \mapsto UN\}$$

The auxiliary functions *extract-local-signals* and *extract-local-binding* used to extract local signal name and type information in the static semantics are defined as follows:

$$extract\text{-}local\text{-}signals : SignalBind^* \rightarrow \mathcal{F}(Name)$$

$$extract\text{-}local\text{-}signals[\![sigbnd_1\ ,\ \ldots\ ,\ sigbnd_{k+1}]\!] =$$

$$name\text{-}set$$

where

$$name\text{-}set = \{nm_1,\ \cdots\ ,\ nm_{k+1}\},$$

$$\forall\ i \in \{1, \ldots, k+1\} \bullet [\![nm_i\ =\ un_i]\!] = sigbnd_i$$

$$extract\text{-}local\text{-}binding : SignalBind^* \rightarrow StaticEnvironment$$

$$extract\text{-}local\text{-}binding[\![sigbnd_1\ ,\ \ldots\ ,\ sigbnd_{k+1}]\!] =$$

$$\{nm_1 \mapsto vt_1,\ \cdots\ ,\ nm_{k+1} \mapsto vt_{k+1}\}$$

where

$$\forall\ i \in \{1, \ldots, k+1\} \bullet$$

$$[\![nm_i : ty_i\ = un_i]\!] = sigbnd_i \wedge vt_i = [\![\ \texttt{vartype}\ ty_i]\!]$$

G Semantics Index